IBM

Implementing CICS Web Services

Configuring and securing Web services in CICS Transaction Server

Connecting CICS to a service integration bus

Enabling atomic Web services

Nigel Williams
Robert Herman
Luis Aused Lopez
Mike Ebbers

Redbooks

ibm.com/redbooks

International Technical Support Organization

Implementing CICS Web Services

October 2007

SG24-7206-02

Note: Before using this information and the product it supports, read the information in "Notices" on page xi.

Third Edition (October 2007)

This edition applies to CICS Transaction Server Version 3.1.

Contents

Notices . . . xi
Trademarks . . . xii

Preface . . . xiii
The team that wrote this book . . . xiv
Become a published author . . . xvii
Comments welcome . . . xvii

Summary of changes . . . xix
December 2006, Second Edition . . . xix
October 2007, Third Edition . . . xix

Part 1. Introduction . . . 1

Chapter 1. Overview of Web services . . . 3
1.1 Introduction . . . 4
1.2 Service-oriented architecture . . . 4
1.2.1 Characteristics . . . 5
1.2.2 Web services versus service-oriented architectures . . . 6
1.3 Web services . . . 7
1.3.1 Properties of a Web service . . . 7
1.3.2 Core standards . . . 8
1.3.3 Web Service Interoperability Basic Profile 1.0 . . . 10
1.3.4 Additional standards . . . 11
1.4 SOAP . . . 12
1.4.1 The envelope . . . 12
1.4.2 Communication styles . . . 17
1.4.3 Encodings . . . 17
1.4.4 Messaging modes . . . 17
1.5 WSDL . . . 18
1.5.1 WSDL Document . . . 19
1.5.2 WSDL document anatomy . . . 19
1.5.3 WSDL definition . . . 24
1.5.4 WSDL bindings . . . 30
1.6 Summary . . . 31

Chapter 2. CICS support for Web services . . . 33
2.1 Overview . . . 34
2.2 CICS as a service provider . . . 37

2.2.1 Preparing to run a CICS application as a service provider 38
2.2.2 Processing the inbound service request 40
2.3 CICS as a service requester 41
2.3.1 Preparing to run a CICS application as a service requester 42
2.3.2 Processing the outbound service request 44
2.3.3 Local optimization 45
2.4 CICS resources for Web services 47
2.4.1 URIMAP 47
2.4.2 PIPELINE 49
2.4.3 WEBSERVICE 53
2.4.4 TCPIPSERVICE 55
2.4.5 Resources checklist 55
2.5 Message handlers 57
2.5.1 SOAP message handlers 57
2.5.2 Channels and containers 59
2.6 Tools for developing CICS Web services 61
2.6.1 CICS Web services assistant 62
2.6.2 Web services assistant utility programs 63
2.6.3 WebSphere Developer for zSeries 65
2.7 Catalog manager example application 67
2.7.1 The base application 67
2.7.2 Web services support for the catalog example application 69

Part 2. Web service configuration 71

Chapter 3. Web services using HTTP 73
3.1 Preparation 74
3.1.1 Software checklist 75
3.1.2 Definition checklist 75
3.1.3 The sample application 76
3.2 Configuring CICS as a service provider 76
3.2.1 Configuring code page support 77
3.2.2 Configuring CICS 77
3.2.3 Configuring WebSphere Application Server on Windows 87
3.2.4 Testing the configuration 90
3.3 Configuring CICS as a service requester 94
3.3.1 Configuring CICS 95
3.3.2 Configuring WebSphere Application Server for z/OS 98
3.3.3 Testing the configuration 100
3.4 Configuring for high availability 101
3.4.1 TCP/IP load balancing 102
3.4.2 High availability configuration 102
3.4.3 Routing inbound Web service requests 102

3.5 Problem determination 104
3.5.1 Error calling dispatch service - INVREQ 104

Chapter 4. Web services using WebSphere MQ 111
4.1 Preparation 112
4.1.1 Software checklist 113
4.1.2 Definition checklist 113
4.2 WebSphere MQ configuration 114
4.2.1 Adding WebSphere MQ support to CICS 114
4.2.2 Defining the queues 115
4.2.3 Defining the trigger process 116
4.3 Configuring CICS as a service provider using WMQ 116
4.3.1 Configuring the service provider pipeline 117
4.4 Configuring CICS as service requester using WMQ 121
4.4.1 Configuring the Catalog application 122
4.4.2 Configuring WebSphere Application Server on Windows 124
4.5 Testing the WMQ configuration. 124
4.6 High availability with WMQ 126

Chapter 5. Connecting CICS to the service integration bus 129
5.1 Overview of the service integration bus 130
5.1.1 Why you would connect CICS to a bus. 131
5.2 Preparation 132
5.2.1 Software checklist 134
5.2.2 Definition checklist 134
5.3 Configuring CICS for a gateway service 135
5.3.1 Updating the CICS-supplied sample WSDL file 135
5.3.2 Creating a URIMAP for the WSDL file 135
5.3.3 Testing the retrieval of the WSDL file from a Web browser 137
5.4 Creating a gateway service on the bus 139
5.4.1 Identifying the bus to be used 139
5.4.2 Creating a Web services gateway instance 140
5.4.3 Creating a gateway service. 141
5.5 Testing the CICS gateway service 145
5.5.1 Publish the bus-generated WSDL. 145
5.5.2 Configuring the catalog manager J2EE application 146
5.5.3 Invoking the gateway service 147

Part 3. Security management. 149

Chapter 6. Elements of cryptography 151
6.1 The role of cryptography 152
6.2 Secret key (or symmetric) cryptography 153
6.2.1 DES. 155

6.2.2 Triple DES (TDEA) 160
6.2.3 AES 161
6.3 Public key (or asymmetric) cryptography 162
6.3.1 RSA 163
6.4 Hash functions 167
6.5 Message authentication codes 171
6.5.1 Block cipher-based MACs 171
6.5.2 Hash function-based MACs 172
6.6 Digital signatures 173
6.6.1 Using DSA for digital signatures 174
6.6.2 Using RSA for digital signatures 176
6.6.3 Comparing RSA with DSA for digital signatures 179
6.7 Public key digital certificates 180
6.7.1 tbsCertificate 182
6.7.2 Standard extensions for X.509 V3 digital certificates 187
6.7.3 Certification paths 189
6.8 Certificate revocation lists 191
6.8.1 Extensions for entries in a CRL 194
6.8.2 Extensions for a CRL 195
6.8.3 Security considerations when using digital certificates 195
6.9 Key agreement protocols 196
6.9.1 The RSA key agreement protocol 197
6.9.2 The Diffie-Hellman key agreement protocol 197
6.10 Transport Layer Security (TLS) 1.0 protocol 199
6.10.1 TLS overview 200
6.10.2 Cipher suites 202
6.10.3 Alert protocol 206
6.10.4 Handshake protocol 207

Chapter 7. Crypto hardware and ICSF 219
7.1 Cryptographic hardware 220
7.1.1 CP Assist for Cryptographic Functions (CPACF) 221
7.1.2 Crypto Express 2 feature 222
7.1.3 Comparison of CPACF, CEX2C, and CEX2A 223
7.1.4 Other cryptographic hardware 224
7.2 ICSF 225
7.2.1 ICSF callable services 225
7.2.2 ICSF administration 226
7.3 How CICS uses ICSF 227
7.4 ICSF services used by CICS WS-Security support 229

Chapter 8. Securing Web services 235
8.1 Traditional CICS security 236

8.1.1 CICS user IDs 236
8.2 Security exposures 237
8.3 Transport security 239
8.3.1 HTTP transport 239
8.3.2 WebSphere MQ transport 251
8.4 SOAP message security 253
8.4.1 CICS and SOAP message security. 256
8.4.2 WebSphere and SOAP message security 266
8.5 Comparison of transport level and SOAP message security 269
8.6 Securing CICS Web services using the service integration bus 270
8.7 WebSphere Datapower SOA appliances 272
8.8 Identity assertion 273
8.8.1 Trust token model 274
8.8.2 Presumed trust model 274

Chapter 9. Security scenarios 275
9.1 Preparation 276
9.1.1 Software checklist 276
9.1.2 Definition checklist 277
9.2 Basic security configuration. 278
9.2.1 Setting up basic security configuration 279
9.2.2 Testing the basic security configuration 279
9.3 Setting the user ID on a URIMAP definition 280
9.3.1 Defining the URIMAP 281
9.3.2 Permitting access to user ID CICSNW 284
9.3.3 Testing user ID on URIMAP resource definition 284
9.4 Enabling SOAP message security with HTTP. 285
9.4.1 Configuring the service requester 286
9.4.2 Configuring CICS 293
9.4.3 Testing SOAP message security 298
9.4.4 SOAP fault messages 299
9.5 Enabling SSL/TLS 301
9.5.1 Creating a key ring and certificates on z/OS for CICS 302
9.5.2 Enabling an SSL/TLS connection from WebSphere 304
9.5.3 Configuring CICS support for SSL/TLS. 309
9.5.4 Testing SSL/TLS 310
9.6 Enabling SOAP message security with WMQ. 312
9.6.1 Configuring CICS to use WMQ 313
9.6.2 Configuring the service requester 315
9.6.3 Header processing program 315
9.6.4 Configuring the service provider 316
9.6.5 Configuring WebSphere MQ for security 318
9.6.6 Testing security with WMQ 319

Chapter 10. Security scenarios using CICS WS-Security support 323
10.1 Preparation 324
10.1.1 Software checklist 324
10.1.2 Definition checklist 325
10.2 Basic security configuration. 327
10.2.1 Creating a RACF key ring 327
10.2.2 Specifying the security SIT parameters 328
10.2.3 Testing the basic security configuration 328
10.2.4 Configuring the pipeline 329
10.2.5 Setting a user ID on a URIMAP definition. 330
10.3 Basic authentication 330
10.3.1 Configuring the service requester for basic authentication 332
10.3.2 Configuring CICS 339
10.3.3 Testing basic authentication 341
10.4 Certificate and key pair generation 347
10.5 Signing a SOAP message. 358
10.5.1 Configuring the service requester for signature processing 360
10.5.2 Configuring CICS for signature processing. 375
10.5.3 Testing the signature scenario 378
10.6 Encrypting a SOAP message 385
10.6.1 Configuring the service requester for encryption 387
10.6.2 Configuring CICS for encryption 398
10.6.3 Testing the encryption scenario 400

Part 4. Transaction management. 407

Chapter 11. Introduction to Web services: Atomic transactions 409
11.1 Beginner's guide to atomic transactions 410
11.1.1 What is a classic transaction. 411
11.1.2 Mapping from classic transactions to WS-Atomic Transaction . . . 415
11.2 WS-Addressing 418
11.2.1 Endpoint references 419
11.2.2 Message information headers. 422
11.2.3 SOAP binding for endpoint references 424
11.3 WS-Coordination 425
11.3.1 Coordination service 426
11.3.2 CreateCoordinationContext. 427
11.3.3 CreateCoordinationContextResponse 429
11.3.4 Register. 432
11.3.5 Register response 434
11.3.6 Two applications with their own coordinators 434
11.3.7 Addressing requirements for WS-Coordination message types . . 436
11.4 WS-Atomic Transaction. 436
11.4.1 Completion protocol 438

11.4.2 Two-Phase Commit protocol 439
11.4.3 Two applications with their own coordinators (continued) 441
11.4.4 Addressing requirements for WS-AT message types 442
11.4.5 CICS TS V3.1 and resynchronization processing 443

Chapter 12. Enabling atomic transactions 447
12.1 Enabling atomic transactions in CICS 448
12.1.1 CICS to CICS configuration 448
12.1.2 More elaborate CICS to CICS configuration 458
12.2 Enabling atomic transactions in WebSphere 460

Chapter 13. Transaction scenarios 463
13.1 Introduction to our scenarios 464
13.1.1 Software checklist 465
13.1.2 Definition checklist 465
13.2 The simple atomic transaction scenario 467
13.2.1 Setting up CICS for the simple scenario 469
13.2.2 Creating the AtomicClient and ITSO.ORDER table 473
13.2.3 Testing the simple scenario 489
13.3 The daisy chain atomic transaction scenario 510
13.3.1 Setting up CICS for the daisy chain scenario 512
13.3.2 Creating DispatchOrderAtomic and the ITSO.DISPATCH table . . 514
13.3.3 Testing the daisy chain scenario 517
13.4 Transaction scenario summary 524

Part 5. Appendixes 525

Appendix A. Sample handler programs 527
A.1 Sample message handler program - CIWSMSGH 528
A.2 Sample header processing program - CIWSSECH 533
A.3 Sample handler program - SNIFFER 539
A.4 Sample XML parser program - MYPARSER 550
A.5 Sample header processing program - CIWSSECR 555
A.6 Sample header processing program - CIWSSECS 562
A.7 Sample header processing program - WSATHND 567

Appendix B. How the DES, AES, SHA-1, and HMAC algorithms work . . 573
B.1 How DES works 574
B.2 How AES works 578
B.3 How SHA-1 works 583
B.3.1 Definitions 583
B.3.2 SHA-1 preprocessing 584
B.3.3 SHA-1 hash computation 586
B.4 How the HMAC algorithm of FIPS PUB 198 works 588

Abbreviations and acronyms 591

Related publications 593
IBM Redbooks 593
Other publications 593
Online resources 594
How to get IBM Redbooks 594
Help from IBM 594

Index 595

Notices

Trademarks

Preface

Today more and more companies are embracing the principles of on demand business by integrating business processes end-to-end across the company and with key partners, enabling them to respond flexibly and rapidly to new circumstances. The move to an on demand business environment requires technical transformation, moving the focus from discrete applications to connected, interdependent information technology components.

Open standards such as Web services enable these components to be hosted in the environments most appropriate to their requirements, while still being able to interact easily — independent of hardware, run-time environment, and programming language.

The Web services support in CICS® Transaction Server Version 3.1 enables your CICS programs to be Web service providers and requesters. CICS supports a number of specifications including SOAP Version 1.1 and Version 1.2, and Web services distributed transactions (WS-Atomic Transaction).

This IBM® Redbook describes how to configure CICS Web services support for HTTP-based and WebSphere® MQ-based solutions, and demonstrates how Web services can be used to integrate J2EE™ applications running in WebSphere Application Server with COBOL programs running in CICS.

It begins with an overview of Web services standards and the Web services support provided by CICS TS V3.1. Complete details for configuring CICS Web services using both HTTP and WebSphere MQ are provided next, along with the steps for using Web services to connect to CICS from a service integration bus. The book then shows how CICS Web services can be secured using a combination of Web Services Security (WS-Security) and transport-level security mechanisms such as SSL/TLS. Finally, it demonstrates how atomic Web services transactions can be configured to allow WebSphere and CICS resource updates to be synchronized.

In this book we concentrate on the implementation specifics such as security, transactions, and availability. The companion book *Developing CICS Web Services* (SG24-7126) presents detailed information about developing CICS Web services.

The team that wrote this book

This International Technical Support Organization (ITSO) IBM Redbooks® publication (third edition) was produced by a team of specialists from around the world working at the Product Solutions and Support Center in IBM Endicott, USA.

Nigel Williams was the project leader for this book. He is a Certified IT Specialist working in the IBM Design Center for On Demand Business in Montpellier. He specializes in core business transformation, connectors, and service-oriented architectures. He is the author of several papers and IBM Redbooks publications, and he speaks frequently on CICS and WebSphere topics. Previously, Nigel worked at the Hursley software lab as a software developer, in systems test, and as customer support for the CICS Early Support Program. He holds a degree in Mathematics and Economics from Surrey University.

Luis Aused Lopez is an IT Specialist for IBM Global Services in Spain, working in Business Consulting Services (BCS) in the travel and transportation sectors. As an assignee he works in the zSeries® Benchmark Center in IBM Montpellier. He has worked for IBM for over ten years. During this time, Luis has developed several J2EE applications for WebSphere running on different platforms, including zSeries, iSeries®, Linux®, and Windows®. His areas of expertise include application development, WebSphere, Java™ performance, DB2®, and eTicketing. He is an author of several IBM Redbooks publications and holds a degree in Physics from Complutense University, Madrid, Spain.

Robert Herman was a Senior IT Specialist, Systems Management Integrator with IBM Global Services in Endicott, New York until his death in 2007. He had 27 years of experience supporting CICS and related products for a variety of IBM internal and external customer accounts. Bob worked on several IBM Redbooks, including *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.2, SG24-6284*.

Mike Ebbers is a certified Consulting IT Specialist in the ITSO Poughkeepsie Center. He has spent 34 years with IBM doing technical support and education for mainframe systems.

The first edition was produced by this team and several additional colleagues in IBM Montpellier, France. Biographies of the additional first edition authors and a photograph of the team follow.

Special note: This book is dedicated to Bob Herman. He died before his work on this third edition could be published. Bob is remembered by the rest of the team as a dedicated professional who paid meticulous attention to detail and was able to write about complex topics such as cryptography in depth but with an elegant simplicity. Bob was also appreciated as a mentor and a guardian of the English language. He will be greatly missed on future ITSO residencies.

The first edition team in the foyer at IBM Montpellier: Robert Herman, Luis Aused Lopez, Grant Ward Able, Nigel Williams, Paolo Chieregatti, and Tommy Joergensen. Steve Wall is missing from the picture.

Grant Ward Able is a Software Engineer working for IBM in Hursley, United Kingdom. He has spent five years in the CICS Transaction Server team as a developer and a tester and in the Solution Test team, working with CICS and WebSphere. Previously, Grant worked for 15 years as a CICS systems programmer. He currently works in the CICS Service Flow runtime team.

Paolo Chieregatti is an IT specialist working for IBM Software Group in Italy. He has 20 years of experience in IT working mainly on IBM mainframes. His areas of expertise include CICS, WebSphere MQ, WebSphere Application Server, and legacy application transformation and integration. He speaks frequently on CICS and WebSphere topics. Before joining IBM, Paolo worked for the Candle® Corporation. He has worked as a CICS systems programmer and project manager.

Tommy Joergensen is a Senior IT Specialist working for IBM Global Services in IBM Denmark. He has more than 25 years of experience working in CICS technical support, including three years at IBM Hursley. In recent years he has delivered services at large accounts in Denmark for both the CICS and WebSphere products. Tommy is the IBM representative in the CICS working group of the Nordic Share Guide organization.

Steve Wall is an IT specialist working in the System z™ Benchmark Center. He worked for the CICS Transaction Server Development organization at Hursley, United Kingdom, for over 20 years before joining the PSSC. Steve has a degree in Linguistic and International Studies from the University of Surrey. He has written and taught extensively about CICS e-business enablement using CICS Web Support and the CICS Transaction Gateway.

Thanks to the following people for their contributions to this project:

Phil Hanson and Mark Cocker of IBM Hursley for supporting this project.

Pascal Tillard for his support setting up the WebSphere Application Server for z/OS® environment and for assisting with the setup of the service integration bus.

Mike Brooks of IBM Hursley for explaining the CICS WS-AtomicTransaction support and making direct contributions to Part 4 of this book.

Ken Ray of IBM UK for his support in setting up WebSphere MQ.

Ian Noble and Oliver Fenton of IBM Hursley for supplying sample programs.

Mike Adams, Fraser Bohm, Ivan Hargreaves, Peter Havercan, Ian Mitchell, Daniel Would and William Yates of IBM Hursley, Derek Ho and Peter Birk from IBM Austin, and Jeff Oestrich of IBM Raleigh for supplying technical advice during the residencies.

The team that wrote the Redbook *Developing for CICS Web Services,* SG24-7126: Chris Rayns, Jim Hollingsworth, Chris Backhouse, David Evans, and Isabel Arnold.

The team that wrote the Redbook *Web Services Handbook for WebSphere Application Server 6.1,* SG24-7257: Ueli Wahli, Owen Burroughs, Owen Cline, Alex Go and Larry Tung.

Tony Delmenico and Steve Webb from the team that worked on the Redbook *Securing Access to CICS Within an SOA* (SG24-5756).

Philippe Richard of IBM Montpellier, and Rich Conway and Bob Haimowitz of the ITSO Poughkeepsie, for providing excellent systems support.

Arnauld Desprets and Patrick Kappeler of IBM Montpellier for advice on the security scenarios.

Ella Buslovich and Alison Chandler of the ITSO Poughkeepsie for help with the graphics and editing respectively.

The following people for the significant amount of time that they have spent reviewing and for their detailed review comments: Cheryll Clark of IBM Australia, Alan Roessle of IBM Montpellier, Phil Wakelin and Robert Harris of IBM Hursley.

Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbooks document dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll team with IBM technical professionals, Business Partners and/or customers.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our Redbooks to be as helpful as possible. Send us your comments about this or other Redbooks in one of the following ways:

- Use the online **Contact us** review redbook form found at:

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbook@us.ibm.com

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYJ; HYJ Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Summary of changes

This section describes the technical changes made in this edition of the book and in previous editions. This edition may also include minor corrections and editorial changes that are not identified.

Note: This book is based on the Web services support in CICS TS V3.1. For information about the Web services enhancements in CICS TS V3.2 refer to *Web Services Guide*, SC34-6838.

December 2006, Second Edition

This revision reflects the addition or modification of the following information:

Part 3, Security Management has been greatly enlarged. Two new chapters were added, and the existing material was reorganized and expanded as well.

New Chapter 6, “Elements of cryptography” on page 151 provides a discussion of basic concepts in cryptography; Chapter 10, “Security scenarios using CICS WS-Security support” on page 323 demonstrates how you can secure CICS Web services using the CICS-supplied message handler DFHWSSE1.

New Appendix B “How the DES, AES, and HMAC algorithms work” provides a detailed explanation of how the algorithms work.

October 2007, Third Edition

Part 3, Security Management has been further enlarged with one new chapter and modifications to the existing material.

New Chapter 7, “Crypto hardware and ICSF” on page 219 introduces IBM cryptographic hardware and describes how CICS WS-Security support uses ICSF (the Integrated Cryptographic Service Facility) and hardware cryptography.

The modifications made to existing chapters include:

- Enhancements to Chapter 6, “Elements of cryptography” on page 151.
- Additional information added to Chapter 8, “Securing Web services” on page 235.
- Additional information about how the SHA-1 algorithm works was added to Appendix B, “How the DES, AES, SHA-1, and HMAC algorithms work” on page 573.

Part 1

Introduction

In this part we give a broad overview of different Web services technologies and then explain how to use Web services in CICS Transaction Server V3.1.

1

Overview of Web services

This chapter focuses on some of the architectural concepts that must be considered on a Web services project. We define and discuss *service-oriented architecture* (SOA) and the relationship between SOAs and Web services. We then take a closer look at Web services, a technology that enables you to invoke applications using Internet protocols and standards. The technology is called “Web services” because it integrates services (applications) using Web technologies (the Internet and its standards).

1.1 Introduction

There is a strong trend for companies to integrate existing systems to implement IT support for business processes that cover the entire business cycle. Today, interactions already exist using a variety of schemes that range from very rigid point-to-point electronic data interchange (EDI) interactions to open Web auctions. Many companies have already made some of their IT systems available to all of their divisions and departments, or even their customers or partners on the Web. However, techniques for collaboration vary from one case to another and are thus proprietary solutions; systems often collaborate without any vision or architecture.

Thus there is an increasing demand for technologies that support the connecting or sharing of resources and data in a very flexible and standardized manner. When technologies and implementations vary across companies and even within divisions or departments, unified business processes cannot be smoothly supported by technology. Integration has been developed only between units that are already aware of each other and that use the same static applications.

Furthermore, there is a need to structure large applications into building blocks in order to use well-defined components within different business processes. A shift towards a *service-oriented* approach will not only standardize interaction, but also allow for more flexibility in the process. The complete value chain within a company is divided into small modular functional units, or services. A service-oriented architecture thus has to focus on how services are described and organized to support their dynamic, automated discovery and use.

Companies and their sub-units should be able to easily provide services. Other business units can use these services to implement their business processes. This integration can be ideally performed during the runtime of the system, not just at the design time.

1.2 Service-oriented architecture

This section is a short introduction to the key concepts of a service-oriented architecture. The architecture makes no statements about the infrastructure or protocols it uses. Therefore, you can implement a service-oriented architecture using technologies other than Web technologies.

As shown in Figure 1-1, a service-oriented architecture contains three basic components:

- A service provider

  The *service provider* creates a Web service and possibly publishes to the service broker the information necessary to access and interface with the Web service.

- A service broker

  The *service broker* (also known as a *service registry*) makes the Web service access and interface information available to any potential service requester.

- A service requester

  The *service requester* binds to the service provider to invoke one of its Web services, having optionally located entries in the broker registry using various find operations.

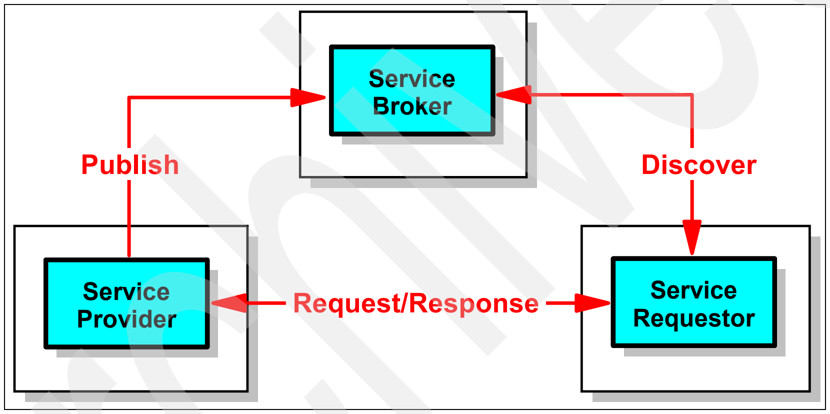


Figure 1-1 Service-oriented architecture components and operations

Each component can also act as one of the two other components. For instance, if a service provider needs information that it can only acquire from some other service, it acts as a service requester while still serving the original request.

1.2.1 Characteristics

A service-oriented architecture enables a loose coupling between the participants. Such a loose coupling provides greater flexibility because of the following characteristics:

- Old and new functional blocks are encapsulated into components that work as services.

- Functional components and their interfaces are separated. Therefore, new interfaces can be plugged in more easily.
- Within complex applications, the control of business processes can be isolated. A business rules engine can be incorporated to control the workflow of a defined business process. Depending on the state of the workflow, the engine calls the respective services.

1.2.2 Web services versus service-oriented architectures

A service-oriented architecture has been used under various guises for many years. It can be (and has been) implemented using a number of different distributed computing technologies, such as CORBA and messaging middleware. The effectiveness of service-oriented architectures in the past has always been limited by the ability of the underlying technology to interoperate across the enterprise.

Web services technology is an ideal technology choice for implementing a service-oriented architecture because:

- Web services are based on standards, and standards promote interoperability. Interoperability is a key business advantage within the enterprise and is crucial in B2B scenarios.
- Web services are widely supported across the industry. For the very first time, all major vendors are recognizing and providing support for Web services. The Web Services Interoperability Organization (WS-I) is an organization that promotes open interoperability between Web services regardless of the platforms, operating systems, or programming languages used.
- Web services are platform and language neutral. There is no bias for or against a particular hardware or software platform. Web services can be implemented in any programming language or toolset. This is important because there will be continued industry support for the development of standards and interoperability between vendor implementations.
- This technology provides a migration path to gradually enable existing business functions as Web services.
- This technology supports synchronous and asynchronous, RPC-based, and complex message-oriented exchange patterns.

Conversely, there are many Web services implementations that are *not* a service-oriented architecture. For example, the use of Web services to connect two heterogeneous systems directly together is not an SOA. These uses of Web services solve real problems and provide significant value on their own. They may form the starting point of an SOA.

In general, an SOA has to be implemented at an enterprise or organizational level in order to achieve many of the benefits.

For more information about the relationship between Web services and service-oriented architectures refer to the IBM Redbook *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303.

1.3 Web services

If we had to describe Web services using just one sentence, we would use the following:

> *Web services are self-contained, modular applications that can be described, published, located, and invoked over a network.*

Web services perform encapsulated business functions, ranging from simple request-reply to full business process interactions. These services can be new applications or wrapped around existing business functions to make them network-enabled. Services can rely on other services to achieve their goals.

The World Wide Web Consortium (W3C) Services Architecture Working Group defines a Web service as follows:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, *typically conveyed using HTTP with an XML serialization* in conjunction with other Web-related standards.

It is important to note from this definition that a Web service is *not* constrained to use SOAP over HTTP/S as the transport mechanism. Web services are equally at home in the messaging world.

1.3.1 Properties of a Web service

All Web services share the following properties:

- **Web services are self-contained**.

  On the client side, no additional software is required. A programming language with XML and HTTP client support is enough to get you started. On the server side, merely an HTTP server and a SOAP server are required.

- **Web services are self-describing**.

  A Web Service Description Language (WSDL) file provides all the information you need to implement a Web service as a provider or to invoke a Web service as a requester.
- **Web services can be published, located, and invoked across the Web**.

  The service requester uses established lightweight Internet standards such as HTTP to invoke the service provider. It leverages the existing infrastructure.
- **Web services are modular**.

  Simple Web services can be aggregated to form more complex ones, either using workflow techniques or by calling lower-layer Web services from a Web service implementation. Web services can be chained together to perform higher-level business functions. This shortens development time and enables best-of-breed implementations.
- **Web services are language-independent and interoperable**.

  The client and server can be implemented in different environments. Any language can be used to implement Web service clients and servers.
- **Web services are inherently open and standards-based**.

  XML and HTTP are the major technical foundation for Web services. A large part of the Web service technology has been built using open source projects. Therefore, vendor independence and interoperability are realistic goals.
- **Web services are loosely coupled**.

  A service requester has to know the interface to a Web service but not the details of how it has been implemented.
- **Web services provide programmatic access**.

  The approach provides no graphical user interface; it operates at the code level.
- **Web services provide the ability to wrap existing applications**.

  Existing applications can easily be integrated into the service-oriented architecture by implementing a Web service as an interface to the application.

1.3.2 Core standards

Web services are built upon four core standards: XML, SOAP, WSDL, and UDDI. Each standard is described briefly in this section.

Extensible Markup Language (XML)

XML is the foundation of Web services. However, since much information has already been written about XML, we do not describe it in this document. You can find information about XML at:

http://www.w3.org/XML/

SOAP

Originally proposed by Microsoft®, SOAP was designed to be a simple and extensible specification for the exchange of structured, XML-based information in a decentralized, distributed environment. As such, it represents the main means of communication between the three actors in an SOA: the service provider, the service requester, and the service broker. A group of companies, including IBM, submitted SOAP to the W3C for consideration by its XML Protocol Working Group. There are currently two versions of SOAP: Version 1.1 and Version 1.2.

The SOAP 1.1 specification contains three parts:

- An *envelope* that defines a framework for describing message content and processing instructions. Each SOAP message consists of an envelope that contains an arbitrary number of headers and one body that carries the payload. SOAP messages might contain faults; faults report failures or unexpected conditions.
- A set of *encoding rules* for expressing instances of application-defined data types.
- A *convention* for representing remote procedure calls and responses.

A SOAP message is, in principle, independent of the transport protocol which is used, and can, therefore, potentially be used with a variety of protocols such as HTTP, JMS, SMTP, or FTP. Right now, the most common way of exchanging SOAP messages is through HTTP.

The way SOAP applications communicate when exchanging messages is often referred to as the message exchange pattern (MEP). The communication can be either one-way messaging, where the SOAP message only goes in one direction, or two-way messaging, where the receiver is expected to send back a reply.

Due to the characteristics of SOAP, it does not matter what technology is used to implement the client, as long as the client can issue XML messages. Similarly, the service can be implemented in any language, as long as it can process XML messages.

We discuss SOAP in more detail in “SOAP” on page 12.

Note: The authors of the SOAP 1.1 specification declared that the acronym SOAP stands for Simple Object Access Protocol. The authors of the SOAP 1.2 specification decided not to give any meaning to the acronym SOAP.

Web Services Description Language (WSDL)

This standard describes Web services as abstract service endpoints that operate on messages. Both the operations and the messages are defined in an abstract manner, while the actual protocol used to carry the message and the endpoint's address are concrete.

WSDL is not bound to any particular protocol or network service. It can be extended to support many different message formats and network protocols. However, because Web services are mainly implemented using SOAP and HTTP, the corresponding bindings are part of this standard.

As of this writing, WSDL 1.1 is in use and WSDL 2.0 is a working draft. We discuss WSDL in more detail in "WSDL" on page 18.

Universal Description, Discovery, and Integration (UDDI)

The Universal Description, Discovery, and Integration standard defines a means to publish and to discover Web services. As of this writing, UDDI Version 3.0 has been finalized, but UDDI Version 2.0 is still more commonly used. For more information, refer to:

`http://www.uddi.org/`
`http://www.oasis-open.org/specs/index.php#wssv1.0`

1.3.3 Web Service Interoperability Basic Profile 1.0

Web services can be used to connect computer systems together across organizational boundaries. Therefore, defining a set of open, non-proprietary standards to which all Web services adhere maximizes the ability to connect disparate systems together.

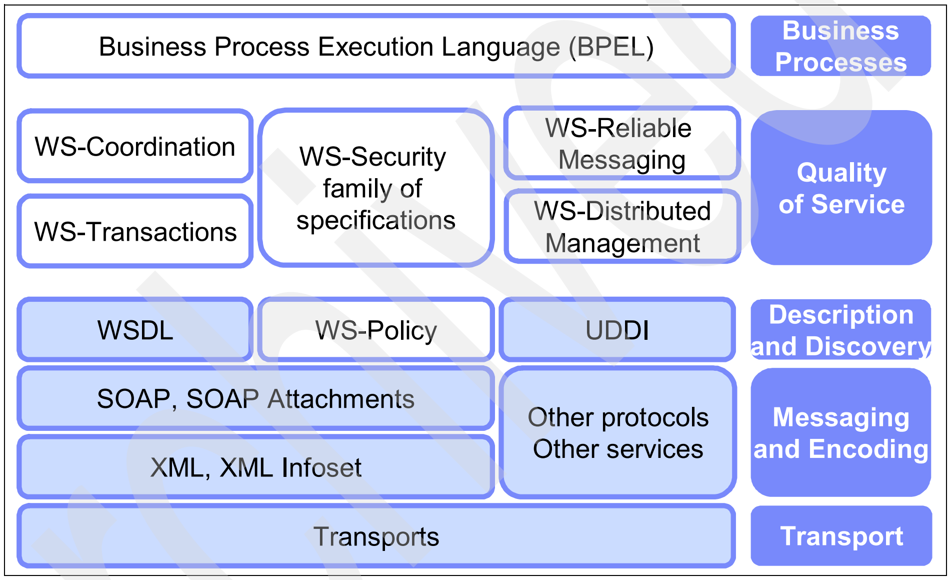
The Web Services Interoperability Organization (WS-I) is an organization that promotes open interoperabiltity between Web services regardless of the platforms, operating systems, or programming languages used. To support this cause the WS-I has released a *basic profile*; this profile outlines a set of specifications to which WSDL documents and SOAP messages sent over HTTP must adhere in order to be WS-I compliant. The full list of specifications can be found on the WS-I Web site:

`http://www.ws-i.org/`

CICS support for Web services conforms with WS-I Basic Profile 1.0. Because SOAP 1.2 is not included in WS-I Basic Profile 1.0, most Web service runtimes still support and recommend using SOAP 1.1. CICS TS V3.1 has support for both SOAP 1.1 and SOAP 1.2.

1.3.4 Additional standards

Figure provides a snapshot of the rapidly changing landscape of Web services-related standards and specifications. We do not intend it to be a strictly correct stack diagram – it just attempts to show the various standards efforts in terms of the general category to which they belong.



Web services standards

Given the current momentum behind Web services and the pace at which standards are evolving, you may also wish to refer to an online compilation of Web services standards. An online compilation is available on the IBM developerWorks® Web site at:

`http://www.ibm.com/developerworks/views/webservices/standards.jsp`

Of particular interest to those developing Web services in CICS are:

- WS-Transactions (the family of specifications that relate to transactional Web services)
- WS-Security (the family of specifications that relate to securing Web services)

1.4 SOAP

In this section we focus mainly on SOAP 1.1.

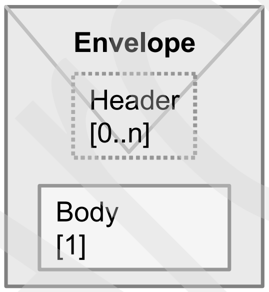
1.4.1 The envelope

A SOAP message is an *envelope* containing zero or more *headers* and exactly one *body*:

- The envelope is the root element of the XML document, providing a container for control information, the addressee of a message, and the message itself.
- Headers contain control information, such as quality of service attributes.
- The body contains the message identification and its parameters.
- Both the headers and the body are child elements of the envelope element.

Figure 1-2 shows a simple SOAP request message.

- The header tells *who* must deal with the message and *how* to deal with it. When the actor is `next` or when actor is omitted, the receiver of the message must do what the body says. Furthermore, the receiver must understand and process the application-defined `<TranID>` element.
- The body tells *what* has to be done: Dispatch an order for `quantityRequired` 1 of `itemRefNumber` 0010 to `customerID` CB1 in `chargeDepartment` ITSO.



```
<Envelope>
   <Header>
      <actor>http:// ...org/soap/actor/next</actor>
      <TranID mustUnderstand=”1”>ABCD</TranID>
   </Header>
   <Body>
      <dispachOrderRequest>
         <itemRefNumber>0010</itemRefNumber>
         <quantityRequired>1</quantityRequired>
         <customerID>CB1</customerID>
         <chargeDepartment>ITSO</chargeDepartment>
</dispatchOrderRequest>
</Body>
</Envelope>
```

Figure 1-2 Example of a simple SOAP message

Namespaces

Namespaces play an important role in SOAP messages. A namespace is simply a way of adding a qualifier to an element name to ensure that it is unique.

For example we may have a message that contains an element `<customer>`. Customers are fairly common so it is very likely that many Web services will have customer elements. To ensure we know what customer we are talking about we declare a namespace for it, for example as follows:

```
xmlns:itso=”http://itso.ibm.com/CICS/catalogApplication
```

This identifies the prefix `itso` with the declared namespace. Then whenever we reference the element `<customer>` we prefix it with the namespace as follows: `<itso:customer>`. This identifies it uniquely as a customer type for our application. Namespaces can be defined as any unique string. They are often defined as URLs since URLs are generally globally unique, and they have to be in URL format. These URLs do not have to physically exist though.

The WS-I Basic Profile 1.0 requires that all application-specific elements in the body must be namespace qualified to avoid collisions between names.

Table 1-1 shows the namespaces of SOAP and WS-I Basic Profile 1.0 used in this book.

Table 1-1 SOAP namespaces

Namespace URI	Explanation
`http://schemas.xmlsoap.org/soap/envelope/`	SOAP 1.1 envelope namespace
`http://schemas.xmlsoap.org/soap/encoding/`	SOAP 1.1 encoding namespace
`http://www.w3.org/2001/XMLSchema-instance`	Schema instance namespace
`http://www.w3.org/2001/XMLSchema`	XML Schema namespace
`http://schemas.xmlsoap.org/wsdl`	WSDL namespace for WSDL framework
`http://schemas.xmlsoap.org/wsdl/soap`	WSDL namespace for WSDL SOAP binding
`http://ws-i.org/schemas/conformanceClaim/`	WS-I Basic Profile

SOAP envelope

The `Envelope` is the root element of the XML document representing the message; it has the following structure:

```
<SOAP-ENV:Envelope .... >
   <SOAP-ENV:Header>
      <SOAP-ENV:HeaderEntry.... />
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      [message payload]
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In general, a SOAP message is a (possibly empty) set of headers plus one body. The SOAP envelope also defines the namespace for structuring messages. The entire SOAP message (headers and body) is wrapped in this envelope.

Headers

Headers are a generic and flexible mechanism for extending a SOAP message in a decentralized and modular way without prior agreement between the parties involved. They allow control information to pass to the receiving SOAP server and also provide extensibility for message structures.

Headers are optional elements in the envelope. If present, the `Header` element *must* be the first immediate child element of a SOAP `Envelope` element. All immediate child elements of the `Header` element are called *header entries*.

There is a predefined header attribute called `SOAP-ENV:mustUnderstand`. The value of the `mustUnderstand` attribute is either `1` or `0`. The absence of the SOAP `mustUnderstand` attribute is semantically equivalent to the value `0`.

If the `mustUnderstand` attribute is present in a header entry and set to `1`, the service provider must implement the semantics defined by the element:

```
<Header>
   <thens:TranID mustUnderstand=”1”>ABCD</thens:TranID>
</Header>
```

In the example, the header entry specifies that a service invocation must fail if the service provider does not support the ability to process the TranID header.

A SOAP intermediary is an application that is capable of both receiving and forwarding SOAP messages on their way to the final destination. In realistic situations, not all parts of a SOAP message may be intended for the ultimate destination of the SOAP message, but, instead, may be intended for one or more of the intermediaries on the message path. Therefore, a second predefined header attribute, `SOAP-ENV:actor`, is used to identify the recipient of the header information. In SOAP 1.2 the actor attribute is renamed `SOAP-ENV:role`. The value of the SOAP actor attribute is the URI of the mediator, which is also the final destination of the particular header element (the mediator does not forward the header).

If the actor is omitted or set to the predefined default value, the header is for the actual recipient and the actual recipient is also the final destination of the message (body). The predefine value is:

```
http://schemas.xmlsoap.org/soap/actor/next
```

If a node on the message path does not recognize a `mustUnderstand` header and the node plays the role specified by the `actor` attribute, the node must generate a SOAP `MustUnderstand` fault. Whether the fault is sent back to the sender depends on the message exchange pattern in use. For request/response, the WS-I BP 1.0 requires the fault to be sent back to the sender. Also, according to WS-I BP 1.0, the receiver node must discontinue normal processing of the SOAP message after generating the fault message.

Headers can carry authentication data, digital signatures, encryption information, and transactional settings. They can also carry client-specific or project-specific controls and extensions to the protocol; the definition of headers is not just up to standards bodies.

Note: The header must not include service instructions (that would be used by the service implementation).

Body

The SOAP `Body` element provides a mechanism for exchanging information intended for the ultimate recipient of the message. The `Body` element is encoded as an immediate child element of the SOAP `Envelope` element. If a `Header` element is present, then the `Body` element *must* immediately follow the `Header` element. Otherwise it *must* be the first immediate child element of the `Envelope` element.

All immediate child elements of the `Body` element are called *body entries*, and each body entry is encoded as an independent element within the SOAP `Body` element. In the most simple case, the body of a basic SOAP message consists of an XML message as defined by the schema in the `types` section of the WSDL document. It is legal to have any valid XML as the body of the SOAP message, but WS-I conformance requires that the elements be namespace qualified.

Error handling

One area where there are significant differences between the SOAP 1.1 and 1.2 specifications is in the handling of errors. Here we focus on the SOAP 1.1 specification for error handling.

SOAP itself predefines one body element, which is the `fault` element used for reporting errors. If present, the `fault` element must appear as a body entry and must not appear more than once. The children of the `fault` element are defined as follows:

- `faultcode` is a code that indicates the type of the fault. SOAP defines a small set of SOAP fault codes covering basic SOAP faults:
  - `soapenv:Client`, indicating that the client sent an incorrectly formatted message
  - `soapenv:Server`, for delivery problems
  - `soapenv:VersionMismatch`, which can report any invalid namespaces specified on the `Envelope` element
  - `soapenv:MustUnderstand`, for errors regarding the processing of header content
- `faultstring` is a human-readable description of the fault. It must be present in a fault element.
- `faultactor` is an optional field that indicates the URI of the source of the fault. The value of the `faultactor` attribute is a URI identifying the source that caused the error. Applications that do not act as the ultimate destination of the SOAP message must include the `faultactor` element in a SOAP `fault` element.
- `detail` is an application-specific field that contains detailed information about the fault. It must not be used to carry information about errors belonging to header entries. Therefore, the absence of the detail element in the `fault` element indicates that the fault is not related to the processing of the body element (the actual message).

For example, a `soapenv:Server` fault message is returned if the service implementation throws a `SOAP Exception`. The exception text is transmitted in the `faultstring` field.

Although SOAP 1.1 permits the use of custom-defined `faultcodes`, the WS-I Basic Profile only permits the use of the four codes defined in SOAP 1.1.

1.4.2 Communication styles

SOAP supports two different communication styles:

Document	Also known as *message-oriented* style: This is a very flexible communication style that provides the best interoperability. The message body is any legal XML as defined in the `types` section of the WSDL document.
RPC	The *remote procedure call* is a synchronous invocation of an operation which returns a result; it is conceptually similar to other RPCs.

1.4.3 Encodings

In distributed computing environments, *encodings* define how data values defined in the application can be translated to and from a protocol format. We refer to these translation steps as *serialization* and *deserialization*, or, synonymously, *marshalling* and *unmarshalling*.

When implementing a Web service, we have to choose one of the tools and programming or scripting languages that support the Web services model. However, the protocol format for Web services is XML, which is independent of the programming language. Thus, SOAP encodings tell the SOAP runtime environment how to translate from data structures constructed in a specific programming language into SOAP XML and vice versa.

The following encodings are defined:

SOAP encoding	The *SOAP encoding* enables marshalling/unmarshalling of values of data types from the SOAP data model. This encoding is defined in the SOAP 1.1 standard.
Literal	The *literal* encoding is a simple XML message that does not carry encoding information. Usually, an XML Schema describes the format and data types of the XML message.

1.4.4 Messaging modes

The two styles (RPC and document) and the two common encodings (SOAP encoding and literal) can be freely intermixed to produce what is called a SOAP messaging mode. Although SOAP supports four modes, only three of the four modes are generally used, and further, only two are preferred by the WS-I Basic Profile.

- **Document/literal**—Provides the best interoperability between language environments. The WS-I Basic Profile states that all Web service interactions should use the Document/literal mode.
- **RPC/literal**—Possible choice between certain implementations. Although RPC/literal is WS-I compliant, it is not frequently used in practice. There are a number of usability issues associated with RPC/literal.
- **RPC/encoded**—Early Java implementations supported this combination, but it does not provide interoperability with other implementations and is not recommended
- **Document/encoded**—Not used in practice.

You can find the SOAP 1.1 specification at the following URL:

`http://www.w3.org/TR/2000/NOTE-SOAP-20000508`

The SOAP 1.2 specification is at the following URL:

`http://www.w3.org/TR/SOAP12`

1.5 WSDL

This section introduces Web Services Description Language (WSDL) 1.1. WSDL uses XML to specify the characteristics of a Web service: what the Web service can do, where it resides, and how it is invoked. WSDL can be extended to allow descriptions of different bindings, regardless of what message formats or network protocols are used to communicate.

WSDL enables a service provider to specify the following characteristics of a Web service:

- Name of the Web service and addressing information
- Protocol and encoding style to be used when accessing the public operations of the Web service
- Type information: Operations, parameters, and data types comprising the interface of the Web service, plus a name for this interface

1.5.1 WSDL Document

A WSDL document contains the following main elements:

Types	A container for data type definitions using some type system, usually XML Schema.
Message	An abstract, typed definition of the data being communicated. A message can have one or more typed parts.
Port type	An abstract set of one or more operations supported by one or more ports.
Operation	An abstract description of an action supported by the service that defines the input and output message and optional fault message.
Binding	A concrete protocol and data format specification for a particular port type. The binding information contains the protocol name, the invocation style, a service ID, and the encoding for each operation.
Port	A single endpoint, which is defined as an aggregation of a binding and a network address.
Service	A collection of related ports.

Note that WSDL does not introduce a new type definition language. WSDL recognizes the need for rich type systems for describing message formats and supports the XML Schema Definition (XSD) specification.

WSDL 1.1 introduces specific binding extensions for various protocols and message formats. There is a WSDL SOAP binding, which is capable of describing SOAP over HTTP. It is worth noting that WSDL does not define any mappings to a programming language; rather, the bindings deal with transport protocols. This is a major difference from interface description languages, such as the CORBA Interface Definition Language (IDL), which has language bindings.

You can find the WSDL 1.1 specification at the following URL:

http://www.w3.org/TR/wsdl

1.5.2 WSDL document anatomy

Figure 1-3 on page 21 shows the elements comprising a WSDL document and the various relationships between them.

The diagram should be interpreted in the following way:

- One WSDL document contains zero or more services. A service contains zero or more port definitions (service endpoints), and a port definition contains a specific protocol extension.
- The same WSDL document contains zero or more bindings. A binding is referenced by zero or more ports. The binding contains one protocol extension, where the style and transport are defined, and zero or more operations bindings. Each of these operation bindings is composed of one protocol extension, where the action and style are defined, and one to three messages bindings, where the encoding is defined.
- The same WSDL document contains zero or more port types. A port type is referenced by zero or more bindings. This port type contains zero or more operations, which are referenced by zero or more operations bindings.
- The same WSDL document contains zero or more messages. An operation usually points to an input and an output message, and optionally to some faults. A message is composed of zero or more parts.
- The same WSDL document contains zero or more types. A type can be referenced by zero or more parts.
- The same WSDL document points to zero or more XML Schemas. An XML Schema contains zero or more XSD types that define the different data types.

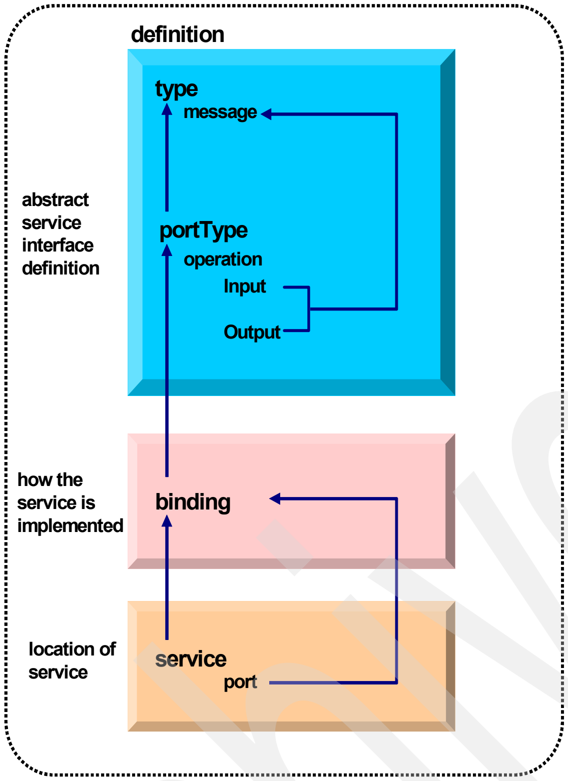


Figure 1-3 WSDL elements and relationships

Example

We now give an example of a simple, complete, and valid WSDL file. As this example shows, even a simple WSDL document contains quite a few elements with various relationships to each other. Example 1-1 contains the WSDL file example. This example is analyzed in detail later in this section.

Example 1-1 Complete WSDL document

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
         xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com"
         xmlns:resns="http://www.exampleApp.dispatchOrder.Response.com"
         xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
         xmlns:tns="http://www.exampleApp.dispatchOrder.com"
         targetNamespace="http://www.exampleApp.dispatchOrder.com">
  <types>
    <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            attributeFormDefault="qualified"
```

```
           elementFormDefault="qualified"
           targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
           xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com">
      <xsd:element name="dispatchOrderRequest" nillable="false">
          <xsd:complexType mixed="false">
              <xsd:sequence>
                 <xsd:element name="itemReferenceNumber" nillable="false">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:short">
                           <xsd:maxInclusive value="9999"/>
                           <xsd:minInclusive value="0"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                 </xsd:element>
                 <xsd:element name="quantityRequired" nillable="false">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:short">
                           <xsd:maxInclusive value="999"/>
                           <xsd:minInclusive value="0"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                 </xsd:element>
              </xsd:sequence>
         </xsd:complexType>
      </xsd:element>
  </xsd:schema>
  <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Response.com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://www.exampleApp.dispatchOrder.Response.com">
      <xsd:element name="dispatchOrderResponse" nillable="false">
          <xsd:complexType mixed="false">
                 <xsd:sequence>
                    <xsd:element name="confirmation" nillable="false">
                        <xsd:simpleType>
                           <xsd:restriction base="xsd:string">
                              <xsd:maxLength value="20"/>
                              <xsd:whiteSpace value="preserve"/>
                           </xsd:restriction>
                        </xsd:simpleType>
                    </xsd:element>
              </xsd:sequence>
          </xsd:complexType>
      </xsd:element>
  </xsd:schema>
</types>
<message name="dispatchOrderResponse">
  <part element="resns:dispatchOrderResponse" name="ResponsePart"/>
  </message>
  <message name="dispatchOrderRequest">
    <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
  </message>
  <portType name="dispatchOrderPort">
    <operation name="dispatchOrder">
       <input message="tns:dispatchOrderRequest" name="DFHOXODSRequest"/>
      <output message="tns:dispatchOrderResponse" name="DFHOXODSResponse"/>
    </operation>
  </portType>
  <binding name="dispatchOrderSoapBinding" type="tns:dispatchOrderPort">
    <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="dispatchOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="DFHOXODSRequest">
        <soap:body parts="RequestPart" use="literal"/>
      </input>
      <output name="DFHOXODSResponse">
        <soap:body parts="ResponsePart" use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="dispatchOrderService">
    <port binding="tns:dispatchOrderSoapBinding" name="dispatchOrderPort">
      <soap:address
       location="http://myserver:54321/exampleApp/services/dispatchOrderPort"/>
    </port>
  </service>
</definitions>
```

Namespaces

WSDL uses the XML namespaces listed in Table 1-2.

Table 1-2 WSDL namespaces

Namespace URI	Explanation
`http://schemas.xmlsoap.org/wsdl/`	Namespace for WSDL framework.
`http://schemas.xmlsoap.org/wsdl/soap/`	SOAP binding.
`http://schemas.xmlsoap.org/wsdl/http/`	HTTP binding.
`http://www.w3.org/2000/10/XMLSchema`	Schema namespace as defined by XSD.

Namespace URI	Explanation
`(URL to WSDL file)`	The *this namespace* (tns) prefix is used as a convention to refer to the current document. Do not confuse it with the XSD *target namespace*, which is a different concept.

The first three namespaces are defined by the WSDL specification itself; the next definition references a namespace that is defined in the SOAP and XSD standards. The last one is local to each specification.

1.5.3 WSDL definition

The WSDL definition contains types, messages, operations, port types, bindings, ports, and services.

Also, WSDL provides an optional element called `wsdl:document` as a container for human-readable documentation.

Types

The *types* element encloses data type definitions used by the exchanged messages. WSDL uses XML Schema Definition (XSD) as its canonical and built-in type system:

```
<definitions .... >
   <types>
      <xsd:schema .... /> (0 or more)
   </types>
</definitions>
```

The XSD type system can be used to define the types in a message regardless of whether or not the resulting wire format is XML. In our example we have two schema sections; one defines the message format for the input and the other defines the message format for the output.

In our example, the types definition, shown in Example 1-2, is where we specify that there is a complex type called `dispatchOrderRequest`, which is composed of two elements: a `itemReferenceNumber` and a `quantityRequired`.

Example 1-2 Types definition of our WSDL example for the input

```
<types>
    <xsd:schema xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            attributeFormDefault="qualified"
```

```
                elementFormDefault="qualified"
                targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
                xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com">
            <xsd:element name="dispatchOrderRequest" nillable="false">
                <xsd:complexType mixed="false">
                    <xsd:sequence>
                        <xsd:element name="itemReferenceNumber" nillable="false">
                            <xsd:simpleType>
                                <xsd:restriction base="xsd:short">
                                    <xsd:maxInclusive value="9999"/>
                                    <xsd:minInclusive value="0"/>
                                </xsd:restriction>
                            </xsd:simpleType>
                        </xsd:element>
                        <xsd:element name="quantityRequired" nillable="false">
                            <xsd:simpleType>
                                <xsd:restriction base="xsd:short">
                                    <xsd:maxInclusive value="999"/>
                                    <xsd:minInclusive value="0"/>
                                </xsd:restriction>
                            </xsd:simpleType>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:schema>
.
.
</types>
```

Messages

A *message* represents one interaction between a service requester and a service provider. If an operation is bidirectional at least two message definitions are used in order to specify the transmissions to and from the service provider. A message consists of one or more logical parts.

```
<definitions .... >
    <message name="nmtoken"> (0 or more)
        <part name="nmtoken" element="qname"(0 or 1) type="qname" (0
or 1)/>
            (0 or more)
    </message>
</definitions>
```

The abstract message definitions are used by the `operation` element. Multiple operations can refer to the same message definition.

Operations and messages are modeled separately in order to support flexibility and simplify reuse of existing definitions. For example, two operations with the same parameters can share one abstract message definition.

In our example, the messages definition, shown in Example 1-3, is where we specify the different parts that compose each message. The request message `dispatchOrderRequest` is composed of an element `dispatchOrderRequest` as defined in the schema in the `parts` section. The response message `dispatchOrderResponse` is similarly defined by the element `dispatchOrderResponse` in the schema. There is no requirement for the names of the message and the schema-defined element to match; in our example we did this merely for convenience.

Example 1-3 Message definition in our WSDL document

```
<message name="dispatchOrderResponse">
   <part element="resns:dispatchOrderResponse" name="ResponsePart"/>
</message>
<message name="dispatchOrderRequest">
   <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
</message>
```

Port types

A *port type* is a named set of abstract operations and the abstract messages involved:

```
   <definitions .... >
      <portType name="nmtoken">
         <operation name="nmtoken" .... /> (0 or more)
      </portType>
   </definitions>
```

WSDL defines four types of operations that a port can support:

One-way	The port receives a message. There is an *input* message only.
Request-response	The port receives a message and sends a correlated message. There is an *input* message followed by an *output* message.
Solicit-response	The port sends a message and receives a correlated message. There is an *output* message followed by an *input* message.
Notification	The port sends a message. There is an *output* message only. This type of operation could be used in a publish/subscribe scenario.

Each of these operation types can be supported with variations of the following syntax. Presence and order of the `input`, `output`, and `fault` messages determine the type of message:

```
<definitions .... >
   <portType .... > (0 or more)
      <operation name="nmtoken" parameterOrder="nmtokens">
         <input name="nmtoken"(0 or 1) message="qname"/> (0 or 1)
         <output name="nmtoken"(0 or 1) message="qname"/> (0 or 1)
         <fault name="nmtoken" message="qname"/> (0 or more)
      </operation>
   </portType >
</definitions>
```

Note that a request-response operation is an abstract notion. A particular binding must be consulted to determine how the messages are actually sent:

- Within a single transport-level operation, such as an HTTP request/response message pair, which is the preferred option
- As two independent transport-level operations, which can be required if the transport protocol only supports one-way communication

In our example, the `portType` and `operation` definitions, shown in Example 1-4, are where we specify the port type, called `dispatchOrderPort`, and a set of operations. In this case, there is only one operation, called `dispatchOrder`.

We also specify the interface that the Web service provides to its possible clients, with the input message `DFHOXODSRequest` and the output message `DFHOXODSResponse`. Since the `input` element appears before the `output` element in the `operation` element, our example shows a request-response type of operation.

Example 1-4 Port type and operation definitions in our WSDL document example

```
<portType name="dispatchOrderPort">
   <operation name="dispatchOrder">
      <input message="tns:dispatchOrderRequest" name="DFHOXODSRequest"/>
      <output message="tns:dispatchOrderResponse" name="DFHOXODSResponse"/>
   </operation>
</portType>
```

Bindings

A *binding* contains:

- Protocol-specific general binding data, such as the underlying transport protocol and the communication style for SOAP.

- Protocol extensions for operations and their messages.

Each binding references one port type; one port type can be used in multiple bindings. All operations defined within the port type must be bound in the binding. The pseudo XSD for the binding looks like this:

```
<definitions .... >
   <binding name="nmtoken" type="qname"> (0 or more)
      <-- extensibility element (1) --> (0 or more)
      <operation name="nmtoken"> (0 or more)
         <-- extensibility element (2) --> (0 or more)
         <input name="nmtoken"(0 or 1) > (0 or 1)
            <-- extensibility element (3) -->
         </input>
         <output name="nmtoken"(0 or 1) > (0 or 1)
            <-- extensibility element (4) --> (0 or more)
         </output>
         <fault name="nmtoken"> (0 or more)
            <-- extensibility element (5) --> (0 or more)
         </fault>
      </operation>
   </binding>
</definitions>
```

As we have already seen, a port references a binding. The port and binding are modeled as separate entities in order to support flexibility and location transparency. Two ports that merely differ in their network address can share the same protocol binding.

The extensibility elements `<-- extensibility element (x) -->` use XML namespaces in order to incorporate protocol-specific information into the language- and protocol-independent WSDL specification.

In our example, the binding definition, shown in Example 1-5, is where we specify our binding name, `dispatchOrderSoapBinding`. The connection must be SOAP HTTP, and the style must be `document`. We provide a reference to our operation, `dispatchOrder`, and we define the input message `DFH0XODSRequest` and the output message `DFH0XODSResponse`, both to be SOAP literal.

Example 1-5 Binding definition in our WSDL document example

```
<binding name="dispatchOrderSoapBinding" type="tns:dispatchOrderPort">
   <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="dispatchOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="DFH0XODSRequest">
```

```
            <soap:body parts="RequestPart" use="literal"/>
         </input>
         <output name="DFHOXODSResponse">
            <soap:body parts="ResponsePart" use="literal"/>
         </output>
      </operation>
</binding>
```

Service definition

A *service* definition merely bundles a set of ports together under a name, as the following pseudo XSD definition of the `service` element shows.

```
   <definitions .... >
      <service name="nmtoken"> (0 or more)
         <port .... /> (0 or more)
      </service>
   </definitions>
```

Multiple service definitions can appear in a single WSDL document.

Port definition

A *port* definition describes an individual endpoint by specifying a single address for a binding:

```
   <definitions .... >
      <service .... > (0 or more)
         <port name="nmtoken" binding="qname"> (0 or more)
         <-- extensibility element (1) -->
         </port>
      </service>
   </definitions>
```

The binding attribute is of type `QName`, which is a qualified name (equivalent to the one used in SOAP). It refers to a binding. A port contains exactly one network address; all other protocol-specific information is contained in the binding.

Any port in the implementation part must reference exactly one binding in the interface part.

The `<-- extensibility element (1) -->` is a placeholder for additional XML elements that can hold protocol-specific information. This mechanism is required because WSDL is designed to support multiple runtime protocols.

In our example, the service and port definition, shown in Example 1-6, is where we specify our service, called `dispatchOrderService`, which contains a collection of our ports. In this case, there is only one that uses the `dispatchOrderSoapBinding` and is called `dispatchOrderPort`. In this port, we specify our connection point as, for example, `http://myserver:54321/exampleApp/services/dispatchOrderPort`.

Example 1-6 Service and port definition in our WSDL document example

```
<service name="dispatchOrderService">
   <port binding="tns:dispatchOrderSoapBinding" name="dispatchOrderPort">
      <soap:address
      location="http://myserver:54321/exampleApp/services/dispatchOrderPort"/>
   </port>
</service>
```

1.5.4 WSDL bindings

We now investigate the WSDL extensibility elements supporting the SOAP transport binding.

SOAP binding

WSDL includes a binding for SOAP 1.1 endpoints, which supports the specification of the following protocol-specific information:

- An indication that a binding is bound to the SOAP 1.1 protocol
- A way of specifying an address for a SOAP endpoint
- The URI for the `SOAPAction` HTTP header for the HTTP binding of SOAP
- A list of definitions for headers that are transmitted as part of the SOAP envelope

Table 1-3 lists the corresponding extension elements.

Table 1-3 SOAP extensibility elements in WSDL

Extension and attributes		Explanation
`<soap:binding ...>`		Binding level; specifies defaults for all operations.
	`transport="uri"` *(0 or 1)*	Binding level; transport is the runtime transport protocol used by SOAP (HTTP, SMTP, and so on).
	`style="rpc|document"` *(0 or 1)*	The style is one of the two SOAP communication styles, rpc or document.
`<soap:operation ... >`		Extends operation definition.

Extension and attributes	Explanation
soapAction="uri" *(0 or 1)*	URN.
style="rpc|document" *(0 or 1)*	See binding level.
<soap:**body** ... >	Extends operation definition; specifies how message parts appear inside the SOAP body.
parts="nmtokens"	Optional; allows externalizing message parts.

use="encoded|literal"	literal: messages reference concrete XSD (no WSDL type);
	encoded: messages reference abstract WSDL type elements;
	encodingStyle extension used.

encodingStyle= "uri-list"*(0 or 1)*	List of supported message encoding styles.
namespace="uri" *(0 or 1)*	URN of the service.
<soap:**fault** ... >	Extends operation definition; contents of fault details element.
name="nmtoken"	Relates soap:fault to wsdl:fault for operation.
use, encodingStyle, namespace	See soap:body.
<soap:**address** ... >	Extends port definition.
location="uri"	Network address of RPC router.
<soap:**header** ... >	Operation level; shaped after <soap:body ...>.
<soap:**headerfault** ... >	Operation level; shaped after <soap:body ...>.

1.6 Summary

We began by discussing service-oriented architectures and how Web services relate to SOAs. We continued by giving an overview of the major technologies that make Web services possible: XML, SOAP, WSDL, and UDDI. We looked in detail at SOAP, which provides an XML text-based, platform- and language-neutral message format. Finally, we explained how WSDL defines the application data to be conveyed in the SOAP message as well as the information required to access the service, such as the transport protocol used and the location of the service.

2

CICS support for Web services

In chapter 1 we introduced several Web services technologies. In this chapter we explain how to use Web services in CICS TS V3.1.

First, we provide an overview of the different Web services functions provided by CICS TS V3.1. We then look at how to prepare for running a CICS application as a service provider and at what processing occurs when a service request arrives in CICS. Similarly, we consider how to configure CICS as a service requester and how CICS processes an outbound request from a service requester application.

We also describe the new CICS resource definitions that are required to support Web services, URIMAP, PIPELINE, WEBSERVICE, and TCPIPSERVICE. And we explain how you can control the processing of a Web service request using message handler and SOAP header processing programs.

2.1 Overview

What the World Wide Web did for interactions between programs and end users, Web services can do for program-to-program interactions. CICS support for Web services makes it possible for CICS applications to be integrated more rapidly, easily, and cheaply than ever before.

Application programs running in CICS TS V3.1 can participate in a heterogeneous Web services environment as service requesters, service providers, or both, using either an HTTP transport or a WebSphere MQ transport.

CICS TS V3.1 provides the following new functions:

- It includes a new Web services assistant utility.

  The Web services assistant utility contains two programs, DFHWS2LS and DFHLS2WS. DFHWS2LS helps you map an existing WSDL document into a high-level programming language data structure, while DFHLS2WS creates a new WSDL document from an existing language structure. The Web services assistant supports the following programming languages:

  – COBOL
  – PL/I
  – C
  – C++

- It supports two different approaches to deploying your CICS applications in a Web services environment.

  – You can use the Web services assistant.

    The Web services assistant helps you deploy an application with the least amount of programming effort. For example, if you want to expose an existing application as a Web service, you can start with a high-level language data structure, and use DFHLS2WS to generate the Web services description. Alternatively, if you want to communicate with an existing Web service, you can start with its Web service description and use DFHWS2LS to generate a high-level language structure that you can use in your program.

    Both DFHLS2WS and DFHWS2LS also generate a file called the *wsbind file*. When your application runs, CICS uses the wsbind file to transform your application data into a SOAP message on output and to transform the SOAP message to application data on input.

  – You can take complete control of the processing of your data.

    You can write your own code to map between your application data and the message that flows between the service requester and provider. For example, if you want to use non-SOAP messages within the Web service infrastructure, you can write your own code to transform between the message format and the format used by your application.

- It reads a pipeline configuration file created by the CICS system programmer to determine which message handlers should be invoked in a pipeline.

  A *message handler* is a program in which you can perform your own processing of Web service requests and responses. A *pipeline* is a set of message handlers that are executed in sequence.

  A pipeline can be configured as a service requester pipeline or a service provider pipeline, but not both.

  You can write your own message handlers to perform processing on request and response messages.

- It supplies message handlers designed especially to help you process SOAP messages.

  CICS provides special SOAP message handler programs that can help you to configure your pipeline as a SOAP node.

  – A service requester pipeline is the initial SOAP sender for the request, and the ultimate SOAP receiver for the response.
  – A service provider pipeline is the ultimate SOAP receiver for the request, and the initial SOAP sender for the response.

  Restriction: You cannot configure a CICS pipeline to function as an intermediary node in a SOAP message path.

  The CICS-provided SOAP message handlers can be configured to invoke one or more user-written SOAP header processing programs and to enforce the presence of particular headers in the SOAP message.

- It allows you to configure many different pipelines.

  You can configure a pipeline to support SOAP 1.1 or SOAP 1.2. Within your CICS system, you can have some pipelines that support SOAP 1.1 and others that support SOAP 1.2.

- It provides the following new resource definitions to help you configure support for Web services:

  – PIPELINE
  – URIMAP
  – WEBSERVICE

- It provides the following new EXEC CICS application programming interface (API) commands:
  - SOAPFAULT ADD | CREATE | DELETE
  - INQUIRE WEBSERVICE
  - INVOKE WEBSERVICE
- It conforms to open standards including:
  - SOAP 1.1 and 1.2
  - HTTP 1.1
  - WSDL 1.1
- It ensures maximum interoperability with other Web services implementations by conforming with the Web Services Interoperability Organization (WS-I) Basic Profile 1.0.
- It supports the WS-Atomic Transaction specification.

Note: CICS TS V3.1 includes some specific enhancements that are related to Web services implementation using HTTP. In particular, Web Support in CICS TS V3.1 contains the following improvements:

- HTTP
  - Concurrent session limit raised from 900 (in CICS TS V2.3) to 65000 per region.
  - There is no affinity to long running CWXN transactions.
- HTTPS
  - Concurrent session limit raised from 250 (in CICS TS V2.3) to 65000 per region.
  - No affinity to SSL TCB for duration of connection.
  - SSL TCB per connection now not required.
  - No affinity to long running CWXN transaction.
  - Easy choice over Cipher suites specified on TCPIPSERVICE definition.
  - Sysplex-wide cache for SSL session ID.

Customers using the Web interface can now exploit persistent sessions for large networks, as opposed to having to make and break connections with every request. These customers will make CPU savings. Customers who had small networks and were already able to exploit persistent connections will see up to about 4% increase on average size applications. They will however benefit from a much greater number of concurrent sessions per region.

2.2 CICS as a service provider

When CICS is a service provider, it receives a service request, which is passed through a pipeline to a target application program. The response from the application is returned to the service requester through the same pipeline. In this section we first discuss how to prepare for running a CICS application as a service provider. Then we discuss how CICS processes the incoming service request.

An existing COMMAREA-based application can be exposed as a service provider, normally without any application changes. Figure 2-1 shows CICS as a service provider.

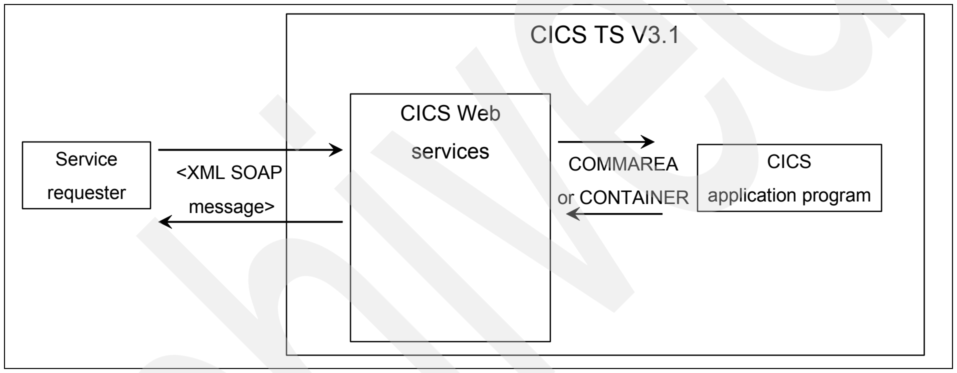


Figure 2-1 CICS as a service provider

When CICS is in the role of service provider, it must perform the following operations:

1. Receive the request from the service requester.
2. Examine the request, and extract the contents that are relevant to the target application program.
3. Invoke the application program, passing data extracted from the request.
4. Construct a response (when the application program returns control) using data returned by the application program.
5. Send a response to the service requester.

Note: This book deals with the implementation of CICS Web services created from COMMAREA-based applications in which there is separation of business logic and presentation logic.

2.2.1 Preparing to run a CICS application as a service provider

Suppose that you have an existing CICS application that you wish to expose as a Web service which uses the HTTP transport. Suppose also that you wish to use the Web services assistant rather than taking control of the processing yourselves. You would perform the following steps:

1. Generate the wsbind and WSDL files.
   a. Create an HFS directory in which to store the generated files. For example, you might create a directory named /u/SharedProjectDirectory/MyFirstWebServiceProvider.
   b. Run the DFHLS2WS program. The input you provide to the program includes the following:
      - The names of the partitioned data set members that contain the high-level language structures the application program uses to describe the Web service request and the Web service response
      - The fully qualified HFS names of the wsbind file and the file into which the Web service description is to be written (the WSDL file)
      - The relative URI that a client will use to access the Web service
      - How CICS should pass data to the target application program (COMMAREA or container)

   Note: Typically, an application developer would perform this step.

2. Create a TCPIPSERVICE resource definition.

   The resource definition should specify PROTOCOL(HTTP) and supply information about the port on which inbound requests are received.

   Note: Typically, a systems programmer would perform this step and the subsequent steps.

3. Create a PIPELINE resource definition.
   a. Create a service provider pipeline configuration file.

      A pipeline configuration file is an XML file that describes, among other things, the message handler programs and the SOAP header processing programs that CICS invokes when it processes the pipeline.
   b. Create an HFS directory in which to store installable wsbind and WSDL files.

We call this directory the “pickup” directory since CICS will pick up the wsbind and WSDL files from this directory and store them on a “shelf” directory.

c. Create an HFS directory for CICS to store installed wsbind files in.

   We call this directory the “shelf” directory.

d. Create a PIPELINE resource definition to handle the Web service request.

   - Specify the CONFIGFILE attribute to point to the file created in step 3a.
   - Specify the WSDIR attribute to point to the directory created in step 3b.
   - Specify the SHELF attribute to point to the directory created in step 3c.

e. Copy the wsbind and WSDL files created in step 1 to the pickup directory created in step 3b.

4. Install the TCPIPSERVICE and PIPELINE resource definitions.

   When the CICS system programmer installs the PIPELINE definition, CICS scans the pickup directory for wsbind files. When CICS finds the wsbind file created in step 1, CICS dynamically creates and installs a WEBSERVICE resource definition for it. CICS derives the name of the WEBSERVICE definition from the name of the wsbind file. The WEBSERVICE definition identifies the name of the associated PIPELINE definition and points to the location of the wsbind file in the HFS.

   During the installation of the WEBSERVICE resource:

   – CICS dynamically creates and installs a URIMAP resource definition. CICS bases the definition on the URI specified in the input to DFHLS2WS in step 1 and stored by DFHLS2WS in the wsbind file.

   – CICS uses the wsbind file to create main storage control blocks to map the inbound service request (XML) to a COMMAREA or a container and to map to XML the outbound COMMAREA or container that contains the response data.

   Note: As an alternative to using the PIPELINE scanning mechanism to install URIMAP resources, you can create and install them using Resource Definition Online (RDO).

5. Publish the WSDL files to the service requester clients.

   a. Customize the `location` attribute on the `<address>` element in the WSDL file so that its value specifies the TCP/IP server name of the machine hosting the service and the port number defined in the TCPIPSERVICE defined in step 2.

   b. Publish the WSDL to any parties wishing to create clients to this Web service.

2.2.2 Processing the inbound service request

Figure 2-2 shows the processing that occurs when a service requester sends a SOAP message over HTTP to a service provider application running in a CICS TS V3.1 region.

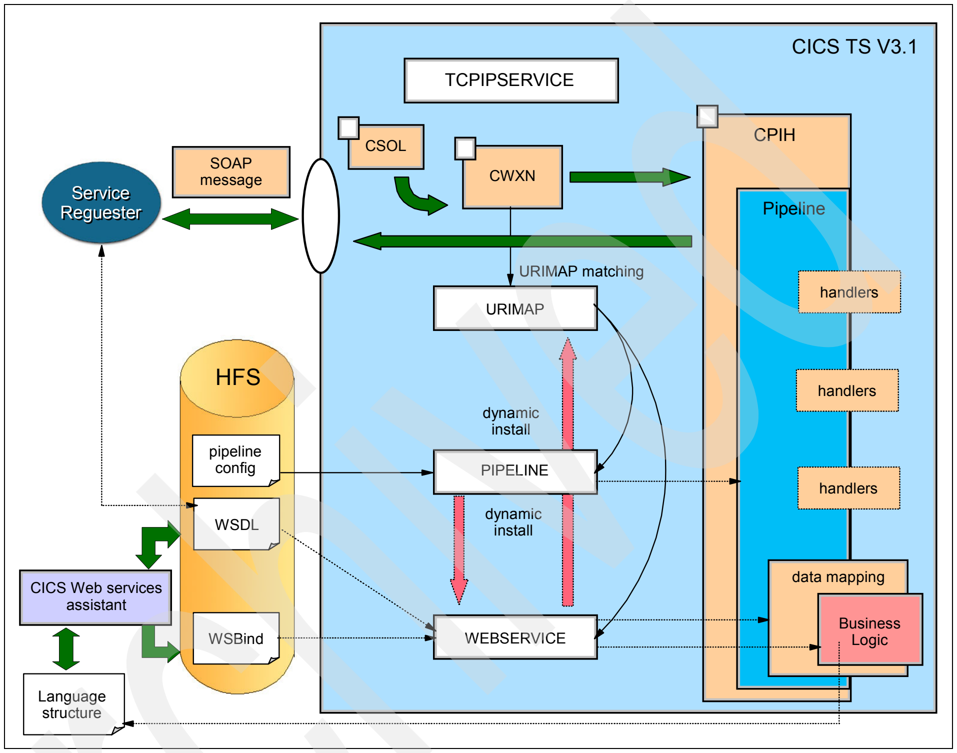


Figure 2-2 Web service run-time service provider processing

The CICS-supplied sockets listener transaction (CSOL) monitors the port specified in the TCPIPSERVICE resource definition for incoming HTTP requests. When the SOAP message arrives, CSOL attaches the transaction specified in the TRANSACTION attribute of the TCPIPSERVICE definition; normally, this will be the CICS-supplied Web attach transaction CWXN.

CWXN finds the URI in the HTTP request and then scans the URIMAP resource definitions for a URIMAP that has its USAGE attribute set to PIPELINE and its PATH attribute set to the URI found in the HTTP request. If CWXN finds such a URIMAP, it uses the PIPELINE and WEBSERVICE attributes of the URIMAP definition to get the name of the PIPELINE and WEBSERVICE definitions, which it will use to process the incoming request. CWXN also uses the TRANSACTION attribute of the URIMAP definition to determine the name of the transaction that it should attach to process the pipeline; normally, this will be the CPIH transaction.

CPIH starts the pipeline processing. It uses the PIPELINE definition to find the name of the pipeline configuration file. CPIH uses the pipeline configuration file to determine which message handler programs and SOAP header processing programs to invoke.

A message handler in the pipeline (typically, a CICS-supplied SOAP message handler) removes the SOAP envelope from the inbound request and passes the SOAP body to the data mapper function.

CICS uses the DFHWS-WEBSERVICE container to pass the name of the required WEBSERVICE definition to the data mapper. The data mapper uses the WEBSERVICE definition to locate the main storage control blocks that it needs to map the inbound service request (XML) to a COMMAREA or a container.

The data mapper links to the target service provider application program, providing it with input in the format that it expects. The application program is not aware that it is being executed as a Web service. The program performs its normal processing and then returns an output COMMAREA or container to the data mapper.

The output data from the CICS application program cannot just be sent back to the pipeline code. The data mapper must first convert the output from the COMMAREA or container format into a SOAP body.

2.3 CICS as a service requester

When CICS is a service requester, an application program sends a request, which is passed through a pipeline to a target service provider. The response from the service provider is returned to the application program through the same pipeline. In this section we discuss how to prepare for running a CICS application as a service requester. Then we discuss how CICS processes the outbound service request.

Figure 2-3 on page 42 shows CICS as a service requester.

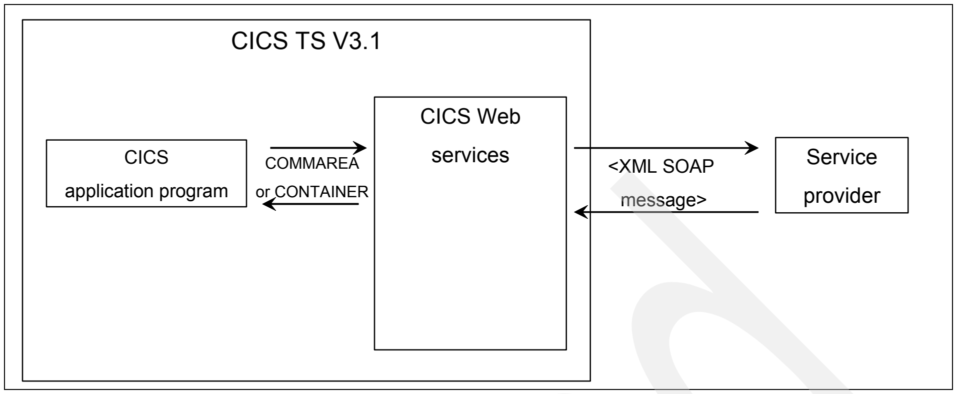


Figure 2-3 CICS as a service requester

When CICS is in the role of service requester, it must perform the following operations:

1. Build a request using data provided by the application program.
2. Send the request to the service provider.
3. Receive a response from the service provider.
4. Examine the response, and extract the contents that are relevant to the original application program.
5. Return control to the application program.

Note: Local optimization is possible when a CICS service requester invokes a CICS service provider application (see “Local optimization” on page 45).

2.3.1 Preparing to run a CICS application as a service requester

Suppose you wish to write a new CICS application that will invoke a Web service. Suppose also that you wish to use the Web services assistant rather than taking control of the processing yourselves. You would perform the following steps:

1. Generate the wsbind file and the language structures.
   a. Create an HFS directory in which to store the wsbind file. For example, you might create a directory named /u/SharedProjectDirectory/MyFirstWebServiceRequester
   b. Run the DFHWS2LS program. The input you provide to the program includes the following:
      - The fully qualified HFS name of the WSDL file that describes the Web service you want to request.

   - The names of the partitioned data set members into which DFHWS2LS should put the high-level language structures it generates. The application program uses the language structures to describe the Web service request and the Web service response.

   Note: Typically, an application developer would perform this step.

2. Create a PIPELINE resource definition.
   a. Create a service requester pipeline configuration file.

      The pipeline configuration file describes the message handler programs and the SOAP header processing programs that CICS will invoke when it processes the pipeline.
   b. Create an HFS directory in which to store installable wsbind files.

      CICS will pick up the wsbind file from this directory and store it on a “shelf” directory.
   c. Create the shelf directory for CICS to store installed wsbind files in.
   d. Create a PIPELINE resource definition to handle the Web service request:
      - Specify the CONFIGFILE attribute to point to the file created in step 2a.
      - Specify the WSDIR attribute to point to the directory created in step 2b.
      - Specify the SHELF attribute to point to the directory created in step 2c.
   e. Copy the wsbind file created in step 1to the pickup directory created in step 2b.

   Note: Typically, a systems programmer would perform this step.

3. Install the PIPELINE resource definition.

   When the CICS system programmer installs the PIPELINE definition, CICS scans the pickup directory for wsbind files. When CICS finds the wsbind file created in step 1, CICS dynamically creates and installs a WEBSERVICE resource definition for it. CICS derives the name of the WEBSERVICE definition from the name of the wsbind file. The WEBSERVICE definition identifies the name of the associated PIPELINE definition and points to the location of the wsbind file in the HFS.

   During the installation of the WEBSERVICE resource, CICS uses the wsbind file to create main storage control blocks to map the outbound service request to an XML document and to map the inbound XML response document to a language structure.

Note: As an alternative to using the PIPELINE scanning mechanism to install URIMAP resources, you can create and install them using Resource Definition Online (RDO).

Note: Typically, a systems programmer would perform this step.

4. Use the language structure generated in step 1 to write the application program.

   a. The application issues the following command to place the outbound data into container DFHWS-DATA:

   ```
   EXEC CICS PUT CONTAINER(DFHWS-DATA) CHANNEL(name_of_channel)
   FROM(data_area)
   ```

   b. It then issues the following command to invoke the Web service:

   ```
   EXEC CICS INVOKE WEBSERVICE(name_of_WEBSERVICE_definition)
   CHANNEL(name_of_channel) OPERATION(name_of_operation)
   ```

Note: Typically, an application developer would perform this step.

2.3.2 Processing the outbound service request

Figure 2-4 shows the processing that occurs when a service requester running in a CICS TS V3.1 region sends a SOAP message to a service provider.

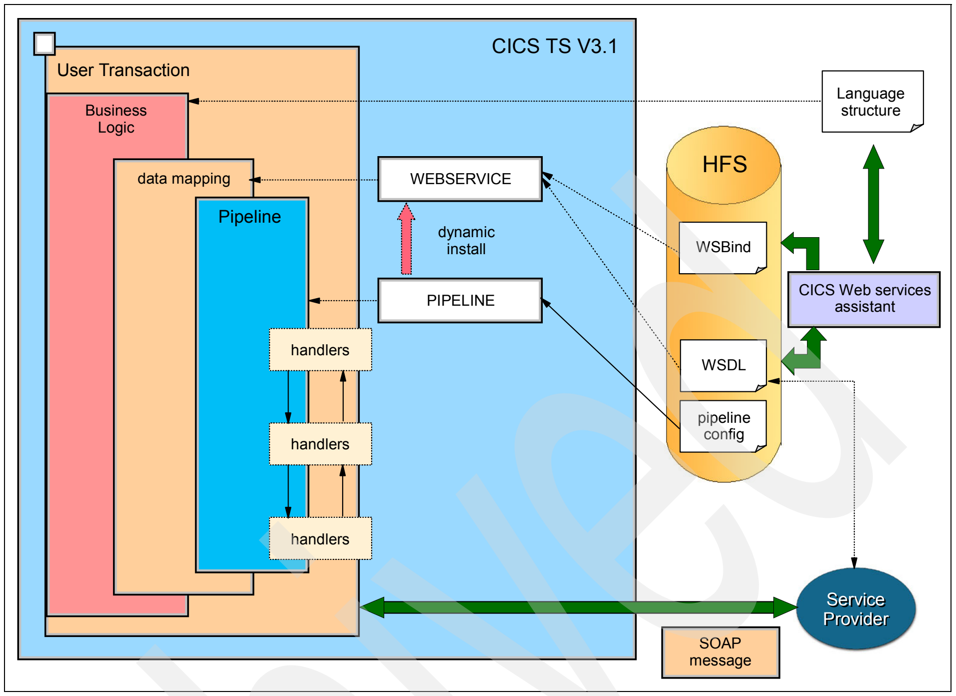


Figure 2-4   Web service run-time service requester processing

When the service requester issues the EXEC CICS INVOKE WEBSERVICE command, CICS uses the information found in the wsbind file that is associated with the specified WEBSERVICE definition to convert the language structure into an XML document. CICS then invokes the message handlers specified in the pipeline configuration file, and they convert the XML document into a SOAP message.

CICS sends the SOAP request message to the remote service provider via either HTTP or WebSphere MQ.

When the SOAP response message is received, CICS passes it back through the pipeline. The message handlers extract the SOAP body from the SOAP envelope, and the data mapping function converts the XML in the SOAP body into a language structure, which is passed to the application program in container DFHWS-DATA.

2.3.3 Local optimization

A special "local" optimization is possible when CICS is in the role of *both* service requester *and* service provider. In this case, CICS avoids the overhead of converting a language structure into an XML document by simply converting the EXEC CICS INVOKE WEBSERVICE command into an EXEC CICS LINK command.

Important: Invoking a CICS Web service using local optimization results in a significant performance benefit.

When an EXEC CICS INVOKE WEBSERVICE command is used to invoke a CICS service provider application, the provider application name in the Web service binding file associated with the WEBSERVICE resource is used to enable the local optimization of the Web service request. If you use this optimization, the request is optimized to an EXEC CICS LINK command (Figure 2-5).

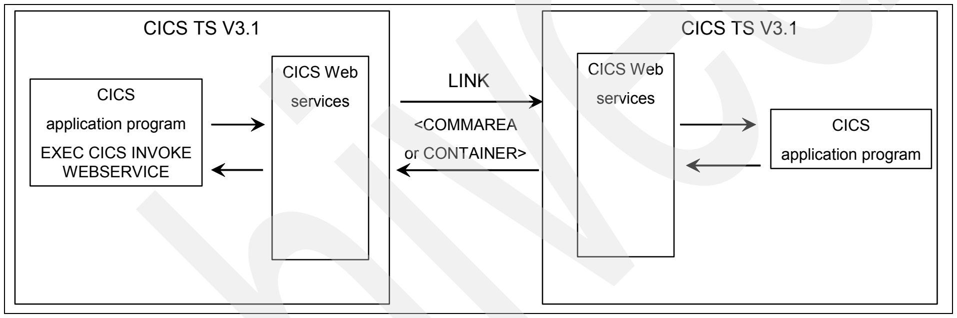


Figure 2-5 Invoking a CICS Web service using local optimization

The CICS service requester and service provider applications can be installed in the same CICS region or different regions. If they are in different regions, then an MRO or ISC connection must exist which enables the LINK request to be shipped to the remote CICS region hosting the service provider application.

Note that this optimization has an effect on the behavior of the EXEC CICS INVOKE WEBSERVICE command when the Web service is not expected to send a response:

- When the optimization is not in effect, control returns from the EXEC CICS INVOKE WEBSERVICE command as soon as the request message is sent.
- When the optimization is in effect, control returns from the EXEC CICS INVOKE WEBSERVICE command only when the target program terminates.

When the Web service is expected to send a response, control returns from the command when the response is available.

Restriction: You can use this optimization only if the service provider application and the service requester application are deployed with the Web services assistant.

2.4 CICS resources for Web services

We now look in more detail at what CICS resources a systems programmer must implement in order to enable Web services in a CICS environment. In Chapter 3, “Web services using HTTP” on page 73 we describe the resources that we created in order to enable Web services in our environment.

2.4.1 URIMAP

The URIMAP resource definition is used to define one of three different Web-related facilities in CICS. It is the value of the USAGE attribute on a URIMAP definition that determines which of the three facilities that particular definition controls.

1. Requests from a Web client, to CICS as an HTTP server

   URIMAP definitions for requests for CICS as an HTTP server have a USAGE attribute of SERVER. These URIMAP definitions match the URLs of HTTP requests that CICS expects to receive from a Web client, and they define how CICS should provide a response to each request. You can use a URIMAP definition to tell CICS to:

   – Provide a *static* response to the HTTP request, using a document template or z/OS UNIX® System Services HFS file

   – Provide a *dynamic* response to the HTTP request, using an application program that issues EXEC CICS WEB application programming interface commands

   – Redirect the request to another server, either temporarily or permanently

   For CICS as an HTTP server, URIMAP definitions incorporate most of the functions that were previously provided by the analyzer program specified on the TCPIPSERVICE definition. An analyzer program may still be involved in the processing path if required.

2. Requests to a server, from CICS as an HTTP client

   URIMAP definitions for requests from CICS as an HTTP client have a USAGE attribute of CLIENT. These URIMAP definitions specify URLs that are used when a user application, acting as a Web client, makes a request through CICS Web support to an HTTP server. Setting up a URIMAP definition for this purpose means that you can avoid identifying the URL in your application program.

3. Web service requests

   URIMAP definitions for Web service requests have a USAGE attribute of PIPELINE. These URIMAP definitions associate a URI for an inbound Web service request (that is, a request by which a client invokes a Web service in CICS) with a PIPELINE or WEBSERVICE resource that specifies the processing to be performed.

   You can use a URIMAP with a USAGE attribute of PIPELINE to specify:

   – The name of the transaction that CICS uses for running the pipeline alias transaction (the default is CPIH)

   – The user ID under which the pipeline alias transaction runs

Figure 2-6 illustrates the purpose of a URIMAP resource definition for Web service requests.

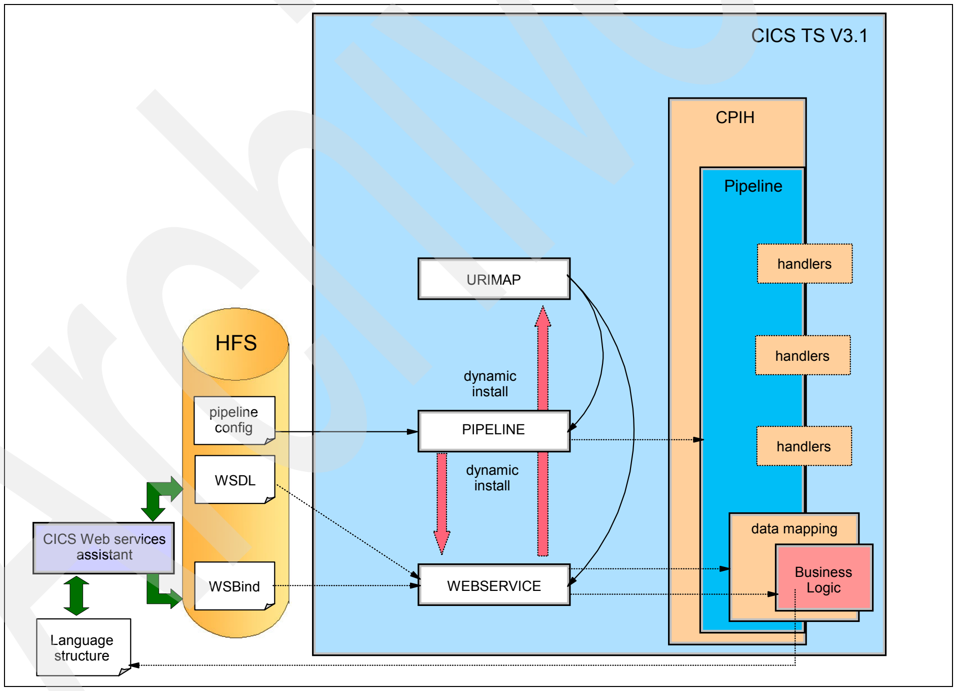


Figure 2-6 URIMAP resource relationships

You can create URIMAP resource definitions in the following ways:

- Use the CEDA transaction

- Use the DFHCSDUP batch utility
- Use CICSPlex® SM Business Application Services
- Use the EXEC CICS CREATE URIMAP command

When you install a PIPELINE resource, or when you issue a PERFORM PIPELINE SCAN command (using CEMT or the CICS system programming interface), CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory), and creates URIMAP and WEBSERVICE resources dynamically. For each Web service binding file in the directory, that is, for each file with the wsbind suffix, CICS installs a WEBSERVICE and a URIMAP if one does not already exist. Existing resources are replaced if the information in the binding file is newer than the existing resources.

Note: If you allow CICS to install the URIMAP resource dynamically, you cannot use the URIMAP definition to specify either the name of the transaction or the user ID under which the pipeline will run.

2.4.2 PIPELINE

A PIPELINE resource definition provides information about the message handlers that will act on a service request and on the response. The information about the message handlers is supplied indirectly; the PIPELINE definition specifies the name of an HFS file, called the pipeline configuration file, which contains an XML description of the message handlers and their configuration.

The most important attributes of the PIPELINE definition are as follows:

- WSDIR

  The WSDIR attribute specifies the name of the Web service binding directory (also known as the pickup directory). The Web service binding directory contains Web service binding files that are associated with the PIPELINE, and that are to be installed automatically by the CICS scanning mechanism. When the PIPELINE definition is installed, CICS scans the directory and automatically installs any Web service binding files it finds there.

  If you specify a value for the WSDIR attribute, it must refer to a valid HFS directory to which the CICS region has at least read access. If this is not the case, any attempt to install the PIPELINE resource will fail.

  If you do not specify a value for WSDIR, no automatic scan takes place on installation of the PIPELINE, and PERFORM PIPELINE SCAN commands will fail.

- SHELF

The SHELF attribute specifies the name of an HFS directory where CICS will copy information about installed Web services. CICS regions into which the PIPELINE definition is installed must have full permission to the shelf directory: read, write, and the ability to create subdirectories.

A single shelf directory may be shared by multiple CICS regions and by multiple PIPELINE definitions. Within a shelf directory, each CICS region uses a separate subdirectory to keep its files separate from those of other CICS regions. Within each region's directory, each PIPELINE uses a separate subdirectory.

After a CICS region performs a cold or initial start, it deletes its subdirectories from the shelf before trying to use the shelf.

- CONFIGFILE

  This attribute specifies the name of the PIPELINE configuration file.

Figure 2-7 illustrates the purpose of the PIPELINE resource definition.

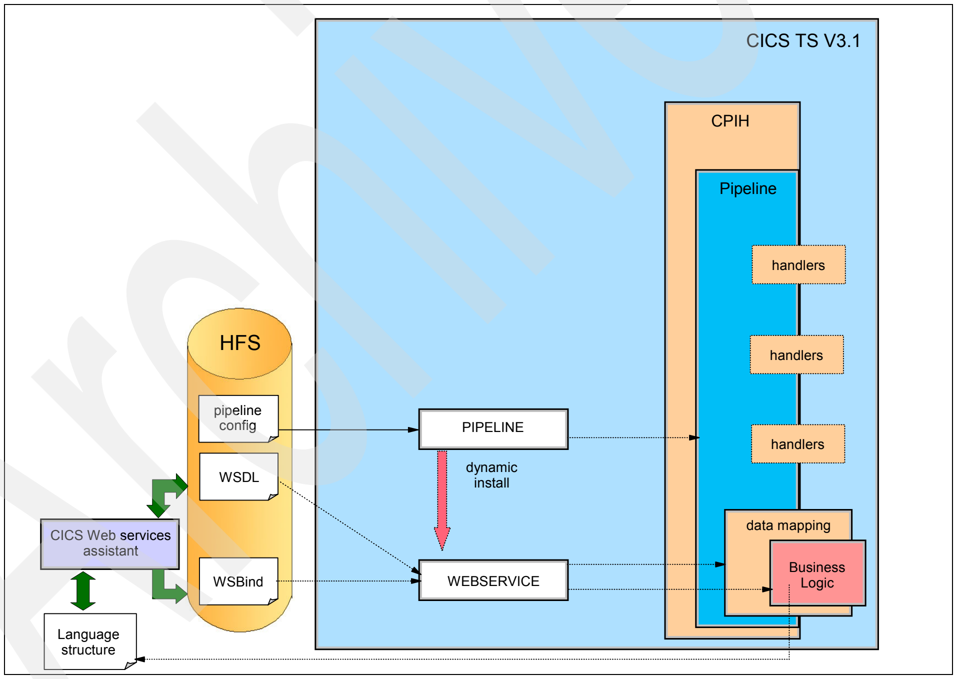


Figure 2-7 PIPELINE resource relationships

You can create PIPELINE resource definitions in the following ways:

- Use the CEDA transaction
- Use the DFHCSDUP batch utility

- Use CICSPlex SM Business Application Services
- Use the EXEC CICS CREATE PIPELINE command

Pipeline configuration file

When CICS processes a Web service request, it uses a pipeline of one or more message handlers to handle the request. The configuration of a pipeline used to handle a Web service request is specified in an XML document, known as a *pipeline configuration file*. Use a suitable XML editor or text editor to work with your pipeline configuration files. The exact configuration of the pipeline will depend upon the specific needs of the application.

There are two kinds of pipeline configuration files: one describes the configuration of a service provider pipeline, the other describes the configuration of a service requester pipeline. Each is defined by its own schema, and each has a different root element. The root element for a provider pipeline is `<provider_pipeline>`, while the root element for a requester pipeline is `<requester_pipeline>`.

The immediate child elements of the `<provider_pipeline>` element are:

- A mandatory `<service>` element, which specifies the message handlers that are invoked for every request, including the terminal message handler. The terminal message handler is the last handler in the pipeline.
- An optional `<transport>` element, which specifies message handlers that are selected at run time based upon the resources that are being used for the message transport. For example, for the HTTP transport, you can specify that CICS should invoke the message handler only when the port on which the request was received is defined on a specific TCPIPSERVICE definition. For the WebSphere MQ transport, you can specify that CICS should invoke the message handler only when the inbound message arrives at a specific message queue.
- An optional `<apphandler>` element, which specifies the name of the program that the terminal message handler will link to by default, that is, the name of the target application program (or wrapper program) that provides the service. Message handlers can specify a different program at run time by using the DFHWS-APPHANDLER container, so the name coded here is not always the program that is linked to.

> **Important:** When you use DFHLS2WS or DFHWS2LS to deploy your service provider, you must specify DFHPITP as the target program. DFHPITP will get the name of your target application program (or wrapper program) from the wsbind file.

The <apphandler> element is used when the last message handler in the pipeline (the terminal handler) is one of the CICS-supplied SOAP message handlers.

If you do not code an <apphandler> element, one of the message handlers *must* use the DFHWS-APPHANDLER container to specify the name of the program.

- An optional <service_parameter_list> element, which contains parameters that CICS will make available to the message handlers in the pipeline via container DFH-SERVICEPLIST.

Example 2-1 shows the sample service provider pipeline configuration file basicsoap11provider.xml.

Example 2-1 Configuration file for service provider

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<provider_pipeline
xmlns="http://www.ibm.com/software/htp/cics/pipeline"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
provider.xsd ">
  <service>
    <terminal_handler>
      <cics_soap_1.1_handler/>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

Important: A pipeline can be configured to support SOAP 1.1 or SOAP 1.2. Within your CICS system, you can have many pipelines, some of which support SOAP 1.1 and some of which support SOAP 1.2.

The immediate sub-elements of a <requester_pipeline> element are:

- An optional <service> element, which specifies the message handlers that are invoked for every request
- An optional <transport> element, which specifies message handlers that are selected at run time, based upon the resources that are being used for the message transport
- An optional <service_parameter_list> element, which contains parameters that CICS will make available to the message handlers in the pipeline via container DFH-SERVICEPLIST

Example 2-2 shows the sample service requester pipeline configuration file basicsoap11requester.xml.

Example 2-2 Configuration file for service requester

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<requester_pipeline
xmlns="http://www.ibm.com/software/htp/cics/pipeline"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
requester.xsd ">
  <service>
    <service_handler_list>
      <cics_soap_1.1_handler/>
    </service_handler_list>
  </service>
</requester_pipeline>
```

2.4.3 WEBSERVICE

Three objects define the execution environment that allows a CICS application program to operate as a Web service provider or a Web service requester:

- The Web service description
- The Web service binding file
- The pipeline

These three objects are defined to CICS on the following attributes of the WEBSERVICE resource definition:

- WSDLFILE
- WSBIND
- PIPELINE

The WEBSERVICE definition has a fourth attribute, VALIDATION, which specifies whether full validation of SOAP messages against the corresponding schema in the Web service description should be performed at run time. VALIDATION(YES) ensures that all SOAP messages that are sent and received are valid XML with respect to the XML schema.

Important: Validation of a SOAP message against a schema incurs considerable processing overhead, and you should normally specify VALIDATION(NO) in a production environment.

If VALIDATION(NO) is specified, sufficient validation is performed to ensure that the message contains well-formed XML.

Figure 2-8 on page 54 illustrates the purpose of the WEBSERVICE resource definition.

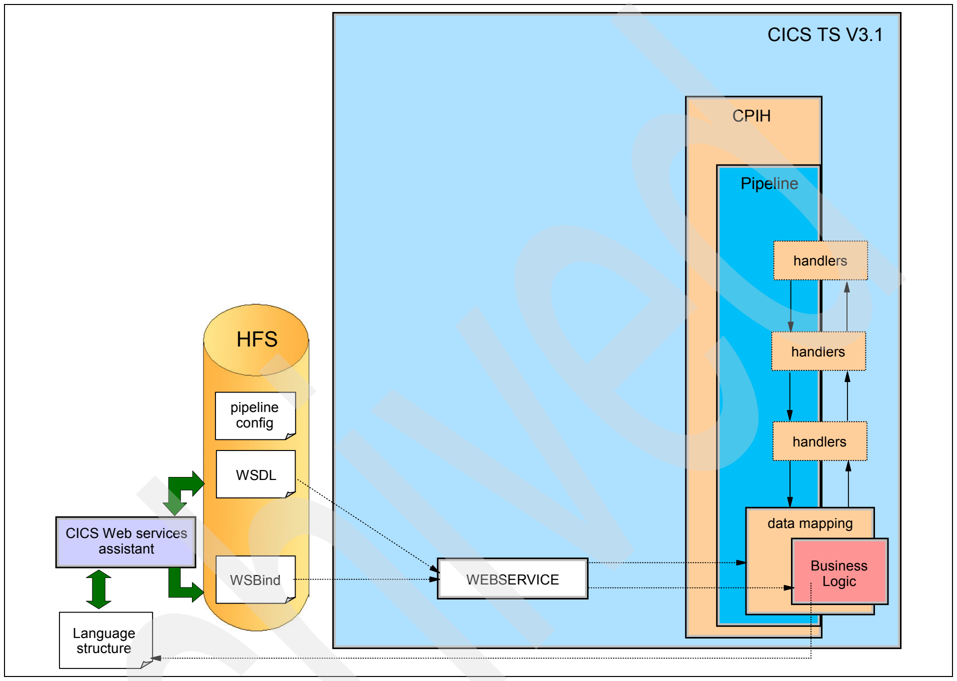


Figure 2-8   WEBSERVICE resource relationships

You can create WEBSERVICE resource definitions in the following ways:

- Using the CEDA transaction
- Using the DFHCSDUP batch utility
- Using CICSPlex SM Business Application Services
- Using the EXEC CICS CREATE WEBSERVICE command

When you install a PIPELINE resource, or when you issue a PERFORM PIPELINE SCAN command (using CEMT or the CICS system programming interface), CICS scans the directory specified in the PIPELINE's WSDIR attribute (the pickup directory), and creates URIMAP and WEBSERVICE resources dynamically. For each Web service binding file in the directory, that is, for each file with the wsbind suffix, CICS installs a WEBSERVICE and a URIMAP if one does not already exist. Existing resources are replaced if the information in the binding file is newer than the existing resources.

The CEMT INQUIRE WEBSERVICE command is used to obtain information about a WEBSERVICE resource definition. The data returned depends on the type of Web service.

Web service binding file

A Web services description contains abstract representations of the input and output messages used by the service. When a service provider or service requester application executes, CICS needs information about how the content of the messages maps to the data structures used by the application. This information is held in a Web service binding file.

Web services binding files are created:

- By utility program DFHWS2LS when language structures are generated from WSDL
- By utility program DFHLS2WS when WSDL is generated from a language structure

At run time, CICS uses information in the Web service binding file to perform the mapping between application data structures and SOAP messages.

2.4.4 TCPIPSERVICE

A TCPIPSERVICE definition is required in a service provider that uses the HTTP transport, and contains information about the port on which inbound requests are received.

You can create TCPIPSERVICE resource definitions in the following ways:

- Using the CEDA transaction
- Using the DFHCSDUP batch utility
- Using CICSPlex SM Business Application Services
- Using the EXEC CICS CREATE TCPIPSERVICE command

2.4.5 Resources checklist

The relationships between CICS Web services definitions are shown in Figure 2-9 on page 56.

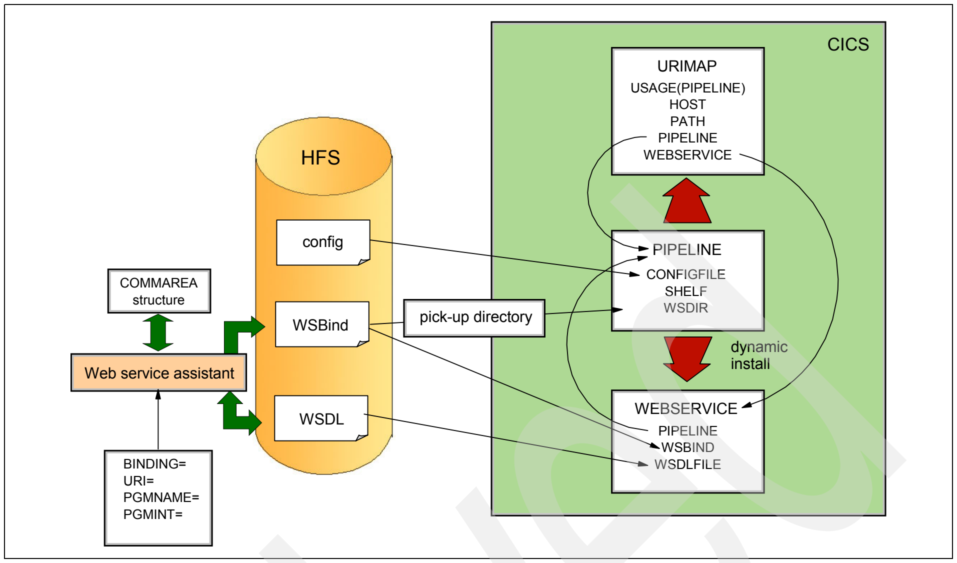


Figure 2-9 CICS Web services resource interrelationships

The resources that are required to support a particular application program depends upon the following:

- Whether the application program is a service provide or a service requester
- Whether the application is deployed with the CICS Web services assistant or you write your own code to map between your application data and SOAP messages

Table 2-1 is a checklist of resource definitions.

Table 2-1 Resource checklist

Service requester or provider	**CICS Web services assistant used**	**PIPELINE required**	**WEBSERVICE required**	**URIMAP required**	**TCPIPSERVICE required**
Provider	yes	yes	yes (1)	yes (1)	(2)
	no	yes	no	yes	(2)
Requester	yes	yes	yes	no	no
	no	yes	no	no	no
(1). When the CICS Web services assistant is used to deploy an application program, the WEBSERVICE and URIMAP resources can be created automatically when the PIPELINE's pickup directory is scanned. This happens when the PIPELINE resource is installed, or as a result of a PERFORM PIPELINE SCAN command. (2). A TCPIPSERVICE resource is required when the HTTP transport is used. When the WebSphere MQ transport is used, you must define a queue.					

2.5 Message handlers

When you want to perform specialized processing on the messages that flow between a service requester and a service provider, and CICS does not supply a message handler that meets your needs, you will need to supply your own.

The message handler interface lets you perform the following tasks in a message handler program:

- Examine the contents of an XML request or response, without changing it
- Change the contents of an XML request or response
- In a non-terminal message handler, pass an XML request or response to the next message handler in the pipeline
- In a terminal message handler (the last handler in the pipeline) call an application program, and generate a response
- In the request phase of the pipeline, force a transition to the response phase, by absorbing the request, and generating a response
- Handle errors

Message handlers can be used for specific custom functions like:

- Logging requests
- Changing the “context” of a request, for example, changing the name of the transaction that CICS uses for running the pipeline alias transaction

Message handlers use channels and containers to interact with one another, and with the system (see “Channels and containers” on page 59).

2.5.1 SOAP message handlers

CICS provides SOAP message handlers that you can include in your pipeline to process SOAP 1.1 and SOAP 1.2 messages. You can use the SOAP message handlers in a service requester or in a service provider pipeline.

On input, the SOAP message handlers parse inbound SOAP messages, and extract the SOAP `<Body>` element for use by your application program. On output, the handlers construct the complete SOAP message, using the `<Body>` element that your application provides.

If you use SOAP headers in your messages, the SOAP handlers can invoke user-written *header processing programs* that allow you to process the SOAP headers on inbound messages, and to add them to outbound messages. For example, a header processing program could check security information in a SOAP header or SOAP body.

> **Tip:** Do not confuse *header processing programs* with *message handlers*. A *header processing program* can only be invoked by a CICS-supplied SOAP message handler to process a specific kind of SOAP header.

A SOAP message handler, and optional header processing programs, are specified in the pipeline configuration file using the `<cics_soap_1.1_handler>` and the `<cics_soap_1.2_handler>` elements and their sub-elements.

Typically, you will need just one SOAP message handler in a pipeline. However, there are some situations where more than one is needed. For example, you can ensure that SOAP headers are processed in a particular sequence by defining multiple SOAP message handlers.

SOAPFAULT commands

SOAP message handlers and header processing programs can use three API commands which are new in CICS TS V3.1 to manage SOAP faults:

- EXEC CICS SOAPFAULT CREATE

  Use this command to create a SOAP fault. If a SOAP fault already exists in the context of the SOAP message that is being processed by the message handler, the existing fault is overwritten.
- EXEC CICS SOAPFAULT ADD

  Use this command to add either of the following items to a SOAPFAULT object that was created with an earlier SOAPFAULT CREATE command:
  - A subcode
  - A fault string for a particular national language

    If the fault already contains a fault string for the language, then this command replaces the fault string for that language. In SOAP 1.1, only the fault string for the original language is used.
- EXEC CICS SOAPFAULT DELETE

  Use this command to delete a SOAPFAULT object that was created with an earlier SOAPFAULT CREATE command.

These commands require information that is held in containers on the channel of the CICS-supplied SOAP message handler. To use these commands, you must have access to the channel. Only the following types of programs have this access:

- Programs that are invoked directly from a CICS-supplied SOAP message handler, including SOAP header processing programs.
- Programs deployed with the Web services assistant that have a channel interface. Programs with a COMMAREA interface do not have access to the channel.

Many of the options on the SOAPFAULT CREATE and SOAPFAULT ADD commands apply to SOAP 1.1 and SOAP 1.2 faults, although their behavior is slightly different for each level of SOAP. Other options apply to one SOAP level or the other, but not to both, and if you specify any of them when the message uses a different level of SOAP, the command will raise an INVREQ condition. To help you determine which SOAP level applies to the message, container DFHWS-SOAPLEVEL contains a binary fullword with one of the following values:

- 1 - The request or response is a SOAP 1.1 message.
- 2 - The request or response is a SOAP 1.2 message.
- 10 - The request or response is not a SOAP message.

2.5.2 Channels and containers

Channels and containers are new resources in CICS TS V3.1 that provide the capability to pass data from one application to another application.

- A *channel* is a logical resource that must contain one or more containers.
- A *container* is a named block of data designed for passing information between programs.

The major advantage of using channels and containers compared to using a COMMAREA is that the length of a container can exceed the 32 KB limit for COMMAREA data. CICS uses channels and containers to pass data between the message handlers of a pipeline.

All programs that are used as message handlers are invoked with the same channel interface. The channel holds a number of containers. The containers can be categorized as:

- **Control containers**

  These are essential to the operation of the pipeline. Message handlers can use the control containers to modify the sequence in which the message handlers are processed.

- **Context containers**

  In some situations, message handler programs need information about the context in which they are invoked. CICS provides this information in a set of context containers that are passed to the programs. Some of the context containers hold information that you can change in your message handler. For example, in a service provider pipeline, you can change the user ID and transaction ID of the target application program by modifying the contents of the appropriate context containers.

- **Header containers**

  Containers that are specific to the header processing program interface.

- **User containers**

  These contain information that one message handler needs to pass to another. The use of user containers is entirely a matter for the message handlers.

For each container, Table 2-2 explains what the function of the container is and the type of access permitted to message handlers and header processing programs.

Table 2-2 CICS Web service containers

Name	**Message handler**	**Header processing program**	**Comment**
Control containers			
DFHERROR	Update	Update	Used to convey information about pipeline errors to other message handlers.
DFHFUNCTION	Update	Update	Indicates where in a pipeline a program is being invoked.
DFHNORESPONSE	Update	Update	In the request phase of a service requester pipeline, indicates that the service provider is not expected to return a response.
DFHREQUEST	Update	Read only	Contains the request message that is processed in the request phase of a pipeline.
DFHRESPONSE	Update	Read only	Contains the response message that is processed in the response phase of a pipeline.
Context containers			
DFHWS-PIPELINE	Read only	Read only	The name of the PIPELINE in which the program is being run.
DFHWS-WEBSERVICE	Update	Update	The name of the WEBSERVICE that specifies the execution environment.
DFHWS-URI	Update	Update	The URI of the service for a service provider pipeline only.

Name	Message handler	Header processing program	Comment
DFHWS-SOAPACTION	Update	Update	The SOAPAction header associated with the SOAP message in container DFHWS-BODY.
DFH-HANDLERPLIST	Read only	Read only	The <handler_parameter_list> contents.
DFH-SERVICEPLIST	Read only	Read only	The <service_parameter_list> contents.
DFHWS-APPHANDLER	Update	Update	The <apphandler> contents.
DFHWS-DATA	Update	Update	Used in INVOKE WEBSERVICE (outbound only) deployed with the CICS Web services assistant. It holds the top-level data structure that is mapped to and from a SOAP request.
DFHWS-TRANID	Update	Update	The transaction ID with which the task in the pipeline is running.
DFHWS-USERID	Update	Update	The user ID with which the task in the pipeline is running.
DFHWS-SOAPLEVEL	Read only	Read only	The level of SOAP used in the message that is being processed.
DFHWS-OPERATION	Read only	Read only	In the response phase of a service requester pipeline, contains the name of the operation that is specified in a SOAP request.
Header containers			
DFHHEADER	None	Update	The single header block that caused the header processing program to be driven.
DFHWS-XMLNS	None	Read only	The list of name-value pairs that map namespace prefixes to namespaces for the XML content of the request.
DFHWS-BODY	None	Update	The contents of the SOAP body.

2.6 Tools for developing CICS Web services

In this section, we provide a brief overview of the main tools for developing CICS Web services. For more detailed information, see *Application Development for CICS Web Services*, SG24-7126.

2.6.1 CICS Web services assistant

The CICS Web services assistant is a set of batch utilities that can help you transform existing CICS applications into Web services and enable CICS applications to use Web services provided by external providers. The assistant supports rapid deployment of CICS applications for use in service providers and service requesters, with minimal programming effort.

When you use the Web services assistant for CICS, you do not have to write your own code for parsing inbound messages and for constructing outbound messages; CICS maps data between the body of a SOAP message and the application program's data structure.

Resource definitions are, for the most part, generated and installed automatically. You do have to define PIPELINE resources, but you can, in many cases, use one of the pipeline configuration files that CICS provides.

The assistant can create a WSDL document from a simple language structure, or a language structure from an existing WSDL document, and supports COBOL, C/C++, and PL/I. It also generates information used to enable automatic run-time conversion of the SOAP messages to containers and COMMAREAs, and vice versa.

However, the assistant cannot deal with every possibility, and there are times when you will need to take a different approach. For example:

- You don't want to use SOAP messages.

  If you prefer to use a non-SOAP protocol for your messages, you can do so. However, your application programs will be responsible for parsing inbound messages, and constructing outbound messages.

- You want to use SOAP messages, but don't want CICS to parse them.

  For an inbound message, the assistant maps the SOAP body to an application data structure. In some applications, you may want to parse the SOAP body yourself.

- The CICS Web services assistant does not support your application's data structure.

  Although the CICS Web services assistant supports the most common data types and structures, there are some that are not supported. For example, OCCURS DEPENDING ON and REDEFINES on data description entries are not supported. For full details on the data types and structures supported by the CICS Web Services assistant, see the *CICS Web Services Guide* (SC34-6458).

In this situation, you should consider one of the following alternatives:

– Provide a wrapper program that maps your application’s data to a format that the assistant can support.

– Use WebSphere Developer for zSeries (see 2.6.3, “WebSphere Developer for zSeries” on page 65).

2.6.2 Web services assistant utility programs

The CICS Web services assistant provides two utility programs: DFHLS2WS and DFHWS2LS. They are described in detail in this section.

DFHLS2WS

This program generates a Web services description and Web services binding file from a language structure. Example 2-3 shows sample JCL for running DFHLS2WS.

Example 2-3 DFHLS2WS JCL sample

```
//LS2WS JOB 'accounting information',name,MSGCLASS=A
// SET QT=''''
//JAVAPROG EXEC DFHLS2WS,
// TMPFILE=&QT.&SYSUID.&QT
//INPUT.SYSUT1 DD *
PDSLIB=//CICSHLQ.CICS.SDFHSAMP
REQMEM=DFH0XCP4
RESPMEM=DFH0XCP4
LANG=COBOL
PGMNAME=DFH0XCMN
URI=exampleApp/inquireSingle
PGMINT=COMMAREA
WSBIND=/u/exampleapp/wsbind/inquireSingle.wsbind
WSDL=/u/exampleapp/wsdl/inquireSingle.wsdl
/*
```

The main input parameters are as follows:

- PDSLIB

  Specifies the name of the partitioned data set that contains the high-level language data structures to be processed.

- REQMEM

  Specifies the name of the partitioned data set member that contains the high-level language structure for the Web service request.

  - For a service provider, the Web service request is the input to the application program.
  - For a service requester, the Web service request is the output from the application program.

- RESPMEM

  Specifies the name of the partitioned data set member that contains the high-level language structure for the Web service response:

  - For a service provider, the Web service response is the output from the application program.
  - For a service requester, the Web service response is the input to the application program.

- LANG

  Specifies the language of the language structure to be created.

- PGMNAME

  Specifies the name of the target CICS application program that is being exposed as a Web service.

- URI

  In a service provider, this parameter specifies the relative URI that a client will use to access the Web service. CICS uses the value specified when it generates a URIMAP resource from the Web service binding file created by DFHLS2WS. The parameter specifies the path component of the URI to which the URIMAP definition applies.

- PGMINT

  For a service provider, specifies how CICS passes data to the target application program (using a COMMAREA or a channel).

- WSBIND

  Specifies the HFS name of the Web service binding file.

- WSDL

  Specifies the HFS name of the Web service description file.

DFHWS2LS

This program generates a language structure and Web services binding file from a Web services description. Example 2-4 shows sample JCL for running DFHWS2LS.

Example 2-4 DFHWS2LS JCL sample

```
//WS2LS JOB 'accounting information',name,MSGCLASS=A
// SET QT=''''
```

```
//JAVAPROG EXEC DFHWS2LS,
// TMPFILE=&QT.&SYSUID.&QT
//INPUT.SYSUT1 DD *
PDSLIB=//CICSHLQ.CICS.SDFHSAMP
REQMEM=CPYBK1
RESPMEM=CPYBK2
LANG=COBOL
PGMNAME=DFH0XCMN
URI=exampleApp/inquireSingle
PGMINT=COMMAREA
WSBIND=/u/exampleapp/wsbind/inquireSingle.wsbind
WSDL=/u/exampleapp/wsdl/inquireSingle.wsdl
/*
```

Mapping Level

Some restrictions that existed on data types supported by the CICS Web services assistant when CICS TS V3.1 was shipped, were removed by APARs PK15904 and PK23547. The MAPPING-LEVEL parameter for both DFHLS2WS and DFHWS2LS specifies the level of mapping that the batch assistant should use when generating the Web service binding file and Web service description. The value of this parameter can be any of the following values:

- 1.0 - The original default mapping level of CICS TS V3.1.
- 1.1 - APAR PK15904 has been applied to the CICS TS V3.1 region where the Web service binding file is deployed. At this level of mapping, there are improvements to DFHWS2LS when mapping XML character and binary data types, in particular when mapping data of variable length.
- 1.2 - Both APARs PK15904 and PK23547 have been applied to the CICS TS V3.1 region where the Web service binding file is deployed.

2.6.3 WebSphere Developer for zSeries

WebSphere Developer for zSeries V6 is based on the IBM Rational® Software Development Platform and facilitates the development of both Java- and z/OS-based applications. It includes capabilities that make traditional z/OS mainframe development, Web development, and integrated composite development faster and more efficient. In particular, WebSphere Developer contains tools that support the development of Web services and the XML enablement of existing CICS COBOL applications.

The XML Services for the Enterprise (XSE) capability of WebSphere Developer provides tools that let you adapt COBOL-based applications so that they can consume and produce XML messages. XSE supports the creation of *driver programs* that work with existing CICS (or IMS™) applications.

The Web Services Enablement wizard is the XSE tool that supports the bottom-up approach for creating Web services based on existing CICS COBOL programs. It takes as input the COMMAREA copybook. The XML structure and data types are then derived from the COBOL data declarations. Based on these, the Web Services Enablement wizard generates the set of artifacts shown in Figure 2-10 on page 66.

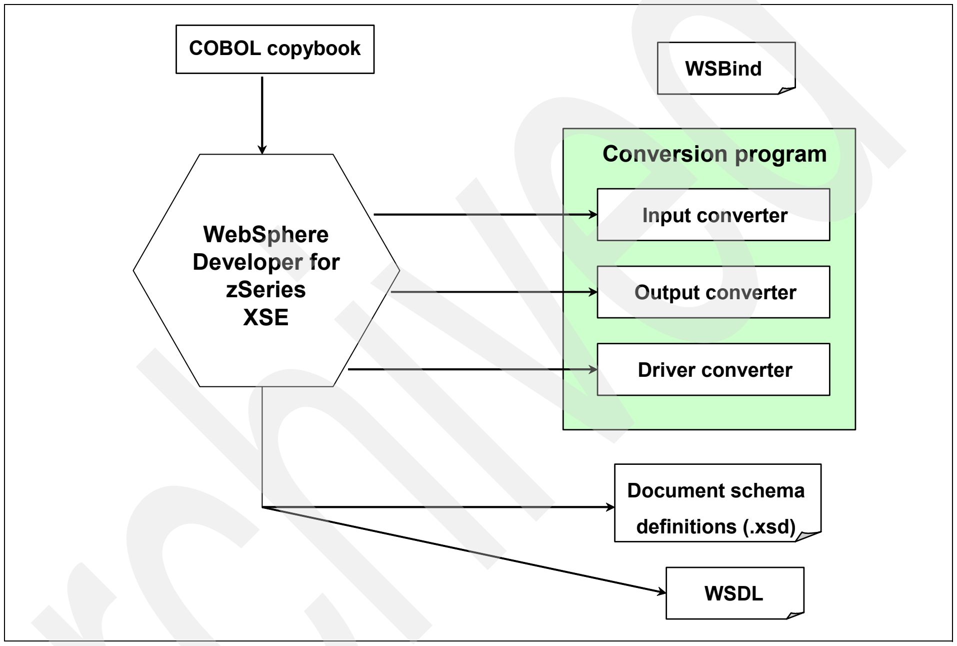


Figure 2-10 WebSphere Developer for zSeries

The artifacts generated by the Web Services Enablement wizard are:

- Input converter

  A COBOL program that takes an incoming XML document and maps it into the corresponding COBOL data structure that the existing CICS application expects.

- Output converter

  A COBOL program that takes the COBOL data results returned from the CICS application and maps them to an XML document.

- Converter driver

A COBOL program that shows how the input and output converters can be used to interact with the existing CICS application.

- Input document XML schema definition (XSD)

  XML schema that describes the incoming XML document.

- Output document XML schema definition (XSD)

  XML schema that describes the outgoing XML document

- WSDL

  Web service description file.

- WSBind

  Web service binding file.

For additional information visit the WebSphere Developer for zSeries Web site at:

http://www.ibm.com/software/awdtools/devzseries/

2.7 Catalog manager example application

The CICS catalog manager example application is a COBOL application designed to illustrate best practice when connecting CICS applications to external clients and servers.

The example is constructed around a simple sales catalog and order processing application, in which the end user can perform these functions:

- List the items in a catalog (implemented as a VSAM file)
- Inquire on individual items in the catalog
- Order items from the catalog

The base application has a 3270 user interface, but the modular structure, with well-defined interfaces between the components, makes it possible to add further components. In particular, the application comes with Web services support, which is designed to illustrate how you can extend an existing application into the Web services environment.

2.7.1 The base application

Figure 2-11 on page 68 shows the structure of the base application.

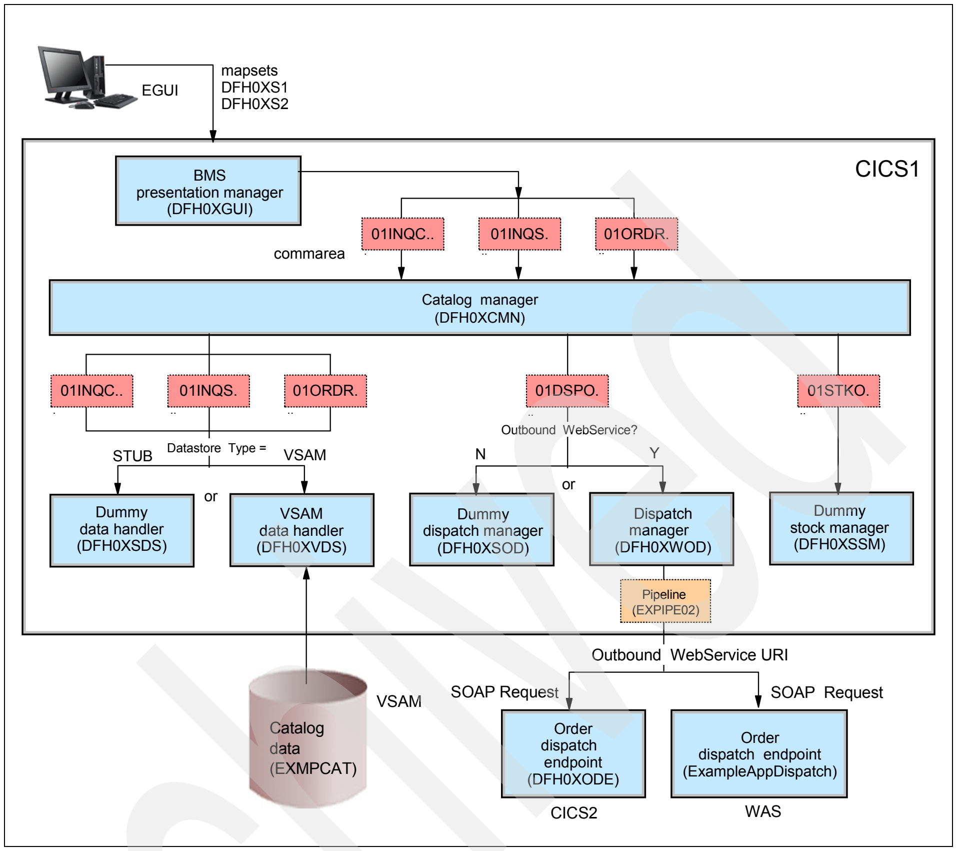

EGUI
mapsets DFH0XS1 DFH0XS2
CICS1
BMS presentation manager (DFH0XGUI)
01INQC..
01INQS.
01ORDR.
commarea
Catalog manager (DFH0XCMN)
01INQC..
01INQS.
01ORDR.
01DSPO.
01STKO.
Outbound WebService?
N
Y
STUB
Datastore Type =
VSAM
or
or
Dummy data handler (DFH0XSDS)
VSAM data handler (DFH0XVDS)
Dummy dispatch manager (DFH0XSOD)
Dispatch manager (DFH0XWOD)
Dummy stock manager (DFH0XSSM)
Pipeline (EXPIPE02)
Outbound WebService URI
VSAM
Catalog data (EXMPCAT)
SOAP Request
SOAP Request
Order dispatch endpoint (DFH0XODE)
Order dispatch endpoint (ExampleAppDispatch)
CICS2
WAS

Figure 2-11 Basic catalog manager application structure

The components of the base application are:

1. A BMS presentation manager (DFH0XGUI) that supports a 3270 terminal or emulator, and that interacts with the main catalog manager program.
2. A catalog manager program (DFH0XCMN) that is the core of the example application, and that interacts with several back-end components.
3. The back-end components are:
    a. A data handler program that provides the interface between the catalog manager program and the data store. The base application provides two versions of this program. They are the VSAM data handler program (DFH0XVDS), which stores data in a VSAM data set; and a dummy data handler (DFH0XSDS), which does not store data, but simply returns valid responses to its caller. Configuration options let you choose between the two programs.

b. A dispatch manager program that provides an interface for dispatching an order to a customer. Again, configuration options let you choose between the two versions of this program: DFHX0WOD is a service requester that invokes a remote order dispatch endpoint, and DFHX0SOD is a dummy program that simply returns valid responses to its caller. There are two equivalent order dispatch endpoints: DFH0XODE is a CICS service provider program; ExampleAppDispatchOrder.ear is an enterprise archive that can be deployed in WebSphere Application Server or similar environments.

c. A dummy stock manager program (DFH0XSSM) that returns valid responses to its caller, but takes no other action.

2.7.2 Web services support for the catalog example application

The Web services support extends the example application, providing a Web client front end and two versions of a Web services endpoint for the order dispatcher component.

The Web client front end and one version of the Web services endpoint are supplied as enterprise archives (EARs) that will run in the following environments:

- WebSphere Application Server Version 5 Release 1 or later
- WebSphere Studio Application Developer Version 5 Release 1 or later with a WebSphere unit test environment
- WebSphere Studio Enterprise Developer Version 5 Release 1 or later with a WebSphere unit test environment

The second version of the Web services endpoint is supplied as a CICS service provider application program (DFH0XODE).

Figure 2-12 on page 70 shows the structure of the Web services catalog application.

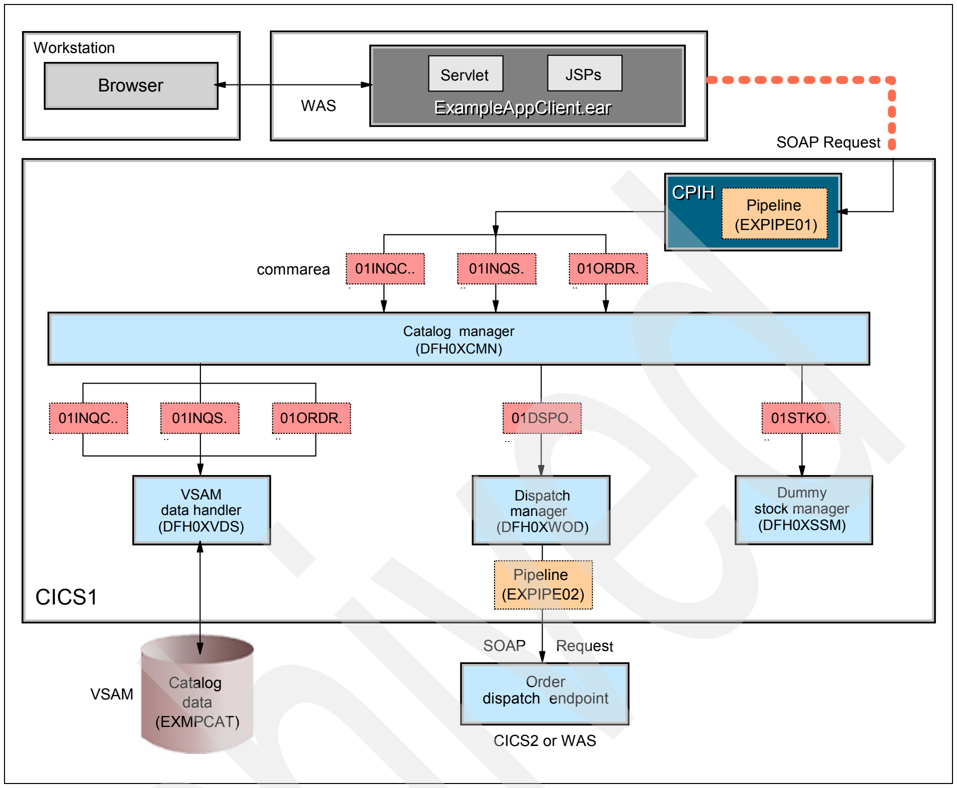


Figure 2-12 Web services catalog manager application structure

In this configuration, the application is accessed through:

- A Web browser client connected to WebSphere Application Server, in which ExampleAppClient.ear is deployed.

The order dispatch endpoint can be:

- A CICS service provider application
- A J2EE service provider application (ExampleAppDispatchOrder.ear) running in WebSphere Application Server

Part 2

Web service configuration

In this part we provide detailed instructions on how we configured our test CICS environment to support Web services using both HTTP and WebSphere MQ as transport mechanisms. We also describe how we deployed service requester and service provider applications in WebSphere Application Server, and how we used Web services to connect between WebSphere Application Server and CICS.

In chapter 5, we introduce the service integration bus and we outline the configuration steps for connecting to a CICS Web service via the bus.

3

Web services using HTTP

In this chapter we describe how we configured our test CICS environment to support Web services using HTTP as the transport mechanism.

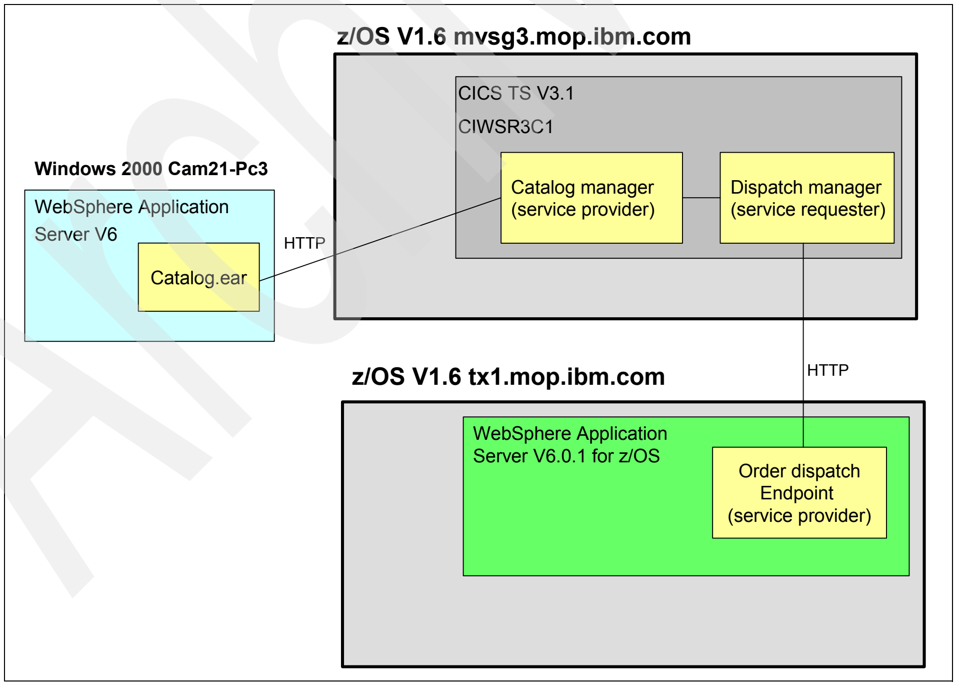
3.1 Preparation

After outlining our test configuration (Figure 3-1), we explain how we configured CICS as a service provider. In Section 3.2, “Configuring CICS as a service provider” on page 76, we show details of how we set up the environment, including:

- ► Configuring code page support
- ► Configuring the HFS file system
- ► Enabling the service provider application in CICS
- ► Deploying the service requester client in WebSphere Application Server for Windows
- ► Managing the WebSphere Application Server connection pool for Web services outbound connections

In Section 3.3, “Configuring CICS as a service requester” on page 94, we show how we configured the same CICS region to act as a service requester, including:

- ► Enabling the service requester application in CICS
- ► Deploying the service provider application in WebSphere Application Server for z/OS



z/OS V1.6 mvsg3.mop.ibm.com
CICS TS V3.1
CIWSR3C1
Catalog manager (service provider)
Dispatch manager (service requester)
Windows 2000 Cam21-Pc3
WebSphere Application Server V6
Catalog.ear
HTTP
HTTP
z/OS V1.6 tx1.mop.ibm.com
WebSphere Application Server V6.0.1 for z/OS
Order dispatch Endpoint (service provider)

Figure 3-1 Software components: Web services using HTTP transport

We do not provide information about how to install the software products and we assume the reader has a working knowledge of both CICS and WebSphere Application Server.

3.1.1 Software checklist

For the configuration shown in Figure 3-1 we used the levels of software shown in Table 3-1.

Table 3-1 Software used in the HTTP scenarios

Windows	z/OS
Windows 2000 SP4	z/OS V1.6
IBM WebSphere Application Server - ND V6.0.2.0	WebSphere Application Server for z/OS V6.0.1
	CICS Transaction Server V3.1
Internet Explorer® V6.0	
Our J2EE application ► Catalog.ear Catalog manager service requester application	CICS-supplied catalog Manager application Our user-supplied CICS programs ► SNIFFER (message handler program) ► CIWSMSGH (message handler program) Our J2EE application ► dispatchOrder.ear Catalog application service provider

3.1.2 Definition checklist

The z/OS definitions we used to configure the scenario are listed in Table 3-2.

Table 3-2 Definition checklist

Value	CICS TS	WebSphere Application Server
IP name	mvsg3.mop.ibm.com	tx1.mop.ibm.com
IP address	9.100.193.167	9.100.193.122
TCP/IP port	13301	13880
Job name	CIWSR3C1	CITGRS1S

Value	CICS TS	WebSphere Application Server
APPLID	A6POR3C1	
TCPIPSERVICE	R3C1	
Provider PIPELINE	EXPIPE01	
Requester PIPELINE	EXPIPE02	

3.1.3 The sample application

For our tests we used the sample program described in Section 2.7, “Catalog manager example application” on page 67. We do not document how to install the sample application itself, because this is explained in detail in *CICS Web Services Guide V3.1,* SC34-6458.

3.2 Configuring CICS as a service provider

In this section we discuss how we configured CICS as a service provider. The configuration we used is shown in Figure 3-2.

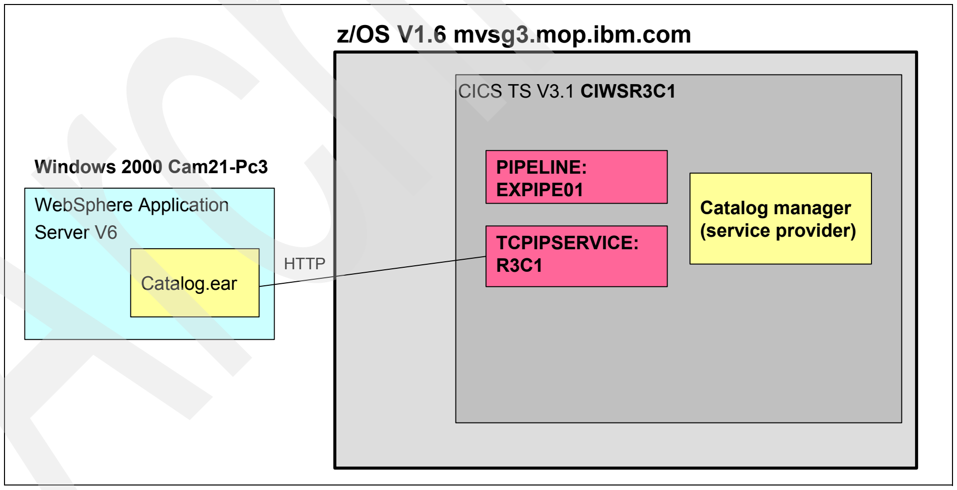


Figure 3-2 CICS as a service provider

3.2.1 Configuring code page support

We configured our z/OS system to support conversions between the two coded character sets used by the example application; these are shown in Example 3-1.

Example 3-1 CCSID description

```
037    EBCDIC Group 1: USA, Canada (z/OS), Netherlands, Portugal,
                 Brazil, Australia, New Zealand
1208 UTF-8 Level 3
```

To do this we added the statements shown in Example 3-2 to the conversion image for our z/OS system.

Example 3-2 Required conversions

```
CONVERSION 037,1208;
CONVERSION 1208,037;
```

We used the `SET UNI=31` z/OS command to activate the updated conversion image, where ***31*** is the suffix of the CUNUNIxx member of SYS1.PARMLIB.

For more information about code page support, see *CICS Installation Guide V3.1*, GC34-6426.

Tip: With z/OS V1R7, the Unicode Services environment can be dynamically updated when a conversion service is requested. If the appropriate table needed for the service is not already loaded into storage, Unicode Services will load the table without requiring an IPL or disrupting the caller's request. For more information, see *z/OS Support for Unicode: Unicode Services*, SA22-7649-06.

3.2.2 Configuring CICS

To enable CICS to receive Web service requests using HTTP we performed the following tasks:

- Updating CICS system initialization table (SIT) parameters
- Creating the HFS directories
- Configuring the TCPIPSERVICE resource definition
- Customizing the pipeline configuration file
- Writing a message handler program that changes the transaction ID
- Configuring the PIPELINE resource definition

- Installing the TCPIPSERVICE and PIPELINE definitions

Updating SIT parameters

Since we decided to use HTTP as the transport for the service request flows, we added the following SIT parameter:

```
TCPIP=YES
```

We then restarted the CICS region to put the parameter into effect.

Creating the HFS directories

Next we created the HFS directories (Example 3-3) used in the PIPELINE definition.

Example 3-3 HFS directories used in the PIPELINE definition

```
/CIWS/R3C1/config
/CIWS/R3C1/shelf
/CIWS/R3C1/wsbind/provider
/CIWS/R3C1/wsbind/requester
```

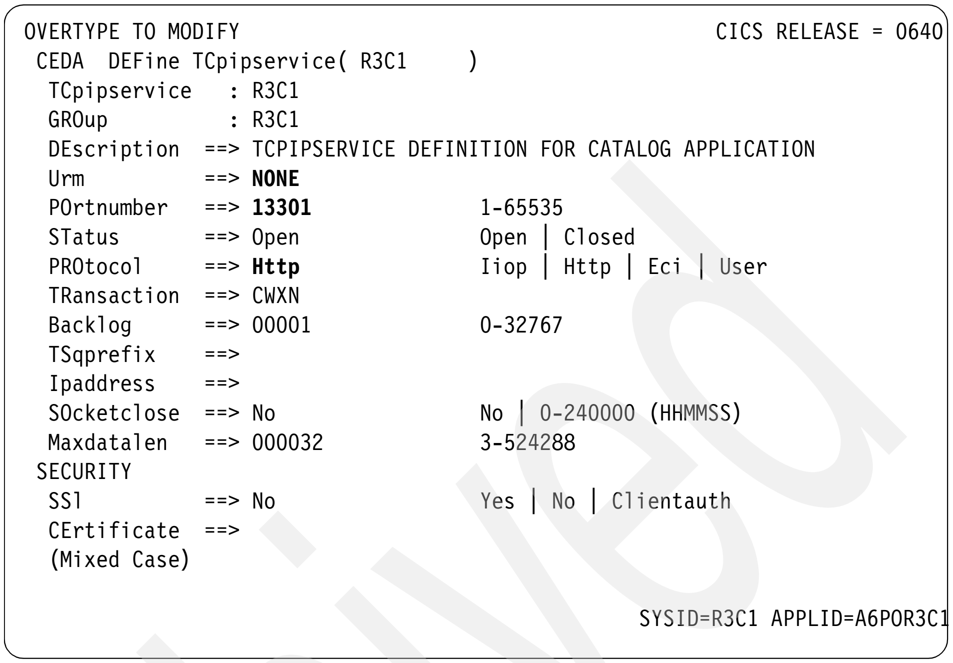
The CICS region user ID must have read permission to the /config directory, and update permission to the /shelf directory.

Configuring the TCPIPSERVICE definition

Next we logged onto CICS and created the TCPIPSERVICE resource definition in CICS using the command:

```
CEDA DEFINE TCPIPSERVICE(R3C1) GROUP(R3C1)
```

We defined the R3C1 TCPIPSERVICE as shown in Figure 3-3.



```
OVERTYPE TO MODIFY                                        CICS RELEASE = 0640
 CEDA  DEFine TCpipservice( R3C1     )
  TCpipservice   : R3C1
  GROup          : R3C1
  DEscription  ==> TCPIPSERVICE DEFINITION FOR CATALOG APPLICATION
  Urm          ==> NONE
  POrtnumber   ==> 13301             1-65535
  STatus       ==> Open              Open | Closed
  PROtocol     ==> Http              Iiop | Http | Eci | User
  TRansaction  ==> CWXN
  Backlog      ==> 00001             0-32767
  TSqprefix    ==>
  Ipaddress    ==>
  SOcketclose  ==> No                No | 0-240000 (HHMMSS)
  Maxdatalen   ==> 000032            3-524288
 SECURITY
  SSl          ==> No                Yes | No | Clientauth
  CErtificate  ==>
  (Mixed Case)

                                                   SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-3   CEDA DEFINE TCPIPSERVICE

We set the PORTNUMBER to 13301, the PROTOCOL to HTTP, and the URM to NONE. We allowed the other attributes to default, and we installed the R3C1 group.

The default setting for the SOCKETCLOSE attribute is NO. Therefore, when a connection is made between a Web service client and CICS, by default CICS keeps the connection open until the Web service client closes the connection. You could set a value (in seconds) for the SOCKETCLOSE attribute if you want to close a persistent connection after the timeout period is reached.

Recommendation: Do not set the SOCKETCLOSE attribute to 0 because this will close the connection after each request.

We used the default setting for SOCKETCLOSE and we configured WebSphere Application Server to timeout idle persisting connections (see “Managing the WebSphere Application Server connection pool” on page 89).

Customizing the pipeline configuration file

The default pipeline alias transaction ID used for inbound HTTP Web service requests is CPIH. We wanted to assign different transaction IDs to different service requests. To do that, we wrote a message handler program CIWSMSGH that replaced the transaction ID in the DFHWS-TRANID container with a transaction ID based on the service request (which can be retrieved from the DFHWS-WEBSERVICE container).

To activate the message handler program, we needed to make changes to the PIPELINE configuration file. We copied the file, basicsoap12provider.xml to the /CIWS/R3C1/config directory shown in Example 3-3 on page 78. The change we made is shown in Example 3-4.

Example 3-4 Service provider configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline
xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
    <transport>
      <default_transport_handler_list>
          <handler>
             <program>CIWSMSGH</program>
             <handler_parameter_list/>
          </handler>
      </default_transport_handler_list>
    </transport>
    <service>
      <terminal_handler>
        <cics_soap_1.2_handler/>
      </terminal_handler>
    </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

Note: The `<default_transport_handler_list>` specifies the message handlers that are invoked by default when any transport is in use.

Writing the message handler program

In this section we show how we used a message handler program to change the default transaction ID (CPIH) to a transaction ID based on the Web service request in the DFHWS-WEBSERVICE container.

Table 3-3 shows the relationship that we established between the transaction ID and the Web service request.

Table 3-3 Transaction ID to Web service name relationship

Transaction ID	Web service request
INQS	inquireSingle
INQC	inquireCatalog
ORDR	placeOrder

There are many reasons why you might want to change the transaction ID based on the service request, for example:

Security	You may want to use transaction security to control access to specific services.
Priority	You may want to assign different performance goals to specific services.
Accounting	You may need to charge users based on access to different services.

We show how we used transaction security to control access to specific services in Section 9.2, “Basic security configuration” on page 278.

Before we activated the message handler program we needed to create the new TRANSACTION definitions with the same characteristics as the CICS-supplied definition for CPIH. We used the CEDA COPY commands shown in Example 3-5 and then installed the definitions.

Example 3-5 CICS definitions - TRANSACTION

```
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(R3C1) AS(INQS)
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(R3C1) AS(INQC)
CEDA COPY TRANSACTION(CPIH) GROUP(DFHPIPE) TO(R3C1) AS(ORDR)
```

The program logic we used for the message handler program is shown in Example 3-6 through Example 3-9. The full program is shown in Section A.1, “Sample message handler program - CIWSMSGH” on page 528.

Note: A message handler can be written in any of the languages CICS supports. The CICS commands in the DPL subset can be used.

Example 3-6 shows the flow of control of the message handler program. The program will only execute for Web service requests.

Example 3-6 Message handler program - Flow of control

```
     IF WS-DFHFUNCTION equal 'RECEIVE-REQUEST'
         PERFORM VALIDATE-REQUEST THRU END-VAL-REQUEST
         PERFORM CHANGE-TRANID    THRU END-CHANGE-TRANID
         EXEC CICS
            DELETE CONTAINER('DFHRESPONSE')
         END-EXEC
     END-IF
     EXEC CICS RETURN END-EXEC.
```

Tip: When the message handler processes a request, it must delete the DFHRESPONSE container if a transition to the response phase of the pipeline will not take place.

Example 3-7 shows the code to get the Web service request from the DFHWS-WEBSERVICE container.

Example 3-7 Message handler program - Get the DFHWS-WEBSERVICE container

```
     EXEC CICS
     GET CONTAINER('DFHWS-WEBSERVICE')
     SET(ADDRESS OF CONTAINER-DATA)
     FLENGTH(CONTAINER-LEN)
     END-EXEC.
```

Example 3-8 shows the code that determines the new transaction ID, replacing the default transaction ID in the DFHWS-TRANID container.

Example 3-8 Message handler program - Determine new transaction ID

```
*---------------- CHANGE DEFAULT TRANID CPIH/CPIL ---------
 CHANGE-TRANID.
       EXEC CICS GET CONTAINER('DFHWS-TRANID')
            SET(ADDRESS OF CONTAINER-DATA)
            FLENGTH(CONTAINER-LEN)
       END-EXEC.
     IF WS-WEBSERVICES = 'inquireSingle'
        MOVE 'INQS' TO CA-TRANID
        PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
     END-IF
     IF WS-WEBSERVICES = 'inquireCatalog'
```

```
       MOVE 'INQC' TO CA-TRANID
       PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
    END-IF
    IF WS-WEBSERVICES = 'placeOrder'
       MOVE 'ORDR' TO CA-TRANID
       PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
    END-IF.
END-CHANGE-TRANID. EXIT.
```

Example 3-9 shows how the program changes the transaction ID in the DFHWS-TRANID container, and performs an EXEC CICS PUT CONTAINER.

Example 3-9 Message handler program - Change transaction ID

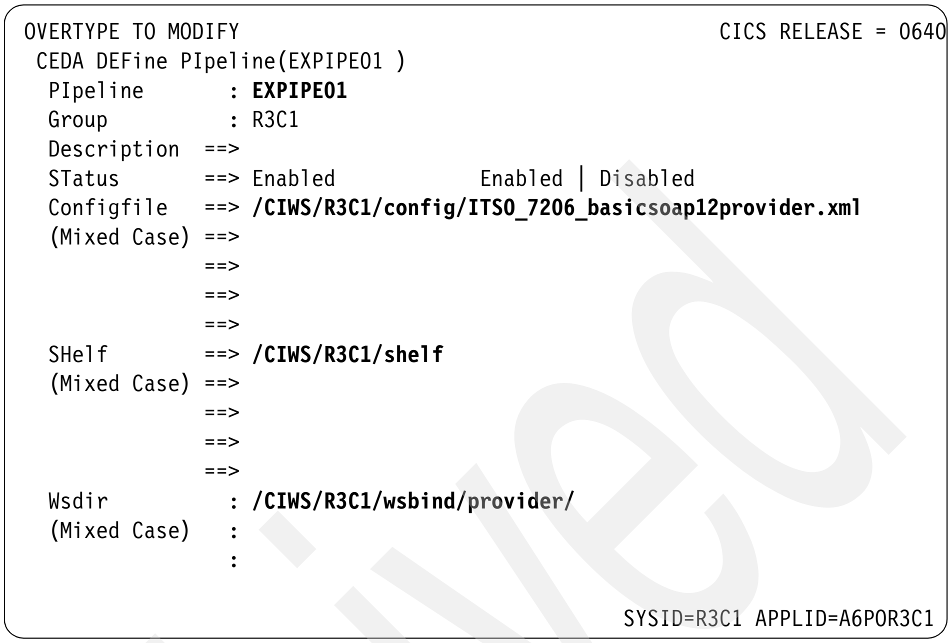
```
CHANGE-CONTAINER.
      MOVE CA-TRANID TO CONTAINER-DATA(1:4)
      EXEC CICS PUT CONTAINER('DFHWS-TRANID')
           FROM(CONTAINER-DATA)
           FLENGTH(CONTAINER-LEN)
      END-EXEC.
END-CHANGE-CONTAINER. EXIT.
```

Configuring the PIPELINE definition

We then defined the pipeline for the CICS service provider using the following CICS command:

```
CEDA DEFINE PIPELINE(EXPIPE01) GROUP(R3C1)
```

We defined the EXPIPE01 pipeline as shown in Figure 3-4 on page 84.


```
OVERTYPE TO MODIFY                                         CICS RELEASE = 0640
 CEDA DEFine PIpeline(EXPIPE01 )
  PIpeline       : EXPIPE01
  Group          : R3C1
  Description  ==>
  STatus       ==> Enabled            Enabled | Disabled
  Configfile   ==> /CIWS/R3C1/config/ITSO_7206_basicsoap12provider.xml
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  SHelf        ==> /CIWS/R3C1/shelf
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  Wsdir          : /CIWS/R3C1/wsbind/provider/
  (Mixed Case)   :
                 :

                                                    SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-4 CEDA DEFINE PIPELINE

Tip: The colons in front of Wsdir on the CEDA screen in Figure 3-4 mean that you are not able to enter input on the lines. You must press **F8** to be able to enter the path for the directory.

- We set CONFIGFILE to the name of our pipeline configuration file:

  /CIWS/R3C1/config/ITSO_7206_basicsoap12provider.xml

  Note: In a subsequent section we show how we modified the basicsoap12provider.xml file, which is why we did not use the pipeline configuration file provided in the /usr/lpp/cicsts/cicsts31/samples/pipelines directory.

- We set SHELF to the name of the shelf directory:

  /CIWS/R3C1/shelf

- We copied the following wsbind files to directory /CIWS/R3C1/wsbind/provider from the CICS supplied directory /usr/lpp/cicsts/cicsts31/samples/webservices/wsbind/provider/:
  - inquireSingle.wsbind
  - inquireCatalog.wsbind
  - placeOrder.wsbind

  Note: /usr/lpp/cicsts/cicsts31 is our CICS HFS install root.

- We set WSDIR to the Web service binding directory that contains the wsbind files for the sample application:

  /CIWS/R3C1/wsbind/provider/

Installing the PIPELINE definition

We then used CEDA to install the PIPELINE definition. When the PIPELINE is installed CICS scans the wsdir directory, and dynamically creates WEBSERVICE and URIMAP definitions for the wsbind files found.

Figure 3-5 shows a CEMT INQUIRE PIPELINE for EXPIPE01.

```
INQUIRE PIPELINE
RESULT - OVERTYPE TO MODIFY
  Pipeline(EXPIPE01)
  Enablestatus( Enabled )
  Configfile(/CIWS/R3C1/config/ITSO_7206_basicsoap12provider.xml)
  Shelf(/CIWS/R3C1/shelf/)
  Wsdir(/CIWS/R3C1/wsbind/provider/)

                                                 SYSID=R3C1 APPLID=A6POR3C1
```

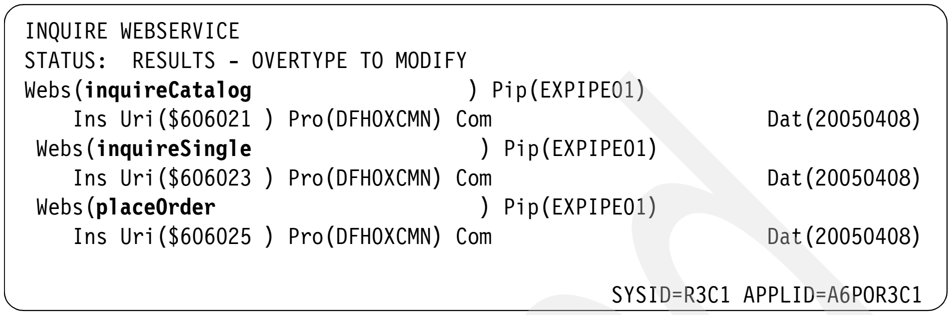
Figure 3-5 CEMT INQUIRE PIPELINE - EXPIPE01

WEBSERVICE resource definitions

In our configuration the WEBSERVICE resource definitions are dynamically installed when the PIPELINE is installed. Optionally, we could define and install the Web services using the CEDA DEFINE WEBSERVICE command; however, this is not normally necessary when using the CICS Web services assistant.

Note: The name for an explicitly defined WEBSERVICE is limited to 8 characters in length, whereas the automatically installed Web service names can be up to 32 characters in length.

When a Web service is dynamically installed, the name of the service is taken from the wsbind file. Figure 3-6 shows the Catalog manager application Web services dispatchOrder, inquireCatalog, inquireSingle and placeOrder.



```
INQUIRE WEBSERVICE
STATUS:  RESULTS - OVERTYPE TO MODIFY
Webs(inquireCatalog                  ) Pip(EXPIPE01)
    Ins Uri($606021 ) Pro(DFHOXCMN) Com                       Dat(20050408)
 Webs(inquireSingle                   ) Pip(EXPIPE01)
    Ins Uri($606023 ) Pro(DFHOXCMN) Com                       Dat(20050408)
 Webs(placeOrder                      ) Pip(EXPIPE01)
    Ins Uri($606025 ) Pro(DFHOXCMN) Com                       Dat(20050408)

                                                   SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-6   CEMT INQUIRE WEBSERVICE

URIMAP resource definition

In our configuration the URIMAP resource definitions are dynamically installed when the PIPELINE is installed. Optionally, we could define and install them manually using the CEDA DEFINE URIMAP command; however, this is not normally necessary when using the CICS Web services assistant.

Using the URIMAP to change the default transaction ID

As an alternative to using the message handler program to change the transaction IDs the services run under, we could have used the URIMAP resource definition. That would, however, mean that we would need to define a URIMAP for each deployed Web service.

A sample resource definition that could have been used is shown in Example 3-10.

Example 3-10   CEDA URIMAP definitions

```
CEDA DEFINE URIMAP(INQS) GROUP(R3C1) HOST(*)
PATH(/exampleApp/inquireSingle)
   PIPELINE(EXPIPE01) TRANSACTION(INQS) USAGE(PIPELINE)
   WEBSERVICE(inquireSingle)
CEDA DEFINE URIMAP(INQC) GROUP(R3C1) HOST(*)
PATH(/exampleApp/inquireCatalog)
   PIPELINE(EXPIPE01) TRANSACTION(INQC) USAGE(PIPELINE)
   WEBSERVICE(inquireCatalog)
CEDA DEFINE URIMAP(ORDR) GROUP(R3C1) HOST(*)
PATH(/exampleApp/placeOrder)
   PIPELINE(EXPIPE01) TRANSACTION(ORDR) USAGE(PIPELINE)
   WEBSERVICE(placeOrder)
```

Figure 3-7 shows a URIMAP resource definition dynamically installed when the PIPELINE is installed.

```
INQUIRE URIMAP
RESULT - OVERTYPE TO MODIFY
  Urimap($606021)
  Usage(Pipe)
  Enablestatus( Enabled )
  Analyzerstat(Noanalyzer)
  Scheme(Http)
  Redirecttype( None )
  Tcpipservice()
  Host(*)
  Path(/exampleApp/inquireCatalog)
  Transaction(CPIH)
  Converter()
  Program()
  Pipeline(EXPIPE01)
  Webservice(inquireCatalog)
  Userid()
  Certificate()
  Ciphers()
  Templatename()
                                                 SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-7 CEMT INQUIRE URIMAP

3.2.3 Configuring WebSphere Application Server on Windows

In this section we discuss how we deployed the Web service client on WebSphere Application Server for Windows. We discuss how we used the WebSphere administrative console to install the Web service client.

Installing the service requester

CICS TS V3.1 provides a sample Web service client, ExampleAppClient.ear. This application archive is built at the J2EE 1.3 level. We planned to use the client in a J2EE 1.4 environment (WebSphere Application Server V6), therefore we migrated the client. We called the new application archive file Catalog.ear.

Note: The CICS-supplied ExampleAppClient.ear file is located in the /usr/lpp/cicsts/cicsts31/samples/webservices/client directory.

Deploying the Catalog.ear file on WebSphere Application Server

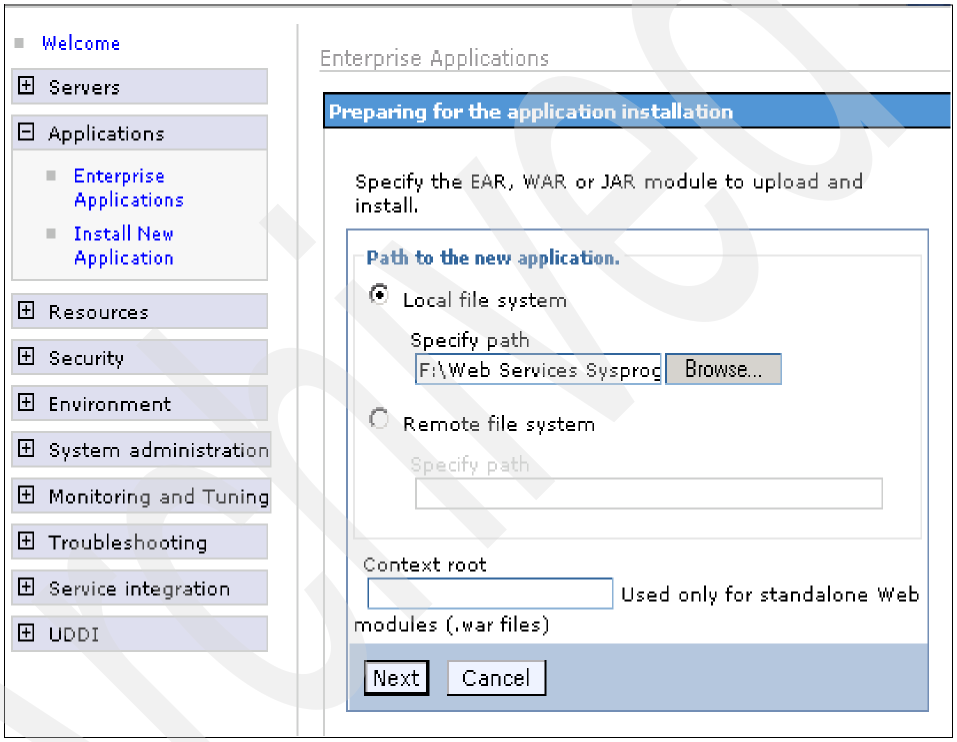
Next we deployed the Catalog.ear file on WebSphere Application Server for Windows. To log on to the WebSphere administrative console, we opened a Web browser window and entered the following url:

```
http://cam21-pc11:9060/admin
```

We entered a user ID and were presented with the window shown in Figure 3-8. We clicked **Local file system** and then clicked **Browse** to locate the EAR file:

```
   F:\Web Services Sysprog\LAN book\addmat\src\ears\Catalog.ear
```

.



Preparing for the application installation

Specify the EAR, WAR or JAR module to upload and install.

Path to the new application.

(●) Local file system

Specify path: F:\Web Services Sysprog

( ) Remote file system

Specify path

Context root: Used only for standalone Web modules (.war files)

Figure 3-8 WebSphere administrative console - Install new application

We clicked **Next** → **Next** → **Next** → **Next** → **Finish** and then saved the configuration.

Next we clicked **Enterprise Applications**, selected the Catalog application, and clicked **Start** to start the application.

Managing the WebSphere Application Server connection pool

Since our service requester runs in WebSphere Application Server, the application can take advantage of the connection pooling for Web services HTTP outbound connections.

The HTTP transport properties are set using the JVM™ custom property panel in the WebSphere administrative console. The following properties apply to our scenario:

- **com.ibm.websphere.webservices.http.connectionTimeout**

  This property specifies the interval, in seconds, after which a connection request times out and the WebServicesFault("Connection timed out") error occurs. The wait time is needed when the maximum number of connections in the connection pool is reached. For example, if the property is set to 300 and the maximum number of connections is reached, the connector waits for 300 seconds until a connection is available. After 300 seconds, the WebServicesFault("Connection timed out") error occurs if a connection is not available. If the property is set to 0 (zero), the connector waits until a connection is available.

  We allowed this property setting to default to 300 seconds.

- **com.ibm.websphere.webservices.http.maxConnection**

  This property specifies the maximum number of connections that are created in the HTTP outbound connector connection pool. If the property is set to 0 (zero), the com.ibm.websphere.webservices.http.connectionTimeout property is ignored. The connector attempts to create as many connections as allowed by the system.

  We allowed this property setting to default to 50.

- **com.ibm.websphere.webservices.http.connectionPoolCleanUp**

  This property specifies the interval, in seconds, between runs of the connection pool maintenance thread. When the pool maintenance thread runs, the connector discards any connections remaining idle for longer than the time set in com.ibm.websphere.webservices.http.connectionIdleTimeout property.

  We allowed this property setting to default to180 seconds.

- **com.ibm.websphere.webservices.http.connectionIdleTimeout**

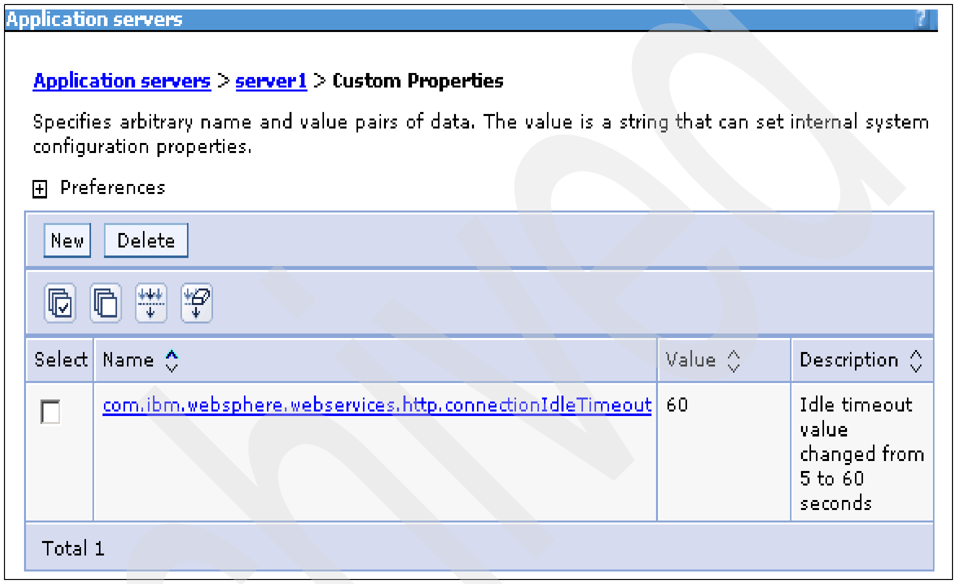
  This property specifies the interval, in seconds, after which an idle connection is discarded.

  We changed this property setting from the default (5 seconds) to 60 seconds because we wanted the connections to persist for a longer period.

We used the WebSphere administrative console to change the connection idle timeout from the default 5 seconds to 60 seconds:

We clicked **Servers** → **Application servers** → **server1** → **Java and Process management** → **Environment Entries** → **New**, and on the presented window, we entered the value shown in Figure 3-9.

We restarted the application server to activate the change.


Application servers

Application servers > server1 > Custom Properties

Specifies arbitrary name and value pairs of data. The value is a string that can set internal system configuration properties.

Preferences

New | Delete

Select	Name	Value	Description
☐	com.ibm.websphere.webservices.http.connectionIdleTimeout	60	Idle timeout value changed from 5 to 60 seconds

Total 1

Figure 3-9 WebSphere admin console - Setting connection idle timeout

Tip: For more information about the WebSphere Application Server connection pooling properties see "Additional HTTP transport properties for Web services applications" in the WebSphere Application Server information center.

3.2.4 Testing the configuration

In this section we discuss how we tested the configuration by invoking the Web client application running on WebSphere Application Server for Windows.

Running the Web client application

We started a browser session and entered the url:

```
http://cam21-pc11:9080/CatalogWeb/Welcome.jsp
```

The window shown in Figure 3-10 was displayed.

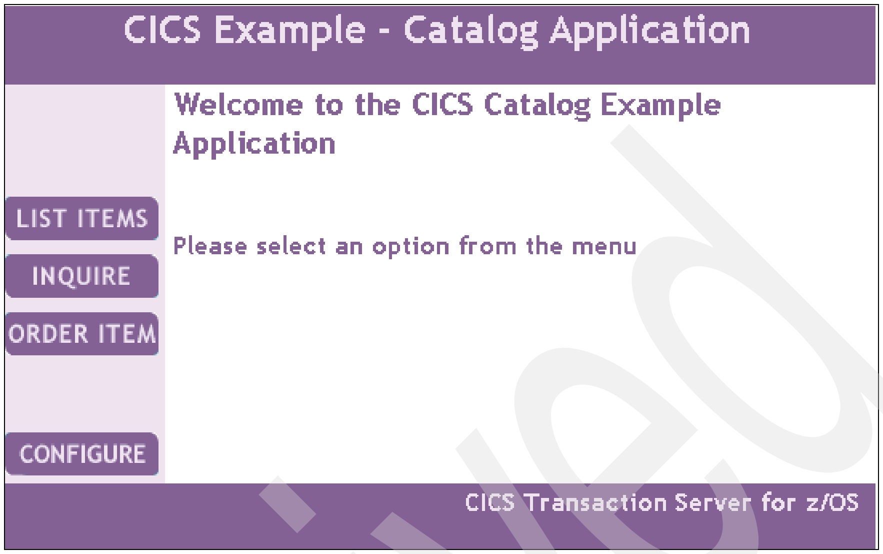
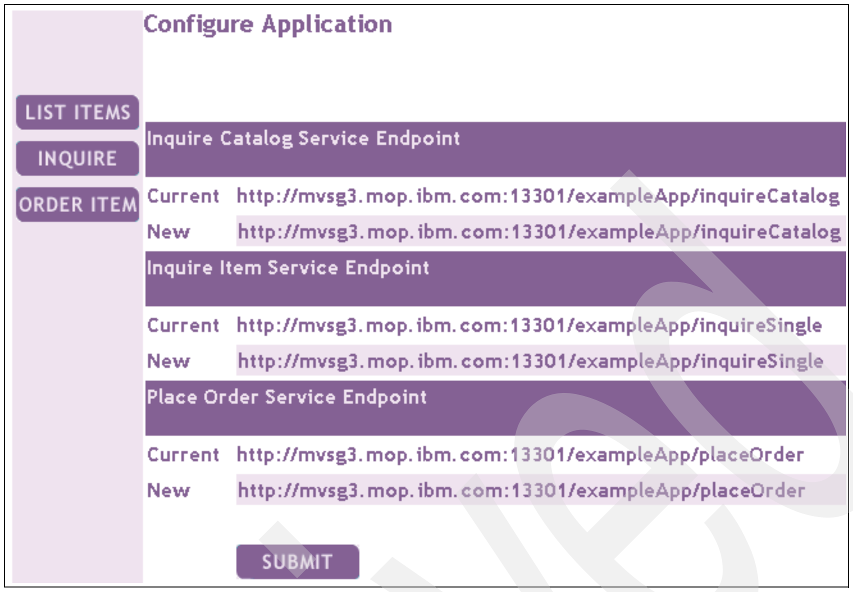

CICS Example - Catalog Application
Welcome to the CICS Catalog Example Application
LIST ITEMS
INQUIRE
ORDER ITEM
CONFIGURE
Please select an option from the menu
CICS Transaction Server for z/OS

Figure 3-10 CICS - Catalog application

We clicked **CONFIGURE**, and the window in Figure 3-11 was presented. We entered the following addresses:

- Inquire catalog
  `http://mvsg3.mop.ibm.com:13301/exampleApp/inquireCatalog`
- Inquire item
  `http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle`
- Place order
  `http://mvsg3.mop.ibm.com:13301/exampleApp/placeOrder`

and clicked **SUBMIT**.


Configure Application

Inquire Catalog Service Endpoint

Current http://mvsg3.mop.ibm.com:13301/exampleApp/inquireCatalog

New http://mvsg3.mop.ibm.com:13301/exampleApp/inquireCatalog

Inquire Item Service Endpoint

Current http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle

New http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle

Place Order Service Endpoint

Current http://mvsg3.mop.ibm.com:13301/exampleApp/placeOrder

New http://mvsg3.mop.ibm.com:13301/exampleApp/placeOrder

Figure 3-11 CICS - Catalog application configuration

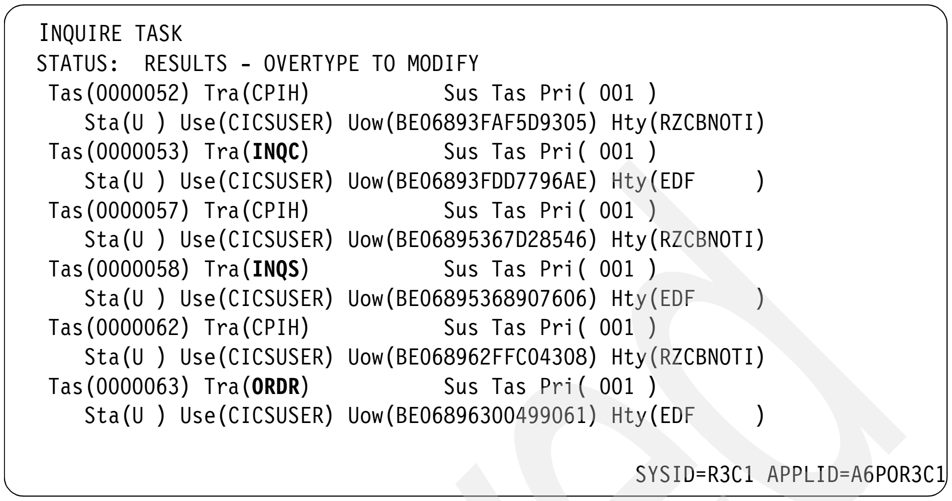
Next we started three Web browser sessions and entered the URL for each browser:

```
http://cam21-pc11:9080/CatalogWeb/Welcome.jsp
```

The Catalog application welcome page (Figure 3-10 on page 91) was presented. We then invoked a different service in each of the browsers:

- LIST ITEMS in browser one
- INQUIRE in browser two
- ORDER ITEM in browser three

From a CICS 3270 screen we used the CICS Execution Diagnostic Facility (EDF) to intercept each of the INQC, INQS and ORDR transactions. We then used the CEMT INQUIRE TASK command to view the in-flight transactions (Figure 3-12).


```
INQUIRE TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000052) Tra(CPIH)          Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE06893FAF5D9305) Hty(RZCBNOTI)
 Tas(0000053) Tra(INQC)          Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE06893FDD7796AE) Hty(EDF     )
 Tas(0000057) Tra(CPIH)          Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE06895367D28546) Hty(RZCBNOTI)
 Tas(0000058) Tra(INQS)          Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE06895368907606) Hty(EDF     )
 Tas(0000062) Tra(CPIH)          Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE068962FFC04308) Hty(RZCBNOTI)
 Tas(0000063) Tra(ORDR)          Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE06896300499061) Hty(EDF     )

                                                SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-12   CEMT INQUIRE TASK

Figure 3-12 shows one instance of each of the INQC, INQS, and ORDR transactions. For each transaction there is an associated pipeline alias transaction CPIH. We noted that these transactions are currently all running under the CICS default user ID CICSUSER.

Example 3-11 shows the output from our message handler program CIWSMSGH for the three service requests. Both the DFHWS-WEBSERVICE and DFHWS-TRANID containers are logged. See Section A.1, “Sample message handler program - CIWSMSGH” on page 528 for more information about the CIWSMSGH program.

Example 3-11   Sample output from message handler program - CIWSMSGH

```
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: inquireCatalog
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: INQC
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: inquireSingle
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: INQS
CIWSMSGH:  >================================<
```

```
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: placeOrder
CIWSMSGH:  ---------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: ORDR
```

3.3 Configuring CICS as a service requester

In this section we discuss how we configured CICS to support outbound Web service requests. The configuration we used is shown in Figure 3-13.

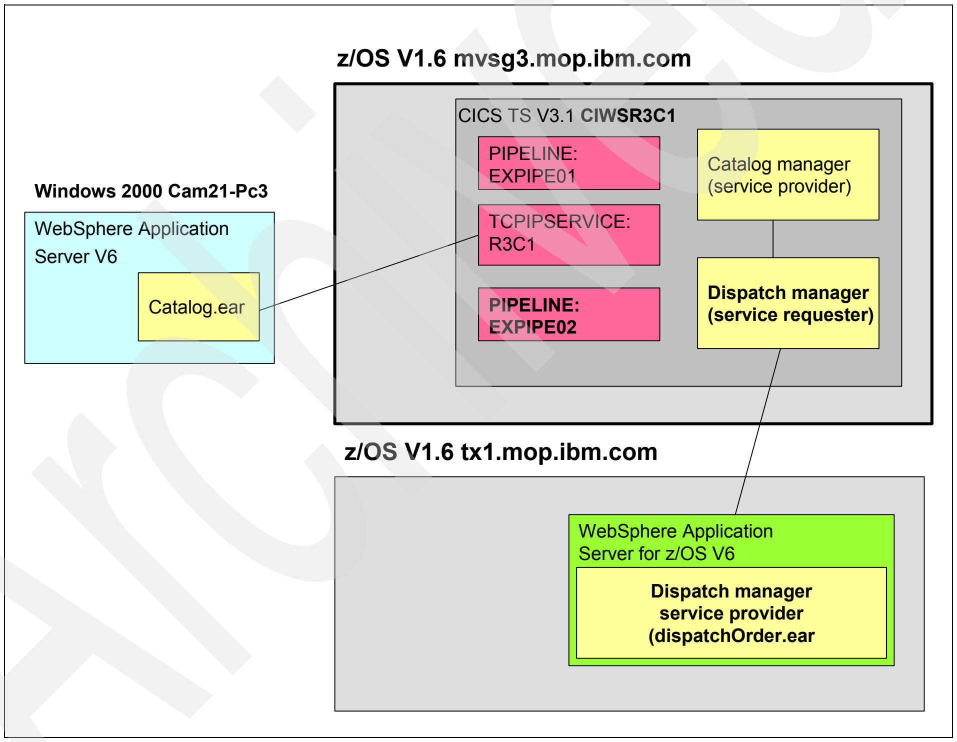

z/OS V1.6 mvsg3.mop.ibm.com
CICS TS V3.1 CIWSR3C1
PIPELINE: EXPIPE01
Catalog manager (service provider)
TCPIPSERVICE: R3C1
PIPELINE: EXPIPE02
Dispatch manager (service requester)
Windows 2000 Cam21-Pc3
WebSphere Application Server V6
Catalog.ear
z/OS V1.6 tx1.mop.ibm.com
WebSphere Application Server for z/OS V6
Dispatch manager service provider (dispatchOrder.ear

Figure 3-13   CICS as service requester

Figure 3-13 shows the TCPIPSERVICE R3C1 used for inbound HTTP Web service requests. Note that a TCPIPSERVICE is not required for outbound HTTP Web service requests from CICS.

3.3.1 Configuring CICS

To enable CICS to generate Web service requests using HTTP, we performed the following tasks:

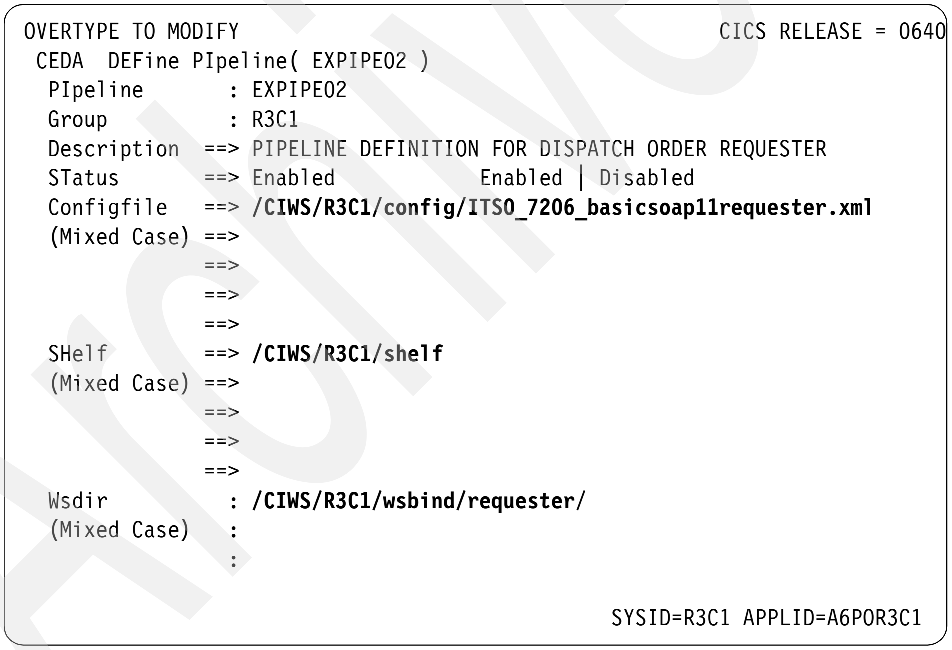
- Configuring the PIPELINE definition
- Configuring the requester TRANSACTION definition
- Configuring the sample application

Configuring the PIPELINE definition

We defined the PIPELINE for the CICS service requester using the following CICS command:

```
CEDA DEFINE PIPELINE(EXPIPE02) GROUP(R3C1)
```

We defined the EXPIPE02 pipeline as shown in Figure 3-14.



```
OVERTYPE TO MODIFY                                           CICS RELEASE = 0640
 CEDA  DEFine PIpeline( EXPIPE02 )
  PIpeline        : EXPIPE02
  Group           : R3C1
  Description  ==> PIPELINE DEFINITION FOR DISPATCH ORDER REQUESTER
  STatus       ==> Enabled            Enabled | Disabled
  Configfile   ==> /CIWS/R3C1/config/ITSO_7206_basicsoap11requester.xml
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  SHelf        ==> /CIWS/R3C1/shelf
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  Wsdir           : /CIWS/R3C1/wsbind/requester/
  (Mixed Case)    :
                  :

                                                      SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-14 CEDA DEFINE PIPELINE command

- We set the CONFIGFILE attribute to:

  /CIWS/R3C1/config/ITSO_7206_basicsoap11requester.xml

- We set the SHELF attribute to:

  /CIWS/R3C1/shelf
- We copied the wsbind file dispatchOrder.wsbind to directory /CIWS/R3C1/wsbind/requester from the CICS-supplied directory: /usr/lpp/cicsts/cicsts31/samples/webservices/wsbind/requester/
- We set WSDIR to the Web service binding directory that contains the wsbind files for the sample application:

  /CIWS/R3C1/wsbind/requester/

Note: In 3.2, “Configuring CICS as a service provider” on page 76, we used the basicsoap12provider.xml configuration file, which supports both SOAP 1.1 and SOAP 1.2 inbound service requests. CICS only supplies a basicsoap11requester.xml configuration file for SOAP 1.1 outbound requests.

Figure 3-15 shows a CEMT INQUIRE PIPELINE for EXPIPE02.

```
INQUIRE PIPELINE
RESULT - OVERTYPE TO MODIFY
  Pipeline(EXPIPE02)
  Enablestatus( Enabled )
  Configfile(/CIWS/R3C1/config/ITSO_7206_basicsoap11requester.xml)
  Shelf(/CIWS/R3C1/shelf/)
  Wsdir(/CIWS/R3C1/wsbind/requester/)


                                                    SYSID=R3C1 APPLID=A6POR3C1
```

Figure 3-15 CEMT INQUIRE PIPELINE - EXPIPE02

Configuring the requester transaction

The duration a Web service requester task will wait for a response is controlled by the DTIMOUT attribute on the TRANSACTION definition. The CICS default is NO, meaning that the request will wait indefinitely. We used the CEDA ALTER command to change the DTIMOUT value for the ORDR transaction to 30 seconds:

```
CEDA ALTER TRANSACTION(ORDR) GROUP(R3C1) DTIMOUT(30)
```

Timeout considerations

When a CICS application is the service *provider*, normal resource timeout mechanisms such as RTIMEOUT (read timeout) apply. If, however, the requester decides to time out before CICS is ready to send the response, the provider transaction abends and CICS issues the messages shown in Example 3-12.

Example 3-12 CICS service provider error message

```
DFHPI0401 12/15/2005 15:51:12 A6POR3C1 ORDR The CICS pipeline HTTP transport
mechanism failed to send a response or receive a request because the connection
was closed.
 DFHPI0503 12/15/2005 15:51:12 A6POR3C1 ORDR The CICS Pipeline Manager has
failed to send a response on the underlying transport. TRANSPORT: HTTP,
PIPELINE: EXPIPE01.
```

For a CICS application which is a service *requester* (Figure 3-16), timeout is controlled by the DTIMOUT attribute on the TRANSACTION definition.

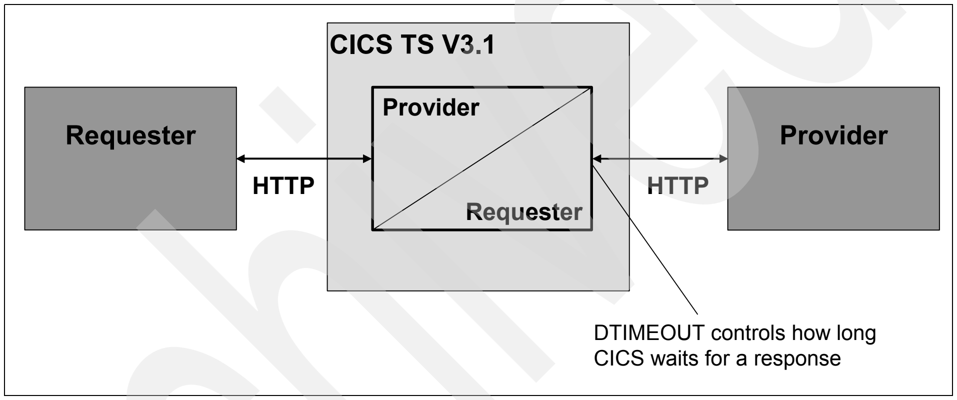


Figure 3-16 CICS timeout considerations

Note: The DTIMOUT attribute on the TRANSACTION definition only controls service requester timeout if HTTP is used as the transport mechanism.

If the request times out, CICS issues the message shown in Example 3-13.

Example 3-13 CICS service requester error message

```
DFHPI0504 12/15/2005 15:55:33 A6POR3C1 ORDR The CICS Pipeline Manager has
failed to communicate with a remote server due to an error in the underlying
transport. TRANSPORT: HTTP, PIPELINE: EXPIPE02.
```

Configuring the sample application

We used the catalog manager configuration transaction (ECFG) to configure the example application. Figure 3-17 shows how we changed the setting of `Outbound`

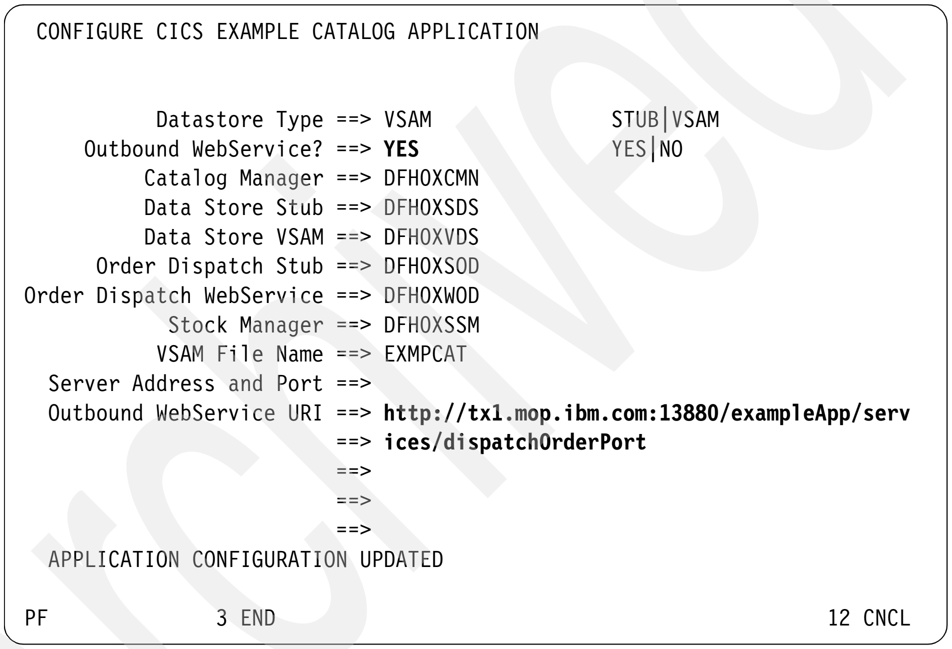
`WebService` to **Yes,** and entered the URI of the service provider for our outbound service request:

```
http://tx1.mop.ibm.com:13880/exampleApp/services/dispatchOrderPort
```

We then pressed **PF3** to save the configuration.

Tip: The 3270 terminal we used to configure the sample application had to be set to NOUCTRAN. We used the following CICS command:

```
CEOT NOUCTRAN
```



```
 CONFIGURE CICS EXAMPLE CATALOG APPLICATION


           Datastore Type ==> VSAM               STUB|VSAM
     Outbound WebService? ==> YES                YES|NO
          Catalog Manager ==> DFH0XCMN
          Data Store Stub ==> DFH0XSDS
          Data Store VSAM ==> DFH0XVDS
      Order Dispatch Stub ==> DFH0XSOD
Order Dispatch WebService ==> DFH0XWOD
            Stock Manager ==> DFH0XSSM
           VSAM File Name ==> EXMPCAT
  Server Address and Port ==>
  Outbound WebService URI ==> http://tx1.mop.ibm.com:13880/exampleApp/serv
                          ==> ices/dispatchOrderPort
                          ==>
                          ==>
                          ==>
  APPLICATION CONFIGURATION UPDATED

PF             3 END                                          12 CNCL
```

Figure 3-17 Catalog application configuration screen

With this configuration, the sample application uses the command EXEC CICS INVOKE WEBSERVICE(“dispatchOrder”) to invoke the dispatchOrder service which in our configuration runs in WebSphere Application Server for z/OS.

3.3.2 Configuring WebSphere Application Server for z/OS

In this section we discuss how we deployed the ExampleAppDispatchOrder service provider application on WebSphere Application Server, including:

- How we used FTP to download the ear file

- How we used the WebSphere administrative console to install the application

Downloading the EAR file

The CICS-supplied ExampleAppDispatchOrder.ear file is located in directory:

/usr/lpp/cicsts/cicsts31/samples/webservices/client

We used the Windows **ftp** command shown in Example 3-14 to download the file to the workstation.

Example 3-14 Using ftp to download the EAR file

```
F:\>cd F:\Web Services Sysprog\LAN book\addmat\src\Catalog
Application\Configuration part
F:\Web Services Sysprog\LAN book\addmat\src\Catalog Application\Configuration
part>ftp mvsg3.mop.ibm.com
Connected to 9.100.193.167.
220-FTPD1 IBM FTP CS V1R6 at MVSG3.pssc.mop.ibm.com, 17:39:13 on 2005-11-24.
220 Connection will close if idle for more than 5 minutes.
User (9.100.193.167:(none)): CIWSTJ
331 Send password please.
Password:
230 CIWSTJ is logged on.  Working directory is "CIWSTJ.".
ftp> cd /usr/lpp/cicsts/cicsts31/samples/webservices/client
250 HFS directory /usr/lpp/cicsts/cicsts31/samples/webservices/client is the
current working directory
ftp> get ExampleAppDispatchOrder.ear
200 Port request OK.
125 Sending data set
/usr/lpp/cicsts/cicsts31/samples/webservices/client/Example
AppDispatchOrder.ear
250 Transfer completed successfully.
ftp: 50623 bytes received in 0,03Seconds 1687,43Kbytes/sec.
ftp> bye
221 Quit command received. Goodbye.
```

Installing the service provider

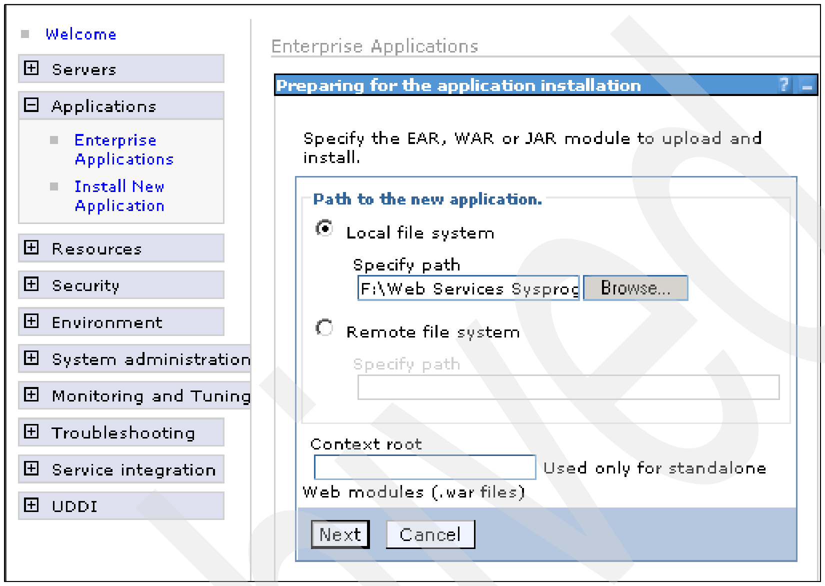
CICS TS V3.1 provides a sample Web service provider application ExampleAppDispatchOrder.ear. This application archive is built at the J2EE 1.3 level. We planned to use the application in a J2EE 1.4 environment (WebSphere Application Server V6), therefore we migrated the application. We called the new application archive file dispatchOrder.ear.

Next we installed the dispatchOrder.ear file on WebSphere Application Server for z/OS. We opened a Web browser window and entered the URL:

`http://tx1.mop.ibm.com:13880/ibm/console`

After logging in, we clicked **Applications** → **Install New Application.** On the next window (Figure 3-18) we clicked **Local file system** and entered the path of the EAR file:

F:\Web Services Sysprog\LAN book\addmat\src\ears\dispatchOrder.ear



Specify the EAR, WAR or JAR module to upload and install.

Path to the new application.

(●) Local file system

Specify path: F:\Web Services Sysprog

( ) Remote file system

Specify path

Context root

Used only for standalone Web modules (.war files)

Figure 3-18 WebSphere Administrative console

We clicked **Next** → **Next** → **Next** → **Next** → **Next** → **Finish,** and then saved the changes to the master configuration.

3.3.3 Testing the configuration

In this section we discuss how we tested the configuration using the same method described in 3.2.4, “Testing the configuration” on page 90.

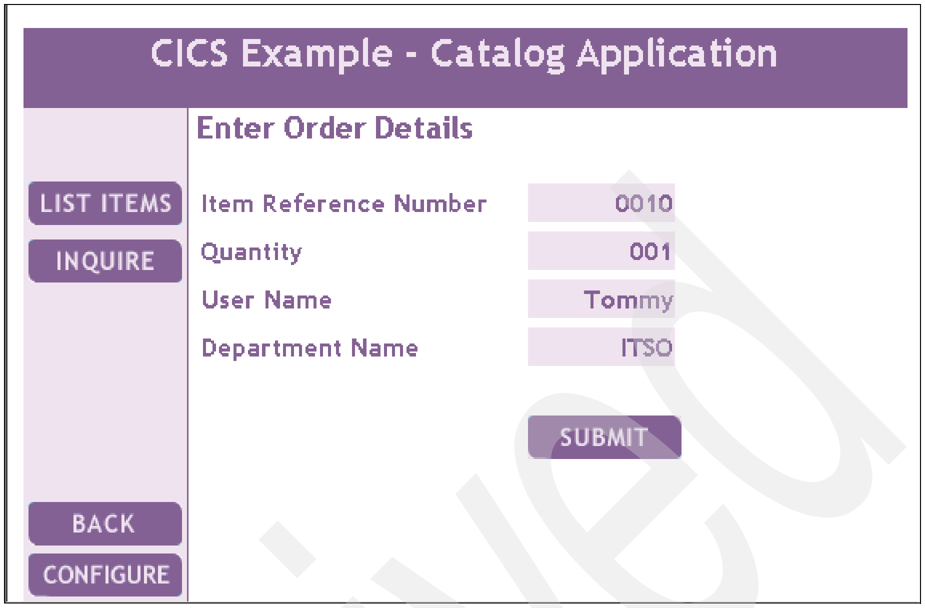
Running the Web client application

We started a Web browser session and entered the URL:

`http://cam21-pc11:9080/CatalogWeb/Welcome.jsp`

The window in Figure 3-10 on page 91 was presented, and we clicked **ORDER ITEM**.

The window in Figure 3-19 was presented. We entered values for `User Name` and `Department Name` and clicked **SUBMIT**.


CICS Example - Catalog Application

Enter Order Details

Item Reference Number 0010

Quantity 001

User Name Tommy

Department Name ITSO

Figure 3-19 CICS - Catalog application order window

We received a message back saying "`ORDER SUCCESFULLY PLACED`." We also noted that the service provider application wrote a message confirming the order to the WebSphere Application Server for z/OS SYSPRINT DD (Example 3-15).

Example 3-15 ExampleAppDispatchOrder output from WebSphere Application Server

```
DispatchOrderSoapBindingImpl: dispatchOrder(): ItemRef=10 Quantity=1
CustomerName=Tommy    Dept=ITSO
```

3.4 Configuring for high availability

After you have successfully configured and tested your CICS Web service configuration, you should consider how you can clone the CICS regions in order to improve scalability and availability.

The principal areas for consideration are:

- How to load balance TCP/IP requests across multiple CICS listener regions
- How to load balance Web service requests dynamically across multiple CICS AORs

3.4.1 TCP/IP load balancing

CICS is designed to work with Sysplex Distributor. Sysplex Distributor is an integral part of z/OS Communications Manager, which offers the ability to load balance incoming socket open requests across different address spaces running on different IP stacks (usually on different LPARs). The routing decision is based on real-time socket status and z/OS Quality of Service (QoS) criteria. This provides the benefit of balancing work across different MVS™ images, providing enhanced scalability and failover in a z/OS Parallel Sysplex®.

3.4.2 High availability configuration

Figure 3-20 shows the recommended high availability configuration. CICSPlex SM provides a dynamic routing program that supports the dynamic routing of transactions. This provides the ability for applications invoked by Web service requests to be dynamically routed across a CICSplex.

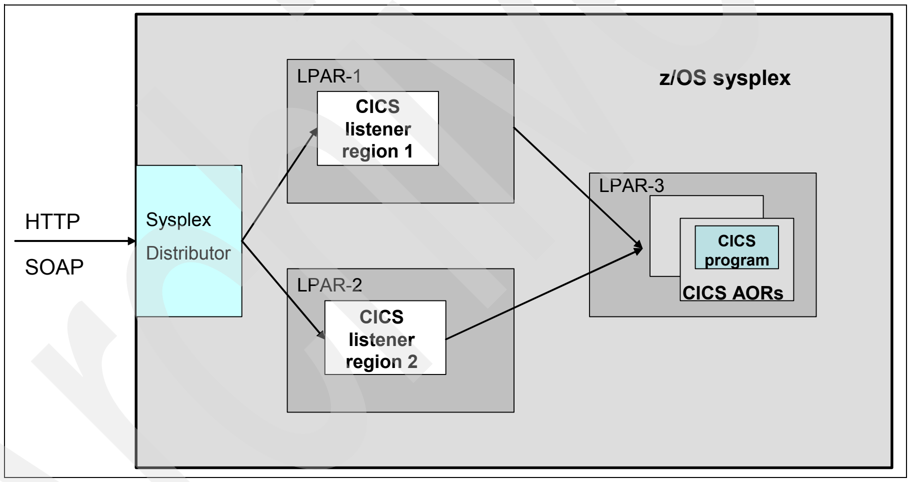


Figure 3-20 High scalability and availability configuration

3.4.3 Routing inbound Web service requests

Inbound Web service requests can be routed to a different CICS region than the one that receives the request using one of two routing models:

- ▶ Distributed routing
- ▶ Dynamic program routing

The distributed routing model

The transaction that runs the target application program is eligible for routing when one of the following is true:

- ► The content of the DFHWS-USERID container has been changed by a program in the pipeline.
- ► The content of the DFHWS-TRANID container has been changed by a program in the pipeline.
- ► The transaction is defined as DYNAMIC or with REMOTESYSTEM(sysid).

Figure 3-21 shows how the distributed routing model can be used to route requests for the ORDR transaction. The routing can be controlled by the routing program specified in the DSRTPGM system initialization parameter. CICSPlex SM can be used to balance the routing requests across multiple AORs.

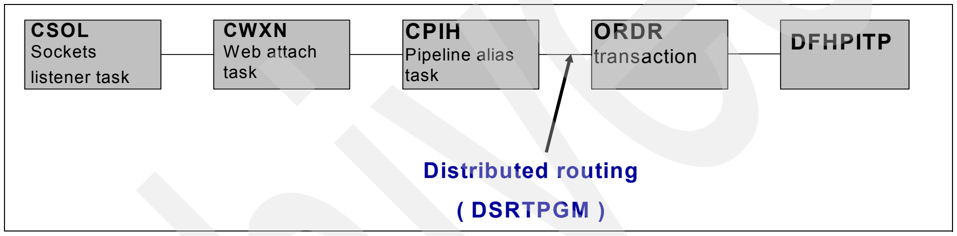


Figure 3-21 Web service provider - Distributed routing

Pipeline configuration

Special considerations have to be made when configuring a pipeline to be used in a distributed routing environment. Table 3-4 shows the resource definition requirements for both the listener region and AOR, and whether each resource definition can be shared between the regions.

Table 3-4 Pipeline resource definitions in dynamic routing configuration

Resource	Listener region	AOR
TCPIPSERVICE	Required	Not required
PIPELINE	Required, shared	Required, shared
WEBSERVICE	Automatically installed from PIPELINE, shared	Required, automatically installed from PIPELINE, shared
Pipeline configuration file	Required, shared	Required, shared
TRANSACTION definition	DYNAMIC(YES)	DYNAMIC(NO)

The dynamic routing model

An alternative way to dynamically route a Web service request, is at the point where CICS links to the user program, in our case DFH0XCMN. At this point (Figure 3-22) the request is routed using the dynamic routing model. In this scenario, the routing can be controlled by the program specified in the DTRPGM system initialization parameter. CICSPlex SM can be used to balance the program link requests across multiple AORs.

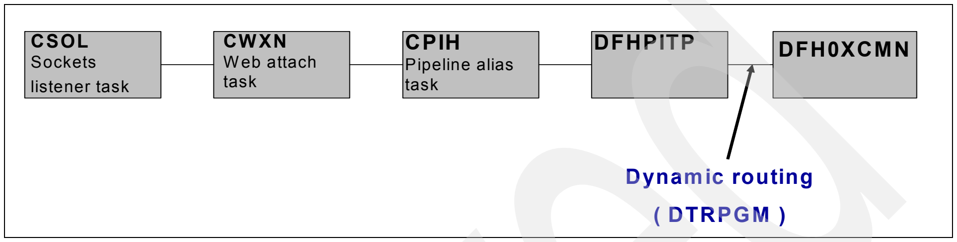


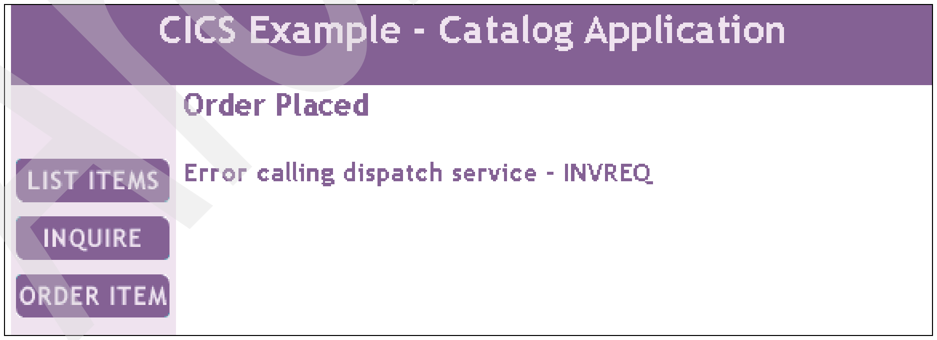
Figure 3-22 Web service provider - Dynamic routing

3.5 Problem determination

In this section we highlight different ways of diagnosing problems that occur when an incorrect URI is used in a Web services call.

3.5.1 Error calling dispatch service - INVREQ

During testing of the CICS service requester scenario we experienced the problem shown in Figure 3-23.

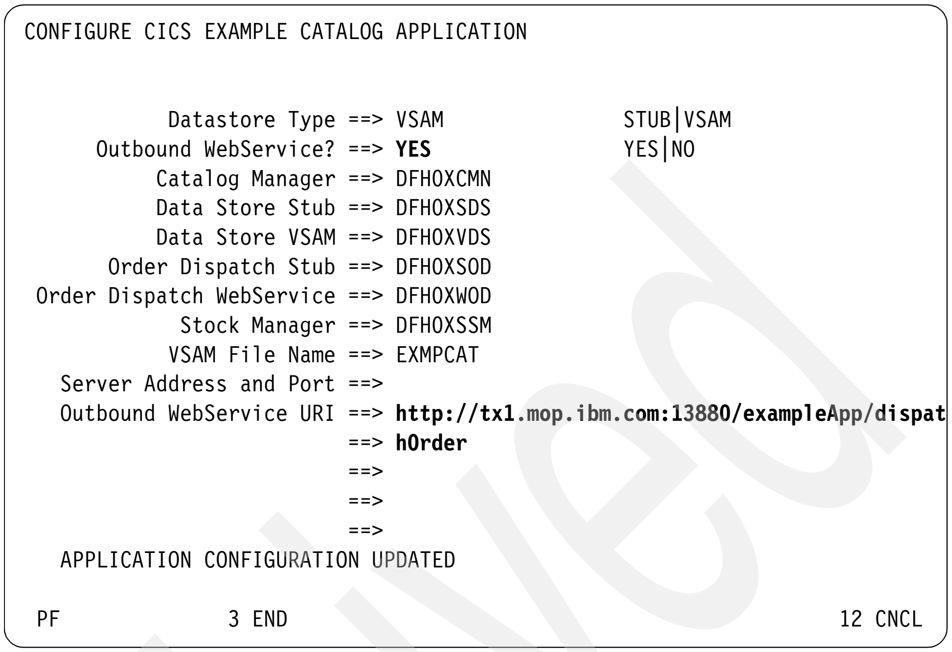

CICS Example - Catalog Application

Order Placed

Error calling dispatch service - INVREQ

Figure 3-23 CICS - Catalog application INVREQ

Figure 3-24 shows the catalog manager configuration (including the URI for the outbound Web service).



```
CONFIGURE CICS EXAMPLE CATALOG APPLICATION


            Datastore Type ==> VSAM               STUB|VSAM
      Outbound WebService? ==> YES                YES|NO
           Catalog Manager ==> DFH0XCMN
            Data Store Stub ==> DFH0XSDS
            Data Store VSAM ==> DFH0XVDS
        Order Dispatch Stub ==> DFH0XSOD
  Order Dispatch WebService ==> DFH0XWOD
              Stock Manager ==> DFH0XSSM
             VSAM File Name ==> EXMPCAT
    Server Address and Port ==>
    Outbound WebService URI ==> http://tx1.mop.ibm.com:13880/exampleApp/dispat
                            ==> hOrder
                            ==>
                            ==>
                            ==>
    APPLICATION CONFIGURATION UPDATED

 PF             3 END                                                 12 CNCL
```

Figure 3-24 CICS - Catalog application ECFG screen

CICS trace

To help diagnose the problem we turned on CICS auxiliary trace using the CETR transaction. In trace entry 002142 of Example 3-16 we see the error returned from WebSphere Application Server:

“`Error 404: SRVE0190E: File not found /services/dispatchOrder.`”

Example 3-16 Sample DispatchOrder CICS trace

```
PI 0A31 PIIS EVENT - REQUEST_CNT - TASK-00182 KE_NUM-008C TCB-L8003/00ADC0A8 RET-9753E4F2
                                TIME-12:13:58.4376239562 INTERVAL-00.0000008750        =002137=
1-0000  3C534F41 502D454E 563A456E 76656C6F  70652078 6D6C6E73 3A534F41 502D454E
                                                            *<SOAP-ENV:Envelope xmlns:SOAP-EN*
  0020  563D2268 7474703A 2F2F7363 68656D61  732E786D 6C736F61 702E6F72 672F736F
                                                            *V="http://schemas.xmlsoap.org/so*
  0040  61702F65 6E76656C 6F70652F 22203E3C  534F4150 2D454E56 3A426F64 793E3C64
                                                            *ap/envelope/" ><SOAP-ENV:Body><d*
  0060  69737061 7463684F 72646572 52657175  65737420 786D6C6E 733D2268 7474703A
                                                            *ispatchOrderRequest xmlns="http:*
  0080  2F2F7777 772E6578 616D706C 65417070  2E646973 70617463 684F7264 65722E52
                                                            *//www.exampleApp.dispatchOrder.R*
  00A0  65717565 73742E63 6F6D223E 3C697465  6D526566 6572656E 63654E75 6D626572
                                                            *equest.com"><itemReferenceNumber*
  00C0  3E31303C 2F697465 6D526566 6572656E  63654E75 6D626572 3E3C7175 616E7469
                                                          *>10</itemReferenceNumber><quanti*
  00E0  74795265 71756972 65643E31 3C2F7175  616E7469 74795265 71756972 65643E3C
                                                          *tyRequired>1</quantityRequired><*
  0100  63757374 6F6D6572 49643E54 6F6D6D79  2020203C 2F637573 746F6D65 7249643E
                                                          *customerId>Tommy   </customerId>*
  0120  3C636861 72676544 65706172 746D656E  743E4954 534F2020 20203C2F 63686172
                                                          *<chargeDepartment>ITSO    </char*
  0140  67654465 70617274 6D656E74 3E3C2F64  69737061 7463684F 72646572 52657175
                                                          *geDepartment></dispatchOrderRequ*
  0160  6573743E 3C2F534F 41502D45 4E563A42  6F64793E 3C2F534F 41502D45 4E563A45
                                                          *est></SOAP-ENV:Body></SOAP-ENV:E*
  0180  6E76656C 6F70653E                                 *nvelope>                        *

PI 0A32 PIIS EVENT - RESPONSE_CNT - TASK-00182 KE_NUM-008C TCB-L8003/00ADC0A8 RET-9753E4F2
                                TIME-12:13:58.4376326594 INTERVAL-00.0000006562        =002142=
1-0000  4572726F 72203430 343A2053 52564530  31393045 3A204669 6C65206E 6F742066
                                                          *Error 404: SRVE0190E: File not f*
  0020  6F756E64 3A202F73 65727669 6365732F  64697370 61746368 4F726465 720A
                                                          *ound: /services/dispatchOrder.  *
```

Using SNIFFER

The user-written SNIFFER handler program is a simple program that browses through the containers available in the pipeline. It can be used as a message handler program or a header processing program.

It browses the containers by issuing a STARTBROWSE CONTAINER command followed by GETNEXT CONTAINER until all containers have been browsed. It then issues an ENDBROWSE CONTAINER command. For each container browsed, it writes the container name and contents to the CICS transient data queue CESE.

We added the SNIFFER message handler program to the requester pipeline EXPIPE02. Example 3-17 shows the pipeline configuration file with SNIFFER added as a message handler program.

Example 3-17 Pipeline configuration file with SNIFFER

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<requester_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
        requester.xsd ">
  <service>
    <service_handler_list>
      <cics_soap_1.1_handler/>
    </service_handler_list>
  </service>
  <default_transport_handler_list>
   <handler>
    <program>SNIFFER</program>
    <handler_parameter_list/>
   </handler>
  </default_transport_handler_list>
</requester_pipeline>
```

The full program is shown in Appendix A.3, “Sample handler program - SNIFFER” on page 539. Example 3-18 shows the containers in the requester pipeline as listed by SNIFFER. The container of interest is DFHWS-URI:

`http://tx1.mop.ibm.com:13880/exampleApp/dispatchOrder`

Example 3-18 sample SNIFFER output

```
SNIFFER :  *** Start ***
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFHFUNCTION
SNIFFER :  Content length    : 00000016
SNIFFER :  Container content: SEND-REQUEST
SNIFFER :  Containers on channel: List starts.
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFHHEADER
SNIFFER :  Content length    : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFHWS-XMLNS
SNIFFER :  Content length    : 00000059
SNIFFER :  Container content: xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
                              envelope/"
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFHWS-SOAPLEVEL
SNIFFER :  Content length    : 00000004
SNIFFER :  Container content:
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFH-HANDLERPLIST
SNIFFER :  Content length    : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFHRESPONSE
SNIFFER :  Content length    : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name    : DFHFUNCTION
SNIFFER :  Content length    : 00000016
SNIFFER :  Container content: SEND-REQUEST
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFH-SERVICEPLIST
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-USERID
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: CICSUSER
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-TRANID
SNIFFER :  Content length   : 00000004
SNIFFER :  Container content: ORDR
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHREQUEST
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-BODY
SNIFFER :  Content length   : 00000293
SNIFFER :  Container content: <SOAP-ENV:Body><dispatchOrderRequest
xmlns="http://www.exampleApp.dispatchOrder.Request.com"><itemRefe
renceNumber>10</itemReferenceNumber><quantityRequired>1</quantityRequired><cust
omerId>Tommy   </customerId><chargeDepartment>ITSO
 </chargeDepartment></dispatchOrderRequest></SOAP-ENV:Body>
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-URI
SNIFFER :  Content length   : 00000255
SNIFFER :  Container content:
           http://tx1.mop.ibm.com:13880/exampleApp/dispatchOrder
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-SOAPACTION
SNIFFER :  Content length   : 00000002
SNIFFER :  Container content: ""
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-OPERATION
SNIFFER :  Content length   : 00000255
SNIFFER :  Container content: dispatchOrder
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-PIPELINE
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: EXPIPE02
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-DATA
SNIFFER :  Content length   : 00000023
SNIFFER :  Container content: 0010001Tommy   ITSO
SNIFFER :  Containers on channel: List ends
SNIFFER :   in a SOAP header processing program.....
SNIFFER :  **** End ****
```

Checking the SOAP address in the WSDL

Next we checked the SOAP address in the WSDL file of the deployed EAR file. In the WebSphere administrative console we clicked **Applications** → **Enterprise applications** → **dispatchOrder** → **Publish WSDL file** → **DispatchOrder_WSDLFiles.zip** and saved the file to disk. We unzipped the file into the directory structure shown in Figure 3-25.

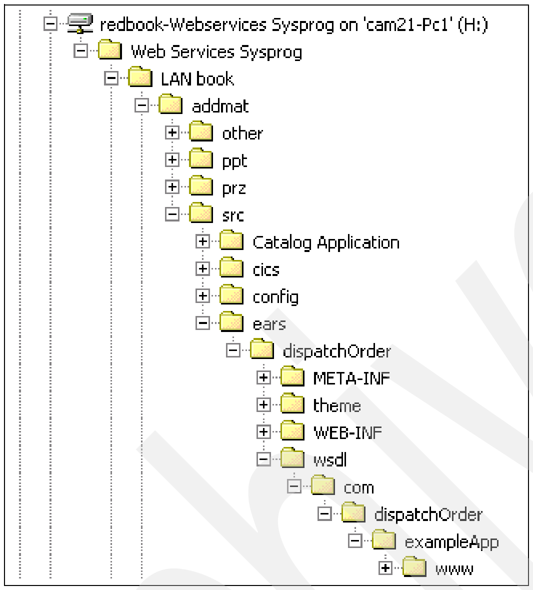


Figure 3-25 ExampleAppDispatchOrder path

In the WSDL file dispatchOrder.wsdl (Example 3-19) we noted the URI of the Web service as found in the `soap: address location`.

Example 3-19 dispatchOrder sample wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://www.exampleApp.dispatchOrder.com"
             xmlns:tns="http://www.exampleApp.dispatchOrder.com"
             xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:resns="http://www.exampleApp.dispatchOrder.Response.com"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema attributeFormDefault="qualified"
             elementFormDefault="qualified"
             targetNamespace="http://www.exampleApp.dispatchOrder.Request.com"
             xmlns:reqns="http://www.exampleApp.dispatchOrder.Request.com"
             xmlns:tns="http://www.exampleApp.dispatchOrder.Request.com"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="dispatchOrderRequest" nillable="false">
.
. Part of wsdl not included
.
  <binding name="dispatchOrderSoapBinding" type="tns:dispatchOrderPort">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/
    soap/http"/>
    <operation name="dispatchOrder">
      <soap:operation soapAction="" style="document"/>
      <input name="DFHOXODSRequest">
        <soap:body parts="RequestPart" use="literal"/>
      </input>
      <output name="DFHOXODSResponse">
        <soap:body parts="ResponsePart" use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="dispatchOrderService">
    <port name="dispatchOrderPort" binding="tns:dispatchOrderSoapBinding">
      <soap:address location="http://tx1.mop.ibm.com:13880/
      exampleApp/services/dispatchOrderPort"/>
    </port>
  </service>
</definitions>
```

The error shown in Figure 3-24 on page 105 was caused by specifying an incorrect URI for the dispatchOrder Web service. In the catalog manager configuration (Figure 3-24) we specified the URI /exampleApp/dispatchOrderPort for the outbound Web service. This is the correct URI for the dispatchOrder service provider deployed inside CICS, but a URI of /exampleApp/services/dispatchOrderPort is the correct URI for our dispatchOrder service deployed in WebSphere Application Server.

4

Web services using WebSphere MQ

In this chapter we describe how we configured our test CICS environment to support Web services using WebSphere MQ as the transport mechanism.

4.1 Preparation

After outlining our test configuration (Figure 4-1), we show how we enabled WebSphere MQ (WMQ) support in a CICS region. We next explain how we configured CICS as a service provider for incoming WMQ message requests. Finally, we show how we configured a CICS region to act as a service requester, sending requests in WMQ messages.

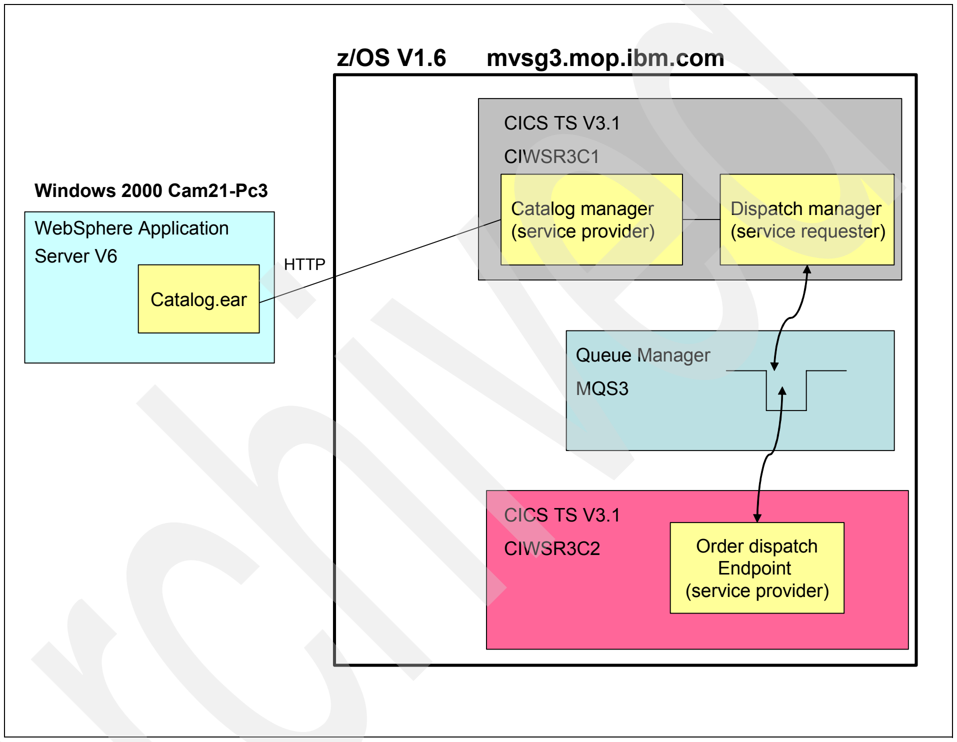

z/OS V1.6
mvsg3.mop.ibm.com
CICS TS V3.1
CIWSR3C1
Catalog manager (service provider)
Dispatch manager (service requester)
Windows 2000 Cam21-Pc3
WebSphere Application Server V6
Catalog.ear
HTTP
Queue Manager
MQS3
CICS TS V3.1
CIWSR3C2
Order dispatch Endpoint (service provider)

Figure 4-1 Software components: Web services using HTTP and WMQ

Note: We used a CICS-to-CICS scenario in order to demonstrate how WMQ can be used with a CICS service provider and a service requester. You can also use WMQ to pass SOAP messages between WebSphere Application Server and CICS.

We do not provide details on how to install the software components, and we also assume the reader has a working knowledge of CICS and WebSphere MQ.

4.1.1 Software checklist

For the configuration shown in Figure 4-1 we used the levels of software shown in Table 4-1.

Table 4-1 Software used in the WebSphere MQ scenarios

Windows	z/OS
Windows 2000 SP4	z/OS V1.6
IBM WebSphere Application Server - ND V6.0.2.0	CICS Transaction Server V3.1
Internet Explorer V6.0	
	WebSphere MQ V5R3M1
Our J2EE applications ► Catalog.ear Catalog manager service requester application	Our user-supplied CICS programs ► CIWSMSGH (message handler program)

Tip: If you use WMQ to pass SOAP messages between WebSphere Application Server and CICS, you should install the fix for APAR PK20393.

4.1.2 Definition checklist

The z/OS definitions we used to configure the scenarios are listed in Table 4-2.

Table 4-2 Definition checklist

Value	CICS region 1	CICS region 2
IP name	mvsg3.mop.ibm.com	mvsg3.mop.ibm.com
IP address	9.100.193.167	9.100.193.167
TCP/IP port	13301	
Job name	CIWSR3C1	CIWSR3C2
APPLID	A6POR3C1	A6POR3C2
TCPIPSERVICE	R3C1	
Provider PIPELINE	EXPIPE01	EXPIPEP03
Requester PIPELINE	EXPIPE02	
WMQ queue manager	MQS3	MQS3

The WMQ definitions we used to configure the scenarios are listed in Table 4-3.

Table 4-3 WMQ definition checklistt

Value	Queue manager MQS3
Queues	V3G3.R3C2.PIPE3.REQUEST V3G3.R3C2.PIPE3.RESPONSE
Process	VSG3.R3C2.PROCESS

4.2 WebSphere MQ configuration

We completed the following tasks in order to enable WMQ support in the two CICS regions CIWSR3C1 and CIWSR3C2:

- Adding WMQ support to CICS
- Defining the queues
- Defining the trigger process

4.2.1 Adding WebSphere MQ support to CICS

We updated the CICS startup procedure for each CICS region by adding the WMQ libraries to the STEPLIB and DFHRPL as shown in Example 4-1.

Example 4-1 CICS startup JCL

```
//STEPLIB  DD DSN=CICSTS31.CICS.SDFHAUTH,DISP=SHR
//         DD DSN=CICSTS31.CICS.SDFJAUTH,DISP=SHR
//         DD DSN=MQM.SCSQANLE,DISP=SHR
//         DD DSN=MQM.SCSQAUTH,DISP=SHR
//DFHRPL   DD DSN=CIWS.CICS.USERLOAD,DISP=SHR
//         DD DSN=CEE.SCEECICS,DISP=SHR
//         DD DSN=CEE.SCEERUN,DISP=SHR
//         DD DSN=CICSTS31.CICS.SDFHLOAD,DISP=SHR
//         DD DSN=MQM.SCSQLOAD,DISP=SHR
//         DD DSN=MQM.SCSQANLE,DISP=SHR
//         DD DSN=MQM.SCSQCICS,DISP=SHR
//         DD DSN=MQM.SCSQAUTH,DISP=SHR
```

- We updated the SIT parameters on CICS region CIWSR3C1:
  - MQCONN=YES
  - INITPARM=(CSQCPARM='SN=MQS3,TN=1,IQ=VSG3.R3C1.INITQ')

- We updated the SIT parameters on CICS region CIWSR3C2:
  - MQCONN=YES
  - INITPARM=(CSQCPARM='SN=MQS3,TN=1,IQ=VSG3.R3C2.INITQ')
- We added the WMQ RDO groups to the startup LIST on CICS region 1 using the following commands, and then we restarted the CICS region:

  ```
  CEDA ADD GROUP(CSQCAT1) TO LIST(LISTR3C1)
  CEDA ADD GROUP(CSQKDQ1) TO LIST(LISTR3C1
  ```
- We added the WMQ RDO groups to the startup LIST on CICS region 2 using the following commands, and then we restarted the CICS region:

  ```
  CEDA ADD GROUP(CSQCAT1) TO LIST(LISTR3C2)
  CEDA ADD GROUP(CSQKDQ1) TO LIST(LISTR3C2)
  ```

4.2.2 Defining the queues

Example 4-2 shows the JCL that we used to define two QUEUE resources of type `local` in the MQS3 queue manager region. One queue is for incoming requests and the other is for responses.

Example 4-2 JCL for defining the queues

```
//CHIQUEUE JOB 1,CIWS,TIME=1440,NOTIFY=&SYSUID,REGION=4M,
//             CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//*
//CSQUTIL  EXEC PGM=CSQUTIL,PARM='MQS3'
//STEPLIB  DD DSN=MQM.SCSQLOAD,DISP=SHR
//         DD DSN=MQM.SCSQANLE,DISP=SHR
//         DD DSN=MQM.SCSQAUTH,DISP=SHR
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSIN    DD *
COMMAND DDNAME(CMDINP)
/*
//CMDINP   DD *
*
DEFINE QLOCAL(VSG3.R3C2.PIPE3.REQUEST) -
DESCR('QUEUE SOAP INCOMING REQUEST')  -
PROCESS(VSG3.R3C2.PROCESS) -
TRIGGER  -
TRIGTYPE(FIRST) -
INITQ('VSG3.R3C2.INITQ') -
*
DEFINE QLOCAL(VSG3.R3C2.PIPE3.RESPONSE) -
```

```
DESCR('QUEUE SOAP RESPONSE')  -
*
/*
```

The INITQ VSG3.R3C2.INITQ is the same name as specified in the INITPARM parameter for the CIWSR3C2 region.

4.2.3 Defining the trigger process

Example 4-3 shows the command that we used to define a PROCESS.

Example 4-3 WMQ definition of PROCESS

```
DEFINE PROCESS(VSG3.R3C2.PROCESS)
APPLTYPE(CICS)
APPLICID(CPIL)
```

The process name is the same name specified when defining the request queue VSG3.R3C2.PIPE3.REQUEST in Example 4-2. APPLICID is specified as CPIL (the SOAP MQ inbound listener transaction) which means that this transaction will be started in CICS when a service request arrives. CPIL matches an incoming URI to a URIMAP definition in order to match the URI to a WEBSERVICE, and attaches the CPIQ transaction (the SOAP MQ inbound router transaction).

4.3 Configuring CICS as a service provider using WMQ

In this section we discuss how we configured the CICS region CIWSR3C2 as a service provider using WMQ (Figure 4-2).

The catalog manager application provides a dispatch manager program that provides an interface for dispatching an order to an external partner. In this scenario, we configured a remote order dispatch endpoint, such that the dispatch request is sent to a CICS service provider program DFH0XODE using WMQ.

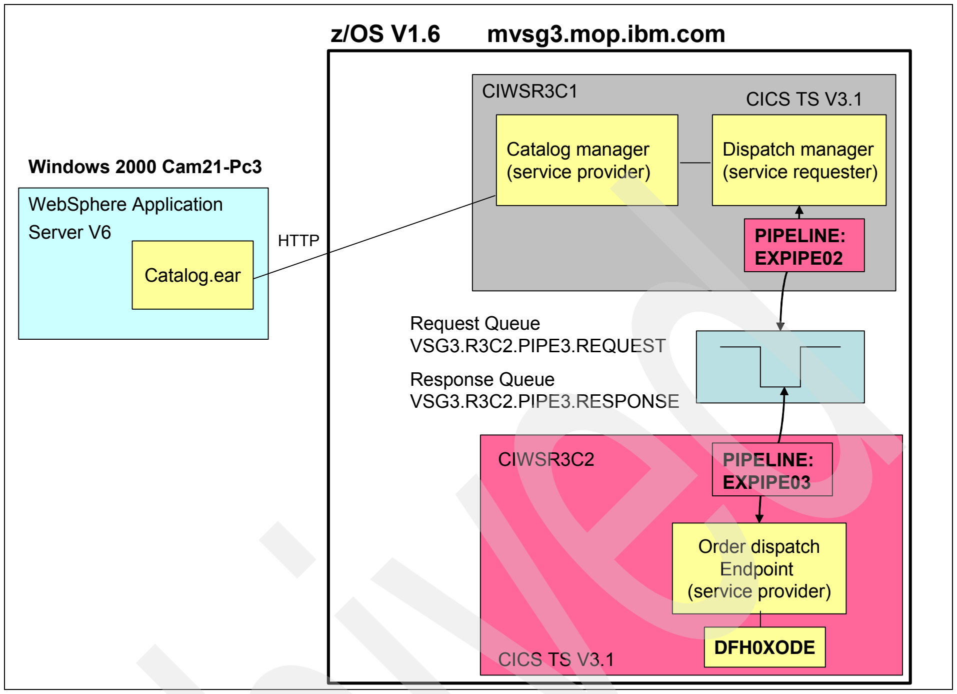

z/OS V1.6
mvsg3.mop.ibm.com
Windows 2000 Cam21-Pc3
WebSphere Application Server V6
Catalog.ear
HTTP
CIWSR3C1
CICS TS V3.1
Catalog manager (service provider)
Dispatch manager (service requester)
PIPELINE: EXPIPE02
Request Queue VSG3.R3C2.PIPE3.REQUEST
Response Queue VSG3.R3C2.PIPE3.RESPONSE
CIWSR3C2
PIPELINE: EXPIPE03
Order dispatch Endpoint (service provider)
DFH0XODE
CICS TS V3.1

Figure 4-2 CICS as a service provider using WMQ

Note: On our system the two CICS regions are actually running on the same z/OS image. In practice they would normally be running on two different systems.

4.3.1 Configuring the service provider pipeline

To enable CICS to receive Web service requests using WMQ we performed the following tasks:

- Creating the HFS directories
- Configuring the pipeline configuration file
- Updating the message handler program CIWSMSGH
- Creating and installing the PIPELINE resource definition

Creating the HFS directories

We created the HFS directories shown in Example 4-4.

Example 4-4 HFS directories used in the PIPELINE definition

```
/CIWS/R3C2/config
/CIWS/R3C2/shelf
/CIWS/R3C2/wsbind/provider
```

We copied the dispatchOrderEndpoint.wsbind file from the CICS-supplied directory /usr/lpp/cicsts/cicsts31/samples/webservices/wsbind/provider to our wsbind directory /CIWS/R3C2/wsbind/provider.

The config directory is used for the pipeline configuration file that we create in a subsequent step, while CICS uses the shelf directory to store installed wsbind files.

We gave the CICS region user ID read permission to the config and wsbind directories, and update permission to the shelf directory.

Configuring the pipeline configuration file

In order to add the CIWSMSGH message handler program to the pipeline for the service provider, we used the same pipeline configuration file that is described in "Customizing the pipeline configuration file" on page 80.

Updating the message handler program

The default transaction ID assigned to inbound WMQ Web services transactions is CPIQ. We wanted to assign a different transaction ID to the dispatch request. To do this, we updated the CIWSMSGH message handler program that we first introduced in "Writing the message handler program" on page 80. We replaced the transaction ID in the DFHWS-TRANID container with an ID based on the service requester (which can be found in the DFHWS-WEBSERVICE container).

Table 4-4 shows the relationship between the transaction ID and the Web service request.

Table 4-4 Transaction ID to Web services name relationship

Transaction ID	**Web services request**
DISP	dispatchOrderEndPoint

Before we activated the message handler program we needed to create the new TRANSACTION definition with the same characteristics as the CICS-supplied definition for CPIQ.

We used the following CEDA COPY command to create the transaction definition:

```
CEDA COPY TRANSACTION(CPIQ) GROUP(DFHPIPE) TO(R3C2) AS(DISP)
```

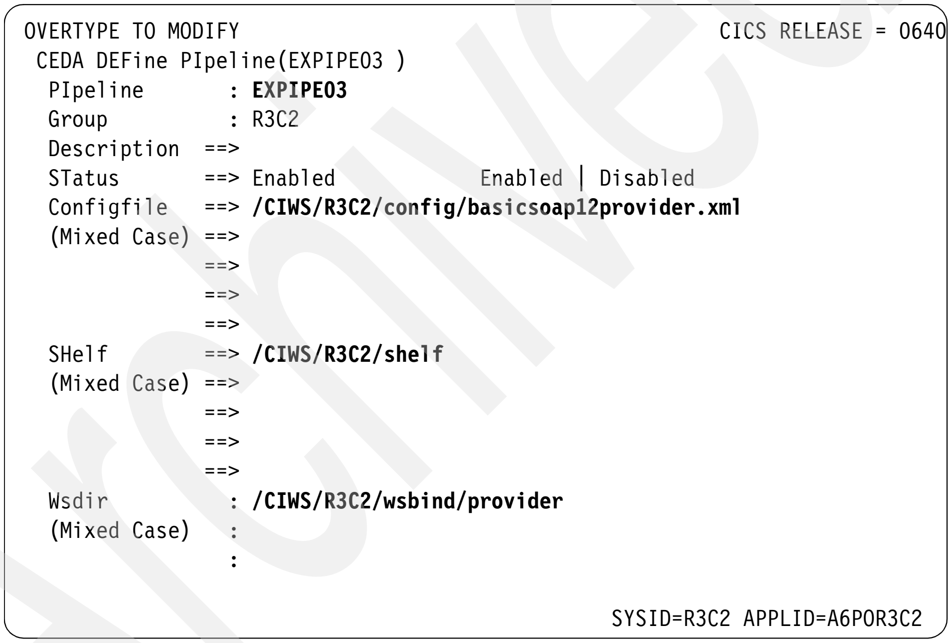
Then we installed the definition.

Creating the PIPELINE resource definition

We then defined the PIPELINE for the CICS service provider using the following CICS command:

```
CEDA DEFINE PIPELINE(EXPIPE03) GROUP(R3C2)
```

We defined EXPIPE03 as shown in Figure 4-3.



```
OVERTYPE TO MODIFY                                          CICS RELEASE = 0640
 CEDA DEFine PIpeline(EXPIPE03 )
  PIpeline       : EXPIPE03
  Group          : R3C2
  Description  ==>
  STatus       ==> Enabled            Enabled | Disabled
  Configfile   ==> /CIWS/R3C2/config/basicsoap12provider.xml
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  SHelf        ==> /CIWS/R3C2/shelf
  (Mixed Case) ==>
               ==>
               ==>
               ==>
  Wsdir          : /CIWS/R3C2/wsbind/provider
  (Mixed Case)   :
                 :

                                                 SYSID=R3C2 APPLID=A6POR3C2
```

Figure 4-3 CEDA DEFINE PIPELINE EXPIPE03

- We set CONFIGFILE to the name of our pipeline configuration file.

  /CIWS/R3C2/config/basicsoap12provider.xml
- We set SHELF to the name of the shelf directory.

  /CIWS/R3C2/shelf

- We set WSDIR to the Web service binding directory that contains the wsbind files for the sample application.

  /CIWS/R3C2/wsbind/provider

Installing the PIPELINE resource

We used CEDA to install the PIPELINE definition. When the PIPELINE is installed, CICS scans the wsdir directory and dynamically creates a WEBSERVICE and a URIMAP definition for each wsbind file that it finds.

Figure 4-4 shows a CEMT INQUIRE PIPELINE for EXPIPE03.

```
INQUIRE PIPELINE
RESULT - OVERTYPE TO MODIFY
  Pipeline(EXPIPE03)
  Enablestatus( Enabled )
  Configfile(/CIWS/R3C2/config/basicsoap12provider.xml)
  Shelf(/CIWS/R3C2/shelf/)
  Wsdir(/CIWS/R3C2/wsbind/provider/)

                                          SYSID=R3C2 APPLID=A6POR3C2
```

Figure 4-4 CEMT INQUIRE PIPELINE - EXPIPE03

After installing the pipeline, we used the CEMT INQUIRE WEBSERVICE command to view the dynamically installed Web service. In Figure 4-5, we noted that the name of the service (namely, dispatchOrderEndpoint) is taken from the wsbind file.

```
INQUIRE WEBSERVICE
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Webs(dispatchOrderEndpoint           ) Pip(EXPIPE03)
    Ins Uri(£439310 ) Pro(DFHOXODE) Com                       Dat(20051209)


                                          SYSID=R3C2 APPLID=A6POR3C2
```

Figure 4-5 CEMT INQUIRE WEBSERVICE

Figure 4-6 shows the dynamically installed URIMAP that is associated with the Web service.

```
INQUIRE URIMAP
RESULT - OVERTYPE TO MODIFY
  Urimap(£439310)
  Usage(Pipe)
  Enablestatus( Enabled )
  Analyzerstat(Noanalyzer)
  Scheme(Http)
  Redirecttype( None )
  Tcpipservice()
  Host(*)
  Path(/exampleApp/dispatchOrder)
  Transaction(CPIH)
  Converter()
  Program()
  Pipeline(EXPIPE03)
  Webservice(dispatchOrderEndpoint)
  Userid()
  Certificate()
  Ciphers()
  Templatename()

                                                       SYSID=R3C2 APPLID=A6POR3C2
```

Figure 4-6 CEMT INQUIRE URIMAP

4.4 Configuring CICS as service requester using WMQ

In this section we explain how we configured CICS region CIWSR3C1 as a service requester using WMQ. In this particular scenario CICS region CIWSR3C1 is both a service provider for the catalog manager application (as detailed in “Configuring CICS as a service provider” on page 76) and a service requester of the Dispatch manager application.

Figure 4-7 on page 122 shows the Dispatch manager order dispatch program DFH0XWOD, which issues an EXEC CICS INVOKE WEBSERVICE command to make an outbound Web service call to the order dispatcher running in CICS region CIWSR3C2.

When communication between the service requester and service provider uses WMQ, the URI of the target is in a form that identifies the target as a queue, and includes information to specify how the request and response should be handled by WMQ.

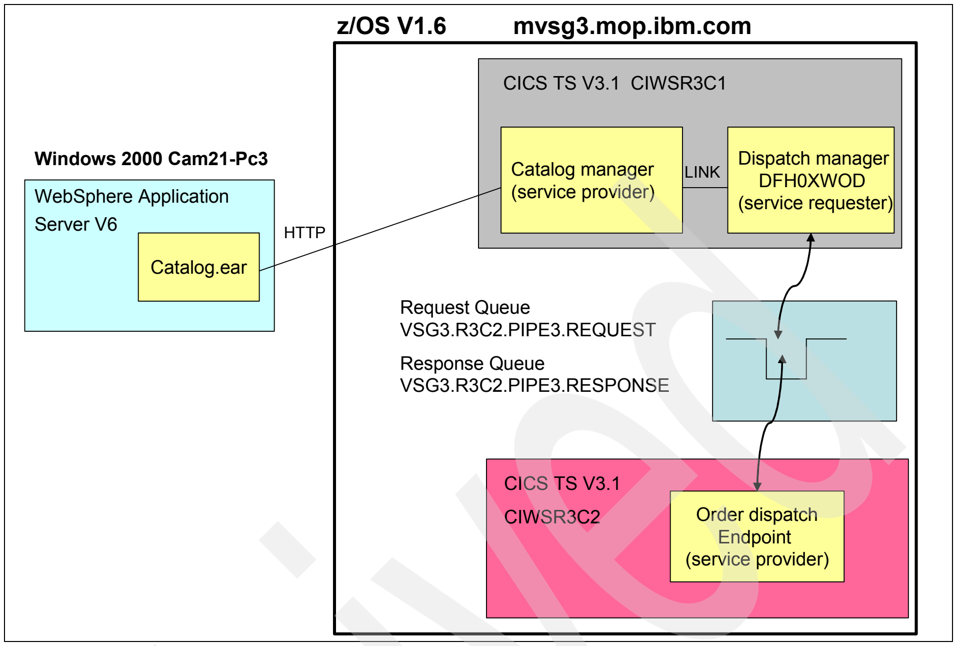
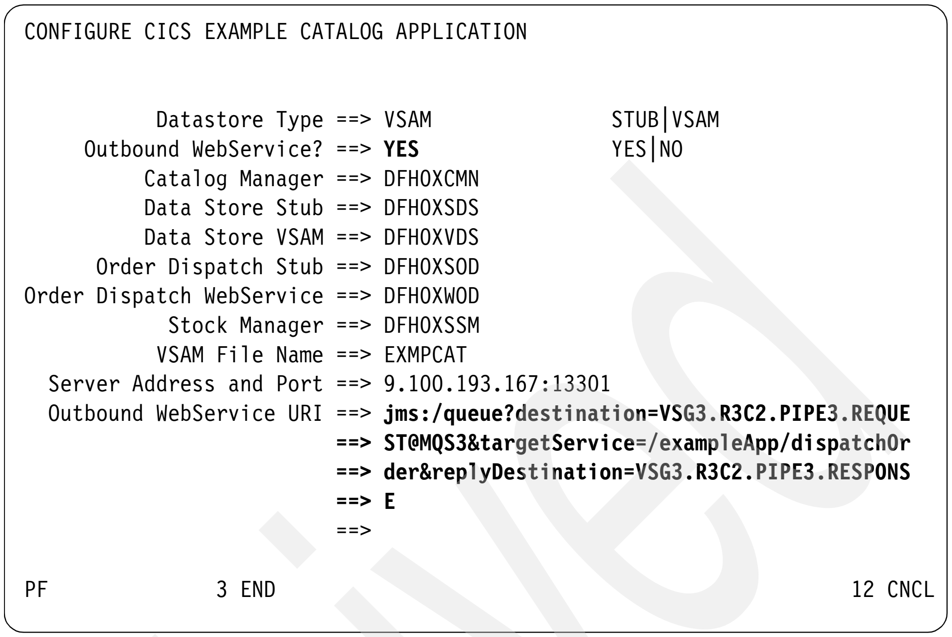

z/OS V1.6
mvsg3.mop.ibm.com
CICS TS V3.1 CIWSR3C1
Windows 2000 Cam21-Pc3
Catalog manager (service provider)
LINK
Dispatch manager DFH0XWOD (service requester)
WebSphere Application Server V6
HTTP
Catalog.ear
Request Queue VSG3.R3C2.PIPE3.REQUEST
Response Queue VSG3.R3C2.PIPE3.RESPONSE
CICS TS V3.1
CIWSR3C2
Order dispatch Endpoint (service provider)

Figure 4-7 CICS as a service provider using WMQ

4.4.1 Configuring the Catalog application

We configured the catalog application to activate the outbound Web services feature using WMQ. We used the CICS-supplied catalog manager configuration transaction ECFG to configure the example application (Figure 4-8).



```
CONFIGURE CICS EXAMPLE CATALOG APPLICATION


           Datastore Type ==> VSAM              STUB|VSAM
     Outbound WebService? ==> YES               YES|NO
          Catalog Manager ==> DFH0XCMN
          Data Store Stub ==> DFH0XSDS
          Data Store VSAM ==> DFH0XVDS
      Order Dispatch Stub ==> DFH0XSOD
Order Dispatch WebService ==> DFH0XWOD
            Stock Manager ==> DFH0XSSM
           VSAM File Name ==> EXMPCAT
  Server Address and Port ==> 9.100.193.167:13301
  Outbound WebService URI ==> jms:/queue?destination=VSG3.R3C2.PIPE3.REQUE
                          ==> ST@MQS3&targetService=/exampleApp/dispatchOr
                          ==> der&replyDestination=VSG3.R3C2.PIPE3.RESPONS
                          ==> E
                          ==>

PF             3 END                                             12 CNCL
```

Figure 4-8 Catalog application configuration screen for WMQ

We changed `Outbound WebService` to **`Yes`**, and entered the URI of the service provider for our outbound service request:

```
jms:/queue?destination=VSG3.R3C2.PIPE3.REQUEST@MQS3&targetService=/e
xampleApp/dispatchOrder&replyDestination=VSG3.R3C2.PIPE3.RESPONSE
```

We then pressed **PF3** to save the configuration.

Tip: You must use the ampersand (`&`) character as a separator between options; otherwise CICS does not recognize the parameters.

The Dispatch manager module DFH0XWOD uses the value of the **Outbound WebService URI** parameter as the URI of the Web service to be invoked when it invokes the dispatch service with an EXEC CICS INVOKE WEBSERVICE command.

The main parameters for the Outbound WebService URI are as follows:

- **jms:/** : A specific URI format to use WMQ.
- **destination:** VSG3.R3C2.PIPE3.REQUEST@MQS3 is a concatenation of the target queue name and the queue manager name.

- **targetService:** /exampleApp/dispatchOrder is the target service in CIWSR3C2 (it matches the dynamically installed URIMAP shown in Figure 4-6 on page 121)

  Tip: If you do not want to specify the targetService in URI data, you can pass the same information by setting /exampleApp/dispatchOrder as the TRIGDATA attribute of the receive queue VSG3.R3C2.PIPE3.REQUEST.

- **replyDestination:** VSG3.R3C2.PIPE3.RESPONSE is the reply queue name for the response.

Important: When the URI specified on a EXEC CICS INVOKE WEBSERVICE command begins with `jms:/`, CICS uses WMQ rather than HTTP to send the request. The application program itself does not need to be aware that WMQ is being used as the transport mechanism in place of HTTP.

Timeout considerations

It is not possible to manage timeout for a WMQ service requester application by specifying a timeout value on the URI. We tested different values for the timeout parameter and found that it always timed out after one minute.

For further information about using WMQ to transport SOAP messages, see *WebSphere MQ - Transport for SOAP,* SC34-6651*.*

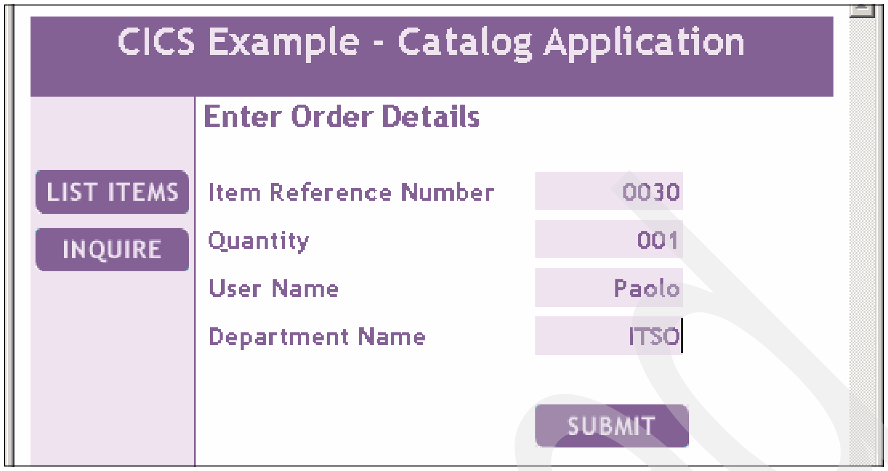
4.4.2 Configuring WebSphere Application Server on Windows

We deployed the catalog manager service requester application (catalog.ear) to WebSphere Application Server for Windows 2000 as documented in 3.2.3, “Configuring WebSphere Application Server on Windows” on page 87.

4.5 Testing the WMQ configuration

To test our WMQ setup we used a Web browser to run the Catalog application as described in “Testing the configuration” on page 100.

Figure 4-9 shows order details for our request.


CICS Example - Catalog Application

Enter Order Details

Item Reference Number: 0030
Quantity: 001
User Name: Paolo
Department Name: ITSO

Figure 4-9 Catalog application - ORDER function

From a CICS 3270 screen we used the CICS Execution Diagnostic Facility (EDF) to intercept the DISP transaction on CICS region CIWSR3C2. We then used the CEMT INQUIRE TASK command to view the inflight transactions (Figure 4-10).

```
INQUIRE TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000026) Tra(CKAM)           Sus Tas Pri( 255 )
    Sta(SD) Use(CIWS3D  ) Uow(BE0772E8804C9C2B)
 Tas(0000344) Tra(CKTI)           Sus Tas Pri( 001 )
    Sta(SD) Use(CICSUSER) Uow(BE0CE7454196FC8B) Hty(MQSeries)
 Tas(0000386) Tra(CEMT) Fac(G350) Run Ter Pri( 255 )
    Sta(TO) Use(CICSUSER) Uow(BE0CE689E4ECE06F)
 Tas(0000388) Tra(**CPIL**)           Sus Tas Pri( 001 )
    Sta(SD) Use(CICSUSER) Uow(BE0CE7454267F84B) Hty(MQSeries)
 Tas(0000389) Tra(**CPIQ**)           Sus Tas Pri( 001 )
    Sta(S ) Use(CICSUSER) Uow(BE0CE745420A3860) Hty(RZCBNOTI)
 Tas(0000390) Tra(**DISP**)           Sus Tas Pri( 001 )
    Sta(U ) Use(CICSUSER) Uow(BE0CE74542800740) Hty(EDF     )
 Tas(0000392) Tra(CEDF) Fac(G353) Sus Ter Pri( 001 )
    Sta(SD) Use(CICSUSER) Uow(BE0CE74542C74820) Hty(ZCIOWAIT)

                                          SYSID=R3C2 APPLID=A6POR3C2
```

Figure 4-10 CEMT INQUIRE TASK

Figure 4-10 shows the inflight transactions:

- The SOAP MQ inbound listener transaction (CPIL)
- The SOAP MQ inbound router transaction (CPIQ)
- The transaction used for running the business logic program (DISP)

After ending the EDF session, we received the `ORDER SUCCESFULLY PLACED` response in the browser.

We noted the SYSPRINT messages by the CIWSMSGH message handler for CICS region CIWSR3C2 (Example 4-5).

Example 4-5 CICS CIWSR3C2 - SYSPRINT

```
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: dispatchOrderEndpoint
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: DISP
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-URI
CIWSMSGH:  Container content: wmq:VSG3.R3C2.PIPE3.REQUEST/exampleApp/dispatchOrder
```

Note that after the request for the dispatchOrderEndpoint service arrives, the message handler changes the transaction ID to DISP. In particular, note that the DFHWS-URI container shows the URI in WMQ format.

4.6 High availability with WMQ

In Section 3.4, “Configuring for high availability” on page 101 we outlined how HTTP Web service requests can be balanced across multiple CICS regions in order to provide a high availability configuration. Here we take a brief look at how WMQ Web service requests can be balanced across multiple CICS regions.

On page 101, we also discussed how once a request is received by a specific CICS region, it can be dynamically routed within a CICSPlex. These transaction and program routing mechanisms can be used irrespective of how the SOAP message is transported.

The principal areas for consideration when designing a high availability configuration for WebSphere MQ are:

- How to share access to queues across multiple CICS regions

- How to load balance WMQ connections across multiple queue managers

Figure 4-11 shows an example high availability configuration for WMQ, in which queues shared in the coupling facility can be accessed by CICS regions running on different LPARs, and WMQ connections are balanced across different queue managers using shared channels.

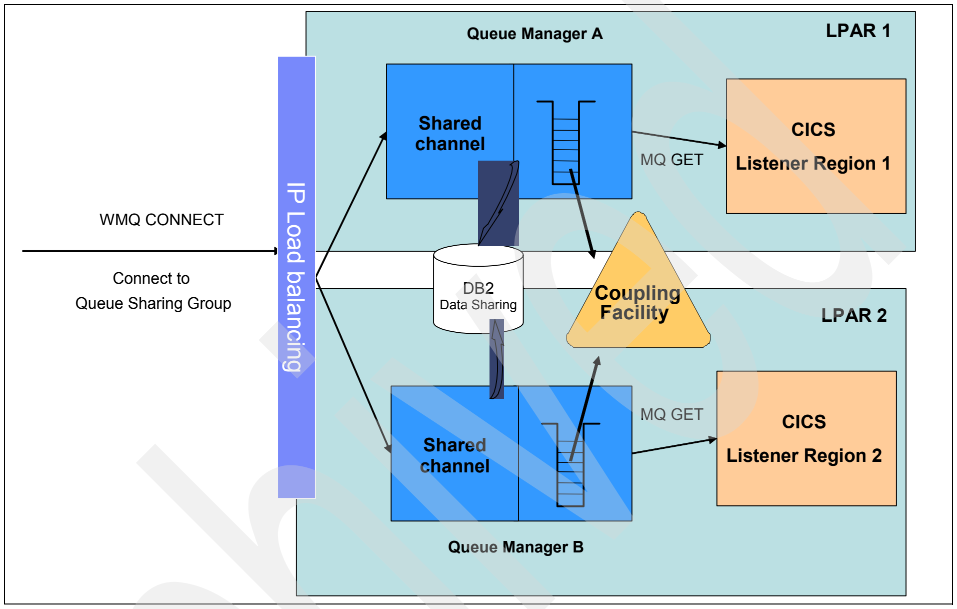


Figure 4-11 High availability configuration for WMQ

Figure 4-11 shows an example WMQ configuration that takes advantage of several parallel sysplex high availability capabilities, specifically:

- **Shared queues**

  A *shared queue* is a type of queue in which messages on that queue can be accessed by one or more queue managers that are identified to the sysplex. The queue managers that can access the same set of shared queues form a group called a *queue-sharing group* (QSG).

  A QSG controls which queue managers can access which coupling facility list structures and hence, which shared queues. Each coupling facility list structure is owned by a QSG and can only be accessed by queue managers in that QSG.

  Multiple queue managers on multiple MVS images within the same queue-sharing group can put messages to and get messages from the same shared queue. This is achieved by storing all the messages in a shared queue in the same coupling facility list structure.

Multiple queue managers on multiple MVS images within the same queue-sharing group can access the same WebSphere MQ objects. This is achieved by storing the object definitions in tables of a DB2 data-sharing group.

The use of shared queues provides a highly available solution because the failure of a single MVS image does not prevent access to shared queues. Another benefit is a capability to implement pull workload balancing. It means that by defining the input queue of an application (such as a CICS service provider application) as a shared queue, you make any message put to that queue available to be retrieved by any queue manager in the queue-sharing group.

- **Shared channels**

  The advantage of using *shared channels* is high availability when compared to being connected to a single queue manager. An inbound channel is classed as shared if it is connected to the queue manager through a group listener. A group listener is an additional task started on each channel initiator in the queue-sharing group. This task listens on an ip address/port combination, specific to that queue manager, known as its group address. Each group address can then be registered with an IP routing mechanism such as Sysplex Distributor.

- **Sysplex Distributor**

  Sysplex Distributor is designed to address the requirement of one single network-visible IP address for a service. Sysplex distributor can be used to map a queue-sharing group-wide generic IP address/port to a specific group address.

For more information about configuring high availability with WMQ refer to *WebSphere MQ in a z/OS Parallel Sysplex Environment*, SG24-6864.

5

Connecting CICS to the service integration bus

This chapter introduces the *service integration bus* (or *bus*) and the benefits of connecting your CICS Web service applications to a bus. It then goes on to explain the steps involved in accessing a CICS Web service over a bus.

5.1 Overview of the service integration bus

WebSphere Application Server V6 provides the ability to use the service integration bus as an intermediary between service requestors and service providers, allowing control over the flow, routing, and transformation of messages.

The use of Web services with the service integration bus is an evolution of the Web Services Gateway (WSGW) provided in WebSphere Application Server Version 5. Whereas the Web Services Gateway was a stand-alone application in V5, the bus is more tightly integrated into the application server, enabling users to use the WebSphere Application Server administration and scalability options, and also build on top of the asynchronous messaging features provided by WebSphere Application Server.

The bus allows the system administrator to create a level of indirection between service requesters and providers by exposing existing services at new destinations. The bus also provides options for managing these services through *mediations*, which can access and manipulate incoming and outgoing message content, or even route the message to a different service. Support for JAX-RPC (the Java API for XML-based Remote Procedure Calls) handlers is also included in the bus.

Figure 5-1 is an overview of the bus and how it can be used to enable Web services clients to access a CICS Web service. Clients can use bus-generated WSDL to access the service, and appropriate mediations could be used for message logging or transformation purposes.

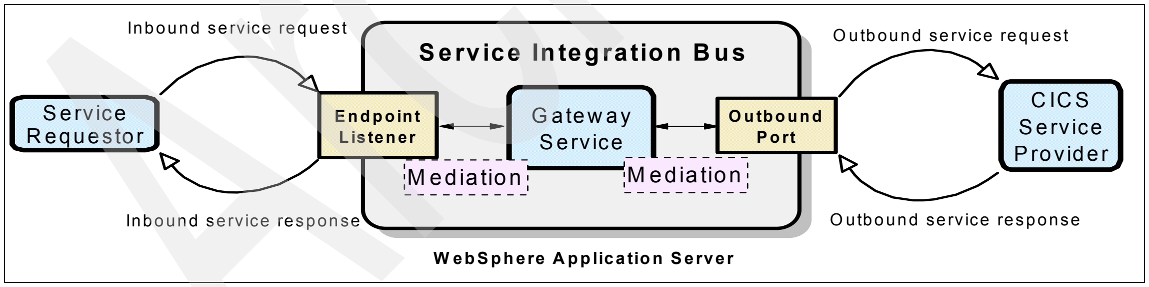


Figure 5-1 Exposing a CICS Web service over the service integration bus

It is possible for CICS to interoperate with the bus both as a service provider and as a service requester. The use of bus-generated WSDL means that the service requester does not need to know the location of the CICS service provider; it only needs to know the location of the bus. The bus itself knows the location of the CICS service provider. Similarly, a CICS service requester does not need to know the location of the service provider; it only needs to know the location of the bus.

Among the components you may come across in discussions and implementations of buses are the following:

- **Bus**: The "intelligent network" on which inbound and outbound services and gateway resources are defined.
- **Endpoint listener**: Entry points to the bus for Web services clients. Endpoint listeners allow clients to connect over SOAP/HTTP or SOAP/JMS. They are associated with inbound services and gateway resources.
- **Inbound service**: Destinations within the bus exposed as Web services (a gateway service can be thought of as a special kind of inbound service).
- **Outbound service**: Destinations within the bus that represent external Web services. CICS is invoked via an outbound service.
- **Gateway instance**: Enables a user to create gateway services.
- **Gateway service**: Exposes a target Web service that is external to the bus, as a bus-managed Web service. (We will be deploying our CICS service provider application as a gateway service.)
- **Mediation**: A stateless session EJB™ attached to a service destination that can apply processing to messages that pass through it, for example, logging or message transformation.
- **JAX-RPC handler**: JAX-RPC is a J2EE standard for intercepting and manipulating Web services messages.

5.1.1 Why you would connect CICS to a bus

We describe here some reasons why you might want to use the Web services features of a WebSphere Application Server service integration bus with your CICS service provider and service requester applications.

- Service location independence

  You may not want to publish details of the location (TCP/IP addresses) of your CICS service provider applications. Indirect access from the service requester to the service provider via a bus means that a change in the address of the service provider requires a configuration update of the bus, but not the service requesters. This service location independence enables a more flexible configuration and eases the task of service administration.
- Securely externalizing existing applications

  Businesses can deploy CICS Web services as secure gateway services on the bus. This enables applications deployed on a CICS deep inside an enterprise to be made available as Web services on the Internet to customers, suppliers, and business partners.

- Return on investment

  Any number of business partners can reuse an existing CICS application process that you make available as a gateway service using the bus. This provides great opportunity for the reuse of existing assets.

- Protocol transformation

  The bus allows a protocol switch between the service requester and the service provider. For example, if CICS exposes a service provider application using HTTP as the transport mechanism, a service requester accessing CICS via the bus can connect to the bus using JMS, and the bus will transparently forward the request on to CICS over HTTP. This function is invaluable for ensuring smooth interoperability between businesses that may implement Web services with different transport mechanisms.

- Standards-based integration

  The bus provides support for the major Web services standards, giving businesses confidence that they can use it to build flexible and interoperable solutions.

5.2 Preparation

After outlining our test configuration (Figure 5-2), we show how we:

- Configured CICS for a gateway service
- Created a gateway service on the bus
- Tested access to a CICS service provider application via the bus

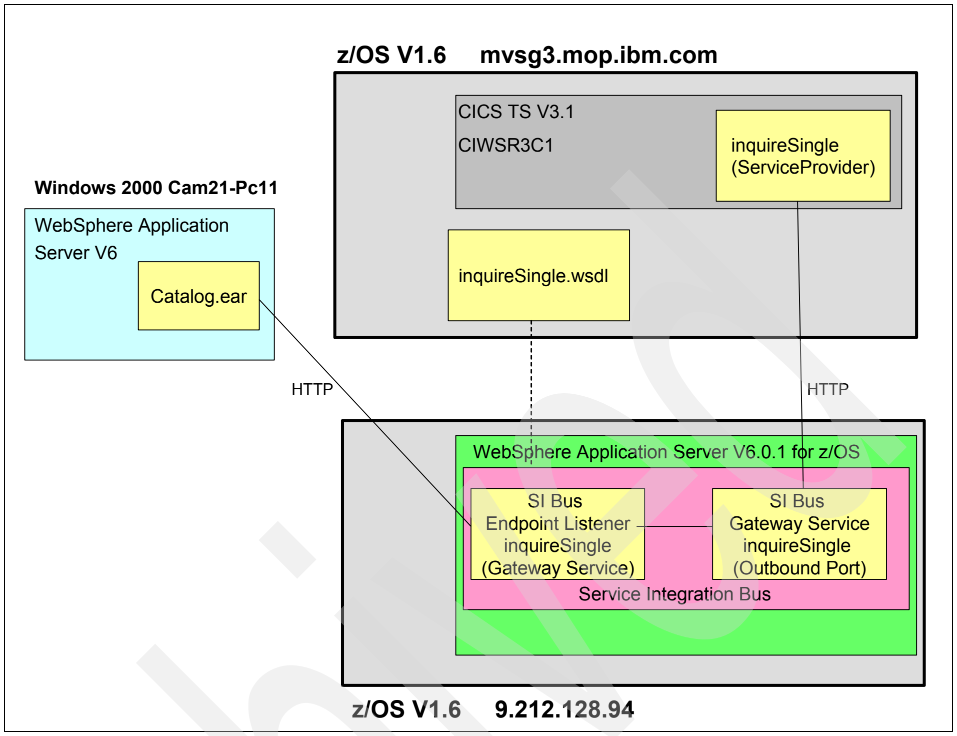

z/OS V1.6 mvsg3.mop.ibm.com
CICS TS V3.1
CIWSR3C1
inquireSingle (ServiceProvider)
Windows 2000 Cam21-Pc11
WebSphere Application Server V6
Catalog.ear
inquireSingle.wsdl
HTTP
HTTP
WebSphere Application Server V6.0.1 for z/OS
SI Bus Endpoint Listener inquireSingle (Gateway Service)
SI Bus Gateway Service inquireSingle (Outbound Port)
Service Integration Bus
z/OS V1.6 9.212.128.94

Figure 5-2 The catalog application accessed via a service integration bus

The configuration in Figure 5-2 shows:

- The Catalog manager application Web service inquireSingle deployed in CICS region CIWSR3C1
- The inquireSingle.wsdl file of the inquireSingle service used by the bus to *locate* the CICS service
- A gateway service inquireSingle used by the bus to *access* the CICS service
- An endpoint listener created in the bus and accessed by the service requester
- The catalog manager service requester deployed in WebSphere Application Server for Windows

In our tests, we configured a service integration bus within a WebSphere Application Server running on z/OS (TCP/IP address 9.212.128.94).

5.2.1 Software checklist

For the configuration shown in Figure 5-2 we used the levels of software shown in Table 5-1.

Table 5-1 Software used in the service integration bus scenarios

Windows	z/OS
Windows 2000 SP4	z/OS V1.6
IBM WebSphere Application Server - ND V6.0.2.0	► CICS Transaction Server V3.1 ► WebSphere Application Server V6.01 for z/OS in a Network Deployment configuration
Internet Explorer V6.0	
Our J2EE application ► Catalog.ear Catalog manager service requester application	CICS-supplied Catalog Manager application

Note: You require a WebSphere Application Server Network Deployment configuration if you want to use the Web services gateway functionality.

5.2.2 Definition checklist

The z/OS definitions we used to configure the scenario are listed in Table 5-2.

Table 5-2 Definition checklist

Value	CICS TS	WebSphere Application Server for z/OS (used for the bus)
IP name	mvsg3.mop.ibm.com	
IP address	9.100.193.167	9.212.128.94
TCP/IP port	13301	9080
Job name	CIWSR3C1	PTS001S
APPLID	A6POR3C1	
TCPIPSERVICE	WSGW	
Provider PIPELINE	EXPIPE01	
URIMAP	INQSINGW	

5.3 Configuring CICS for a gateway service

In order to enable access to the inquireSingle service of the CICS catalog manager application using a gateway service, the WSDL file describing the CICS service must be made available to the WebSphere Application Server that is hosting the bus. To do this, we performed the following tasks:

- Updated the catalog manager inquireSingle WSDL file
- Created a URIMAP for serving the WSDL file
- Tested the retrieval of the WSDL file from a Web browser

For all IP requests from the bus to CICS we used the same TCPIPSERVICE R3C1 that we configured in "Configuring the TCPIPSERVICE definition" on page 78.

For the Web service request that is passed from the bus to CICS, we used the EXPIPE01 pipeline that we configured in "Configuring the PIPELINE definition" on page 83. An alternative approach is to create a pipeline to be used for all requests from the bus. You may chose to take this approach if you want a separation of bus requests from non-bus requests, for example, if you want to run different message handlers for the two different types of request.

5.3.1 Updating the CICS-supplied sample WSDL file

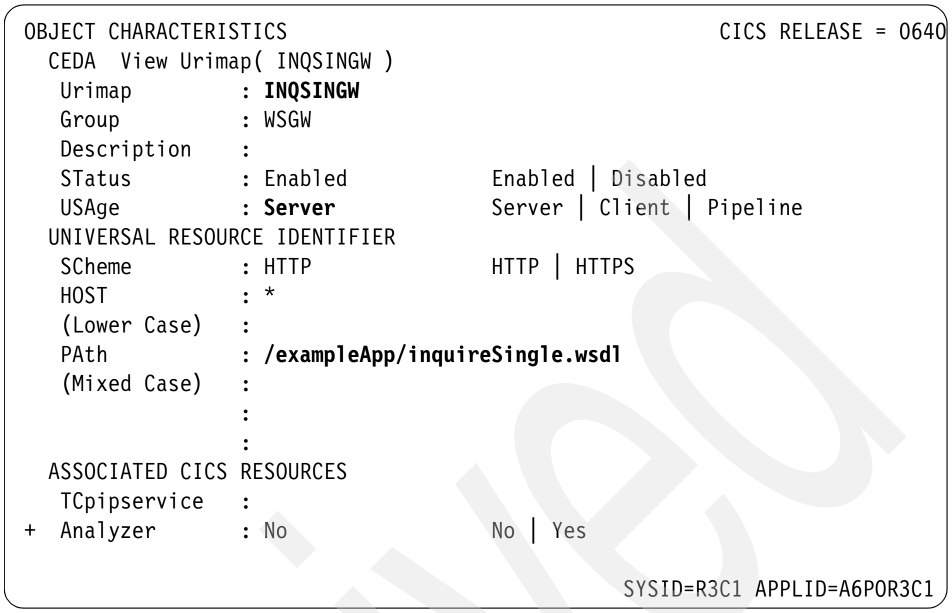
We changed the endpoint information in the CICS-supplied inquireSingle.wsdl to point to the IP address and port used by the R3C1 TCPIPSERVICE. We copied the sample WSDL file provided by CICS into file /u/exampleApp/inquireSingle.wsdl, and updated the *location* attribute of the *service* element as shown in Example 5-1.

Example 5-1 Specifying the CICS service location in inquireSingle WSDL

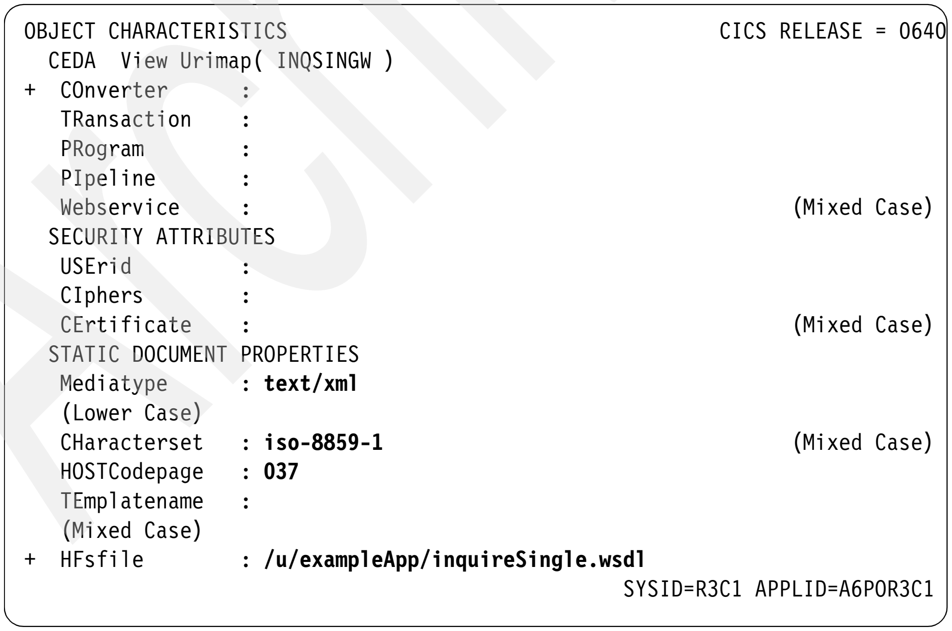
```
<service name="DFHOXCMNService">
        <port binding="tns:DFHOXCMNHTTPSoapBinding" name="DFHOXCMNPort">
          <soap:address
          location="http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle"/>
        </port>
</service>
```

5.3.2 Creating a URIMAP for the WSDL file

CICS allows us to map incoming HTTP requests to an HFS file, and then serve the file (just like an HTTP server). Figure 5-3 and Figure 5-4 show the URIMAP that we used to serve the file /u/exampleApp/inquireSingle.wsdl.


```
OBJECT CHARACTERISTICS                                        CICS RELEASE = 0640
  CEDA  View Urimap( INQSINGW )
   Urimap         : INQSINGW
   Group          : WSGW
   Description    :
   STatus         : Enabled            Enabled | Disabled
   USAge          : Server             Server | Client | Pipeline
  UNIVERSAL RESOURCE IDENTIFIER
   SCheme         : HTTP               HTTP | HTTPS
   HOST           : *
   (Lower Case)   :
   PAth           : /exampleApp/inquireSingle.wsdl
   (Mixed Case)   :
                  :
                  :
  ASSOCIATED CICS RESOURCES
   TCpipservice   :
+  Analyzer       : No                 No | Yes

                                                       SYSID=R3C1 APPLID=A6POR3C1
```
Figure 5-3 URIMAP for serving inquireSingle.wsdl (page 1 of 2)


```
OBJECT CHARACTERISTICS                                        CICS RELEASE = 0640
  CEDA  View Urimap( INQSINGW )
+  COnverter      :
   TRansaction    :
   PRogram        :
   PIpeline       :
   Webservice     :                                                 (Mixed Case)
  SECURITY ATTRIBUTES
   USErid         :
   CIphers        :
   CErtificate    :                                                 (Mixed Case)
  STATIC DOCUMENT PROPERTIES
   Mediatype      : text/xml
   (Lower Case)
   CHaracterset   : iso-8859-1                                      (Mixed Case)
   HOSTCodepage   : 037
   TEmplatename   :
   (Mixed Case)
+  HFsfile        : /u/exampleApp/inquireSingle.wsdl
                                                       SYSID=R3C1 APPLID=A6POR3C1
```
Figure 5-4 URIMAP for serving inquireSingle.wsdl (page 2 of 2)

We used the following values:

- URIMAP name was set to INQSINGW.
- USAGE was set to SERVER, indicating that CICS is to act like an HTTP server, in this case serving a static file.
- SCHEME was set to HTTP because no transport security was being used.
- HOST was set to an asterisk (*) to indicate that the URIMAP definition is to be used as a wildcard to match on any host name.
- PATH was set to /exampleApp/inquireSingle.wsdl. Together with the HOST and SCHEME values this means that the URIMAP resource will cater to any HTTP request of the form `http://*/exampleApp/inquireSingle.wsdl`.
- TCPIPSERVICE was not set so that the URIMAP definition applies to a request on any inbound port.
- MEDIATYPE was set to text/xml to indicate that an XML file is being served. CICS creates a Content-Type header for the response using the value of this attribute.

  Note: If the MEDIATYPE attribute specifies a text type, the CHARACTERSET and HOSTCODEPAGE attributes must also be specified so that code page conversion can take place.

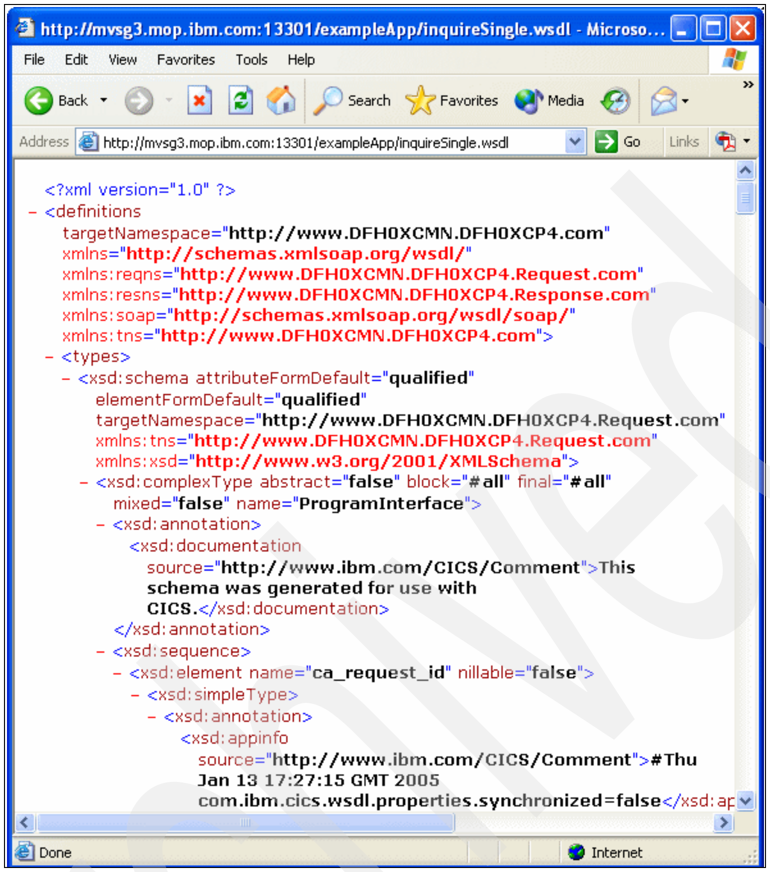
- CHARACTERSET was set to iso-8859-1 so that CICS converts the body of the response that is sent to the bus into ASCII.
- HOSTCODEPAGE was set to 037 to indicate the EBCDIC code page in which the WSDL file is encoded. This information is needed by CICS to perform code page conversion for the body of the response.
- HFSFILE was set to /u/exampleApp/inquireSingle.wsdl, which is the fully qualified name of the HFS file containing the WSDL.
- All other values were allowed to default.

5.3.3 Testing the retrieval of the WSDL file from a Web browser

Prior to configuring the gateway service on the bus, we tested the CICS configuration by retrieving the inquireSingle.wsdl file from a Web browser using the URL:

```
http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle.wsdl
```

Figure 5-5 shows the WSDL file displayed in the Web browser.


http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle.wsdl - Microso...
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle.wsdl Go Links
Done Internet

```
  <?xml version="1.0" ?>
- <definitions
    targetNamespace="http://www.DFH0XCMN.DFH0XCP4.com"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:reqns="http://www.DFH0XCMN.DFH0XCP4.Request.com"
    xmlns:resns="http://www.DFH0XCMN.DFH0XCP4.Response.com"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.DFH0XCMN.DFH0XCP4.com">
  - <types>
    - <xsd:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://www.DFH0XCMN.DFH0XCP4.Request.com"
        xmlns:tns="http://www.DFH0XCMN.DFH0XCP4.Request.com"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      - <xsd:complexType abstract="false" block="#all" final="#all"
          mixed="false" name="ProgramInterface">
        - <xsd:annotation>
            <xsd:documentation
              source="http://www.ibm.com/CICS/Comment">This
              schema was generated for use with
              CICS.</xsd:documentation>
          </xsd:annotation>
        - <xsd:sequence>
          - <xsd:element name="ca_request_id" nillable="false">
            - <xsd:simpleType>
              - <xsd:annotation>
                  <xsd:appinfo
                    source="http://www.ibm.com/CICS/Comment">#Thu
                    Jan 13 17:27:15 GMT 2005
                    com.ibm.cics.wsdl.properties.synchronized=false</xsd:ap
```

Figure 5-5 inquireSingle.wsdl from Web browser

This test demonstrates that CICS is successfully serving the WSDL file. After configuring the bus, the bus itself will be able to retrieve this WSDL file from CICS.

5.4 Creating a gateway service on the bus

At this point, you need to ask your WebSphere Application Server administrator for their assistance in defining a CICS gateway service on the bus.

For the purposes of this book, we assume that you have access to a WebSphere Application Server running in Network Deployment mode (in our case the WebSphere Application Server was running on z/OS), and that the WebSphere administrator has done the following:

- Created a service integration bus
- Configured the SOAPHTTPChannel1
- Created an endpoint listener

For information about these and other bus administration tasks, see *WebSphere Version 6 Web Services Handbook Development and Deployment,* SG24-6461.

In this section we explain how we created a CICS gateway service. To do this, we performed the following tasks:

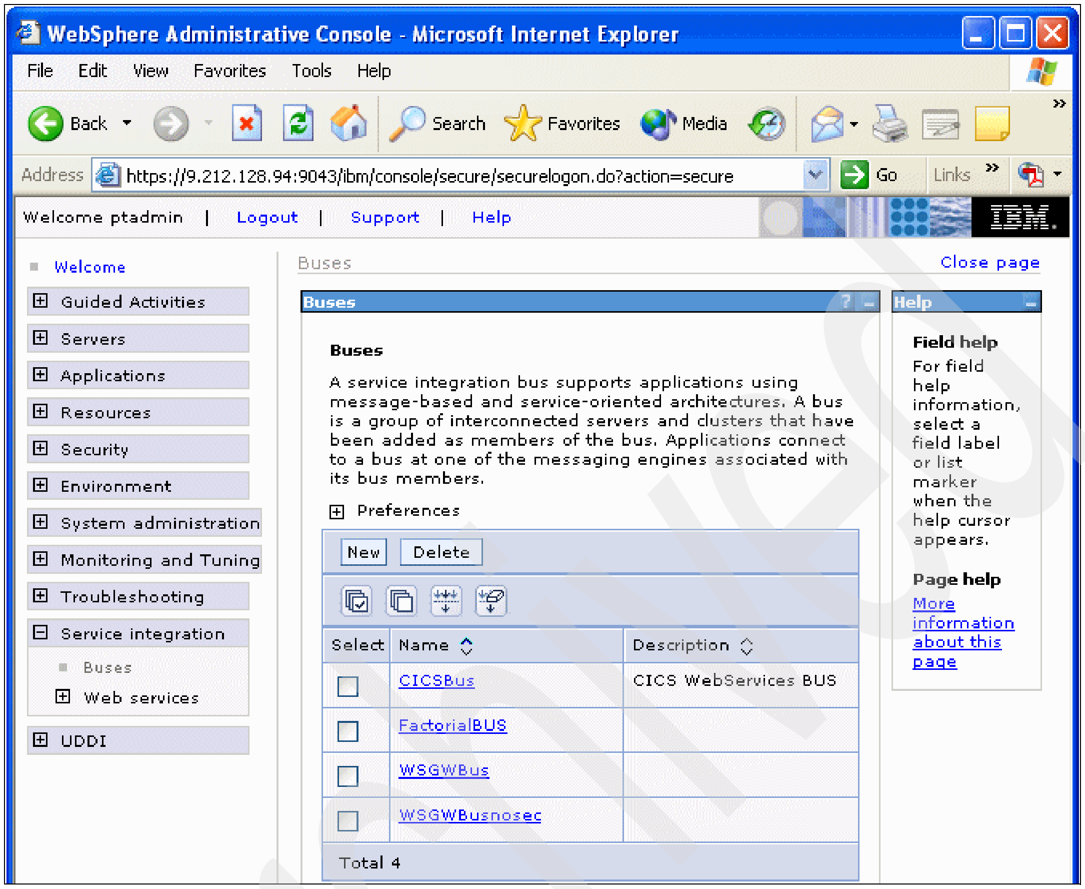
- Identified the bus to be used
- Created a Web services gateway instance
- Created the gateway service

5.4.1 Identifying the bus to be used

We logged on to the Administrative console of the WebSphere Application Server, which is hosting our bus, using the URL:

```
http://9.212.128.94:9060/admin
```

We selected **Service Integration** → **Buses** to see what buses were currently defined on this application server. Figure 5-6 shows the bus *CICSBus* that was previously created by our WebSphere administrator. This is the bus that we will now use for creating a gateway instance and a gateway service.


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address https://9.212.128.94:9043/ibm/console/secure/securelogon.do?action=secure
Go Links
Welcome ptadmin | Logout | Support | Help
Welcome
Guided Activities
Servers
Applications
Resources
Security
Environment
System administration
Monitoring and Tuning
Troubleshooting
Service integration
Buses
Web services
UDDI
Buses
Close page
Help
Field help
For field help information, select a field label or list marker when the help cursor appears.
Page help
More information about this page

Buses

A service integration bus supports applications using message-based and service-oriented architectures. A bus is a group of interconnected servers and clusters that have been added as members of the bus. Applications connect to a bus at one of the messaging engines associated with its bus members.

Preferences

New Delete

Select	Name	Description
☐	CICSBus	CICS WebServices BUS
☐	FactorialBUS	
☐	WSGWBus	
☐	WSGWBusnosec	

Total 4

Figure 5-6 Service integration bus CICSBus

5.4.2 Creating a Web services gateway instance

A *gateway instance* is the base on which you create gateway and proxy services; these services cannot be created until an instance exists. Within a bus, you can create multiple gateway instances in order to partition these services into logical groups to allow simpler management.

We clicked **CICSBus** and then selected **Web service gateway instances.** We then selected **New** to create a new Web service gateway instance. In the General Properties, we entered the following information:

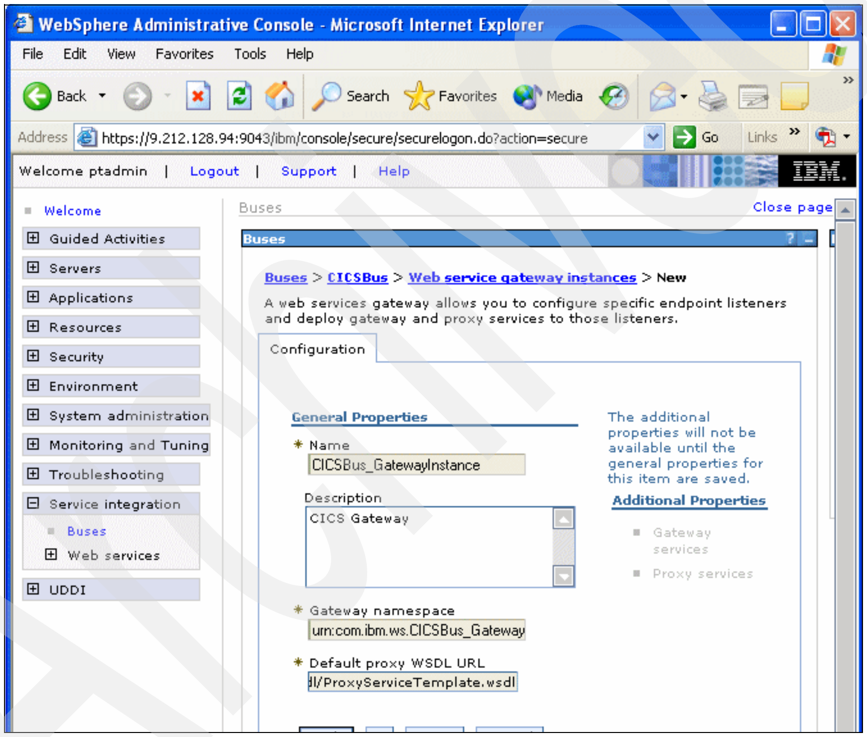
- The name for the gateway instance CICSBus_GatewayInstance.
- The Gateway namespace urn:com.ibm.ws.CICSBUS_Gateway. This is the namespace that will be used in all gateway-generated WSDL.

Tip: It is good practice to use a namespace that you are happy with from the start, because changing it later will require you to redeploy any associated gateway services.

- We entered the location for the Default proxy WSDL URL: http://9.212.128.94:9080/sibws/proxywsdl/ProxyServiceTemplate.wsdl

Note: We will not use this proxy service, but the admin console requires that one be specified.

- We clicked **OK** and then **Save** to save our new gateway instance.


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address https://9.212.128.94:9043/ibm/console/secure/securelogon.do?action=secure
Go Links
Welcome ptadmin | Logout | Support | Help
Welcome
Guided Activities
Servers
Applications
Resources
Security
Environment
System administration
Monitoring and Tuning
Troubleshooting
Service integration
Buses
Web services
UDDI
Buses
Close page

Buses > CICSBus > Web service gateway instances > New

A web services gateway allows you to configure specific endpoint listeners and deploy gateway and proxy services to those listeners.

Configuration

General Properties

* Name
CICSBus_GatewayInstance

Description
CICS Gateway

* Gateway namespace
urn:com.ibm.ws.CICSBus_Gateway

* Default proxy WSDL URL
dl/ProxyServiceTemplate.wsdl

The additional properties will not be available until the general properties for this item are saved.

Additional Properties
- Gateway services
- Proxy services

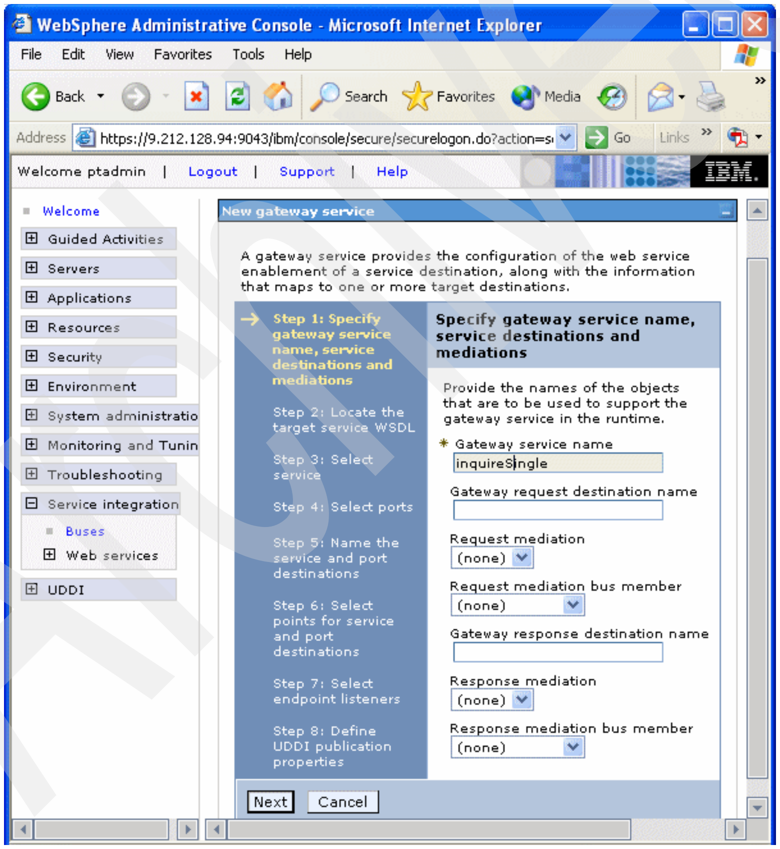
Figure 5-7 Create a gateway instance CICSBus_GatewayInstance

5.4.3 Creating a gateway service

Gateway services are used to take an existing WSDL-described Web service external to the bus (such as the inquireSingle CICS service) and expose it as a new Web service at a new endpoint, enabling you to relocate the underlying target service (if needed) without changing the details of the gateway service.

We used the WebSphere admin console to create our gateway service:

- We selected **Service integration** → **Buses** and clicked the name of our bus **CICSBus**.
- From the additional properties list of CICSBus, we clicked **Web service gateway instances** and then clicked again on our gateway instance **CICSBus_GatewayInstance**.
- From the additional properties list of CICSBus_GatewayInstance, we clicked **Gateway Services** and then **New**.
- We selected the type of gateway service as a **WSDL-defined web service provider** and clicked **Next**.
- In step 1, we entered the name of the gateway service **inquireSingle** (Figure 5-8) and clicked **Next**.


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address https://9.212.128.94:9043/ibm/console/secure/securelogon.do?action=s Go Links
Welcome ptadmin | Logout | Support | Help
Welcome
Guided Activities
Servers
Applications
Resources
Security
Environment
System administratio
Monitoring and Tunin
Troubleshooting
Service integration
Buses
Web services
UDDI

New gateway service

A gateway service provides the configuration of the web service enablement of a service destination, along with the information that maps to one or more target destinations.

→ Step 1: Specify gateway service name, service destinations and mediations
Step 2: Locate the target service WSDL
Step 3: Select service
Step 4: Select ports
Step 5: Name the service and port destinations
Step 6: Select points for service and port destinations
Step 7: Select endpoint listeners
Step 8: Define UDDI publication properties

Specify gateway service name, service destinations and mediations

Provide the names of the objects that are to be used to support the gateway service in the runtime.

* Gateway service name: inquireSingle

Gateway request destination name:

Request mediation: (none)

Request mediation bus member: (none)

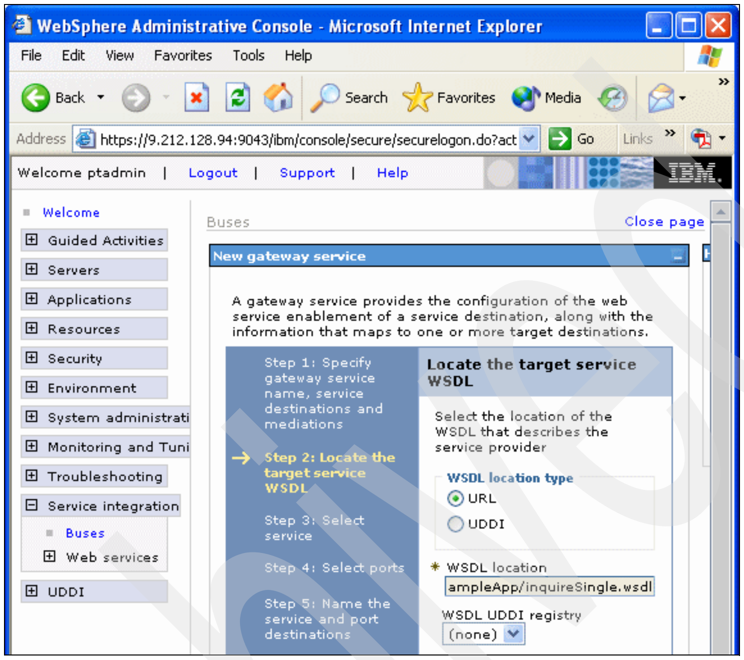
Gateway response destination name:

Response mediation: (none)

Response mediation bus member: (none)

Figure 5-8 Create gateway service - Step 1

- In step 2 we specified the location of the inquireSingle WSDL (http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle.wsdl) which allows the bus to locate the CICS service (Figure 5-9).


WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address https://9.212.128.94:9043/ibm/console/secure/securelogon.do?act
Go Links
Welcome ptadmin | Logout | Support | Help
Welcome
Guided Activities
Servers
Applications
Resources
Security
Environment
System administrati
Monitoring and Tuni
Troubleshooting
Service integration
Buses
Web services
UDDI
Buses
Close page
New gateway service

A gateway service provides the configuration of the web service enablement of a service destination, along with the information that maps to one or more target destinations.

Step 1: Specify gateway service name, service destinations and mediations

→ Step 2: Locate the target service WSDL

Step 3: Select service

Step 4: Select ports

Step 5: Name the service and port destinations

Locate the target service WSDL

Select the location of the WSDL that describes the service provider

WSDL location type
[✓ URL]
UDDI

* WSDL location: ampleApp/inquireSingle.wsdl

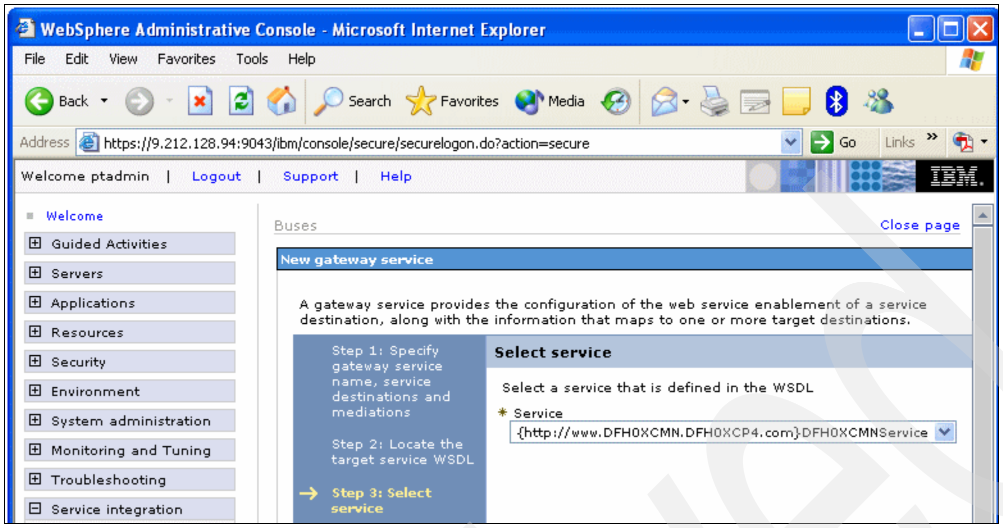
WSDL UDDI registry: (none)

Figure 5-9 Create gateway service - Step 2

- When we clicked **Next**, WebSphere retrieved the WSDL file (in our case, by sending an HTTP request to CICS requesting the WSDL file).
- We were then prompted to chose the operation defined in the WSDL which we want to enable as a gateway service (Figure 5-10). In this case there is only one option to chose from:

  `{http://www.DFHOXCMN.DFHOXCP4.com}DFHOXCMNService`

  We selected the operation and clicked **Next.**


WebSphere Administrative Console - Microsoft Internet Explorer

Address https://9.212.128.94:9043/ibm/console/secure/securelogon.do?action=secure

Welcome ptadmin | Logout | Support | Help

Buses

New gateway service

A gateway service provides the configuration of the web service enablement of a service destination, along with the information that maps to one or more target destinations.

Step 1: Specify gateway service name, service destinations and mediations

Step 2: Locate the target service WSDL

Step 3: Select service

Select service

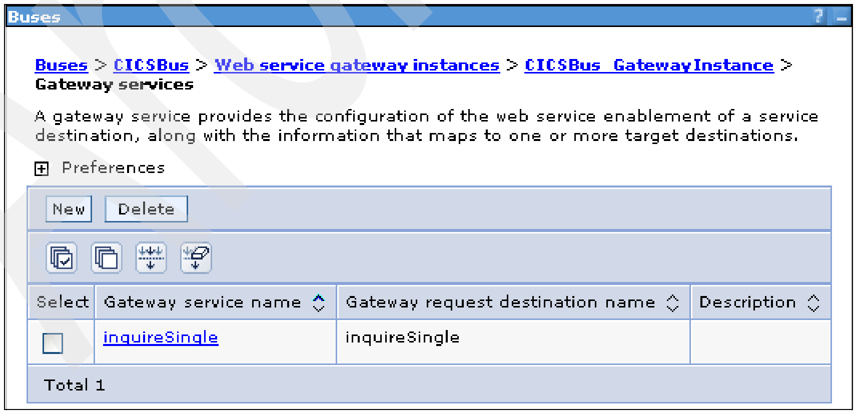
Select a service that is defined in the WSDL

* Service {http://www.DFH0XCMN.DFH0XCP4.com}DFH0XCMNService

Figure 5-10 Create gateway service - Step 3

- For each of the steps 4 through 6 we accepted the default options.
- In step 7 (**Select endpoint listeners)**, we selected the **servernameSOAPHTTPChannel1** endpoint listener (the only one associated with our bus).
- For step 8 (**Define UDDI publication properties**), we accepted the default (no UDDI reference), clicked **Finish** and saved our changes.

We have now successfully defined our CICS gateway service (Figure 5-11).


Buses

Buses > CICSBus > Web service gateway instances > CICSBus_GatewayInstance > **Gateway services**

A gateway service provides the configuration of the web service enablement of a service destination, along with the information that maps to one or more target destinations.

Select	Gateway service name	Gateway request destination name	Description
☐	inquireSingle	inquireSingle	
Total 1			

Figure 5-11 inquireSingle gateway service

5.5 Testing the CICS gateway service

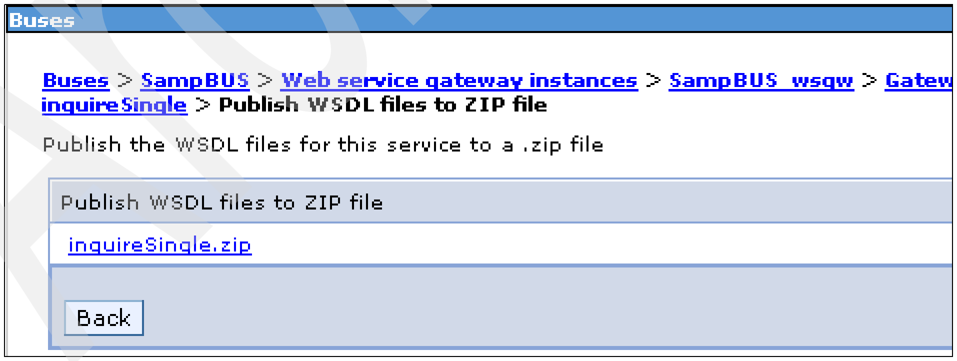
In this section we discuss how we tested the gateway service configuration using the catalog manager application. We cover the following tasks:

- Publishing the bus-generated WSDL for the inquireSingle gateway service
- Configuring the Catalog manager J2EE application
- Invoking the gateway service

5.5.1 Publish the bus-generated WSDL

In order to test the gateway service, we needed to publish the bus-generated WSDL by requesting a zip file from WebSphere, which we then saved on the local machine.

- Using the WebSphere admin console we clicked **Service integration** → **Buses** and then clicked on our bus CICSBus.
- From the additional properties list of CICSBus, we clicked **Web service Gateway instances** and then clicked again on our gateway instance CICSBus_GatewayInstance.
- From the additional properties list of CICSBus_GatewayInstance, we clicked **Gateway Services** and then chose our gateway service inquireSingle.
- From the additional properties list of inquireSingle, we clicked **Inbound web service enablement**.
- From the additional properties list, we clicked **Publish WSDL files to ZIP file** (Figure 5-12).


Buses

Buses > SampBUS > Web service gateway instances > SampBUS_wsgw > Gatew
inquireSingle > Publish WSDL files to ZIP file

Publish the WSDL files for this service to a .zip file

Publish WSDL files to ZIP file

inquireSingle.zip

Figure 5-12 Publish inquireSingle WSDL to ZIP file

- We clicked **inquireSingle.zip** and saved the file on the local machine.

- We unzipped the file and noted that the following files were contained in the zip:
  - CICSBus.inquireSingleBindings.wsdl
  - CICSBusinquireSingleNonBound.wsdl
  - CICSBus.inquireSinglePortTypes.wsdl
  - CICSBus.inquireSingleService.wsdl
- The endpoint information (for the inquireSingle gateway service) is stored in file CICSBus.inquireSingleService.wsdl. We opened the file and located the SOAP address in the `service` element (Example 5-2). This is the endpoint address for our gateway service.

Example 5-2 Gateway service inquireSingle endpoint address

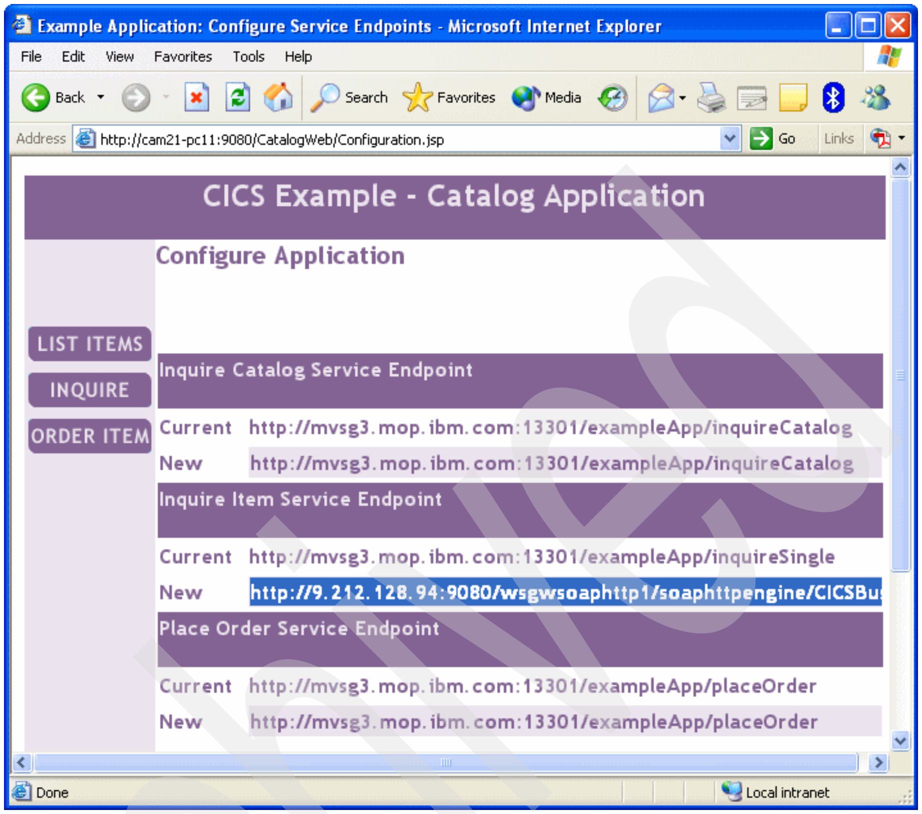
```
<soap:address
location="http://9.212.128.94:9080/wsgwsoaphttp1/soaphttpengine/CICSBus
/inquireSingle/MV06_server1_SOAPHTTPChannel1_InboundPort"/>
```

5.5.2 Configuring the catalog manager J2EE application

Next we needed to configure the catalog manager J2EE application so that it accesses the inquireSingle gateway service. We started a Web browser session and entered the URL:

```
http://cam21-pc11:9080/CatalogWeb/Welcome.jsp
```

We clicked **CONFIGURE** and specified the endpoint address of the inquireSingle gateway service for the Catalog manager **Inquire Item Service Endpoint** (Figure 5-13). We clicked **SUBMIT** to complete the update.



Example Application: Configure Service Endpoints - Microsoft Internet Explorer

Address http://cam21-pc11:9080/CatalogWeb/Configuration.jsp

CICS Example - Catalog Application

Configure Application

Inquire Catalog Service Endpoint

Current http://mvsg3.mop.ibm.com:13301/exampleApp/inquireCatalog

New http://mvsg3.mop.ibm.com:13301/exampleApp/inquireCatalog

Inquire Item Service Endpoint

Current http://mvsg3.mop.ibm.com:13301/exampleApp/inquireSingle

New http://9.212.128.94:9080/wsgwsoaphttp1/soaphttpengine/CICSBu

Place Order Service Endpoint

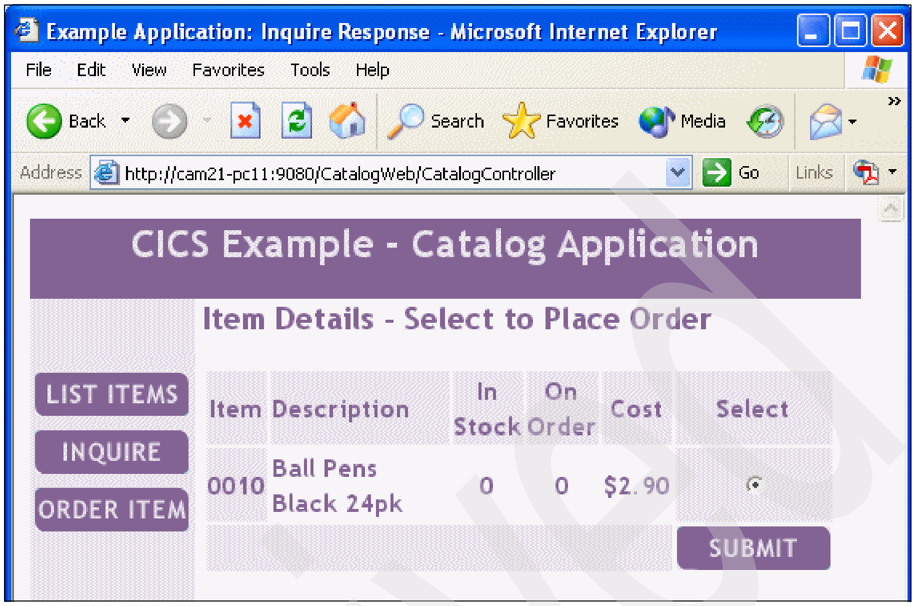
Current http://mvsg3.mop.ibm.com:13301/exampleApp/placeOrder

New http://mvsg3.mop.ibm.com:13301/exampleApp/placeOrder

Figure 5-13 Configure Catalog manager application to invoke gateway service

5.5.3 Invoking the gateway service

We clicked **INQUIRE** to submit a catalog single item inquiry. Figure 5-14 shows the results of the inquireSingle Web service call.


Example Application: Inquire Response - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Media
Address
http://cam21-pc11:9080/CatalogWeb/CatalogController
Go
Links
LIST ITEMS
INQUIRE
ORDER ITEM
SUBMIT

CICS Example - Catalog Application

Item Details - Select to Place Order

Item	Description	In Stock	On Order	Cost	Select
0010	Ball Pens Black 24pk	0	0	$2.90	●

Figure 5-14 Calling the gateway service inquireSingle

Example 5-3 shows the output from our message handler program CIWSMSGH for the inquireSingle service request.

Example 5-3 CIWSMSGH message handler output for inquireSingle request

```
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: inquireSingle
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: INQS
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-URI
CIWSMSGH:  Container content:
/exampleApp/inquireSingle
```

Note that this request is treated by CICS in exactly the same way as though it had been sent directly from the service requester, rather than having been sent over the SiBus.

Part 3

Security management

We begin this section by introducing you to some basic cryptography concepts and to some algorithms and protocols that are used to protect computing resources. The concepts include secret key cryptography, public key cryptography, hashing functions, message authentication codes, digital signatures, digital certificates, and certificate revocation lists. The algorithms and protocols include DES, AES, RSA, SHA-1, DSS, and TLS/SSL.

A CICS application may use one of these algorithms by calling one of the services provided by the Integrated Cryptographic Service Facility (ICSF). The service might implement the requested algorithm by invoking either a software routine or some cryptographic hardware. Therefore, we include in this section some information about IBM cryptographic hardware.

Next we explain how you can use transport layer security or SOAP message security, or both, to secure Web services running in CICS both when you use

HTTP as the transport mechanism and when you use WebSphere MQ as the transport mechanism.

Finally, we discuss security scenarios that we implemented to demonstrate how you can secure a CICS Web services environment.

6

Elements of cryptography

When implementing security for a CICS Web services solution, it is useful to understand some basic cryptography terminology and concepts such as:

- What we mean by identification, authentication, authorization, integrity, confidentiality, and non-repudiation
- The difference between secret key cryptography and public key cryptography
- The definition of a hashing function, a digital signature, a digital certificate, a certificate authority, and a certificate revocation list

The purpose of this chapter is to provide some background information about cryptography to help you when reading the following chapters in this book, in which we explain how cryptographic functions can be used with CICS and describe the scenarios that we tested.

6.1 The role of cryptography

A complete security policy will put mechanisms in place to achieve the following objectives:

- **Identification**

  Identification is the ability to assign an identity to the entity accessing the system. Typically the identity is used to control access to resources. Depending on the security model in which the identification is performed, the identity can be called a *user ID,* a *UID*, or a *principal*.

- **Authentication**

  Authentication is the process of validating the identity claimed by the accessing entity. Authentication is performed by verifying authentication information provided with the claimed identity. The authentication information is generally referred to as the accessor's *credentials*. A credential can be the accessor's name and password; it can also be a *token* provided by a trusted party, such as a Kerberos ticket or an X.509 certificate.

  You need authentication when you want to give different rights to access resources (such as files and databases) to different requesting identities.

  Note: Authentication is usually one of the earliest steps in a request workflow. When authenticated, an identity can be *asserted* to the downstream process steps, meaning that these steps trust the upstream steps to have already successfully authenticated the identity.

- **Authorization**

  Authorization is the process of checking whether an identity that has already been authenticated should be given access to a resource that it is requesting. A typical implementation of authorization is to pass to the access control mechanism a *security context* that contains the identity that has been authenticated.

- **Integrity**

  Integrity ensures that transmitted or stored information has not been altered in an unauthorized or accidental manner. Typically it is a mechanism to verify that what is received over a network is the same as what was sent.

- **Confidentiality**

  Confidentiality ensures that an unauthorized party cannot obtain the meaning of the transferred or stored data. Typically confidentiality is achieved by encrypting the data.

- **Auditing**

  With auditing, you capture and record security-related events (such as a user signing onto or off of a system) so that you can analyze them later, perhaps after a breach of your security has occurred.

- **Non-repudiation**

  Non-repudiation means that a sender and a receiver of data are able to provide legal proof to a third party that the sender did send the information, and the receiver received the identical information. Neither side is *able to deny*.

As used in computer security, cryptography provides the following processes:

- *Encrypting* converts plaintext (that is, data in normal, readable form) into ciphertext, which conceals the meaning of the data to any unauthorized recipient. Encrypting is also called *enciphering*.

  Most cryptographic systems combine two elements:

  – An algorithm that specifies the mathematical steps needed to encrypt the data.

  – A cryptographic key (a string of numbers or characters), or keys. The algorithm uses the key to select one relationship between plaintext and ciphertext out of the many possible relationships the algorithm provides. The selected relationship determines the composition of the algorithm's result.

- *Decrypting* converts ciphertext back into plaintext. Decrypting is also called *deciphering*.
- *Hashing* uses a one-way (irreversible) calculation to condense a long message into a compact bit string called a *message digest*.
- Generating a *digital signature* involves encrypting a message digest with a private key to create the electronic equivalent of a handwritten signature. You can use a digital signature to verify the identity of the signer and to ensure that nothing has altered the signed document since it was signed.

In this chapter we show how you can use cryptography to achieve authentication, data integrity, confidentiality, and non-repudiation.

6.2 Secret key (or symmetric) cryptography

In *secret key* cryptography the sender and receiver of a message know and use the same secret key; the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. See Figure 6-1. Secret key cryptography is also known as symmetric cryptography.

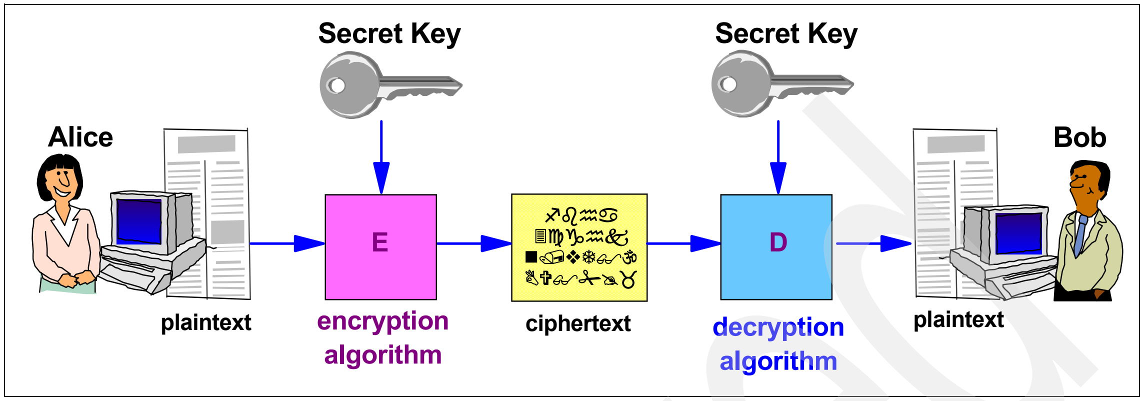


Figure 6-1 Secret key (or symmetric) cryptography

The main challenge of secret key cryptography is getting the sender and receiver to agree on the secret key without anyone else finding out. If the sender and receiver are in separate physical locations, they must trust a courier, a phone system, or some other transmission medium to prevent the disclosure of the secret key. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all messages encrypted using that key.

Block ciphers

A *block cipher* is a type of secret key encryption algorithm that transforms a *fixed*-length block of plaintext data into a block of ciphertext data of the same length. This transformation takes place under the action of a user-provided secret key. Decryption is performed by applying the reverse transformation to the ciphertext block using the same secret key. The fixed length is called the block size. Common block sizes include 64 bits and 128 bits.

Iterated block ciphers

Iterated block ciphers encrypt a plaintext block by a process that has several rounds. In each round, the same transformation (also known as a round function) is applied to the data using a *subkey*. The set of subkeys is usually derived from the user-provided secret key by a special function. The set of subkeys is called the *key schedule*. The number of rounds in an iterated cipher depends on the desired security level and the consequent trade-off with performance. In most cases, an increased number of rounds will improve the security offered by a block cipher.

6.2.1 DES

The Data Encryption Algorithm (DEA), developed by IBM, is one example of an iterated block cipher. IBM submitted the DEA to the National Bureau of Standards (NBS) during an NBS public solicitation for cryptographic algorithms to be used in a Federal Information Processing Standard (FIPS). In 1977 the NBS issued FIPS Publication 46 *Data Encryption Standard (DES)* which specified that the DEA be used within the United States Federal Government for the cryptographic protection of sensitive, but unclassified, computer data. As a result, the DEA is often called DES.

The DES was reaffirmed in 1983, 1988, 1993, and 1999. As time passed the NBS became the National Institute of Standards and Technology or NIST; it is a division of the U. S. Department of Commerce.

The DES has a 64-bit block size. A DES key consists of 64 bits, of which 56 bits are randomly generated and used directly by the algorithm. The other 8 bits, which are not used by the algorithm, can be used for error detection. The binary format of the key is:

(B1,B2,...,B7,P1,B8,...,B14,P2,B15,...,B49,P7,B50,...,B56,P8)

where (B1,B2,...,B56) are the independent bits of a DES key and (P1,P2,...,P8) are reserved for parity bits computed on the preceding seven independent bits and set so that the parity of the octet is odd, that is, there is an odd number of “1” bits in the octet.

How DES works

Readers who want or need to have an overview of how the DES algorithm works should consult B.1, “How DES works” on page 574.

DES modes of operation

When we use a block cipher to encrypt a message of *arbitrary* length, we use techniques known as modes of operation for the block cipher. In December, 1980, FIPS Publication 81 *DES Modes of Operation* announced four modes of operation for DES:

- Electronic Codebook (ECB)
- Cipher Block Chaining (CBC)
- Cipher Feedback (CFB)
- Output Feedback (OFB)

We now describe the first two of these modes of operation.

ECB

In ECB mode, the message M of arbitrary length is first divided into blocks m_i. Each block contains 64 bits, the block size of the DES algorithm. Each plaintext block m_i is used directly as the input block to the DES encryption algorithm with key k. The resultant output block is used directly as ciphertext. See Figure 6-2, where E_k represents encryption using the DES algorithm with k as the key.

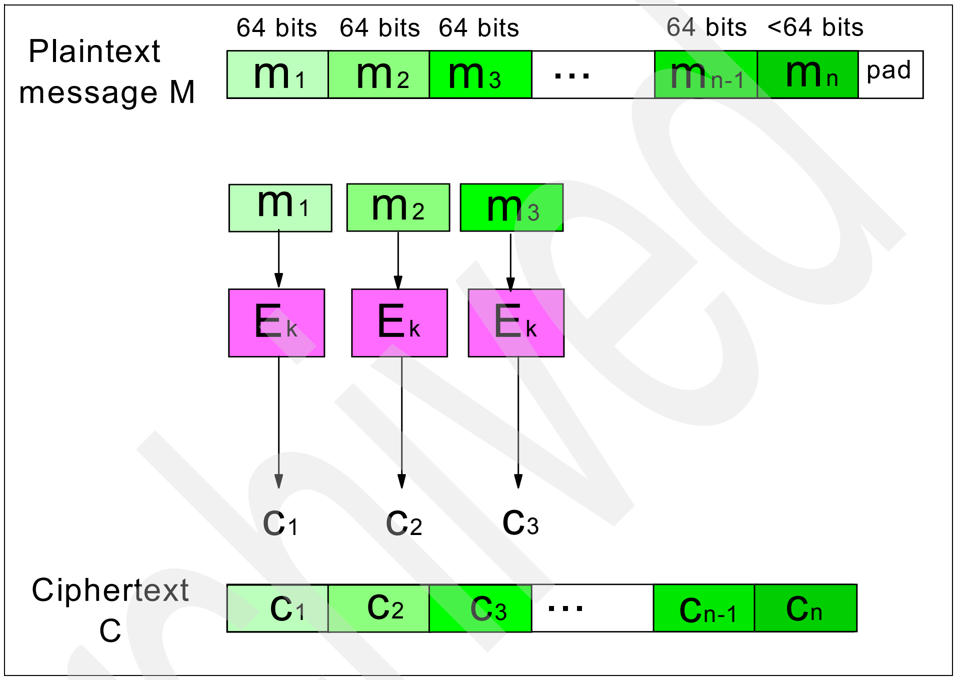


Figure 6-2 Electronic codebook (ECB) mode of operation

The analogy to a codebook arises because the same plaintext block always produces the same ciphertext block for a given cryptographic key. Thus a list (or codebook) of plaintext blocks and corresponding ciphertext blocks theoretically could be constructed for any given key.

Since the ECB mode is a 64-bit block cipher, an ECB device must encrypt data in integral multiples of 64 bits. If a user has less than 64 bits to encrypt, then the least significant bits of the unused portion of the input data block must be padded, for example, filled with random or pseudo-random bits prior to ECB encryption. The corresponding decrypting device must then discard these padding bits after decryption of the ciphertext block.

CBC

In practice, CBC is the most widely used mode of DES. In CBC, the message M of arbitrary length is first divided into blocks m_i. Each block contains 64 bits, the block size of the DES algorithm. Each plaintext block m_i is XORed (exclusive ORed) with the previous ciphertext block c_{i-1} and then encrypted. A 64-bit initialization vector c_0 is used as a "seed" for the process. See Figure 6-3, where a circle enclosing a cross represents an XOR operation.

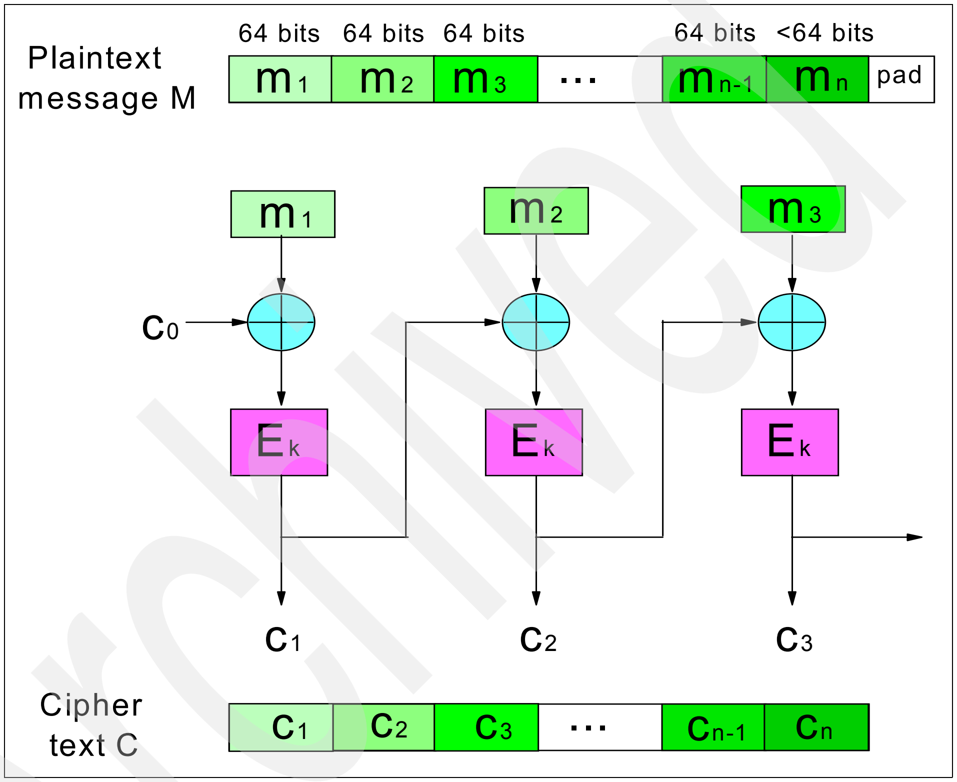


Figure 6-3 DES encryption using the Cipher Block Chaining (CBC) mode of operation

Thus, the encryption of each block depends on previous blocks, and the same 64-bit plaintext block can encrypt to different ciphertext blocks depending on its context in the overall message. XORing of the previous ciphertext block with the plaintext block conceals any patterns in the plaintext.

Partial data blocks (blocks of less than 64 bits) require special handling. One method of encrypting a final partial data block of a message is described next.

The following method can be used for applications where the length of the ciphertext can be greater than the length of the plaintext. In this case the final partial data block of a message is padded in the least significant bits positions with “0”s, “1”s, or pseudo-random bits. The decrypter will have to know when and to what extent padding has occurred. This can be accomplished explicitly, for example, using a padding indicator, or implicitly, for example, using constant length transactions.

The padding indicator will depend on the data being encrypted.

- Binary

  If the data is pure binary, then the partial data block should be left justified in the input block and the unused bits of the block set to the complement of the last data bit, that is, if the last data bit of the message is “0” then “1”s are used as padding bits and if the last data bit is “1” then “0”s are used. The input block is then encrypted.

  The resulting output block is the ciphertext. The ciphertext message must be marked as being padded so that the decrypter can reverse the padding process, remove the padding bits and produce the original plaintext. The decrypter scans the decrypted padded block and discards the least significant bits that are all identical.

- Bytes

  If the data consists of bytes (for example, 8-bit ASCII characters) then the padding indicator should be a character denoting the number of padding bytes, including itself, and should be placed in the least significant byte of the input block before encrypting. For example, if there are five ASCII data characters in the final partial block of a message to be encrypted, then an ASCII “3” is put in the least significant byte of the input block (any pad characters may be used in the other two pad positions) before encryption. Again the ciphertext message must be marked as being padded.

Figure 6-4 shows the decryption of a message using the CBC mode of operation; D_k represents decryption using the DES algorithm with key k.

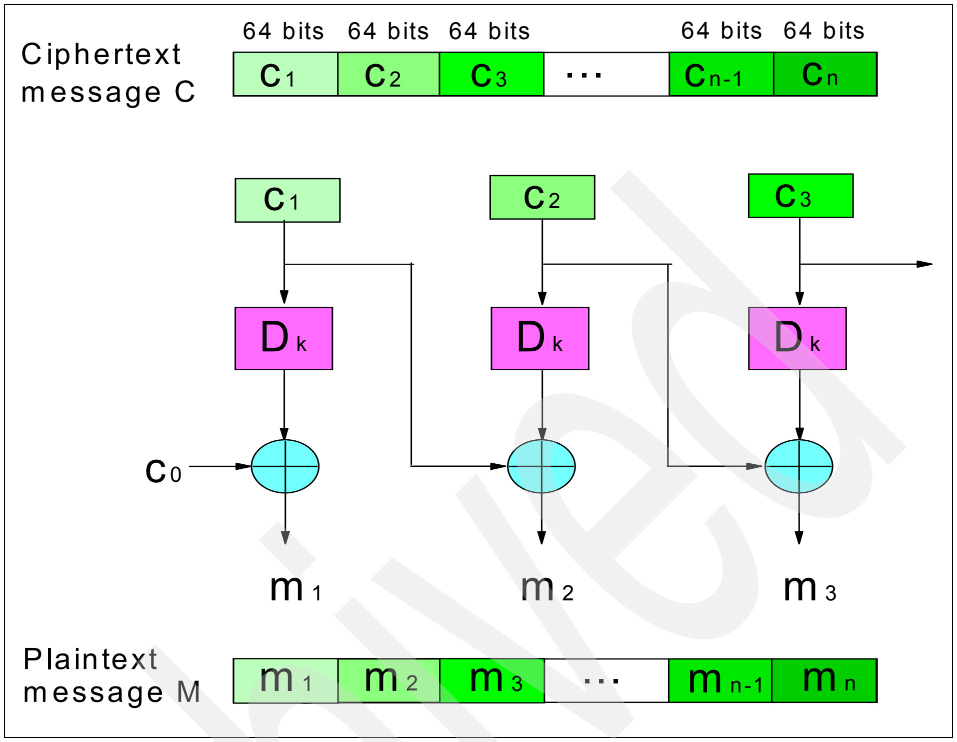


Figure 6-4 DES decryption using the CBC mode of operation

Status of DES

Because the speed of computers has increased significantly since 1977, it may now be possible to try every possible DES 56-bit key in turn until the correct key is identified. This technique of attempting to decipher a message is called exhaustive key search or brute force search. Indeed, a DES cracking machine has been used to recover a DES key in 22 hours.

Therefore, the consensus of the cryptographic community is that DES is no longer secure. FIPS 46-3 reaffirmed DES usage as of October 1999, but permitted single DES only for legacy systems. FIPS 46-3 included a definition of triple-DES (TDEA) which became "the FIPS approved symmetric encryption algorithm of choice." On November 26, 2001, the NIST published FIPS 197 announcing the Advanced Encryption Standard (AES); the standard became effective on May 26, 2002. The NIST withdrew FIPS 46-3 on May 19, 2005.

6.2.2 Triple DES (TDEA)

For some time it has been common practice to protect information with triple-DES instead of DES. This means that the input data is, in effect, encrypted three times. There are a variety of ways of doing this; FIPS Pub 46-3 defines triple-DES encryption with keys k_1, k_2, and k_3 as:

$C = E_{k3}(\ D_{k2}(\ E_{k1}(M)\ )\ )$

where $E_k(I)$ and $D_k(I)$ denote DES encryption and DES decryption, respectively, of the input I with the key k. See Figure 6-5.

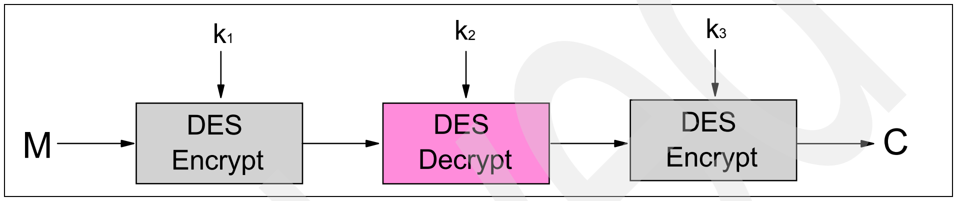


Figure 6-5 Triple DES - EDE

This mode of encryption is sometimes referred to as DES-EDE (encrypt, decrypt, encrypt). FIPS Pub 46-3 defines three keying options for DES-EDE:

- k_1, k_2 and k_3 are independent.

  Since each key has a length of 64 bits, this is sometimes referred to as TDEA-192. However, since only 56 bits of the 64 bits of each key are actually used, it is sometimes also called TDEA-168.
- k_1 and k_2 are independent, but $k_3 = k_1$.

  This may be called TDEA-128 or TDEA-112.
- $k_1 = k_2 = k_3$.

Another variant is DES-EEE, which consists of three consecutive encryptions.

TDEA modes of operation

Like all block ciphers, triple-DES can be used in a variety of modes. The American National Standards Institute (ANSI) X9.52 standard *Triple Data Encryption Algorithm Modes of Operation* describes seven different modes:

- TDEA Electronic Codebook (TECB)
- TDEA Cipher Block Chaining (TCBC)
- TDEA Cipher Block Chaining - Interleaved (TCBC - I)
- TDEA Cipher Feedback (TCFB)

- TDEA Cipher Feedback - Pipelined (TCFB-P)
- TDEA Output Feedback (TOFB)
- TDEA Output Feedback - Interleaved (TOFB-I)

Figure 6-6 shows TDEA running in TECB mode.

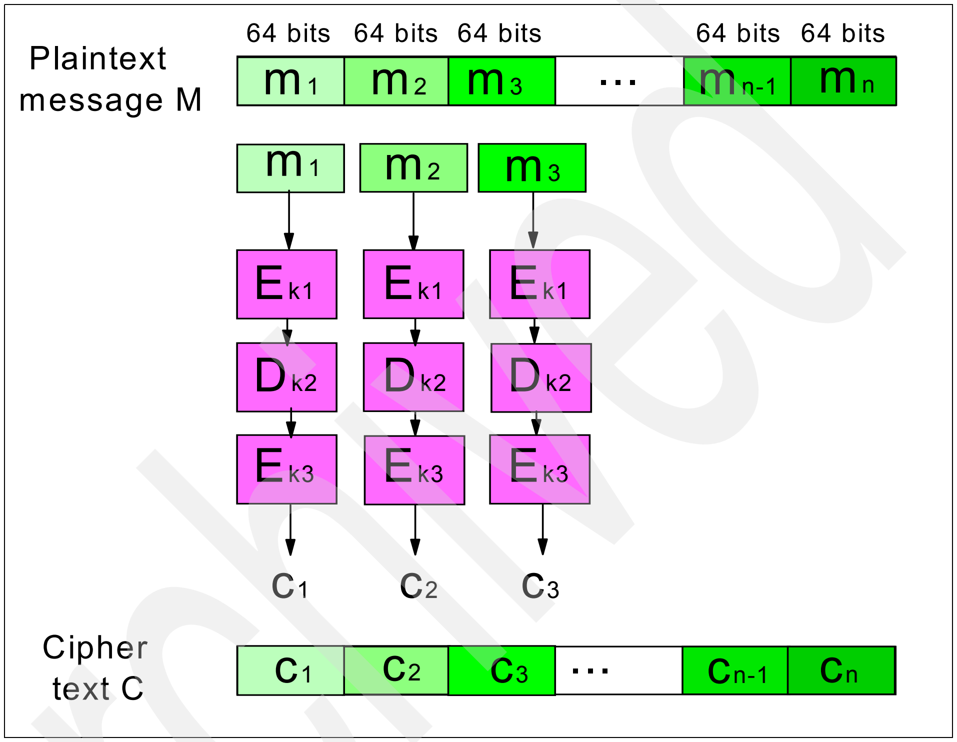


Figure 6-6 Triple DES running in TECB mode

6.2.3 AES

The Advanced Encryption Standard (AES) is another example of an iterated block cipher. The AES algorithm resulted from a multi-year evaluation process led by the NIST with submissions and review by an international community of cryptography experts. The Rijndael algorithm, invented by Joan Daemen and Vincent Rijmen, was selected as the standard. The NIST specified the AES in FIPS PUB 197 in November, 2001.

The AES processes data blocks of 128 bits. That is, the input and the output for the AES algorithm each consists of a sequence of 128 bits (16 bytes or 4 words).

The cipher key for the AES algorithm is a sequence of 128, 192, or 256 bits. These different "flavors" of AES can be referred to as "AES-128", "AES-192", and "AES-256". The number of words Nk in the key is thus 4, 6, or 8.

The number of rounds Nr to be performed during the execution of the algorithm depends on the key size. When Nk = 4, then Nr = 10. If Nk=6, then Nr=12, and when Nk=8, then Nr=14.

How AES works

Readers who want or need to have an overview of how the AES algorithm works should consult B.2, "How AES works" on page 578.

6.3 Public key (or asymmetric) cryptography

In *public key* cryptography each person gets a pair of keys, one called the *public* key and the other called the *private* key. The public key is published, while the private key is kept secret. The need for the sender and receiver to share secret information is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. In this system, it is no longer necessary to trust the security of some means of communication. The only requirement is that public keys be associated with their users in a trusted manner (for instance, in a trusted directory).

In public key cryptography:

- Data encrypted with a public key can only be decrypted with the corresponding private key. This guarantees data privacy for the receiver, since he is the only one able to decrypt the data. But the receiver cannot be sure who the sender is; it could be anyone.
- Data encrypted with a private key can only be decrypted with the corresponding public key. Anyone can decrypt the data, but the receiver knows who the sender is because the data can come only from one sender, the owner of the private key.

When Alice wishes to send a secret message to Bob, she looks up Bob's public key in a directory, uses it to encrypt the message and sends it off. Bob then uses his private key to decrypt the message and read it. No one listening in can decrypt the message. Anyone can send an encrypted message to Bob, but only Bob can read it (because only Bob knows Bob's private key). See Figure 6-7. Public key cryptography is also known as asymmetric cryptography.

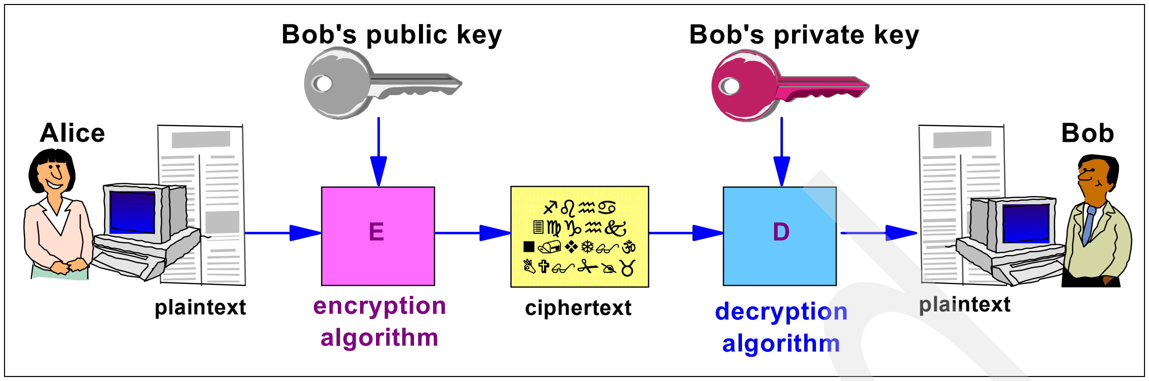


Figure 6-7 Public key (or asymmetric) cryptography

Note that public key cryptography solves the problem of how to safely transmit a secret key. When Alice wishes to send a secret key to Bob, she looks up Bob's public key in a directory, uses it to encrypt the secret key and sends it off. Bob then uses his private key to decrypt the secret key and read it. No one listening in can decrypt the secret key.

In a public key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public key system by deriving the private key from the public key. Typically, the defense against this is to make the problem of deriving the private key from the public key as difficult as possible. For instance, some public key cryptosystems are designed such that deriving the private key from the public key requires the attacker to factor a large number; in this case it is computationally not feasible to perform the derivation.

6.3.1 RSA

The RSA cryptosystem is a public key cryptosystem developed in 1977 by Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA stands for the first letter in each of its inventors' last names. The RSA algorithm is by far the most widely used public key cryptosystem in the world.

Note: Some information in this section was derived from the RSA Web site at www.rsasecurity.com/rsalabs/. The authors would like to thank RSA for permission to use this material.

Before we discuss how the RSA algorithm works, we review the following definitions:

- Prime number

  A prime number is any integer greater than 1 that is divisible only by 1 and itself. The first twelve primes are 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, and 37.

- Factor

  Given an integer n, any number that divides it is called a factor of n. For example, 7 is a factor of 91, because 91/7 is an integer.

- Factoring

  Factoring is the breaking down of an integer into its prime factors. For example, 140 = 2^2 x 5 x 7. This is a hard problem (that is, a computationally intensive problem; one that is computationally difficult to solve).

- Relatively prime

  Two integers are relatively prime if they have no common factors except 1. For example, 14 and 25 are relatively prime, while 14 and 91 are not (7 is a common factor of 14 and 91).

- Congruent modulo n

  Given integers a, b, and n with n > 0, we say that a and b are congruent modulo n if a-b is divisible by n, that is, if (a-b)/n = i, an integer. Equivalently, a and b are congruent modulo n if there is an integer i such that a-b = i x n, that is, such that a = b + (i x n). If a and b are congruent modulo n, we write

  a = b mod n

  For example, 50 = 0 mod 5 because (50 - 0)/5 =10, an integer. But 50 is not congruent to 1 mod 5 because (50-1)/5 is not an integer. However, 51 is congruent to 1 mod 5.

  In arithmetic mod n, integers between 0 and n - 1 are used with normal addition, subtraction, multiplication, and exponentiation, except that after each operation the result keeps only the remainder after dividing by n. For example, 5^6 mod 23 = 8 because 5^6=5 x 5 x 5 x 5 x 5 x 5 = 15,625 and 15,625 divided by 23 gives a quotient of 679 and a remainder of 8. Also, 3 + 4 = 2 mod 5 because 3 + 4 = 7 and 7 divided by 5 gives a quotient of 1and a remainder of 2.

The RSA algorithm works as follows: take two large primes, p and q, and compute their product n = pq. Choose a number, e, less than n and relatively prime to (p-1)(q-1). Find another number d such that (ed - 1) is divisible by (p-1)(q-1), that is, ed = 1 mod[(p-1)(q-1)] or d = e^{-1}mod[(p-1)(q-1)]. The public key is the pair (n, e); the private key is (n, d). For example, suppose we take the two primes p = 7 and q = 17. Their product is n = 119. The number e = 5 is relatively prime to (7-1)(17-1) = 96. The number d = 77 is such that ed-1 = 385-1 = 384 is divisible by 96. The public key is the pair (119, 5) and the private key is the pair (119,77). The factors p and q may be destroyed or kept with the private key.

RSA uses the following terminology:

- n is called the modulus
- e is called the public exponent
- d is called the private exponent

It is currently difficult to obtain the private exponent d from the public key (n, e). However if one could factor n into p and q, then one could obtain the private exponent d. Thus the security of the RSA system is based on the assumption that factoring is difficult. The discovery of an easy method of factoring would "break" RSA.

Here is how the RSA system could be used for encryption. Suppose Alice wants to send a message m to Bob. Alice creates the ciphertext c by exponentiating: $c = m^e \bmod n$, where e and n are Bob's public key. She sends c to Bob. To decrypt, Bob also exponentiates: $m = c^d \bmod n$; the relationship between e and d ensures that Bob correctly recovers m. Since only Bob knows d, only Bob can decrypt this message.

As an example, let's encrypt the message "sell" using the public key (119, 5). If we use the convention that a=1, b=2, c=3, and so forth to convert the letters to numbers, we find that the plaintext "sell" becomes 19 5 12 12.

- Raising 19 to the 5th power gives 2,476,099. Dividing 2,476,099 by n=119, we get a remainder of 66.
- Raising 5 to the 5th power gives 3125. Dividing 3125 by n=119, we get a remainder of 31.
- Raising 12 to the 5th power gives 248,832. Dividing 248,832 by n=119, we get a remainder of 3.

Thus our ciphertext is 66 31 3 3.

To decrypt 66 31 3 3, we use the private key (119, 77).

- Raising 66 to the 77th power gives 127316015002712725024996... (a very large number). Dividing that very large number by n=119, we get a remainder of 19.
- Raising 31 to the 77th power gives 683676142775442000196395... (another very large number). Dividing that number by n=119, we get a remainder of 5.

- Raising 3 to the 77th power gives 5474401089420219382077155.... Dividing that number by n=119, we get a remainder of 12.

Thus our plaintext is 19 5 12 12 or “sell”.

Digital envelopes

In practice the RSA system is often used together with a secret-key cryptosystem, such as DES. Suppose Alice wishes to send an encrypted message to Bob. She first encrypts the message with DES, using a randomly chosen DES key. Then she looks up Bob's public key and uses it to encrypt the DES key. The DES-encrypted message and the RSA-encrypted DES key are sent to Bob. Upon receiving them, Bob decrypts the DES key with his private key, then uses the DES key to decrypt the message itself. See Figure 6-8. This process is sometimes referred to as sending a digital envelope; it combines the high speed of DES with the key management convenience of the RSA system.

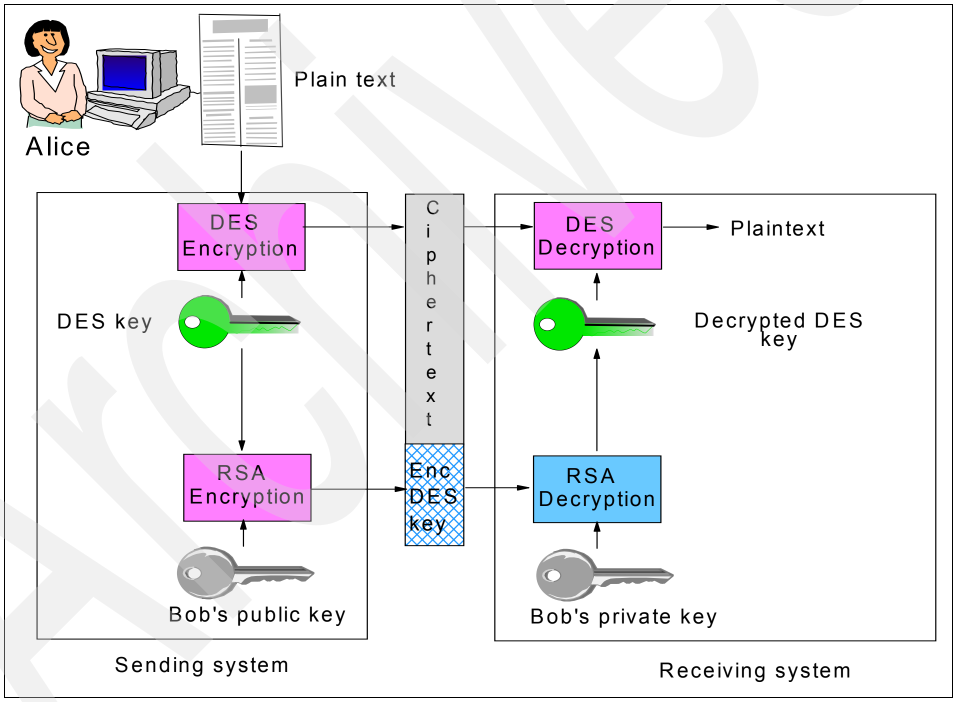


Figure 6-8 Using DES to encrypt data and RSA to manage the DES key

What is the appropriate key size in the RSA cryptosystem?

The size of a key in the RSA algorithm typically refers to the size of n. The two primes, p and q, which compose n should be of roughly equal length; this makes n harder to factor than if one of the primes is much smaller than the other. If one chooses to use a 768-bit value for n, the primes should each have a length of approximately 384 bits.

The best size for n depends on the user's security needs. The larger the value of n, the greater the security, but also the slower the RSA algorithm operations. Choose a length for n upon consideration, first, of the value of the protected data and how long it needs to be protected, and, second, of how powerful the potential threats might be.

Key sizes of 512-bits no longer provide sufficient security for anything more than very short-term security needs. RSA Laboratories currently recommends key sizes of 1024 bits for corporate use and 2048 bits for extremely valuable keys like the root key pair used by a certifying authority. Less valuable information may well be encrypted using a 768-bit key, since such a key is still beyond the reach of all known key breaking algorithms. RSA Laboratories publishes recommended key lengths on a regular basis.

As for the slowdown caused by increasing the key size, doubling the length of n will, on average, increase the time required for public key operations (encryption and signature verification) by a factor of four, and increase the time taken by private key operations (decrypting and signing) by a factor of eight. Key generation time would increase by a factor of 16 upon doubling the length of n, but this is a relatively infrequent operation for most users.

6.4 Hash functions

A hash function H is a transformation that takes an input message m and returns a fixed-size string, which is called the hash value h. Using mathematical notation for functions, we express this as h=H(m). See Figure 6-9.

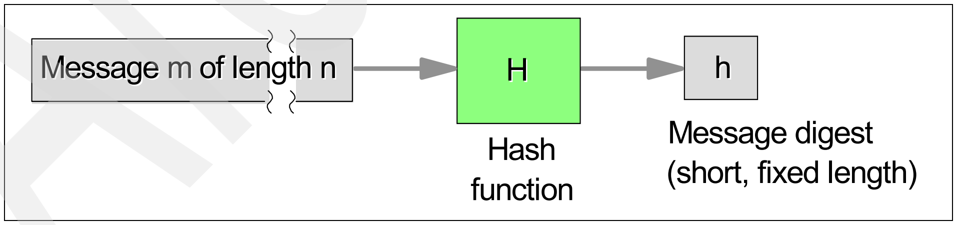


Figure 6-9 A hash function

Figure 6-10 shows an example of a hash function at work.

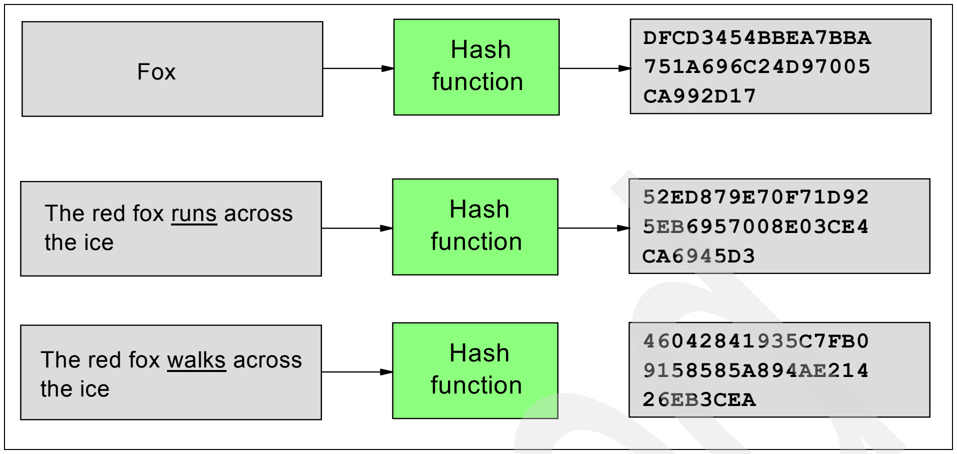


Figure 6-10 A hash function at work

Notice that the hash function in Figure 6-10 produces a message digest whose length is 20 bytes regardless of the length of the message. Notice also that if we make a small change in the message we get a very different message digest.

When employed in cryptography, hash functions are usually chosen to have some additional properties. The basic requirements for a cryptographic hash function are as follows:

- The input can be of any length.
- The output has a fixed length.
- H(m) is relatively easy to compute for any given m.
- H(m) is one-way.

  A hash function H is said to be one-way if it is hard to invert, where “hard to invert” means that given a hash value h, it is computationally not feasible to find some input m such that H(m)=h. An everyday example of a one-way function is mashing a potato; you can do it easily, but once you have mashed the potato, you will find it rather difficult to reconstruct the original potato.
- H(m) is collision-free.

  A collision-free hash function H is one for which it is computationally not feasible to find any two messages x and y such that H(x)=h and H(y)=h; that is, it is computationally not feasible to find any two messages that hash to the same value.

The hash value represents concisely the longer message or document from which it was computed; this value is called the *message digest*. One can think of a message digest as a *digital fingerprint* of the larger document; it identifies the message much like a real fingerprint identifies a person. Thus a good cryptographic hash function ensures that it is very difficult to:

- Recover the message from the message digest
- Construct a block of data M_2 that has the same message digest h as another given block M_1

You can use a hashing function to verify that data has not been altered during transmission. The sender of the data calculates the message digest using the data itself and the hashing function. The sender then ensures that the message digest is transmitted *with integrity* to the intended receiver of the data; one way to do this is to publish the message digest in a reliable source of public information. When the receiver gets the data, he can generate the message digest and compare it to the original one. If the two are equal, he can accept the data as genuine; if they differ, he can assume the data is bogus. See Figure 6-11.

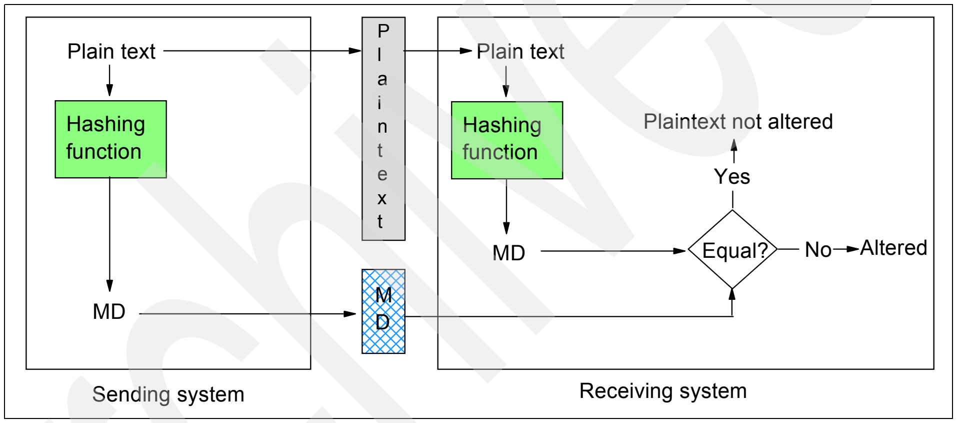


Figure 6-11 Using a hash function to verify that data has not been altered

In the preceding example, the message digest should not be sent in the clear. Since the hash functions are well-known and no key is involved, a man-in-the-middle could not only forge the message but also replace the message digest with that of the forged message. This would make it impossible for the receiver to detect the forgery.

I. Damgard and R.C. Merkle greatly influenced cryptographic hash function design by defining a hash function in terms of what is called a *compression function*. A compression function takes a fixed-length input (for example, 512 bits) and returns a shorter, fixed-length output (for example, 160 bits). Given a compression function F, a hash function can be defined by repeated applications of the compression function F until the entire message has been processed. In this process, a message of arbitrary length is broken into blocks whose length depends on the compression function, and *padded* (for security reasons) so the size of the message is a multiple of the block size. The blocks are then processed sequentially, taking as input the result of the hash so far and the current message block, with the final output being the hash value for the message. See Figure 6-12.

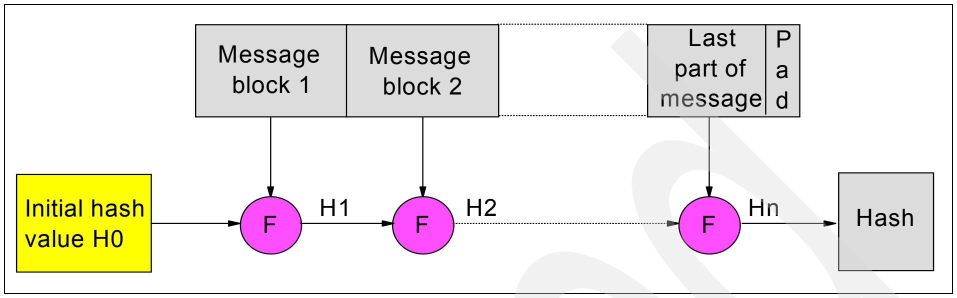


Figure 6-12 Iterative structure for hash functions

Among the well-known hash functions are the following:

- MD2 and MD5

  MD2 and MD5 were developed by Ronald Rivest of the Laboratory for Computer Science at the Massachusetts Institute of Technology (MIT). Both functions take a message of arbitrary length and produce a 128-bit message digest. MD2 was optimized for 8-bit machines, whereas MD5 was aimed at 32-bit machines. Description and source code for MD2 and MD5 can be found as Internet RFCs 1319 and 1321, respectively.

- SHA-1

  The Secure Hash Algorithm (SHA) was developed by the NIST and specified in the Secure Hash Standard (FIPS PUB 180). SHA-1 corrected an unpublished flaw in SHA and was published in 1994 as FIPS PUB 180-1.

  SHA-1 is an iterative hash function, as shown in Figure 6-12. It operates on messages whose length is less than 2^{64} bits. The message is first padded so that the length in bits of the message is a multiple of 512. Then the message is parsed into 512-bit message blocks.

  SHA-1 produces a 160-bit (20-byte) message digest. Readers who want or need to have an overview of how the SHA-1 algorithm works should consult B.3, “How SHA-1 works” on page 583.

  The algorithm is slightly slower than MD5 but the larger message digest makes it more secure against brute-force collision and inversion attacks.

- SHA-256

  FIPS PUB 180-2 specifies the SHA-256 algorithm. This algorithm also takes a message of less than 2^{64} bits in length but it produces a 256-bit message digest.

6.5 Message authentication codes

A message authentication code (MAC) is a short piece of information used to authenticate a message. It is an authentication tag derived by applying an authentication scheme, together with a secret key, to a message. Unlike digital signatures, MACs are computed and verified with the *same* key, so that they can only be verified by the intended recipient. The MAC value protects both a message's integrity and its authenticity.

There are four types of MACs:

- Unconditionally secure
- Stream cipher-based
- Block cipher-based
- Hash function-based

We briefly discuss block cipher-based MACs and hash function-based MACs.

6.5.1 Block cipher-based MACs

Figure 6-13 shows how we might use the DES algorithm to compute a MAC on a message M. We begin by dividing the message M into blocks m_i. Each block contains 64 bits, the block size of the DES algorithm. We XOR each block m_i with the previous ciphertext block c_{i-1} and then encrypt it by using the DES encryption algorithm with key k. A 64-bit initialization vector c_0 is used as a "seed" for the process. The output from the last step is the MAC.

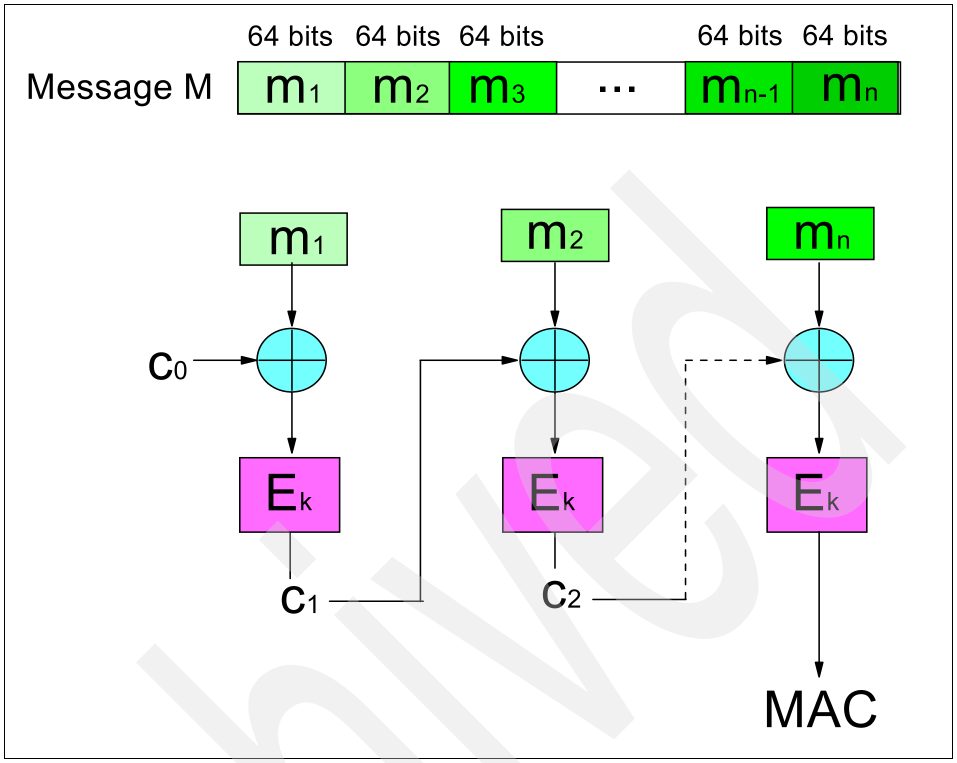


Figure 6-13 Computing a MAC by using the DES block cipher

6.5.2 Hash function-based MACs

MACs based on cryptographic hash functions are known as HMACs. HMACs have two functionally distinct parameters: a *message input*, and a *secret key* known only to the message originator and intended receivers.

An HMAC function is used by the message sender to produce a value (the MAC) that is formed by condensing the secret key and the message input. The MAC is typically sent to the message receiver along with the message. The receiver computes the MAC on the received message using the same key and HMAC function as was used by the sender, and compares the result computed with the received MAC. If the two values match, the message has been correctly received, and the receiver is assured that the sender is a member of the community of users that share the key. See Figure 6-14.

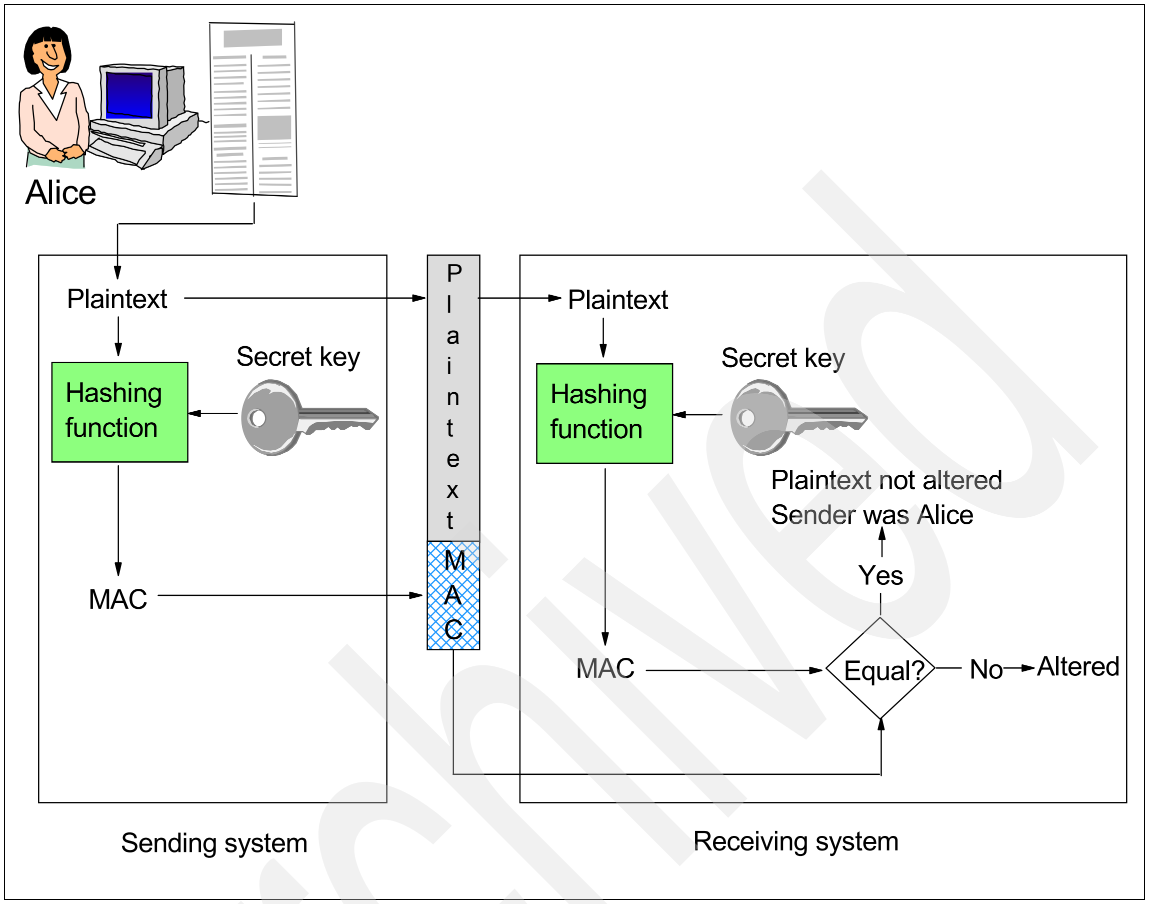


Figure 6-14 Keyed-hash message authentication code (HMAC)

Note that, since the receiver has the key that is used in creation of the MAC, this process does not offer a guarantee of non-repudiation because it is theoretically possible for the receiver to forge a message and claim it was sent by the sender.

For an overview of the HMAC function described in FIPS PUB 198, see B.4, "How the HMAC algorithm of FIPS PUB 198 works" on page 588.

6.6 Digital signatures

In this book *digital signature* is used to mean a cryptographically-based signature assurance scheme. We discuss two such schemes: the Digital Signature Algorithm (DSA) and RSA. While the DSA can only be used to provide digital signatures, the RSA system can be used for both encryption and digital signatures.

6.6.1 Using DSA for digital signatures

The NIST published the first version of the DSA in the *Digital Signature Standard (DSS)* FIPS PUB 186 in May, 1994. The current version was published in FIPS PUB 186-2 in January, 2000; in October, 2001, Change Notice 1 amended FIPS PUB 186-2.

DSA parameters

The DSA makes use of the following parameters:

- p = a prime number where $2^{1023} < p < 2^{1024}$
- q = a prime divisor of p-1, where $2^{159} < q < 2^{160}$
- $g = h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < p-1$ such that $h^{(p-1)/q} \bmod p > 1$
- x = a randomly or pseudo randomly generated integer with $0 < x < q$
- $y = g^{x} \bmod p$
- k = a randomly or pseudo randomly generated integer with $0 < k < q$

Appendix 2 and Appendix 3 of FIPS PUB 186-2 specify methods for generating p, q, x, and k.

The integers p, q, and g can be public and can be common to a group of users. The integers x and y are a user's private and public keys, respectively. Parameters x and k are used for signature generation only, and must be kept secret. Parameter k must be regenerated for each signature.

DSA signature generation

The signature of a message M is the pair of numbers r and s computed according to the equations:

$$r = (g^{k} \bmod p) \bmod q$$

$$s = (k^{-1}(\text{SHA-1}(M) + xr)) \bmod q$$

In these equations, k^{-1} is the multiplicative inverse of k, mod q; that is, $(k^{-1} k) \bmod q = 1$ and $0 < k^{-1} < q$. The value of SHA-1(M) is a 160-bit string output by the Secure Hash Algorithm SHA-1. For use in computing s, this string must be converted to an integer.

DSA signature verification

Prior to verifying the signature in a signed message, p, q, and g plus the sender's public key y and identity are made available to the verifier in an authenticated manner.

Let M', r', and s' be the received versions of M, r, and s respectively. See Figure 6-15.

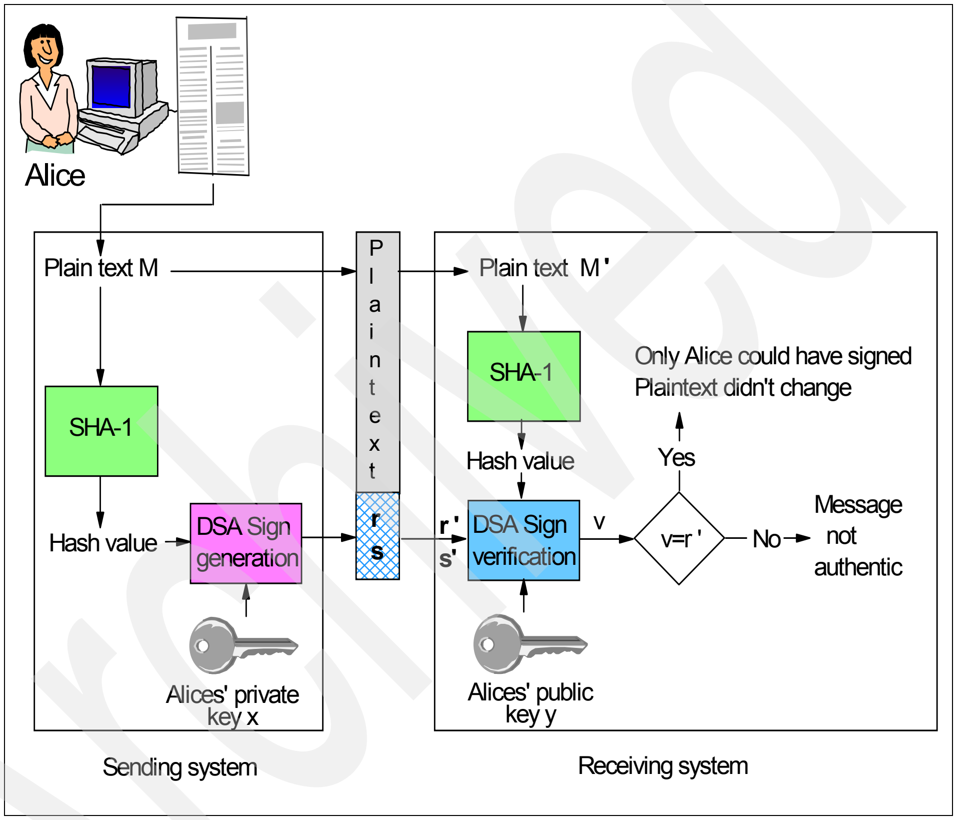


Figure 6-15 Verifying a DSA signature

To verify the signature, the verifier first checks to see that $0 < r' < q$ and $0 < s' < q$; if either condition is violated the signature should be rejected. If these two conditions are satisfied, the verifier computes:

$w = (s')^{-1} \bmod q$

$u1 = ((\text{SHA-1}(M'))w) \bmod q$

$u2 = ((r')w) \bmod q$

$v = ((g^{u1} y^{u2}) \bmod p) \bmod q$

If $v = r'$, then the signature is verified and the verifier can have high confidence that the received message was sent by the party holding the secret key x corresponding to y.

If v does not equal r', then the message may have been modified, the message may have been incorrectly signed by the signatory, or the message may have been signed by an impostor. The message should be considered invalid.

The ANSI X9.62 standard *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)* specifies a method of providing digital signatures that makes use of the properties of mathematical objects known as elliptic curves. Since CICS TS V3.1 does not support this method, we do not discuss it here.

6.6.2 Using RSA for digital signatures

When RSA cryptography is used to calculate a digital signature, the sender encrypts the message digest of the document with his or her own private key. Anyone with access to the public key of the signer can verify the signature.

Suppose Alice wants to send a signed document or message to Bob. She applies a hash function to the message, creating a message digest. She then encrypts the message digest with her private key, thereby creating the digital signature. (Since the message digest is usually considerably shorter than the original message, Alice saves a considerable amount of time when she encrypts the message digest rather than the message itself). Alice sends Bob the encrypted message digest (digital signature) and the message. Bob, upon receiving the message and signature, decrypts the signature with Alice's public key to recover the message digest. He then hashes the message with the same hash function Alice used and compares the result to the message digest decrypted from the signature. If they are exactly equal, the signature has been successfully verified and he can be confident the message did indeed come from Alice. If they are not equal, then the message either originated elsewhere or was altered after it was signed, and he rejects the message. See Figure 6-16.

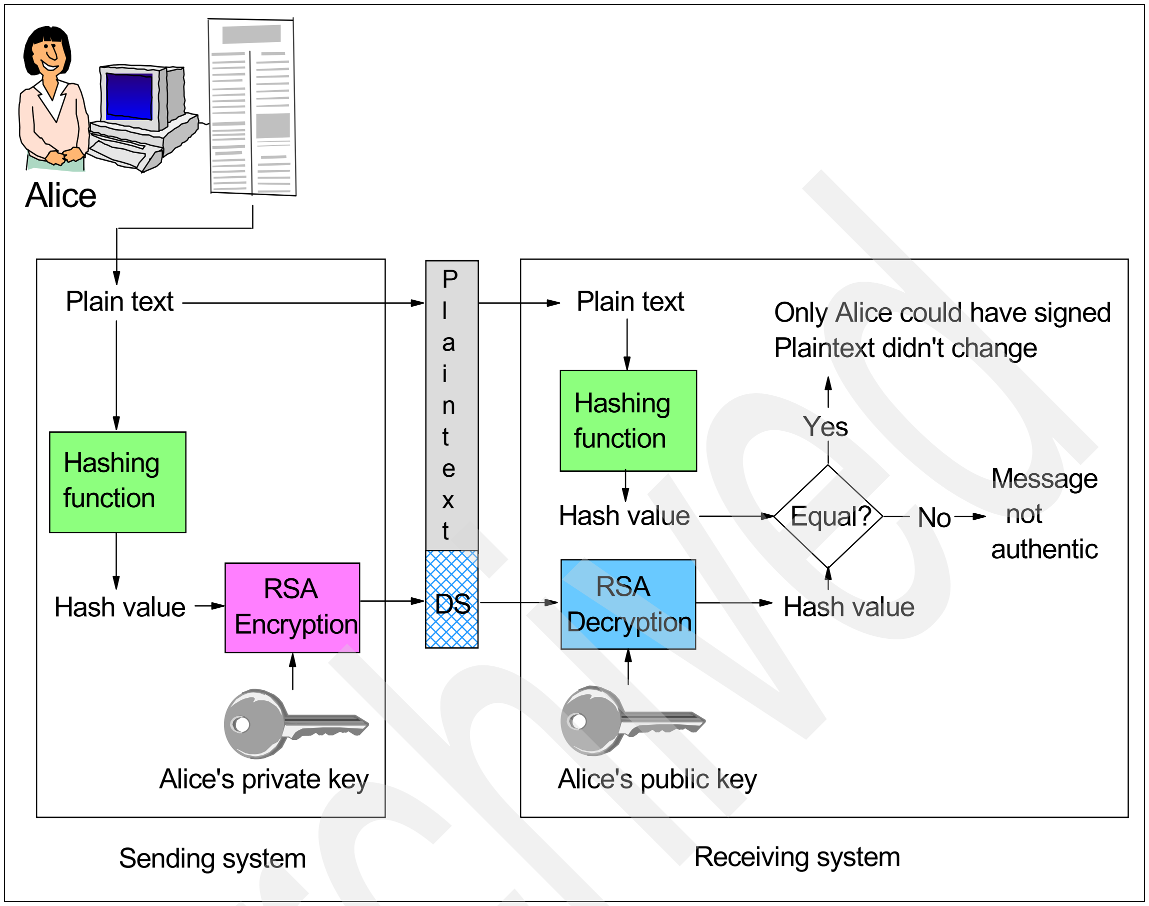


Figure 6-16 Creating and verifying a digital signature using RSA

In mathematical terminology, when Alice wants to send a message m to Bob, she applies a hash function H to the message m, creating a message digest h=H(m). She then creates a digital signature s by exponentiating: $s = h^d \bmod n$, where d and n are Alice's private key. She sends m and s to Bob. To verify the signature, Bob exponentiates and checks that the message digest h is recovered: $h = s^e \bmod n$, where e and n are Alice's public key.

In practice, the public exponent in the RSA algorithm is usually much smaller than the private exponent. This means that verification of a signature is faster than signing. This is desirable when a message will be signed by an individual only once, but the signature may be verified many times.

Note that the recipient of signed data can use a digital signature to prove to a third party that the signature was in fact generated by the signatory. This is known as non-repudiation since the signatory cannot, at a later time, repudiate the signature.

There is a potential problem with this type of digital signature. Alice not only signed the message she intended to sign, but she also signed all other messages that happen to hash to the same message digest. When two messages hash to the same message digest it is called a collision; the collision-free properties of hash functions are a necessary security requirement for most digital signature schemes. A hash function is secure if it is very time consuming, if at all possible, to figure out the original message given its digest.

In addition, someone could pretend to be Alice and sign documents with a key pair he claims is Alice's. To avoid scenarios such as this, there are digital documents called *certificates* that associate a person with a specific public key. We discuss digital certificates in 6.7, "Public key digital certificates" on page 180.

Suppose that Alice wishes to keep the contents of the document secret instead of sending the document in the clear as in Figure 6-16. In this case she may wish to sign the document, then encrypt it using Bob's public key. Bob will then need to decrypt the document using his private key and verify the signature on the recovered message using Alice's public key. See Figure 6-17.

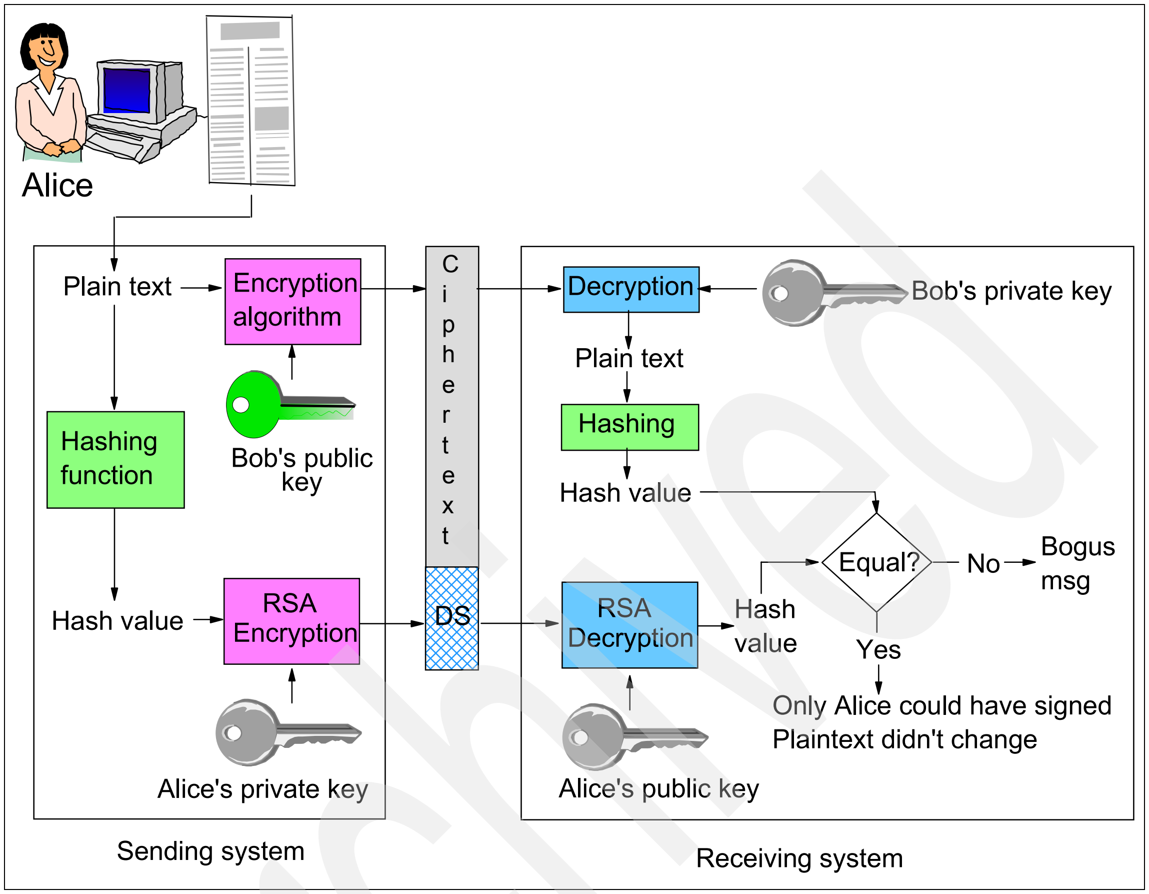


Figure 6-17 Creating and verifying a digital signature while encrypting the message

Alternatively, if it is necessary for intermediary third parties to validate the integrity of the message without being able to decrypt its content, a message digest can be computed on the encrypted message, rather than on its plaintext form.

6.6.3 Comparing RSA with DSA for digital signatures

In DSA, signature generation is faster than signature verification, whereas with the RSA algorithm, signature verification is very much faster than signature generation (if the public and private exponents, respectively, are chosen for this property, which is the usual case). It might be claimed that it is advantageous for signing to be the faster operation, but since in many applications a piece of digital information is signed once, but verified often, it may well be more advantageous to have faster verification.

6.7 Public key digital certificates

The tricky aspect of digital signatures is the trustworthy distribution of public keys, since the receiver requires a genuine copy of the sender's public key. This is provided by public key digital certificates.

A digital certificate is analogous to a passport in the following ways:

- Passports are issued by a trusted authority such as a government passport office. Digital certificates are issued by trusted authorities known as Certificate Authorities or CAs.
- A government passport office does not issue a passport unless the person who requests it has proven his identity and citizenship to the passport office. CAs have a responsibility to check the credentials provided in an application for a digital certificate. The CA might, for example, require the person who is requesting the certificate to appear in person and show a birth certificate.
- A passport certifies the bearer's name, address, and citizenship. A digital certificate establishes the subject's distinguished name (DN) and public key.
- Specialized equipment is used in the creation of a passport to make it very difficult to alter the information in it or to forge a passport altogether. CAs sign the digital certificates they issue with their private key.
- If other authorities, such as the border police in other countries, trust the authority that issued the passport, they implicitly trust the passport. If a Web user trusts a CA, he implicitly trusts digital certificates issued by the CA.
- Both passports and digital certificates are valid for a limited time.

Certificate issuance proceeds as follows. The requester generates a public and private key pair and then sends the public key to an appropriate CA with some proof of identification. The CA checks the identification and takes any other necessary steps to assure itself that the request really did come from the requester and that the public key was not modified in transit. Then the CA sends the requester a certificate which attests that the public key belongs to the requester.

In the following discussion of digital certificates we use the description of the version 3 format of a digital certificate as given in RFC 3280 *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*.

The certificate is a sequence of three required fields:

- `tbsCertificate`

  The `tbsCertificate` field contains the name of the subject of the certificate, a public key associated with the subject, the name of the issuer of the certificate, a validity period, and other associated information which we describe in Section 6.7.1, “tbsCertificate” on page 182.

- `signatureAlgorithm`

  The `signatureAlgorithm` field contains the identifier for the cryptographic algorithm used by the CA to sign this certificate. RFC 3279 *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile* lists the supported algorithms:

  – `md2WithRSAEncryption`

    This algorithm uses md2 for the hash function and RSA for the encryption algorithm.

  – `md5WithRSAEncryption`

  – `sha-1WithRSAEncryption`

  – `id-dsa-with-sha1`

    This algorithm uses SHA-1 for the hash function, and its uses the Digital Signature Algorithm.

  – `ecdsa-with-SHA1`

- `signatureValue`

  The `signatureValue` field contains a digital signature computed upon the `tbsCertificate`. The ASN.1 DER encoded `tbsCertificate` is used as the input to the signature function. This signature value is encoded as a BIT STRING and included in the signature field as shown in Figure 6-18.

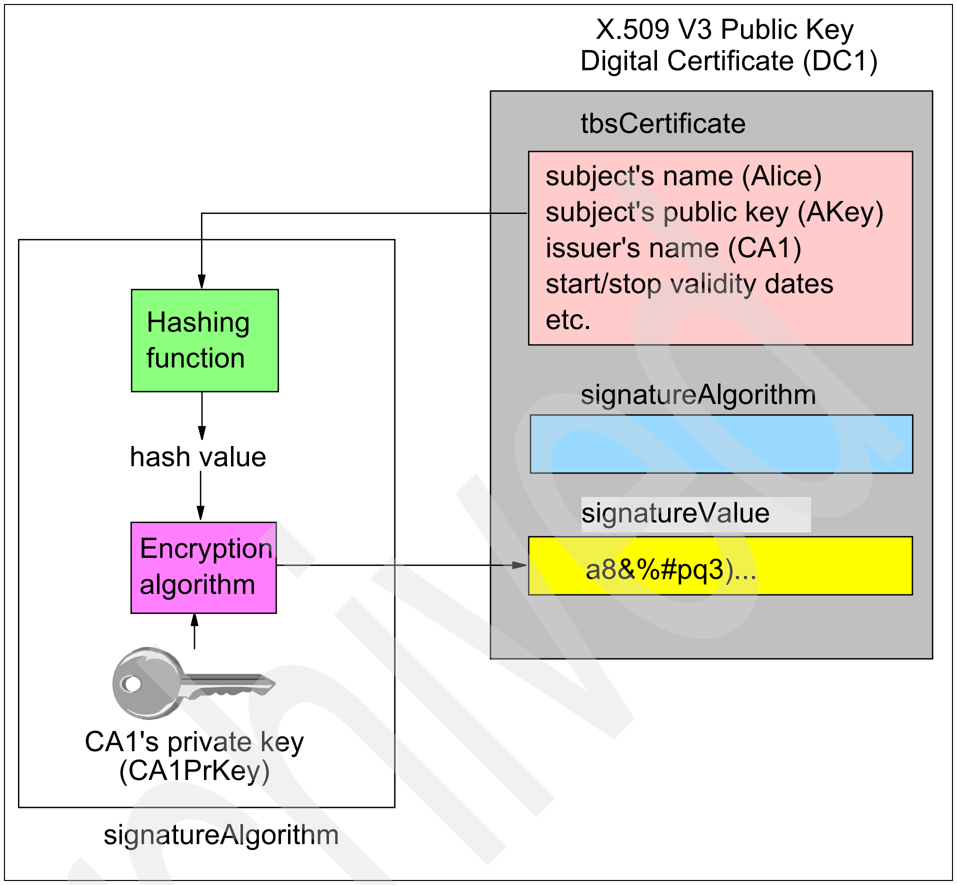


Figure 6-18 X.509 V3 public key digital certificate

By generating this signature, a CA certifies the validity of the information in the `tbsCertificate` field. In particular, the CA certifies the binding between the public key material and the subject of the certificate.

6.7.1 tbsCertificate

A `tbsCertificate` contains the following fields:

- `version`

  ITU-T X.509, which was first published in 1988 as part of the X.500 Directory recommendations, defines a standard certificate format. (ITU-T is the International Telecommunications Union; it was formerly known as CCITT and is a multinational union that provides standards for telecommunications equipment and systems.) The certificate format in the 1988 standard is called the version 1 format. When X.500 was revised in 1993, two more fields were added, resulting in the version 2 format. Experience gained in attempts to deploy RFC 1422 *Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management* revealed the need to develop a third version. In June, 1996, standardization of the basic version 3 format was completed. The value stored in the `version` field is one less than the version number; for example, when the version is 3, the value stored in the `version` field is 2.

- `serialNumber`

  The serial number is a positive integer assigned by the CA to the certificate. It is unique for each certificate issued by a CA; that is, the issuer name and serial number identify a unique certificate.

- `signature`

  This field contains the algorithm identifier for the algorithm used by the CA to sign the certificate. This field must contain the same algorithm identifier as the `signatureAlgorithm` field.

- `issuer`

  The `issuer` field identifies the entity that has signed and issued the certificate. It contains a distinguished name (DN). We explain what a distinguished name is in “Distinguished names” on page 185.

- `validity`

  The certificate validity period is the time interval during which the CA warrants that it will maintain information about the status of the certificate. The field is represented as a sequence of two dates:

  – `notBefore` - The date on which the certificate validity period begins.
  – `notAfter` - The date on which the certificate validity period ends.

  Both `notBefore` and `notAfter` can be encoded YYMMDDHHMMSSZ or YYYYMMDDHHMMSSZ.

- `subject`

  The `subject` field identifies the entity associated with the public key stored in the `subjectPublicKeyInfo` field. The `subject` field contains a DN.

  If the subject is a CA, then the `subject` field must contain a DN that matches the contents of the `issuer` field in all certificates issued by the subject CA.

- `subjectPublicKeyInfo`

  This field is used to carry the public key and identify the algorithm with which the key is used. RFC 3279 lists the following supported algorithm identifiers:

  – `rsaEncryption`

    When the `algorithmIdentifier` is `rsaEncryption`, the public key must be encoded as a sequence of two integers: the modulus n and the public exponent e.

  – `id-dsa`

    When the `algorithmIdentifier` is `id-dsa`, the public key must be encoded as the integer y.

  – `dhpublicnumber`

    This identifies the Diffie-Hellman key exchange algorithm. The public key is the integer y = g^x mod p. We discuss the Diffie-Hellman key exchange algorithm in Section 6.9.2, “The Diffie-Hellman key agreement protocol” on page 197.

  – `id-keyExchangeAlgorithm`

    This identifies the Key Exchange Algorithm (KEA), which is a key agreement algorithm. We do not discuss it further.

  – `id-ecPublicKey`

    When the `algorithmIdentifier` is `id-ecPublicKey`, the public key is intended for use in either the Elliptic Curve Digital Signature Algorithm (ECDSA) or the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm, neither of which is discussed further.

- `issuerUniqueId` (optional)

  This field is used to handle the possibility of reuse of issuer names over time. RFC 3280 recommends that names not be reused for different entities and that Internet certificates not make use of unique identifiers.

- `subjectUniqueId` (optional)

  This field is used to handle the possibility of reuse of subject names over time. RFC 3280 also recommends against the use of this field.

- `extensions (optional)`

  If present, this field is a sequence of one or more certificate extensions. The extensions defined for X.509 V3 certificates provide methods for associating additional attributes with users or public keys and for managing a certification hierarchy. We discuss a few of the standard extensions defined in RFC 3280 in Section 6.7.2, “Standard extensions for X.509 V3 digital certificates” on page 187.

Example 6-1 shows the decode of an X.509 certificate as found at:
`http://en.wikipedia.org/wiki/X.509`

Example 6-1 Sample X.509 certificate

```
Certificate:
   Data:
       Version: 1 (0x0)
       Serial Number: 7829 (0x1e95)
       Signature Algorithm: md5WithRSAEncryption
       Issuer: C=ZA, ST=Western Cape, L=Cape Town, O=Thawte Consulting cc,
               OU=Certification Services Division,
               CN=Thawte Server CA/Email=server-certs@thawte.com
       Validity
           Not Before: Jul 9 16:04:02 1998 GMT
           Not After : Jul 9 16:04:02 1999 GMT
       Subject: C=US, SP=Maryland, L=Pasadena, O=Brent Baccala,
                OU=FreeSoft, CN=www.freesoft.org/Email=baccala@freesoft.org
       Subject Public Key Info:
           Public Key Algorithm: rsaEncryption
           RSA Public Key: (1024 bit)
               Modulus (1024 bit):
                   00:b4:31:98:0a:c4:bc:62:c1:88:aa:dc:b0:c8:bb:
                   33:35:19:d5:0c:64:b9:3d:41:b2:96:fc:f3:31:e1:
                   66:36:d0:8e:56:12:44:ba:75:eb:e8:1c:9c:5b:66:
                   70:33:52:14:c9:ec:4f:91:51:70:39:de:53:85:17:
                   16:94:6e:ee:f4:d5:6f:d5:ca:b3:47:5e:1b:0c:7b:
                   c5:cc:2b:6b:c1:90:c3:16:31:0d:bf:7a:c7:47:77:
                   8f:a0:21:c7:4c:d0:16:65:00:c1:0f:d7:b8:80:e3:
                   d2:75:6b:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:b8:
                   e8:35:1c:9e:27:52:7e:41:8f:
               Exponent: 65537 (0x10001)
   Signature Algorithm: md5WithRSAEncryption
       93:5f:8f:5f:c5:af:bf:0a:ab:a5:6d:fb:24:5f:b6:59:5d:9d:
       92:2e:4a:1b:8b:ac:7d:99:17:5d:cd:19:f6:ad:ef:63:2f:92:
       ab:2f:4b:cf:0a:13:90:ee:2c:0e:43:03:be:f6:ea:8e:9c:67:
       d0:a2:40:03:f7:ef:6a:15:09:79:a9:46:ed:b7:16:1b:41:72:
       0d:19:aa:ad:dd:9a:df:ab:97:50:65:f5:5e:85:a6:ef:19:d1:
       5a:de:9d:ea:63:cd:cb:cc:6d:5d:01:85:b5:6d:c8:f3:d9:f7:
       8f:0e:fc:ba:1f:34:e9:96:6e:6c:cf:f2:ef:9b:bf:de:b5:22:
       68:9f
```

Distinguished names

Both the `issuer` field and the `subject` field of `tbsCertificate` must contain an X.520 Distinguished Name (DN). A DN is a sequence of Relative Distinguished Names (RDNs). An RDN™ has the form <attribute type> = <value>. Table 6-1 shows the string representations of common attribute types.

Table 6-1 Attribute types and their string representations

Attribute Type	String
countryName	C
organizationName	O
organizationalUnitName	OU
stateOrProvinceName	SP
localityName	L
commonName	CN

You can think of a DN as a unique name that unambiguously identifies a single entry in a directory information tree. Each RDN in a DN corresponds to a branch in the tree leading from the root of the tree to the directory entry. As shown in Figure 6-19, the distinguished name C=US, O=IBM, OU=IO, SP=NY, L=End, CN=Bob Herman describes Bob Herman, who works in the village of Endicott in the state of New York, USA, for the Integrated Operations unit of IBM; while the distinguished name C=FR, O=IBM, OU=S&D, SP=Her, L=MOP, CN=Nigel Williams describes Nigel Williams, who works in the city of Montpellier in the province of Herault, France, for the Sales and Distribution unit of IBM.

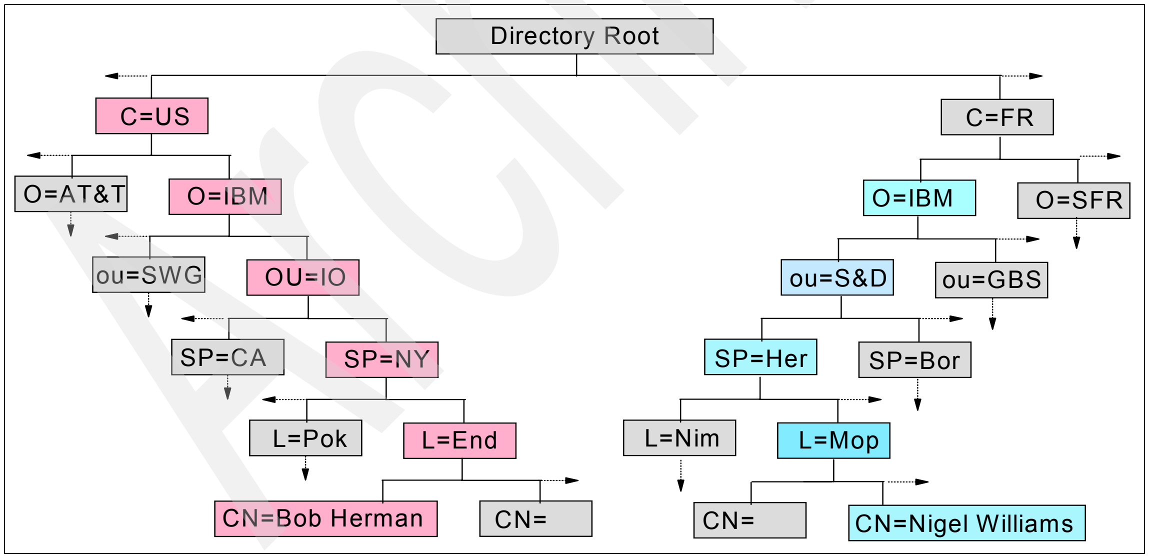


Figure 6-19 Distinguished names

6.7.2 Standard extensions for X.509 V3 digital certificates

RFC3280 defines sixteen standard extensions, but we limit our discussion here to only the most relevant ones; specifically, those supported by the RACF® RACDERT command that you can use to generate a digital certificate. These extensions are:

- Key usage

  The key usage extension defines the purpose of the subject public key contained in the certificate:

  – digitalSignature (0)

    The key is used with a digital signature mechanism to support security services other than certificate signing (bit 5) or certificate revocation list (CRL) signing (bit 6). Digital signature mechanisms are often used for entity authentication and data origin authentication with integrity. (We explain what a CRL is in Section 6.8, “Certificate revocation lists” on page 191.)

  – nonRepudiation (1)

    The key is used to verify digital signatures used to provide a non-repudiation service, which protects against the signing entity falsely denying some action, excluding certificate or CRL signing. In case of later conflict, a reliable third party may determine the authenticity of the signed data.

  – keyEncipherment (2)

    The key is used for key transport. For example, when an RSA key is to be used for key management, then this bit is set.

  – dataEncipherment (3)

    The key is used for enciphering user data, other than cryptographic keys.

  – keyAgreement (4)

    The key is used for key agreement. For example, when a Diffie-Hellman key is to be used for key management, then this bit is set.

  – keyCertSign (5)

    The key is used for verifying a signature on public key certificates.

  – cRLSign (6)

    The key is used for verifying a signature on a CRL.

  – encipherOnly (7)

    The meaning of the encipherOnly bit is undefined in the absence of the keyAgreement bit. When the encipherOnly bit is asserted and the keyAgreement bit is also set, the key may be used only for enciphering data while performing key agreement.

  – decipherOnly (8)

    The meaning of this bit is the same as the encipherOnly bit except that it applies to a decipher operation.

  The usage restriction might be employed when a key that could be used for more than one operation is to be restricted.

- Subject alternative name

  The subject alternative name extension allows additional identities to be bound to the subject of the certificate. Defined options include:

  – An Internet electronic mail address
  – A domain name system (DNS) name
  – An IP address
  – A uniform resource identifier (URI)

  The subject alternative name is considered to be definitively bound to the public key.

Example 6-2 shows the main options of a RACF RACDCERT command that you can use to generate a digital certificate.

Example 6-2 RACF command for generating a digital certificate

```
RACDCERT ID(userid) GENCERT
         SUBJECTSDN(
                    CN(‘common-name’)
                    T(‘title’)
                    OU(‘organizational-unit-name1’,...)
                    O(‘organization-name’)
                    L(‘locality’)
                    SP(‘state-or-province’)
                    C(‘country’) )
         SIZE(size-of-new-private-key-in-decimal-bits)
         NOTBEFORE( DATE(yyyy-mm-dd) TIME(hh:mm:ss) )
         NOTAFTER ( DATE(yyyy-mm-dd) TIME(hh:mm:ss) )
         WITHLABEL(‘label-name’)
         SIGNWITH( CERTAUTH|SITE LABEL(‘label-name’) )
         PCICC | ICSF | DSA
         KEYUSAGE( HANDSHAKE DATAENCRYPT DOCSIGN CERTSIGN)
         ALTNAME( IP(numeric-ip-address)
                  DOMAIN(‘internet-domain-name’)
                  EMAIL(‘email-address’)
                  URI(‘universal-resource-identifier’) )
```

The values that you specify for the KEYUSAGE parameter specify the values for the KeyUsage certificate extension as follows:

- HANDSHAKE

  The key facilitates identification and key exchange during security handshakes, such as SSL. RACF sets the digitalSignature and keyEncipherment indicators in the extension.
- DATAENCRYPT

  The key is used to encrypt data. RACF sets the dataEncipherment indicator in the extension.
- DOCSIGN

  The key is used to produce a legally binding signature. RACF sets the nonRepudiation indicator in the extension.
- CERTSIGN

  The key is used to sign other digital certificates and CRLs. RACF sets the keyCertSign and cRLSign indicators in the extension.

When you choose either PCICC or ICSF, the resulting private key is generated with the RSA algorithm and stored in the ICSF PKDS.

- If you choose PCICC, the key pair is generated using cryptographic hardware and the resulting private key is stored in the Integrated Cryptographic Services Facility (ICSF) Private Key Data Set (PKDS).
- If you choose ICSF, the key pair is generated using software and the resulting private key is stored in the ICSF PKDS.

If DSA is specified, the key pair is generated using software with DSA algorithm and the private key is stored in the RACF database as a non-ICSF DSA key. If you omit both, the key pair is generated using software and the private key is stored in the RACF database.

The ICSF PKDS is recommended for the storage of a private key associated with a digital certificate. ICSF ensures that private keys are encrypted under a master key and that access to them is controlled by profiles in the RACF general resource classes CSFKEYS and CSFSERV. See Chapter 7, “Crypto hardware and ICSF” on page 219 for a discussion of cryptographic hardware and ICSF.

6.7.3 Certification paths

In Figure 6-18 on page 182 certificate authority CA1 issued digital certificate DC1 to certify that public key AKey belongs to Alice. But how do we know that we can trust digital certificate DC1? Well, since DC1 is essentially the message tbsCertificate signed with CA1’s private key, we must verify the digital signature just like we verified the digital signature in Figure 6-16 on page 177. That is, if we hold an assured copy of CA1's public key, we must use it to proceed as shown in Figure 6-20.

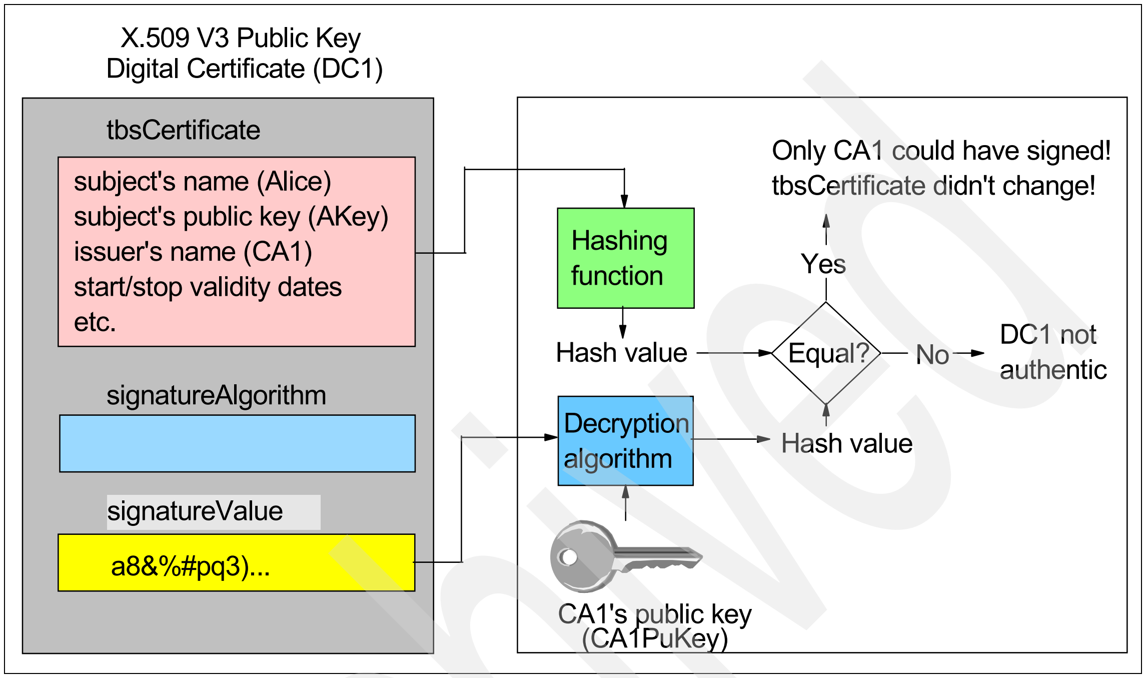


Figure 6-20 Verifying digital certificate

If we do not already hold an assured copy of CA1's public key, then we need a digital certificate signed by another CA to certify that CA1PuKey belongs to CA1. See Figure 6-21.

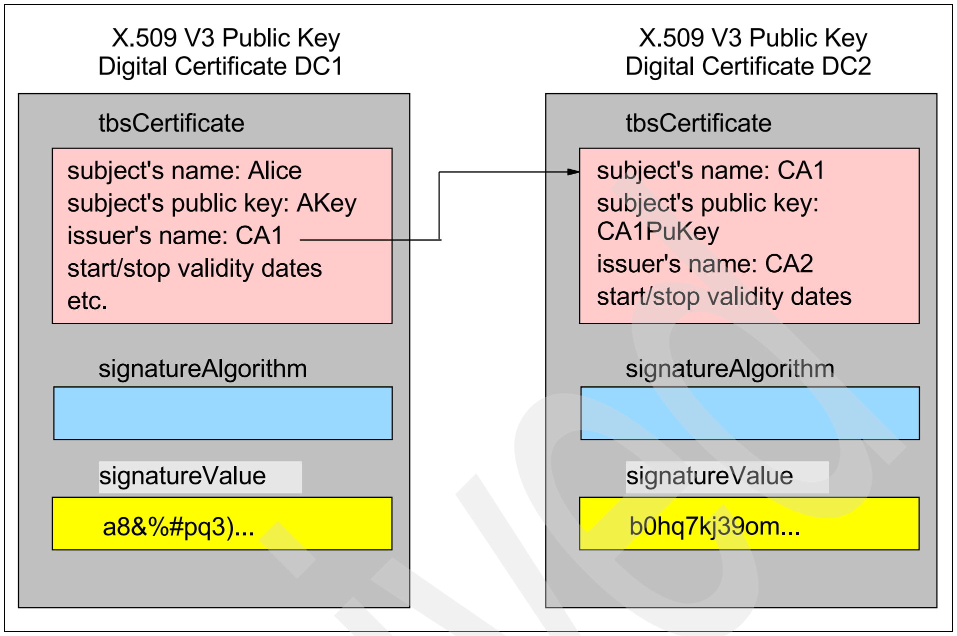


Figure 6-21 Certification path

In general, we need a sequence of n certificates, which satisfies the following conditions:

- Certificate 1 is the certificate to be validated.
- For all x in {1, 2,..., n-1}, the issuer of certificate x is the subject of certificate x+1.
- Certificate n is issued by a certificate authority that is trusted without a certificate from any other certifying authority. The certificate may be a self-signed certificate (one in which the CA uses its own private key to attest that the subject public key belongs to the CA).
- For all x in {1,2,...n} the certificate is valid at the time in question.

Such a sequence is called a *certification path*.

6.8 Certificate revocation lists

When a certificate is issued, it is expected to be in use for its entire validity period. However, various circumstances may cause a certificate to become invalid prior to the expiration of the validity period. Such circumstances include change of name, change of association between subject and CA (for example, an employee terminates employment with an organization), and compromise or suspected compromise of the corresponding private key. Under such circumstances, the CA needs to revoke the certificate.

RFC 3280 defines one method of certificate revocation. This method involves each CA periodically issuing a signed data structure called a *certificate revocation list* (CRL). A CRL is a time-stamped list identifying revoked certificates that is signed by a CA and made freely available in a public repository. Each revoked certificate is identified in a CRL by its certificate serial number. When a certificate-using system uses a certificate, that system not only checks the certificate signature and validity but also acquires a suitably-recent CRL and checks that the certificate serial number is not on that CRL. A new CRL is issued on a regular periodic basis. An entry is added to the CRL as part of the next update following notification of revocation.

Each CRL has a particular scope. The CRL scope is the set of certificates that could appear on a given CRL. For example, the scope could be "all certificates issued by CA X," "all CA certificates issued by CA X," or "all certificates issued by CA X that have been revoked for reasons of key compromise and CA compromise."

A complete CRL lists all unexpired certificates, within its scope, that have been revoked for one of the revocation reasons covered by the CRL scope. The CRL issuer might also generate delta CRLs. A delta CRL only lists those certificates, within its scope, whose revocation status has changed since the issuance of a referenced complete CRL (known as the base CRL).

A CRL is a sequence of three required fields:

- `tbsCertList`

  The `tbsCertList` is itself a sequence of required and optional fields:

  – `version` (optional)

    The `version` field describes the version of the encoded CRL. When the version is 2, the integer value for the field is 1.

  – `signature`

    The `signature` field contains the algorithm identifier for the algorithm used to sign the CRL.

  – `issuer`

    The `issuer` field contains an X.500 distinguished name (DN) that identifies the entity that has signed and issued the CRL.

  – `thisUpdate`

    This field indicates the issue date of this CRL.

- `nextUpdate`

  This field indicates the date by which the next CRL will be issued. The next CRL could be issued before the indicated date, but it will not be issued any later than the indicated date.

- `revokedCertificates` (optional)

  The revoked certificate list is optional to support the case where a CA has not revoked any unexpired certificates that it has issued. The `revokedCertificates` field is a sequence of three fields:

  - `userCertificate`

    This field contains the serial number of the revoked certificate. Certificates revoked by the CA are uniquely identified by the certificate serial number.

  - `revocationDate`

    This field contains the date on which the revocation occurred.

  - `crlEntryExtensions` (optional)

    We discuss some of the extensions for CRL entries in Section 6.8.1, "Extensions for entries in a CRL" on page 194.

- `crlExtensions` (optional)

  We discuss some of the extensions for CRLs in Section 6.8.2, "Extensions for a CRL" on page 195.

► `signatureAlgorithm`

The `signatureAlgorithm` field contains the algorithm identifier for the algorithm used by the CRL issuer to sign the CRL.

► `signatureValue`

The `signatureValue` field contains a digital signature computed upon the `tbsCertList`. The ASN.1 DER encoded `tbsCertList` is used as the input to the signature function. This signature value is encoded as a BIT STRING and included in the CRL `signatureValue` field.

You can find an example of a certificate revocation list at:

```
crl.geotrust.com/crls/secureca.crl
```

6.8.1 Extensions for entries in a CRL

The extensions for CRL entries, which are defined in RFC 3280, include the following:

- `reasonCode`

  This extension identifies the reason for the certificate revocation as follows:

  - `unspecified` (0)
  - `keyCompromise` (1)
  - `caCompromise` (2)
  - `affiliationChanged` (3)
  - `superseded` (4)
  - `cessationOfOperation` (5)
  - `certificateHold` (6)
  - `removeFromCRL` (8)
  - `privilegeWithdrawn` (9)
  - `aACompromise` (10)

  Apparently the `certificateHold` status is a reversible status that can be used to note the temporary invalidity of the certificate, for instance when the user is not sure if the private key has been lost. If, in this example, the private key was found again and nobody had access to it, the status can be reinstated, and the certificate is valid again, thus removing the certificate from further CRLs.

- `holdInstructionCode`

  This extension indicates the action to be taken after encountering a certificate that has been placed on hold:

  - `reject`

    Reject the certificate.

  - `callissuer`

    Call the certificate issuer or reject the certificate.

- `invalidityDate`

  This extension provides the date on which it is known or suspected that the private key was compromised or that the certificate otherwise became invalid. This date can be earlier than the revocation date in the CRL entry, which is the date on which the CA processed the revocation. When a revocation is first posted by a CRL issuer in a CRL, the invalidity date may precede the date of issue of earlier CRLs.

6.8.2 Extensions for a CRL

The CRL extensions defined in RFC 3280 include the following:

- `authorityKeyIdentifier`

  This extension provides a means of identifying the public key corresponding to the private key used to sign a CRL.

- `issuerAltName`

  This extension allows the following additional identities to be associated with the issuer of the CRL: an electronic mail address, a DNS name, an IP address, and a URI.

- `cRLNumber`

  This extension conveys a monotonically increasing sequence number for a given CRL scope and CRL issuer. It allows users to easily determine when a particular CRL supersedes another CRL. CRL numbers also support the identification of complementary complete CRLs and delta CRLs.

- `issuingDistributionPoint`

  This extension identifies the CRL distribution point and scope for a particular CRL and indicates whether the CRL covers revocation for end entity certificates only, CA certificates only, attribute certificates only, or a limited set of reason codes.

- `freshestCRL`

  This extension identifies how delta CRL information for this complete CRL is obtained.

- `deltaCRLIndicator`

  This extension identifies a CRL as being a delta CRL. Delta CRLs contain updates to revocation information previously distributed, rather than all the information that would appear in a complete CRL. The use of delta CRLs can sometimes reduce network load and processing time. The extension contains the number of the base CRL, that is, it contains the number that identifies the CRL, complete for a given scope, that was used as the starting point in the generation of this delta CRL.

6.8.3 Security considerations when using digital certificates

RFC 3280 advises users to consider the following points when using digital certificates:

- The procedures performed by CAs to validate the binding of the subject's identity to their public key greatly affect the confidence that ought to be placed in the certificate. Different CAs may issue certificates with varying levels of identification requirements. One CA may insist on seeing a driver's license, another may want the certificate request form to be notarized, yet another may want fingerprints of anyone requesting a certificate.

Relying parties might wish to review the CA's certificate practice statement in order to avoid situations such as the following. Suppose Mallory wishes to impersonate Alice. If Mallory can convincingly sign messages as Alice, he can send a message to Alice's bank saying "I wish to withdraw $10,000 from my account. Send me the money." To carry out this attack, Mallory generates a key pair and sends the public key to a CA saying "I'm Alice. Here is my public key. Please send me a certificate." If the CA is fooled and sends him such a certificate, he can then fool the bank.

- The use of a single key pair for both signature and other purposes is strongly discouraged. Use of separate key pairs for signature and key management provides several benefits to the users. The ramifications associated with loss or disclosure of a signature key are different from loss or disclosure of a key management key. Using separate key pairs permits a balanced and flexible response.
- The protection afforded private keys is a critical security factor. Failure of users to protect their private keys will permit an attacker to masquerade as them, or decrypt their personal information.
- The availability and freshness of revocation information affects the degree of assurance that ought to be placed in a certificate. If revocation information is untimely or unavailable, the assurance associated with the binding is clearly reduced.
- The certification path validation algorithm depends on the certain knowledge of the public keys (and other information) about one or more trusted CAs. The decision to trust a CA is an important decision as it ultimately determines the trust afforded a certificate.
- The binding between a key and certificate subject cannot be stronger than the cryptographic module implementation and algorithms used to generate the signature. Short key lengths or weak hash algorithms will limit the utility of a certificate.

6.9 Key agreement protocols

A *key agreement protocol*, also called a *key exchange protocol*, is a protocol that allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a secret key algorithm. We discuss two key agreement protocols: RSA and Diffie-Hellman.

6.9.1 The RSA key agreement protocol

In Section 6.3, "Public key (or asymmetric) cryptography" on page 162 we noted that public key cryptography can be used to safely transmit a secret key. When Alice wishes to send a secret key to Bob, she looks up Bob's public key in a directory, uses it to encrypt the secret key and sends it off to Bob. Bob then uses his private key to decrypt the secret key and read it. No one listening in can decrypt the secret key.

Since RSA is a public key cryptosystem, we can use RSA to safely transmit a secret key. See Figure 6-22.

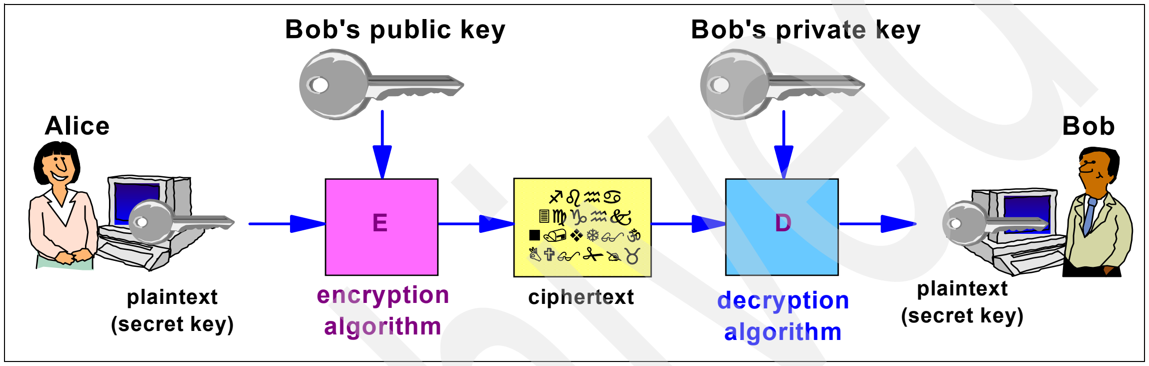


Figure 6-22 RSA key agreement protocol

6.9.2 The Diffie-Hellman key agreement protocol

The Diffie-Hellman key agreement protocol was first published by Whitfield Diffie and Martin Hellman in 1976. The protocol has two system parameters *p* and *g*. They are both public and can be used by all the users in a system.

- Parameter p is a prime number.
- Parameter g (usually called a generator) is an integer less than p, with the following property: for every integer *n* between 1 and p-1 inclusive, there is a power *k* of g such that $n = g^k \bmod p$.

For example, if p = 7, then g = 3 is a generator because: $1 = 3^0 \bmod 7$, $2 = 3^2 \bmod 7$, $3 = 3^1 \bmod 7$, $4 = 3^4 \bmod 7$, $5 = 3^5 \bmod 7$, and $6 = 3^3 \bmod 7$.

Suppose Alice and Bob want to agree on a shared secret key using the Diffie-Hellman key agreement protocol. They proceed as follows:

- They agree upon a prime number p and a generator g.
- Alice generates a random private integer *a* and then derives her public value $g^a \bmod p$.

- Alice sends her public value to Bob.
- Bob generates a random private integer *b* and then derives his public value g^b mod p.
- Bob sends his public value to Alice.
- Alice computes $(g^b)^a$ mod p.
- Bob computes $(g^a)^b$ mod p.

Since $(g^b)^a$ mod p = g^{ba} mod p = g^{ab} mod p = $(g^a)^b$ mod p, Bob and Alice now have a shared secret key.

As an example, suppose that Alice and Bob agree to use a prime number p = 23 and a generator g = 5.

- Suppose Alice chooses a secret integer a = 6. She then computes her public value 5^6 mod 23 = 8.
- Alice sends her public value 8 to Bob.
- Suppose Bob chooses a secret integer b = 15. He then computes his public value 5^{15} mod 23 = 19
- Bob sends his public value 19 to Alice.
- Alice computes 19^6 mod 23 = 2.
- Bob computes 8^{15} mod 23 = 2.

Alice and Bob now have the shared secret key 2.

The Diffie-Hellman key agreement protocol as just described is vulnerable to a man-in-the-middle attack. In this attack, Eve (for eavesdropper) intercepts Alice's public value and sends her own public value to Bob. When Bob transmits his public value, Eve substitutes it with her own and sends it to Alice. Eve and Alice thus agree on one shared key and Eve and Bob agree on another shared key. After this exchange, Eve simply decrypts any messages sent out by Alice or Bob, and then reads and possibly modifies them before re-encrypting with the appropriate key and transmitting them to the other party. This vulnerability is present because the protocol does not *authenticate* the participants. The protocol as described is sometimes called anonymous Diffie-Hellman.

The authenticated Diffie-Hellman key agreement protocol, or *Station-to-Station* (STS) protocol, was presented by Diffie, van Oorschot, and Wiener in 1992. Prior to execution of this protocol, Alice and Bob each obtain a public/private key pair and a certificate for the public key; they also agree upon the two system parameters p and g. The protocol then proceeds as follows:

1. Alice generates a random number a and computes and sends g^a mod p to Bob.

2. Bob generates a random number b and computes g^b mod p.
3. Bob computes the shared secret key K = $(g^a)^b$mod p.
4. Bob concatenates the exponentials (g^b mod p, g^a mod p) (order is important), signs them using his private key B, and then encrypts them with K. He sends the ciphertext along with his own exponential g^b mod p to Alice.
5. Alice computes the shared secret key K = $(g^b)^a$ mod p.
6. Alice decrypts (g^b mod p , g^a mod p) using the shared secret key K and verifies Bob's signature using Bob's public key.
7. Alice concatenates the exponentials (g^a mod p , g^b mod p) (order is important), signs them using her private key A, and then encrypts them with K. She sends the ciphertext to Bob.
8. Bob decrypts and verifies Alice's signature.

Alice and Bob are now mutually authenticated and have a shared secret. This secret, K, can then be used to encrypt further communication.

6.10 Transport Layer Security (TLS) 1.0 protocol

The primary goal of the Transport Layer Security (TLS) protocol is to provide privacy (confidentiality) and data integrity between two applications communicating over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

At the time we are writing this book, there are two versions of the TLS protocol. The Internet Society's Request For Comments (RFC) 2246 specified the TLS 1.0 protocol in January, 1999. RFC 4346 specified the TLS 1.1 protocol in April, 2006. CICS TS V3.1 added support for the TLS protocol; however, since CICS TS V3.1 became generally available before TLS 1.1, CICS TS V3.1 supports only TLS 1.0. Therefore, we limit our discussion of TLS to TLS 1.0.

TLS 1.0 is based on the SSL 3.0 Protocol Specification as published by Netscape. TLS provides the following enhancements over SSL 3.0:

- Key-Hashing for Message Authentication

  TLS uses Key-Hashing for Message Authentication Code (HMAC), which ensures that a record cannot be altered while travelling over an open network such as the Internet. SSL Version 3.0 also provides keyed message authentication, but HMAC is considered more secure than the Message Authentication Code (MAC) function that SSL 3.0 uses.

- Enhanced Pseudorandom Function (PRF)

  PRF is used for generating key data. In TLS, the PRF is defined with the HMAC. The PRF uses two hash algorithms in a way that guarantees its security. If either algorithm is exposed then the data remains secure as long as the second algorithm is not exposed.

- Improved finished message verification

  Both TLS 1.0 and SSL 3.0 provide a finished message to both endpoints that authenticates that the exchanged messages were not altered. However, TLS bases this finished message on the PRF and HMAC values, which is more secure than SSL 3.0.

- Consistent certificate handling

  Unlike SSL 3.0, TLS specifies the type of certificate which must be exchanged between TLS implementations.

- Specific alert messages

  TLS provides more specific and additional alerts to indicate problems that either session endpoint detects. TLS also documents when certain alerts should be sent.

The differences between TLS 1.0 and SSL 3.0 are significant enough that TLS 1.0 and SSL 3.0 do not interoperate (although TLS 1.0 does incorporate a mechanism by which a TLS implementation can back down to SSL 3.0).

6.10.1 TLS overview

The TLS protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. At the lowest level, layered on top of some reliable transport protocol (for example, TCP), is the TLS Record Protocol. The TLS Record Protocol provides connection security that has two basic properties:

- The connection is private. Secret key cryptography is used for data encryption. The keys for this secret key encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol).
- The connection is reliable. Message transport includes a message integrity check using a keyed MAC (HMAC). Secure hash functions (for example, SHA-1 or MD5) are used for MAC computations.

The TLS Handshake Protocol operates on top of the TLS Record Protocol and allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol (such as http) transmits or receives its first byte of data. The TLS Handshake Protocol provides connection security that has three basic properties:

- The peer's identity can be authenticated using public key cryptography.
- The negotiation of a shared secret is secure: the negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.
- The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.

The Record Protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher level clients. See Figure 6-23, which shows a client sending a message to a server.

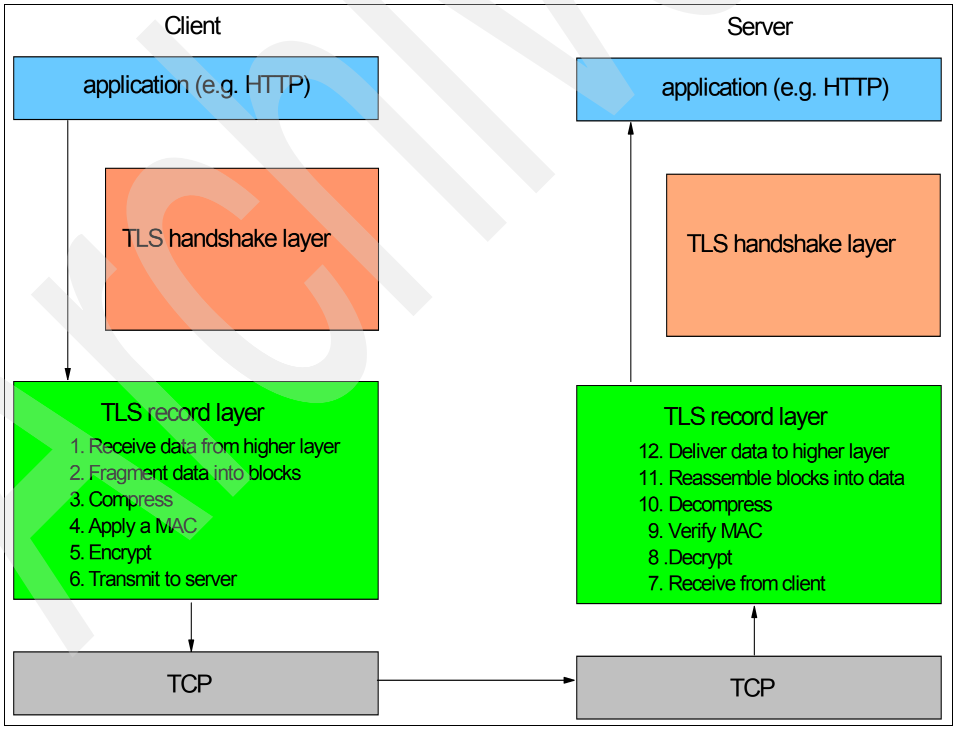


Figure 6-23 Overview of TLS

The TLS Handshake Protocol consists of a suite of three sub-protocols:

- Change cipher spec protocol
- Alert protocol
- Handshake protocol

Before we examine the Handshake protocol, we explain what a *cipher suite* is.

6.10.2 Cipher suites

One dictionary defines a *suite* as “a group of things forming a unit or constituting a collection”. A *cipher suite* then is a collection of cipher algorithms. More specifically, RFC 2246 defines a cipher suite as a collection consisting of one key exchange algorithm, one encryption algorithm, and one hash algorithm.

The hash algorithm must come from the following set:

- NULL (no hash algorithm)
- MD5
- SHA (meaning SHA-1)

The encryption algorithm must come from the following set:

- NULL (no encryption)
- IDEA_CBC

  IDEA is a 64-bit block cipher designed by Xuejia Lai and James Massey; it uses a 128 bit key. IDEA_CBC is IDEA running in cipher block chaining mode.
- RC2_CBC_40

  RC2 is a variable key-size block cipher designed by Ronald Rivest for RSA Security; it uses a 64-bit block size. RC2_CBC_40 is RC2 running with a 40-bit key in cipher block chaining mode. (“RC” stands for “Ron’s Code” or “Rivest’s Cipher”.)
- RC4_40

  RC4 is a variable key-size stream cipher designed by Rivest for RSA Security. RC4_40 is RC4 running with a 40-bit key.
- RC4_128

  RC4_128 is RC4 running with a 128-bit key. When RFC 2246 was published, RC4_40 was “exportable” but RC4_128 was not. (For many years, the U.S. government did not approve export of cryptographic products unless the key size was strictly limited.)

- DES40_CBC

  DES40_CBC is DES running with a 40-bit key in cipher block chaining mode.
- DES_CBC

  DES_CBC is DES running with a 56-bit key in cipher block chaining mode. When RFC 2246 was published, DES40_CBC was exportable but DES_CBC was not.
- 3DES_EDE_CBC

  3DES_EDE_CBC is TDEA running in cipher block chaining mode. The first use of DES is for encryption, the second for decryption, and the third for encryption.

The key exchange algorithm must come from the following set:

- DHE_DSS
- DHE_DSS_EXPORT
- DHE_RSA
- DHE_RSA_EXPORT
- DH_anon
- DH_anon_EXPORT
- DH_DSS
- DH_DSS_EXPORT
- DH_RSA
- DH_RSA_EXPORT
- NULL
- RSA
- RSA_EXPORT

DH denotes key exchange algorithms in which the server's certificate contains the Diffie-Hellman parameters signed by the certificate authority. DHE denotes ephemeral Diffie-Hellman, where the Diffie-Hellman parameters are signed by a DSS or RSA certificate, which in turn has been signed by the CA. The signing algorithm used is specified after DH or DHE.

DH_anon indicates completely anonymous Diffie-Hellman communications in which neither party is authenticated. RSA indicates that the server should provide an RSA certificate that can be used for key exchange. (An RSA certificate is an X.509 certificate that has been signed by using the RSA algorithm.)

RFC 2246 assigns names to the 27 cipher suites which contain an acceptable combination of algorithms from the sets identified previously. The names have the form:

TLS_*key-exchange-algorithm*_WITH_*encryption-algorithm_hash-algorithm*

For example, the cipher suite TLS_RSA_WITH_DES_CBC_SHA contains the RSA key exchange algorithm, the DES_CBC encryption algorithm, and the SHA hash algorithm. The cipher suite TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA contains the DH_DSS key exchange algorithm, the 3DES_EDE_CBC encryption algorithm, and the SHA hash algorithm.

The TLS 1.0 protocol seeks to provide a framework into which new public key and secret key encryption methods can be incorporated. This prevents the need to create a new protocol (which would risk the introduction of possible new weaknesses). The TLS 1.0 protocol allows additional cipher suites to be registered by publishing an RFC that specifies the cipher suite. Indeed, several such RFCs have been published, including the following:

- RFC 2712 *Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)*
- RFC 3268 *Advanced Encryption Standard (AES) Ciphersuites for Transport Layer Security (TLS)*
- RFC 4132 *Addition of Camellia Cipher Suites to Transport Layer Security (TLS)*
- RFC 4162 *Addition of SEED Cipher Suites to Transport Layer Security (TLS)*
- RFC 4279 *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*
- RFC 4492 *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*

These RFCs and RFC 2246 assign a number to each cipher suite. Table 6-2 shows the numbers assigned to the cipher suites supported by Cryptographic Services System SSL or Cryptographic Services Security Level 3, or both, in z/OS V1.4.

Table 6-2 Cipher suites supported by System SSL in z/OS V1.4

Number	Name
01	TLS_RSA_WITH_NULL_MD5
02	TLS_RSA_WITH_NULL_SHA
03	TLS_RSA_EXPORT_WITH_RC4_40_MD5
04	TLS_RSA_WITH_RC4_128_MD5
05	TLS_RSA_WITH_RC4_128_SHA

Number	Name
06	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
09	TLS_RSA_WITH_DES_CBC_SHA
0A	TLS_RSA_WITH_3DES_EDE_CBC_SHA
2F	TLS_RSA_WITH_AES_128_CBC_SHA
35	TLS_RSA_WITH_AES_256_CBC_SHA

Cryptographic Services System SSL or Cryptographic Services Security Level 3, or both, in z/OS V1.6, z/OS V1.7, and z/OS V1.8 support the cipher suites shown in Table 6-2 plus the additional cipher suites shown in Table 6-3.

Table 6-3 Cipher suites supported by System SSL in z/OS V1.6, V1.7, and V1.8

Number	Name
0C	TLS_DH_DSS_WITH_DES_CBC_SHA
0D	TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
0F	TLS_DH_RSA_WITH_DES_CBC_SHA
10	TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
12	TLS_DHE_DSS_WITH_DES_CBC_SHA
13	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
15	TLS_DHE_RSA_WITH_DES_CBC_SHA
16	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
30	TLS_DH_DSS_WITH_AES_128_CBC_SHA
31	TLS_DH_RSA_WITH_AES_128_CBC_SHA
32	TLS_DHE_DSS_WITH_AES_128_CBC_SHA
33	TLS_DHE_RSA_WITH_AES_128_CBC_SHA
36	TLS_DH_DSS_WITH_AES_256_CBC_SHA
37	TLS_DH_RSA_WITH_AES_256_CBC_SHA
38	TLS_DHE_DSS_WITH_AES_256_CBC_SHA
39	TLS_DHE_RSA_WITH_AES_256_CBC_SHA

In “Defining a TCPIPSERVICE resource for SSL” on page 244 we explain how you can control which cipher suites are used by CICS with the CIPHERS attribute of the TCPIPSERVICE resource definition.

6.10.3 Alert protocol

Error handling in the TLS Handshake protocol is very simple. When an error is detected, the detecting party sends a message called an *alert message* to the other party. An alert message conveys the severity of the message and a description of the alert.

The severity of the message must be one of the following:

- `warning` (1)
- `fatal` (2)

Upon transmission or receipt of a `fatal` alert message, both parties immediately close the connection. Servers and clients are required to forget any session identifiers, keys, and secrets associated with a failed connection.

The description of the alert must be one of the following:

- `close_notify` (0)
- `unexpected_message` (10)
- `bad_record_mac` (20)
- `decryption_failed` (21)
- `record_overflow` (22)
- `decompression_failure` (30)
- `handshake_failure` (40)
- `bad_certificate` (42)
- `unsupported_certificate` (43)
- `certificate_revoked` (44)
- `certificate_expired` (45)
- `certificate_unknown` (46)
- `illegal_parameter` (47)
- `unknown_ca` (48)
- `access_denied` (49)
- `decode_error` (50)
- `decrypt_error` (51)

- `export_restriction` (60)
- `protocol_version` (70)
- `insufficient_security` (71)
- `internal_error` (80)
- `user_canceled` (90)
- `no_renegotiation` (100)

RFC 2246 provides a short explanation of each of these descriptions.

The `user-canceled` and `no-renegotiation` alerts carry a level of `warning`. The sender may determine at its discretion whether the following alerts are fatal or not: `bad_certificate`, `unsupported_certificate`, `certificate_revoked`, `certificate_expired`, `certificate_unknown`, and `decrypt_error`. The remaining alerts always carry a level of `fatal`.

6.10.4 Handshake protocol

When a TLS client and server first start communicating, they agree on which version of the TLS protocol they will use, select a cipher suite, optionally authenticate each other, and use public key encryption techniques to generate shared secrets.

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on a cipher suite and a compression algorithm, exchange random values, and check for session resumption.
- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.
- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.
- Generate a master secret from the premaster secret and exchanged random values.
- Provide security parameters to the record layer as shown in Figure 6-24.
- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

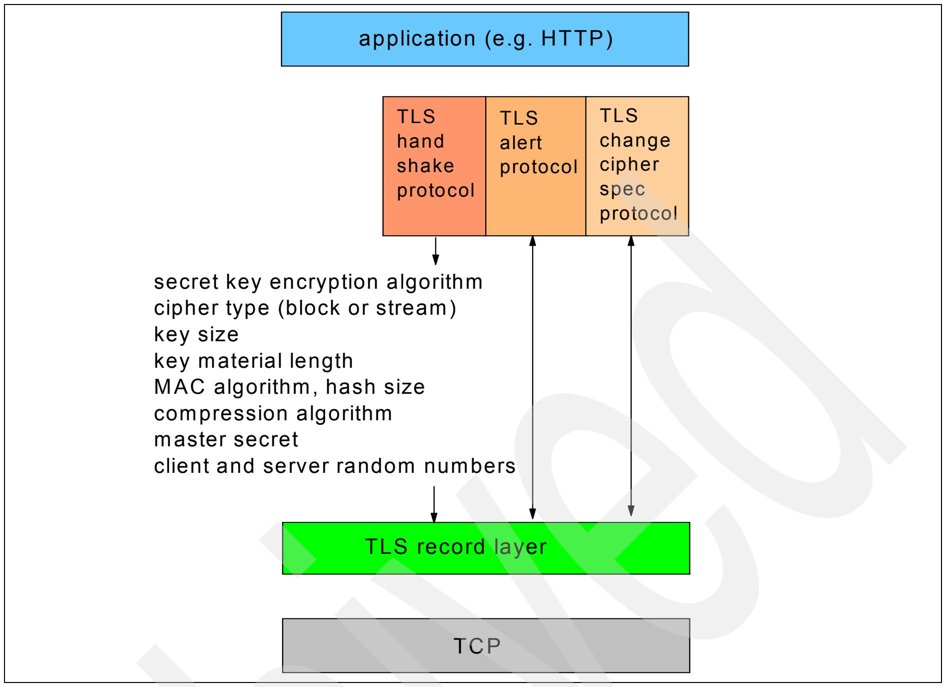


Figure 6-24 Handshake protocol passes security parameters to the record layer

Starting a new session

When the client and server wish to start a new session, they begin by exchanging hello messages as shown in Figure 6-25.

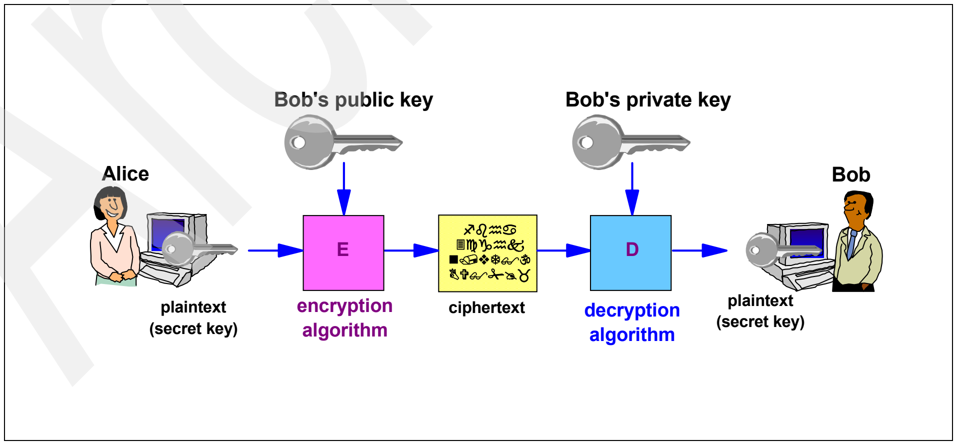


Figure 6-25 Starting a new TLS session(1): Establishing algorithms

The server can send the `hello_request` message at any time. It is a simple notification that the client should begin the negotiation process anew by sending a `client_hello` message when convenient.

The `client_hello` message includes the following parameters:

- The version of the TLS protocol by which the client wishes to communicate during this session. This should be the highest valued version supported by the client. For TLS 1.0 the version should be 3.1. (The version value 3.1 is historical: TLS version 1.0 is a minor modification to the SSL 3.0 protocol, which bears the version value 3.0.)
- The current time and date according to the client's internal clock, followed by 28 bytes generated by a secure random number generator.
- A list of the cipher suites supported by the client in order of the client's preference (favorite choice first). Recall that each cipher suite contains a key exchange algorithm, a secret key encryption algorithm (including secret key length), and a MAC algorithm.
- A list of the compression methods supported by the client, sorted by client preference.

The server will send a `server_hello` message in response to a `client_hello` message when it is able to find an acceptable set of algorithms. If it cannot find such a match, it will respond with a handshake failure alert. The `server_hello` message includes the following parameters:

- Either the TLS protocol version suggested by the client or the highest TLS protocol version supported by the server, whichever is lower.
- The current time and date according to the server's internal clock, followed by 28 bytes generated by a secure random number generator.
- The identity of the session corresponding to this connection. The actual contents of the sessionID are defined by the server.
- The single cipher suite selected by the server from the list supplied by the client.
- The single compression algorithm selected by the server from the list supplied by the client.

Thus the `client_hello` and `server_hello` messages establish the following connection attributes: the TLS protocol version, the sessionID, the key exchange algorithm, the secret key encryption algorithm, the key length for the secret key encryption algorithm, and the compression method. Additionally, two random values are generated and exchanged: `client_hello.random` and `server_hello.random`. Comparing this list of items with the list of items shown in Figure 6-24 on page 208, which the TLS Handshake Protocol must pass to the

Record layer, we see that the TLS Handshake Protocol must still come up with a *master secret*.

The master secret is generated by using, among other things, a pre-master secret. The general goal of the key exchange process shown in Figure 6-26 is to create a pre-master secret known to the communicating parties and not to attackers. Note that the italicized lines in Figure 6-26 represent actions taken by the client and server rather than protocol messages.

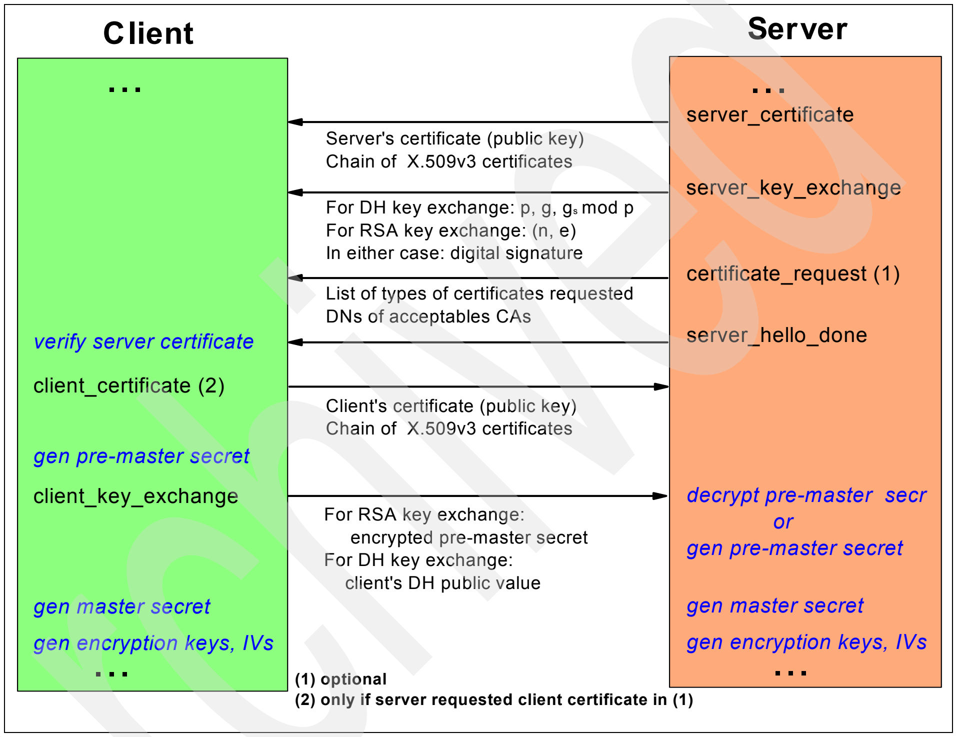


Figure 6-26 Starting a new TLS session(2): Establishing the pre-master secret

The `server_certificate` message sends a chain of X.509v3 certificates. The server's certificate must come first in the chain. Each following certificate must directly certify the one preceding it. The server's certificate must contain a key that matches the key exchange method that was specified in the negotiated cipher suite; see Table 6-4.

Table 6-4   *Key exchange methods and corresponding certificate key types*

Key exchange method of negotiated cipher suite	**Type of key in server certificate**
RSA	RSA public key; the keyUsage field of the certificate must allow the key to be used for encryption.
RSA_EXPORT	RSA public key of length greater than 512 bits which can be used for signing, or a key of 512 bits or shorter which can be used for either encryption or signing.
DHE_DSS	DSS public key.
DHE_DSS_EXPORT	DSS public key.
DHE_RSA	RSA public key which can be used for signing.
DHE_RSA_EXPORT	RSA public key which can be used for signing.
DH_DSS	Diffie_Hellman key. The algorithm used to sign the certificate should be DSS.
DH_RSA	Diffie_Hellman key. The algorithm used to sign the certificate should be RSA.

Note: At the time RFC 2246 was written, United States export restrictions limited RSA keys used for encryption to 512 bits, but did not place any limit on lengths of RSA keys used for signing operations.

The `server_key_exchange` message is sent by the server only when the `server_certificate` message does not contain enough data to allow the client to exchange a pre-master secret. This is true for the following key exchange methods: RSA_EXPORT (if the public key in the server certificate is longer than 512 bits), DHE_DSS, DHE_DSS_EXPORT, DHE_RSA, DHE_RSA_EXPORT, and DH_anon.

The `server_key_exchange` message conveys cryptographic information to allow the client to confidentially communicate the pre-master secret to the server: either an RSA public key with which to encrypt the pre-master secret, or a Diffie-Hellman public key with which the client can complete a key exchange (with the result being the pre-master secret).

When the key exchange method is RSA_EXPORT, the `server_key_exchange` message includes the following parameters:

- The modulus *n* of the server's temporary RSA key.

  According to US export law at the time RFC 2246 was written, RSA moduli larger than 512 bits could not be used for key exchange in software exported from the US. This message allows the larger RSA keys encoded in certificates to be used to sign temporary shorter RSA keys.
- The public exponent *e* of the server's temporary RSA key.
- A 36 byte structure of two hashes (one SHA-1 and one MD5), which has been signed with the server's private key. The SHA-1 hash takes as input the concatenation of `client_hello.random`, `server_hello.random`, and (*n*,*e*); it produces 20 bytes of output. The MD5 hash takes the same input and produces 16 bytes of output.

When the key exchange method is DHE_DSS, DHE_DSS_EXPORT, DHE_RSA, or DHE_RSA_EXPORT, the `server_key_exchange` message includes the following parameters:

- The prime modulus p used for the Diffie-Hellman operation.
- The generator g used for the Diffie-Hellman operation.
- The server's Diffie-Hellman public value g^s mod p.
- Two integers r and s produced as follows. An SHA-1 hash takes as input the concatenation of `client_hello.random`, `server_hello.random`, p, g, and g^s mod p; it produces 20 bytes of output. The 20 bytes are run through the Digital Signature Algorithm.

A non-anonymous server can optionally request a certificate from the client. The `certificate_request` message includes the following parameters:

- A list of the types of certificates requested, sorted in order of the server's preference
- A list of the distinguished names (DNs) of acceptable certificate authorities (CAs)

The server sends the `server_hello_done` message to indicate that it is done sending messages to support the key exchange, and the client can proceed with its phase of the key exchange. Upon receipt of this message, the client should verify that the server provided a valid certificate and that the certificate has not expired or been revoked. The client should also check that the server hello parameters are acceptable.

The `client_certificate` message is the first message the client can send after receiving the `server_hello_done` message. The client only sends the client_certificate message if the server requests a certificate. If the client does not have a suitable certificate to send to the server, it sends a message containing no certificates. If the server requires client authentication in order to continue the handshake, it may respond with a fatal_handshake failure alert.

The structure of the client_key_exchange message depends on which key exchange method has been selected.

- If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret, encrypts it using either the public key from the server's certificate or the temporary RSA key provided in a server_key_exchange message, and then sends the result in an encrypted pre-master secret message. Since the pre-master secret has been encrypted using the server's public key, the server can decrypt it. In fact, only the server can decrypt it. The pre-master secret consists of two bytes that indicate the latest (newest) version of the TLS protocol supported by the client followed by 46 securely-generated random bytes.
- If Diffie-Hellman is being used for key agreement, the client_key_exchange message conveys the client's Diffie-Hellman public value g^c mod p. Having the client's Diffie-Hellman public value allows the server to compute the same pre-master secret as the client. (In the event that the key exchange method is DH_RSA or DH_DSS, and the server requested client certification, and the client was able to respond with a certificate that contained a Diffie-Hellman public key whose group and generator matched those specified by the server in its certificate, then the client will send an empty client_key_exchange message.)

Now that the client and the server have agreed upon the pre-master secret they can compute the master secret. For all key exchange methods, the same algorithm is used to convert the pre-master secret into the master secret.

Example 6-3 Computing the master secret

```
master_secret=PRF(pre_master_secret, “master secret”,
                  client_hello.random+server_hello.random)
```

In Example 6-3 PRF is a pseudo-random function defined in RFC 2246, and + represents the concatenation operation. PRF takes as input a secret (such as our pre-master secret), an identifying label (such as “master secret”), and a seed (such as the concatenation of the random numbers generated by the client and the server).

Having computed the master secret, the Record Protocol layer for the client and the Record Protocol layer for the server can now each use PRF to compute a key_block as shown in Example 6-4.

Example 6-4 Computing the key_block

```
key_block=PRF(master_secret, “key expansion”,
                  client_hello.random+server_hello.random)
```

Then the `key_block` is partitioned as follows:

- `client_write_MAC_secret`

  The first `SecurityParameters.hash_size` bytes of the `key_block` become the secret data used to authenticate data written by the client.

- `server_write_MAC_secret`

  The next `SecurityParameters.hash_size` bytes of the `key_block` become the secret data used to authenticate data written by the server.

- `client_write_key`

  The next `SecurityParameters.key_material_length` bytes become the key used to encrypt data written by the client.

- `server_write_key`

  The next `SecurityParameters.key_material_length` bytes become the key used to encrypt data written by the server.

- `client_write_IV`

  The next bytes become the initialization vector for the encryption algorithm when the client encrypts data. The required number of bytes is equal to the block size for block ciphers and zero for stream ciphers.

- `server_write_IV`

  The next bytes become the initialization vector for the encryption algorithm when the server encrypts data.

Note: Since the `client_hello.random` and `server_hello.random` values are unique for each connection, the data encryption keys and MAC secrets will be unique for each connection. Also note that the `server_write_key` and the `client_write_key` are independent of each other.

Figure 6-27 on page 215 shows the final phase of starting a new TLS session.

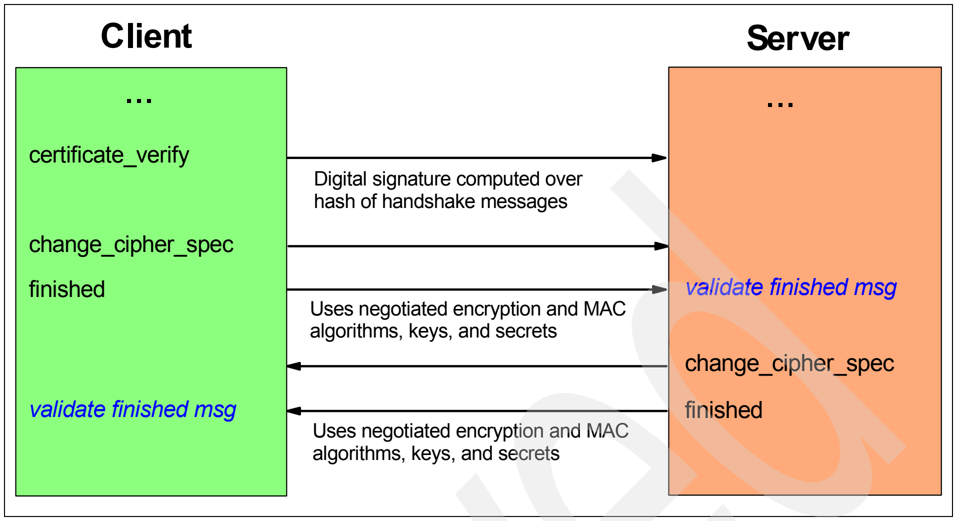

Client
Server
...
...
certificate_verify
Digital signature computed over hash of handshake messages
change_cipher_spec
finished
validate finished msg
Uses negotiated encryption and MAC algorithms, keys, and secrets
change_cipher_spec
validate finished msg
finished
Uses negotiated encryption and MAC algorithms, keys, and secrets

Figure 6-27 Starting a new TLS session (3): Verification

The `certificate_verify` message is used to provide explicit verification of a client certificate. This message is only sent following a client certificate that has signing capability (that is, all certificates except those containing fixed Diffie-Hellman parameters).

When the key exchange method is RSA, the `certificate_verify` message includes a 36-byte structure of two hashes (one SHA-1 and one MD5), which has been signed with the client's private key. The SHA-1 hash takes as input the concatenation of all handshake messages sent or received starting at `client_hello` up to but not including this message; it produces 20 bytes of output. The MD5 hash takes the same input and produces 16 bytes of output. These handshake messages include the server certificate, which binds the signature to the server, and `server_hello.random`, which binds the signature to the current handshake process.

When the key exchange method is Diffie-Hellman, the `certificate_verify` message includes a SHA-1 hash that takes the input described in the preceding paragraph and produces 20 bytes of output. The 20 bytes are then signed using the Digital Signature Algorithm.

The `change_cipher_spec` message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the newly negotiated encryption and MAC algorithms and keys. The message consists of a single byte of value 1.

A finished message is always sent immediately after a change_cipher_spec message to verify that the key exchange and authentication processes were successful. The finished message is the first protected with the just-negotiated algorithms, keys, and secrets. The content of the finished message is generated using PRF as shown in Example 6-5.

Example 6-5 Computing the verify_data

```
PRF(master_secret, finished_label,
    MD5(handshake_messages)+SHA-1(handshake_messages))
```

In Example 6-5:

- The value of finished_label is the string “client finished” for finished messages sent by the client and “server finished” for finished messages sent by the server.
- The value of handshake_messages is all of the data from all handshake messages up to but not including this message.

Recipients of finished messages must verify that the contents are correct. Once a side has sent its finished message and received and validated the finished message from its peer, it may begin to send and receive application data over the connection.

Outgoing data is protected with a MAC before transmission. To prevent message replay or modification attacks, the MAC is computed from the MAC secret, the message contents, the message length, and the sequence number of the message.

Resuming a session

Cryptographic operations tend to be highly CPU intensive, particularly public key operations. For this reason, the TLS protocol has incorporated an optional session caching scheme to reduce the number of connections that need to be established from scratch. When the client and server decide to resume a previous session, the message flow is as shown in Figure 6-28 on page 217.

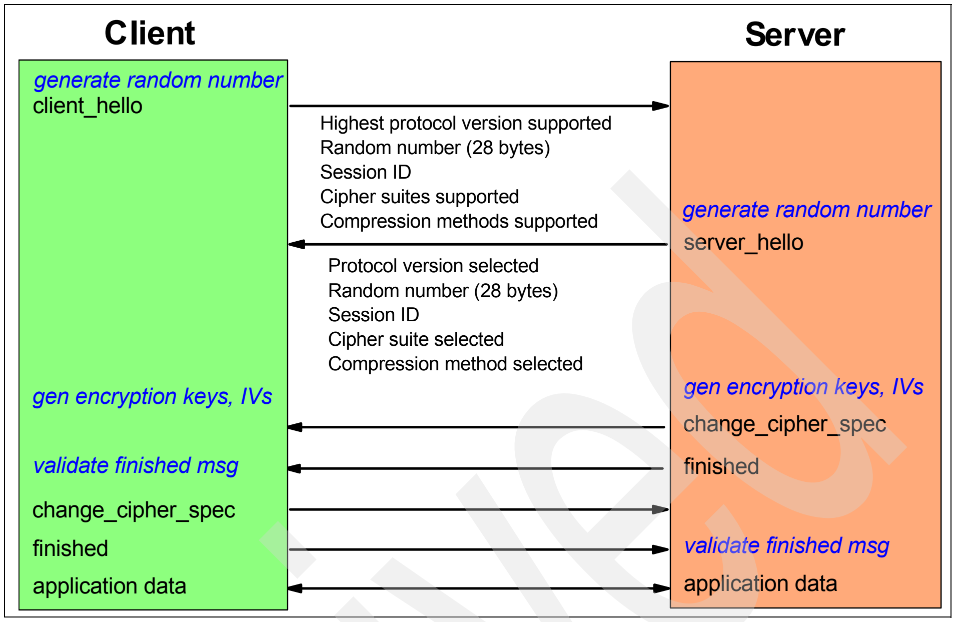


Figure 6-28 Resuming a session

The client sends a `client_hello` using the session ID of the session to be resumed. The server then checks its session cache for a match.

- If a match is found, and the server is willing to re-establish the connection under the specified session state, it will send a `server_hello` with the same session ID value plus the cipher suite and compression method from the state of the session being resumed. At this point both client and server must send `change_cipher_spec` messages and proceed directly to `finished` messages.
- If a session ID match is not found, the server generates a new session ID and the TLS client and server perform a full handshake.

When a connection is established by resuming a session, new `client_hello.random` and `server_hello.random` values are used with the session's master secret to produce a new `key_block` (see Example 6-4) and hence new encryption keys and MAC secrets.

7

Crypto hardware and ICSF

If you want to use the CICS-supplied module DFHWSSE1 to implement WS-Security in CICS TS V3.1, your system will have to meet certain hardware and software requirements. These requirements vary among the various server models (z900, z990, z9™, and so forth).

In this chapter we introduce you to IBM cryptographic hardware and to ICSF, the software product through which DFHWSSE1 invokes the cryptographic hardware. We then describe how CICS WS-Security support uses ICSF and hardware cryptography.

7.1 Cryptographic hardware

A cryptographic hardware feature is a secure, high-speed device that performs cryptographic functions. The cryptographic hardware features available to your CICS regions depend on the server that you have. This section provides a summary of the cryptographic hardware features currently available on System z hardware.

In recent years IBM has shipped the following cryptographic hardware products for mainframe servers:

- Central Processor Assist for Cryptographic Functions (CPACF)
- Cryptographic Express 2 Coprocessor (CEX2C)
- Cryptographic Express 2 Accelerator (CEX2A)
- Peripheral Component Interconnect Cryptographic Coprocessor (PCICC)
- Peripheral Component Interconnect Cryptographic Accelerator (PCICA)
- Peripheral Component Interconnect - Extended Cryptographic Coprocessor (PCIXCC)
- Cryptographic Coprocessor Feature (CCF)

Table 7-1 shows which of these cryptographic hardware products are available for each of several mainframe servers.

Table 7-1 Cryptographic hardware per server type

Server	**9672 G5,G6**	**z800, z900**	**z890, z990**	**z9 109, z9 BC, z9 EC**
CPACF	No	No	Yes (Requires z/OS 1.4 or later)	Yes (Requires z/OS 1.6 or later)
CEX2C (feature 0863)	No	No	Yes (Requires feature 3863 and z/OS 1.4 or later with ICSF FMID HCR7720 or later)	Yes (Requires feature 3863 and z/OS 1.6 or later with ICSF FMID HCR7730 or later)
CEX2A (feature 0863)	No	No	No	Yes (Requires feature 3863 and z/OS 1.6 or later with ICSF FMID HCR7730 or later)
PCICC (feature 0861)	Yes	Yes (requires CCF)	No	No
PCICA (feature 0862)	No	Yes (requires CCF)	Yes (Requires feature 3863)	No

Server	9672 G5,G6	z800, z900	z890, z990	z9 109, z9 BC, z9 EC
PCIXCC (feature 0868)	No	No	Yes (Requires feature 3863 and ICSF FMID HCR770A or later)	No
CCF (feature 0800)	Yes	Yes	No	No

Next we summarize the main features of the most recent cryptographic hardware options, the CPACF and CEX2.

7.1.1 CP Assist for Cryptographic Functions (CPACF)

CPACF offers a set of symmetric cryptographic functions available on all CPs of a z990, z890, z9-109, z9 Enterprise Class (EC), and z9 Business Class (BC).

The CPACF feature provides hardware acceleration for DES, triple-DES, AES (128 bit), MAC, SHA-1, and SHA-256 cryptographic services. It provides high-performance hardware encryption, decryption, and hashing support.

The SHA-1 algorithm is always available. The SHA-256 algorithm is available on the z9-109, z9 EC, and z9 BC. CPACF DES/triple-DES enablement is provided with feature 3863. It provides for clear key DES and triple-DES instructions. On the z9-109, z9 EC, and z9 BC, this feature includes clear key AES for 128-bit keys.

Important: The CPACF operates with *clear* keys only. A clear key is a key that has not been encrypted under another key and has no additional protection within the cryptographic environment.

Use of the CPACF instructions provides improved performance. Since the CPACF cryptographic functions are implemented in each CP, the potential throughput scales with the number of CPs in the server.

The CPACF feature can be used indirectly by z/OS applications and subsystems (such as CICS) that use the Integrated Cryptographic Service Facility (ICSF) for cryptographic functions (see “ICSF” on page 225).

7.1.2 Crypto Express 2 feature

The optional Crypto Express 2 (CEX2) comes as a PCI-X (Peripheral Component Interconnect eXtended) pluggable feature that provides a high performance and secure cryptographic environment.

An installation can configure the CEX2 feature as an asynchronous cryptographic coprocessor (CEX2C) or accelerator (CEX2A). The CEX2C is available on the System z9™ servers, z890 and z990. The CEX2A is available on the System z9 servers only.

- The Crypto Express 2 *Coprocessor* (CEX2C) feature is designed to secure the cryptographic keys.

  Note: A *secure* key is a key that has been encrypted under another key (usually the master key).

  Security-relevant cryptographic keys are encrypted under a Master Key when outside of the secure boundary of the CEX2C card. The Master Keys are always kept in battery backed-up memory within the tamper-protected boundary of the CEX2C, and are destroyed if the hardware module detects an attempt to penetrate it. The tamper-responding hardware has been certified at the highest level under the FIPS 140-2 standard.

  The CEX2C allows the user to do the following, using secure keys:

  – Encrypt and decrypt data utilizing shared secret-key algorithms.

  – Generate, install, and distribute cryptographic keys securely using both public and secret-key cryptographic methods.

  – Generate, verify, and translate personal identification numbers (PINs).

  – Ensure the integrity of data by using MACs, hashing algorithms, and RSA public key algorithm (PKA) digital signatures.

  Clear key PKA operations are also supported by the CEX2C, and are often used to provide SSL protocol communications.

  The CEX2C consolidates the functions previously offered on the z900 by the Cryptographic Coprocessor feature (CCF), the PCI Cryptographic Coprocessor (PCICC), and the PCI Cryptographic Accelerator (PCICA) feature.

- The Crypto Express 2 *Accelerator* (CEX2A) is actually a CEX2C that has been reconfigured by the user to only provide a subset of the CEX2C functions at enhanced speed.

  The CEX2A provides hardware support to accelerate certain cryptographic operations that occur frequently in the e-business environment.

Computationally intensive public key operations as used by the SSL/TLS protocol can be offloaded from the CP to the CEX2A, potentially increasing system throughput.

The CEX2A is used for the following RSA cryptographic operations (with clear keys only):

– PKA Decrypt (CSNDPKD), with PKCS-1.2 formatting
– PKA Encrypt (CSNDPKE), with ZERO-PAD formatting
– Digital Signature Verify

Important: The CEX2 feature requires ICSF to be active.

A z9 109, z9 BC, or z9 EC server can support a maximum of eight CEX2 features. Since each feature provides two coprocessors or accelerators, a z9 server can support a maximum of 16 cryptographic coprocessors or accelerators.

7.1.3 Comparison of CPACF, CEX2C, and CEX2A

Table 7-2 summarizes the functions and attributes of the cryptographic hardware that is available for a System z9.

Table 7-2 Comparison of System z9 cryptographic hardware

Function or attribute	CPACF	CEX2C	CEX2A
DES/TDES encrypt/decrypt with clear key	X		
AES-128 encrypt/decrypt with clear key	X		
DES/TDES encrypt/decrypt with secure key		X	
AES encrypt/decrypt with secure key		X	
Generate pseudo random numbers	X	X	
Provide hashing and message authentication	X	X	
Secure key RSA		X	
Clear key RSA		X	X
Provide highest performance for SSL handshaking with clear key			X
Provide highest performance for asymmetric encryption with secure key		X	

Function or attribute	CPACF	CEX2C	CEX2A
Physically imbedded on each CP	X		
Tamper-resistant hardware packaging		X	
Designed for FIPS 140-2 Level 4 certification		X	
Requires ICSF to be active		X	X
Storage for system master keys		X	
Requires system master keys to be loaded		X	
Usable for key management operations		X	

7.1.4 Other cryptographic hardware

In this section we identify the cryptographic hardware that is available with System z servers prior to the z9.

PCI Cryptographic Accelerator (PCICC)

The PCI Cryptographic Coprocessor (PCICC) is an orderable feature that adds additional cryptographic function and cryptographic performance to the z800 and z900 servers, and S/390® G5/G6 servers.

PCI Cryptographic Accelerator (PCICA)

The Peripheral Component Interconnect Cryptographic Accelerator (PCICA) is an orderable feature on the z990 and other zSeries servers. The PCICA feature is used for the acceleration of modular arithmetic operations, in particular the complex RSA cryptographic operations used with the SSL protocol.

PCI-X Cryptographic Coprocessor (PCIXCC)

The PCIXCC is a single coprocessor card that replaced the CCF and PCICC for the z890 and z990 servers. For more information about the PCIXCC see the article "The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer™" which appeared in volume 48 of the *IBM Journal of Research and Development* for May/July, 2004.

Cryptographic Coprocessor Feature (CCF)

The CCF is a single-chip cryptographic coprocessor that was imbedded as a standard, no-cost component in the early CMOS mainframe systems. Depending on its size, each CMOS mainframe has one or two CCFs. Each CCF contains both DES and Public Key Algorithm (PKA) cryptographic processing units.

7.2 ICSF

The Integrated Cryptographic Service Facility (ICSF) is a software element of z/OS that works with cryptographic hardware features and RACF to provide secure, high-speed cryptographic services in the z/OS environment. ICSF provides the application programming interfaces by which applications, and subsystems such as CICS, request the cryptographic services.

ICSF provides support for a number of cryptography services, including:

- DES and triple-DES encryption for privacy
- The transport of symmetric data keys through the use of the RSA public key algorithm
- The generation and verification of digital signatures through the use of both the RSA and the DSA algorithms
- The generation of RSA and DSA keys
- The PKA Encrypt and PKA Decrypt callable services that can be used to enhance the security and performance of SSL/TLS security protocol applications
- AES encryption and decryption

7.2.1 ICSF callable services

The format for invoking an ICSF callable service depends on the programming language, for example:

- C

  CSNBxxxx (*return_code*,*reason_code*,*exit_data_length*,*exit_data*, *parameter_5*,*parameter_6*,...,*parameter_N*)
- COBOL

  CALL ‘CSNBxxxx’ USING *return_code*,*reason_code*,*exit_data_length*, *exit_data*,*parameter_5*,*parameter_6*,...,*parameter_N*
- PL/I

  DCL CSNBxxxx ENTRY OPTIONS(ASM);

  CALL CSNBxxxx *return_code*,*reason_code*,*exit_data_length*,*exit_data*, *parameter_5*,*parameter_6*,...,*parameter_N*
- Assembler

  CALL CSNBxxxx,(*return_code*,*reason_code*,*exit_data_length*,*exit_data*, *parameter_5*,*parameter_6*,...,*parameter_N*)

Note: You cannot use Java to invoke an ICSF callable service, but Java applications can indirectly use the callable services via JSSE and JCECCA.

Controlling who can use cryptographic keys and services

The ICSF administrator can use RACF to control which applications can use specific keys and services. To set up these controls, the ICSF administrator must create RACF general resource profiles in the CSFKEYS resource class and in the CSFSERV resource class. The CSFKEYS class controls access to cryptographic keys, and the CSFSERV class controls access to ICSF services.

The following RACF command defines a profile in the CSFKEYS class:

```
RDEFINE CSFKEYS label UACC(NONE) other-optional-operands
```

where *label* is the label by which the key is defined in the CKDS or PKDS. Use the RACF PERMIT command to give user IDs or groups access to the profile:

```
PERMIT label CLASS(CSFKEYS) ID(groupID) ACCESS(READ)
```

To refresh the in-storage RACF profiles, issue a SETROPTS command:

```
SETROPTS RACLIST(CSFKEYS) REFRESH
```

The following RACF command defines a profile in the CSFSERV class:

```
RDEFINE CSFSERV service-name UACC(NONE) other-optional-operands
```

where *service-name* is chosen from a list in the *ICSF Administrator's Guide*, SA22-7521. (If the application program called the CSNBxxxx service, you should generally specify CSFxxxx as the *service-name* in the RDEFINE command. Note, however, that access to ICSF services CSNBSYE and CSNBSYD is not protected by profiles in the CSFSERV class.) Use the RACF PERMIT command to give user IDs or groups access to the profile:

```
PERMIT service-name CLASS(CSFSERV) ID(groupID) ACCESS(READ)
```

To refresh the in-storage RACF profiles, issue a SETROPTS command:

```
SETROPTS RACLIST(CSFSERV) REFRESH
```

7.2.2 ICSF administration

You define installation options and configure ICSF using the ICSF panels. You can use the ICSF panels to activate or deactivate your PCICC, PCIXCC, CEX2C, PCICA, and CEX2A coprocessors.

Example 7-1 shows a sample ICSF Coprocessor Management panel. The panel prefixes the coprocessor serial ID with a letter that indicates the type of coprocessor as follows: A for PCICA, E for CEX2C, F for CEX2A, and X for PCIXCC.

Example 7-1 Sample ICSF Coprocessor Management panel

```
------------------------ ICSF Coprocessor Management -------- Row 1 to 4 of 4

 Select the coprocessors to be processed and press ENTER.
 Action characters are: A, D, E, K, R and S. See the help panel for details.

   COPROCESSOR        SERIAL NUMBER   STATUS
   -----------        -------------   ------
 . E00                95000224        ACTIVE
 . E01                95000225        DEACTIVATED
 . E02                95000182        DEACTIVATED
 . E03                95000180        DEACTIVATED
****************************** Bottom of data *******************************
```

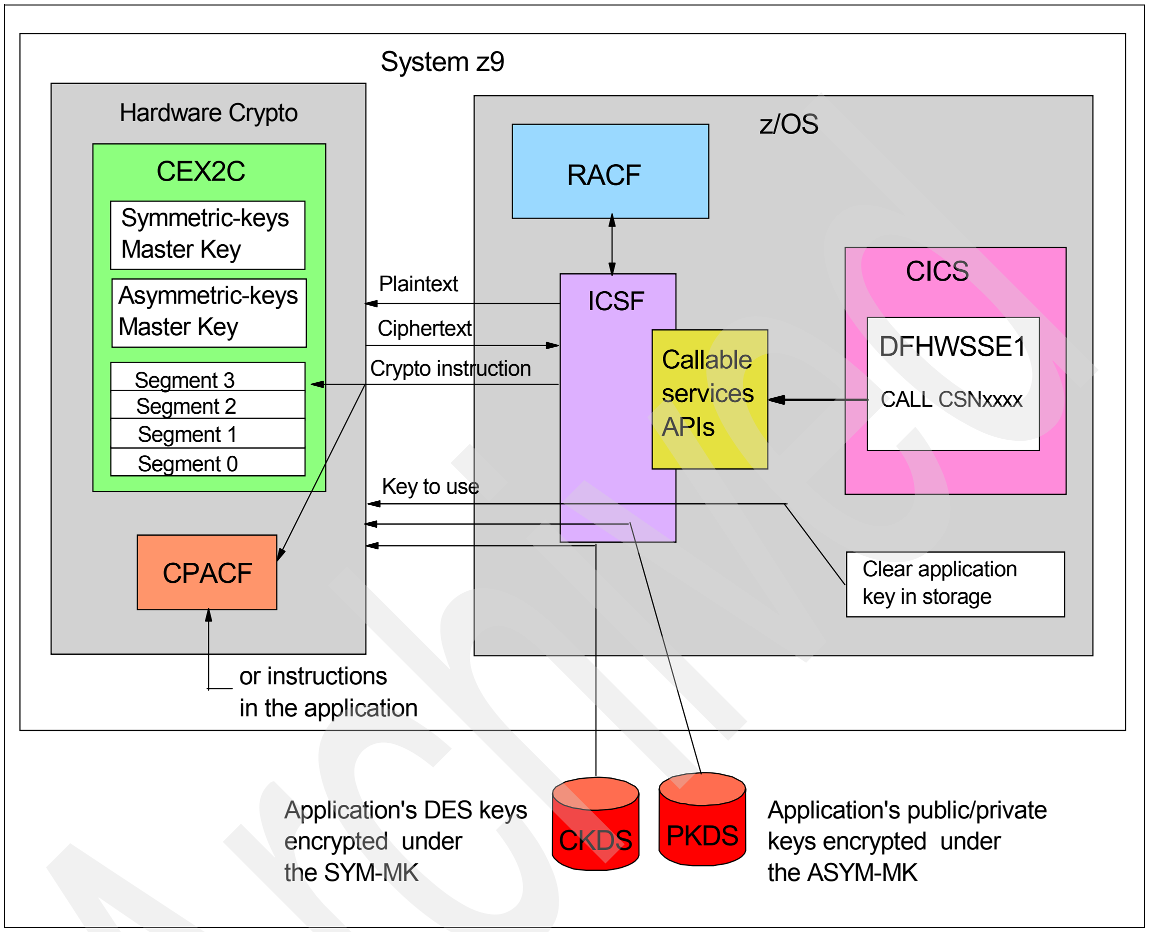
Note: ICSF recognizes the CEX2A beginning with the FMID HCR7730 level of ICSF. Previous levels of ICSF ignore the CEX2A cards.

7.3 How CICS uses ICSF

The CICS support for XML digital signature processing and XML encryption is dependent on ICSF services and therefore the configuration and startup of ICSF is a requirement for using this support.

Figure 7-1 shows how the CICS-supplied message handler, DFHWSSE1, issues a cryptographic API call to the ICSF started task. The ICSF started task invokes RACF to determine whether the user ID associated with the request is authorized to use the requested cryptographic service and any keys associated with the request. If the user ID has the proper authority, the ICSF started task will decide whether it should perform the request using ICSF software or cryptographic hardware.

If ICSF decides to use cryptographic hardware, it gives control to its routines that contain the crypto instructions. If ICSF routes the request to the CEX2C and the request is, for instance, a request to encrypt data, the ICSF started task provides the CEX2C with the data to be encrypted and the key to be used by the encryption algorithm. Recall that the key is encrypted, in this case under a variant of the Symmetric Keys Master Key(SYM-MK) stored in the CEX2C.



System z9
Hardware Crypto
CEX2C
Symmetric-keys Master Key
Asymmetric-keys Master Key
Segment 3
Segment 2
Segment 1
Segment 0
CPACF
z/OS
RACF
ICSF
Callable services APIs
CICS
DFHWSSE1
CALL CSNxxxx
Plaintext
Ciphertext
Crypto instruction
Key to use
Clear application key in storage
or instructions in the application
Application's DES keys encrypted under the SYM-MK
CKDS
PKDS
Application's public/private keys encrypted under the ASYM-MK

Figure 7-1 Overview of how CICS uses ICSF

The keys can be stored in ICSF-managed VSAM data sets and pointed to by the application program by using the label under which they are stored. The Cryptographic Key Data Set (CKDS) is used to store the symmetric keys in their encrypted form, and the Public Key Data Set (PKDS) is used to store the asymmetric keys. If the level of ICSF that you are using is HCR7720 or higher, you can also store keys in the CKDS in clear (unencrypted) form.

7.4 ICSF services used by CICS WS-Security support

This section provides information about which ICSF services are used by CICS and the cryptographic hardware requirements for each ICSF service.

ICSF callable services

This section describes the ICSF callable services used by CICS WS-Security.

- CSNBCKM
  Usage: XML encryption/decryption with DES algorithms

  The multiple clear key import callable service imports a clear 64-bit, 128-bit, or 192-bit DATA key that is to be used to encipher or decipher data. This service accepts a clear DATA key, enciphers it under the master key, and returns the encrypted DATA key in operational form in an internal key token.
- CSNBDEC
  Usage: XML decryption with DES algorithms

  The decipher callable service decrypts data in the caller's primary address space using either DES or TDES in the cipher block chaining mode. The caller must supply a 64-byte string that is an internal key token containing the data-encrypting key, or the label of a CKDS record containing a data-encrypting key, to be used for decrypting the data. Thus CSNBDEC uses a secure key. If the key token or CKDS record contains a 64-bit key, a DES decryption is performed. If the key token or CKDS record contains a 128-bit or 192-bit key, TDES decryption is performed.
- CSNBENC
  Usage: XML encryption with DES algorithms

  The encipher callable service encrypts data in the caller's primary address space using either DES or TDES in the cipher block chaining mode. The caller must supply a 64-byte string that is an internal key token containing the data-encrypting key, or the label of a CKDS record containing a data-encrypting key, to be used for encrypting the data. Thus CSNBENC uses a secure key. If the key token or CKDS record contains a 64-bit key, a DES encryption is performed. If the key token or key label contains a 128-bit or 192-bit key, TDES encryption is performed.
- CSNBOWH
  Usage: XML encryption/decryption and XML digital signature processing

  The one-way hash generate callable service generates a one-way hash on specified text. This service supports the following methods: MD5 (software only), SHA-1, RIPEMD-160 (software only), and SHA-256.

- CSNBRNG
  Usage: XML encryption/decryption and XML digital signature processing

  The random number generate callable service generates a 64-bit random number. If the caller requests, the service will generate a number with either even parity or odd parity in each byte. Parity is calculated on the 7 high-order bits in each byte and is presented in the low-order bit in the byte. (Note that a 64-bit random number with odd parity in each byte would be suitable for use as a DES key.)
- CSNBSYD
  Usage: XML decryption

  The symmetric key decipher callable service decrypts data in the caller's primary address space using one of the following algorithms: DES, TDES-128, TDES-192, AES-128, AES-192, AES-256. The caller can specify the use of either the electronic code book mode or the cipher block chaining mode. The caller must supply a key to be used for decrypting the data. The key must be supplied in one of the following ways: a clear key, a 64-byte string that is an internal key token containing a clear key, or a 64-byte string that is the label of a CKDS record containing a clear key.
- CSNBSYE
  Usage: XML encryption

  The symmetric key encipher callable service encrypts data in the caller's primary address space using one of the following algorithms: DES, TDES-128, TDES-192, AES-128, AES-192, AES-256. The caller can specify the use of either the electronic code book mode or the cipher block chaining mode. The caller must supply a key to be used for encrypting the data. The key must be supplied in one of the following ways: a clear key, a 64-byte string that is an internal key token containing a clear key, or a 64-byte string that is the label of a CKDS record containing a clear key.
- CSNDDSG
  Usage: XML digital signature generation

  The digital signature generate callable service generates a digital signature using an RSA private key. The private key must be valid for signature usage. The caller must supply input text that has been previously hashed using the one-way hash generate callable service.
- CSNDDSV
  Usage: XML digital signature validation

  The digital signature verify callable service verifies a digital signature using an RSA public key. The caller must supply input text that has been previously hashed using the one-way hash generate callable service. The caller must also supply the digital signature that is to be verified.

- CSNDPKB
  Usage: XML digital signature validation

  The PKA key token build callable service can be used to do one of the following:

  – Take a clear public key and a clear private key as input and build a PKA private external key token that contains a clear public key and a clear private key. You can use this token as input to the PKA key import service to obtain a private internal key token containing an enciphered private key.

  – Take a clear public key as input and build a PKA public external key token containing a clear public key. You can use this token directly in other PKA services.

  A "PKA key" means either an RSA key or a DSS key.

- CSNDPKD
  Usage: XML decryption

  The PKA decrypt service decrypts a formatted key, deformats it, and returns the deformatted value to the application in the clear.

- CSNDPKE
  Usage: XML encryption

  The PKA encrypt service encrypts a supplied clear key value under an RSA public key.

Cryptographic hardware required

The *z/OS 1.8 ICSF Application Programmer's Guide,* SA22-7522-08 provides details about the cryptographic hardware required by each callable service for a given server model.

Table 7-3 shows the cryptographic hardware required for each callable service listed in the previous section.

Table 7-3 Cryptographic hardware required by ICSF callable services used by CICS

Service	**IBM eServer zSeries 800 or 900**	**IBM eServer zSeries 890 or 990**	**IBM System z9-109**	**IBM System z9 BC or EC**
CSNBCKM	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBDEC	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBENC	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBOWH	SHA-1 requires CCF (SHA-256 not supported by CCF)	SHA-1 requires CPACF (SHA-256 not supported by CPACF)	SHA-1 and SHA-256 require CPACF	SHA-1 and SHA-256 require CPACF
CSNBRNG	CCF	PCIXCC CEX2C	CEX2C	CEX2C
CSNBSYD	CCF	CPACF	None[1]	None[1]
CSNBSYE	CCF	CPACF	None[1]	None[1]
CSNDDSG	CCF PCICC	PCIXCC CEX2C	CEX2C	CEX2C
CSNDDSV	CCF	PCICA PCIXCC CEX2C	CEX2C CEX2A	CEX2C CEX2A
CSNDPKB	None	None	None	None
CSNDPKD	CCF PCICC PCICA	PCICA PCIXCC CEX2C	CEX2C CEX2A	CEX2C CEX2A
CSNDPKE	CCF PCICC	PCICA PCIXCC CEX2C	CEX2C CEX2A	CEX2C CEX2A
Note 1. ICSF implements the AES algorithm in software. However, if hardware is available, ICSF will implement AES in hardware.				

Our discussion in this chapter leads to the conclusion that for DFHWSSE1 to run successfully on a z9-109, z9 BC, or z9 EC server, the following statements must all be true:

- Feature 3863 is installed on the server.
- At least one Feature 0863 (CEX2) is installed on the server.

- At least one of the PCIXCC cards of the CEX2 feature is configured as a coprocessor (CEX2C) rather than an accelerator (CEX2A).
- The operating system level is z/OS 1.6 or higher.
- The ICSF level is FMID HCR7730 or higher.
- ICSF is active.
- The CEX2C is *online* to z/OS.
- The CEX2C is *active* to ICSF.
- The user ID under which the CICS region runs has READ access to the RACF profiles in the CSFSERV class which protect the callable services shown in “ICSF callable services” on page 229.

Furthermore, you should specify API(OPENAPI) on the PROGRAM definition for the DFHWSSE1 program so that CICS will execute the program on an open TCB.

8

Securing Web services

While you can exercise very strict control over the access to your applications from conventional 3270 terminals, your Web services clients are likely to be in remote locations, and you will be faced with new security issues.

When implementing a CICS Web services solution, you need to consider questions like the following:

- Will *authentication* be done by CICS itself or in an external server such as WebSphere Application Server?
- What *authorization* mechanisms will be used to protect access to the CICS system and access to resources such as transactions, files, and databases?
- How will you protect the *confidentiality* of data that is transported between the different tiers of the physical configuration?
- Should you use transport security, for example SSL/TLS, or SOAP message security to protect your CICS Web services?

In this chapter, we explain what security mechanisms can be used to protect CICS Web services, and we give practical advice on which technologies to use.

8.1 Traditional CICS security

In a CICS environment, the assets you normally want to protect are the application programs and the resources that are accessed by the application programs. To prevent disclosure, destruction, or corruption of these assets, you must control access to the CICS region and to different CICS components.

You can limit the activities of a CICS user to only those functions that the user is authorized to use by implementing one or more of the following CICS security mechanisms:

- Transaction security

  This ensures that users who attempt to run a transaction are entitled to do so.
- Resource security

  This ensures that users who use CICS resources, such as files and transient data queues, are entitled to do so.
- Command security

  This ensures that users who use CICS system programming commands are entitled to do so.
- Surrogate security

  This ensures that a *surrogate* user is authorized to act on behalf of another user.

When CICS security is active, requests to attach transactions, and requests by transactions to access resources, are associated with a user ID. When a user makes such a request, CICS calls the external security manager (such as RACF) to determine if the user ID has the authority to complete the request. If the user ID does not have the correct authority, CICS denies the request.

In many cases, a user is a human operator, interacting with CICS through a terminal or a workstation. However, the user can also be a Web browser user or, in a Web services solution, a program executing in a client system.

8.1.1 CICS user IDs

When a human operator signs on to a CICS region at the start of a terminal session, he or she is challenged to provide a user ID and password. The user ID remains associated with the terminal until the terminal operator signs off. Transactions executed from the terminal, and requests made by those transactions, are associated with that user ID.

For connections from Web users, there are other ways that the user of a CICS transaction can be identified, including:

- An HTTP client can provide HTTP basic authentication information (a user ID and password). The transaction that services the client's request, and further requests made by that transaction, are associated with that user ID.
- A client program that is communicating with CICS using the Secure Sockets Layer (SSL) supplies a client certificate to identify itself. The security manager maps the certificate to a user ID. The transaction that services the client's request, and further requests made by that transaction, are associated with that user ID.

In addition to these transport-level authentication mechanisms, Web service clients can also pass authentication data in the SOAP message.

Special CICS user IDs

There are two particular user IDs that CICS uses in addition to those that identify individual end users. These are:

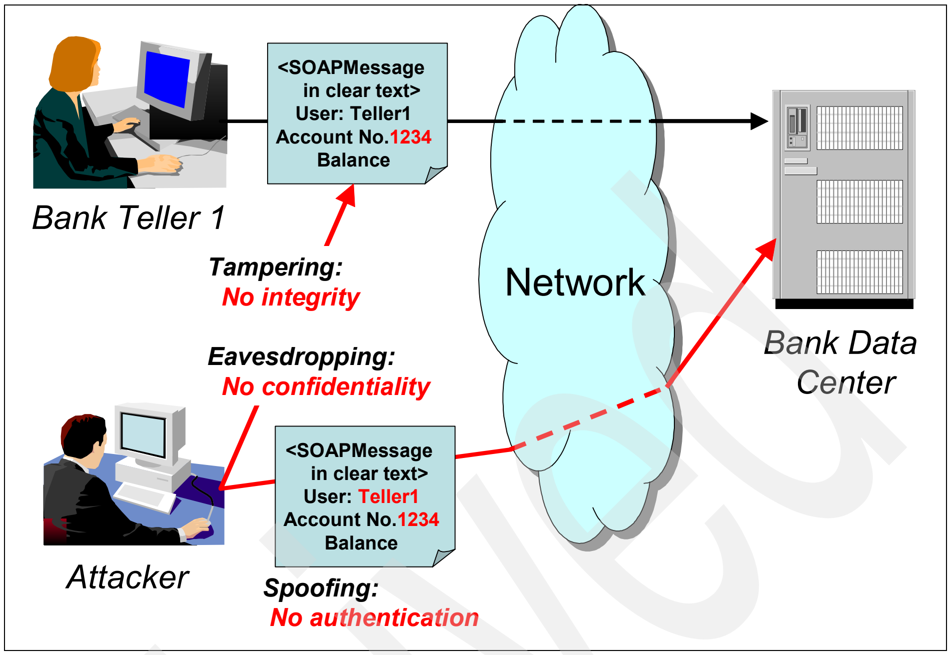
Region user ID	The CICS region user ID is used for authorization checking when the CICS system (rather than an individual user of the system) requests access to system resources such as CICS data sets and other servers.
Default user ID	When a user does not sign on, CICS assigns a default user ID to the user. It is specified in the SIT parameter DFLTUSER. In the absence of more explicit identification, it is used to identify TCP/IP clients that connect to CICS. You should give very little authority to the default user ID.

For a complete discussion of traditional CICS security, refer to *CICS TS V3.1 RACF Security Guide*, SC34-6249.

8.2 Security exposures

An end-to-end security solution addresses the security exposures found along the path of a request from an end client to a target service, including any intermediary services that route, or participate in, the service request.

To illustrate potential security exposures in a Web services environment, we use the bank teller scenario shown in Figure 8-1. The bank teller (Web service requester or client) connects over the Internet to the bank's data center where the Web service provider runs.



<SOAPMessage in clear text> User: Teller1 Account No.1234 Balance
Bank Teller 1
Tampering: No integrity
Network
Bank Data Center
Eavesdropping: No confidentiality
<SOAPMessage in clear text> User: Teller1 Account No.1234 Balance
Attacker
Spoofing: No authentication

Figure 8-1 Potential security exposures in a Web services environment

If the bank has not applied any security, it has the following exposures:

- **Spoofing**

  An attacker posing as the bank teller could send a SOAP message to the service provider to get confidential information or to withdraw money from another customer's account.

  The bank can eliminate this security exposure by requiring that the bank teller authenticate herself.

- **Tampering**

  An attacker could intercept the SOAP message between the Web service requester and provider and modify the message, for example, to deposit the money into another account by changing the account number. Because there is no integrity constraint, the Web service provider does not verify whether the message has been altered and accepts the modified transaction.

  The bank can eliminate this security exposure by implementing digital signatures.

- **Eavesdropping**

  An attacker could intercept the SOAP message and read the information contained in the message because it has been sent in clear text. The attacker could obtain confidential customer or bank information such as account numbers and balances.

The bank can eliminate this security exposure by encrypting the SOAP message.

In order to protect against the risk of these security exposures, the bank can use either or both of the following types of security to secure its Web services environment:

- Transport-level security

  Transport-level security mechanisms such as SSL/TLS can be used to secure Web services. In the next section we review how different transport-level security mechanisms can be used to secure a CICS Web services solution.

- SOAP message security

  The Web Services Security model introduces a set of interrelated specifications to form a layering approach to security. When products implement these specifications, they send security information within the SOAP message itself as SOAP message headers. In “SOAP message security” on page 253 we discuss how SOAP headers can be used to secure a CICS Web services solution.

We discuss some factors to consider when choosing between transport-level security and SOAP message security in “Comparison of transport level and SOAP message security” on page 269.

8.3 Transport security

In this section we review how basic authentication and SSL/TLS can be used to secure a CICS Web services solution when HTTP is the transport and then when MQ is the transport.

8.3.1 HTTP transport

When a CICS Web service is invoked using HTTP, you can use basic authentication to authenticate the Web service client, and you can use SSL/TLS to both authenticate the Web service client and to ensure message integrity and confidentiality.

Basic authentication

HTTP basic authentication is a simple challenge and response mechanism with which a server can request authentication information (a user ID and password) from a client. The client passes the authentication information to the server in an

HTTP Authorization header. The authentication information is in base-64 encoding.

The AUTHENTICATE attribute on the CICS TCPIPSERVICE resource definition specifies the authentication and identification scheme to be used for inbound TCP/IP connections for the HTTP protocol. You enable HTTP basic authentication by specifying BASIC for the AUTHENTICATE attribute.

A CICS service provider application can be protected by HTTP basic authentication. However, the HTTP basic authentication scheme can only be considered a secure means of authentication when the connection between the Web service client and the CICS region is secure. If the connection is insecure, the scheme does not provide sufficient security to prevent unauthorized users from discovering and using the authentication information for a server. If there is a possibility of a password being intercepted, basic authentication should be used in combination with SSL/TLS, so that SSL encryption is used to protect the user ID and password information.

CICS support for SSL/TLS

CICS uses System SSL to support both the SSL 3.0 and TLS 1.0 protocols. HTTPS connections will automatically use the TLS 1.0 protocol, unless the client specifically requires SSL 3.0.

A CICS service provider application can be secured using HTTPS, and a CICS service requester application can use HTTPS to invoke a service provider application. HTTPS has the following advantages:

- It provides a very fast and secure transport for CICS Web services.
- It provides for authentication through either HTTP basic authentication or a client X.509 certificate.
- It provides integrity for the data passed between the service requester and the service provider.
- It provides confidentiality for the data passed between the service requester and the service provider by using efficient secret key cryptography.
- It can be used with hardware cryptographic devices that can significantly reduce the cost of SSL handshakes. You can customize your encryption settings to use only the cipher suites that use the Integrated Cryptographic Service Facility (ICSF). See “ICSF” on page 225 for information about ICSF.
- It is mature and similarly implemented by most vendors, and therefore, is subject to few interoperability problems.

To activate SSL/TLS support in a CICS TS V3.1 region, you must perform the following tasks:

- Obtain a server certificate if CICS is the service provider. If CICS is the service requester, you may need to get a client certificate.
- Create a key ring.

  The CICS TS V3.1 RACF Guide describes how to do this.
- Ensure that your CICS region has access to the z/OS System SSL library.

  You can do this by using a STEPLIB or JOBLIB statement in the startup JCL for your CICS region or by putting the z/OS System SSL library in the MVS link list. The final qualifier of the name for the z/OS System SSL library is SGSKLOAD for z/OS V1.4 and SIEALNKE for z/OS V1.6 and later releases.
- Specify values for the CICS system initialization parameters related to SSL.

  We discuss these parameters in the next section.
- Define a TCPIPSERVICE resource for inbound requests or a URIMAP resource for outbound requests.

  We discuss how to do this in “Defining a TCPIPSERVICE resource for SSL” on page 244 and in “Defining a URIMAP resource for SSL” on page 246.

System initialization parameters related to SSL

To activate SSL/TLS support in a CICS TS V3.1 region, you must specify values for the following system initialization parameters:

- ENCRYPTION={STRONG | WEAK | MEDIUM}

  Specifies the cipher suites that CICS uses for secure TCP/IP connections. When a secure connection is established between a service requester and a service provider, the most secure cipher suite supported by both is used.

  – Use ENCRYPTION=STRONG when you can tolerate the overhead of using high encryption if the other system requires it.

  – Use ENCRYPTION=WEAK when you want to use encryption keys up to 40 bits in length.

  – Use ENCRYPTION=MEDIUM when you want to use encryption keys up to 56 bits in length.

  When you use the CICS CEDA transaction to define a TCPIPSERVICE or URIMAP resource, CICS automatically initializes the CIPHERS attribute of that resource definition with a default list of acceptable cipher suites; the contents of the default list depends on the value of the ENCRYPTION parameter.

- KEYRING=*key-ring-name*

  Specifies the name of a key ring in the RACF database that contains keys and certificates used by CICS. It must be owned by the CICS region user ID.

  When CICS finds the KEYRING parameter in the system initialization table, it knows that SSL/TLS processing is required and creates one open transaction environment (OTE) TCB, called the SP TCB, that is used to own socket pthread tasks. The SP TCB manages a pool of S8 TCBs that are used to process SSL connections. Each SSL connection uses an S8 TCB, which is allocated from the SSL pool and requires a UNIX pthread. All of the S8 TCBs run within a single LE enclave, which is owned by the SP TCB and contains the SSL cache.

- MAXSSLTCBS={<u>8</u> | *number*}

  Specifies the maximum number of S8 TCBs that are available to CICS to process SSL connections. The S8 TCBs are created and managed in the SSL pool.

  S8 TCBs are now locked to a transaction only for the amount of time that it needs to perform SSL functions. After the SSL negotiation is complete, the TCB is released back into the SSL pool to be reused.

  Increasing the number of available TCBs allows more simultaneous SSL connections to take place. However, increasing the number of TCBs too much will impact storage below the line.

  The maximum value that you can specify for the MAXSSLTCBS parameter is 1024.

- SSLDELAY={<u>600</u> | *number*}

  Specifies the length of time in seconds for which CICS retains session IDs for secure socket connections in the SSL cache. Session IDs are tokens that represent a secure connection between CICS and an SSL client. The session ID is created and exchanged between the SSL client and CICS during the SSL handshake.

  While the session ID is retained by CICS within the SSLDELAY period, CICS will re-establish an SSL connection with a client by using only a partial handshake as discussed in “Resuming a session” on page 216. The value is a number of seconds in the range 0 through 86400. The default value is 600.

  Increasing the value of the SSLDELAY parameter retains the session IDs in the cache for longer, thereby optimizing the time it takes to perform SSL negotiations.

  The SSLDELAY parameter only applies when the SSLCACHE parameter has the value CICS.

- SSLCACHE={<u>CICS</u> | SYSPLEX}

  Specifies whether CICS should use the local SSL cache in the CICS region, or share the cache across multiple CICS regions by using the sysplex session cache support provided by System SSL.

  When SSLCACHE=CICS, a client who successfully connects to CICS region 1 on z/OS system 1 and then connects to CICS region 2 on z/OS system 2 must go through a full SSL handshake in both cases; this is because CICS stores the SSL session id in a cache that is local to the CICS address space.

  When SSLCACHE=SYSPLEX, an SSL session established with a CICS region on one system in the sysplex can be resumed using a CICS region on another system in the sysplex as long as the SSL client presents the session identifier obtained for the first session when initiating the second session. CICS uses the sysplex session cache support provided by the System SSL started task (GSKSRVR), an optional component of System SSL. GSKSRVR processes the following environment variables:

  – GSK_LOCAL_THREADS

    This variable specifies the maximum number of threads which will be used to handle program call requests from SSL applications running on the same system as the GSKSRVR started task.

  – GSK_SIDCACHE_SIZE

    This variable specifies the size of the sysplex session cache in megabytes.

  – GSK_SIDCACHE_TIMEOUT

    This variable specifies the sysplex session cache entry timeout in minutes.

  Sharing SSL session ids across different CICS regions is particularly useful when Web service requests are being routed across a set of CICS regions using TCP/IP connection workload balancing techniques, such as TCP/IP port sharing or Sysplex Distributor. If the cache is shared between the CICS regions, the number of full SSL handshakes can be significantly reduced.

  In order to use the sysplex session cache, each system in the sysplex must be using the same external security manager (for example, z/OS Security Server RACF) and a userid on one system in the sysplex must represent the same user on all other systems in the sysplex (that is, userid ZED on System A has the same access rights as userid ZED on System B). The external security manager must support the RACROUTE REQUEST=EXTRACT,TYPE=ENVRXTR and RACROUTE REQUEST=FASTAUTH functions. Refer to *System SSL Programming*, SC24-5901, for additonal information about the System SSL started task.

  Caching across a sysplex can only take place when the regions accept SSL connections at the same IP address.

In “Enabling SSL/TLS” on page 301 we show how we enabled our CICS region to support SSL connections from our Web service client running in WebSphere Application Server.

Defining a TCPIPSERVICE resource for SSL

The following attributes of the TCPIPSERVICE resource definition relate to using SSL for inbound requests when the transport is HTTP:

- AUTHENTICATE(NO | BASIC | CERTIFICATE | AUTOREGISTER | AUTOMATIC)

  Specifies the authentication and identification scheme to be used.

  – NO
    The client is not required to send authentication or identification information. However, if the client sends a valid certificate that is already registered to the security manager, and associated with a user ID, then the user ID identifies the client.

  – BASIC
    HTTP basic authentication is used to obtain a user ID and password from the client.

  – CERTIFICATE
    SSL client certificate authentication is used to authenticate and identify the client. The client must send a valid certificate that is already registered to RACF and associated with a user ID. If a valid certificate is not received, or the certificate is not associated with a user ID, the connection is rejected.

    When the end user has been successfully authenticated, the user ID associated with the certificate identifies the client.

  – AUTOREGISTER
    SSL client certificate authentication is used to authenticate and identify the client.

    • If the client sends a valid certificate that is *not* registered to the security manager, then HTTP Basic authentication is used to obtain a user ID and password from the client. If the password is valid, CICS registers the certificate with the security manager and associates it with the user ID. The user ID identifies the client.

    • If the client sends a valid certificate that *is* already registered to the security manager, and associated with a user ID, then that user ID identifies the client.

  – AUTOMATIC
    This combines the AUTOREGISTER and BASIC functions.

   - If the client sends a certificate that is already registered to the security manager, and associated with a user ID, then that user ID identifies the client.
   - If the client sends a certificate that is not registered to the security manager, then HTTP Basic authentication is used to obtain a user ID and password from the client. Provided that the password is valid, CICS registers the certificate with the security manager, and associates it with the user ID. The user ID identifies the client.
   - If the client does not send a certificate, then HTTP Basic authentication is used to obtain a user ID and password from the user. When the end user has been successfully authenticated, the user ID supplied identifies the client.

- CERTIFICATE

  Specifies the label of an X.509 certificate that is used as a *server* certificate during the SSL handshake. If this attribute is omitted, the default certificate defined in the key ring for the CICS region user ID is used.

- CIPHERS

  Specifies a string of up to 56 hexadecimal digits that is interpreted as a list of up to 28 2-digit cipher suite numbers (the tables in “Cipher suites” on page 202 show the cipher suite numbers assigned to each cipher suite).

  When you use CEDA to define the resource, CICS automatically initializes this attribute with a default list of acceptable numbers. The contents of the default list depends on the level of encryption that is specified by the ENCRYPTION system initialization parameter.

  – For ENCRYPTION=WEAK, the default value is 03060102.

  – For ENCRYPTION=MEDIUM, the default value is 0903060102.

  – For ENCRYPTION=STRONG, the default value is:

     - 0504352F0A0903060102 for z/OS V1.4.
     - 050435363738392F303132330A1613100D0915120F0C03060102 for z/OS V1.6, z/OS V1.7, and z/OS V1.8.

  Note: When you use CEDA to define a resource using CIPHERS, and ENCRYPTION=STRONG is specified, the field is automatically filled in from the list of ciphers supported by the underlying z/OS (with the exception of cipher 00, which is removed). When you upgrade z/OS, you can use CEDA ALTER to clear the cipher list; it will then be upgraded to the currently-supported list of ciphers.

You can customize the default list to set a minimum level as well as a maximum level of encryption to be used in the encryption negotiation process of the SSL/TLS handshake.

- You can *reorder* the cipher suite numbers or *remove* one or more of them from the initial list. For example, if the system initialization parameter ENCRYPTION is set to STRONG and z/OS is at the V1.4 level, you could remove the suites 09, 03, 06, 02, and 01 and reorder the remaining suites to specify 352F0A0504. Specifying these numbers means that CICS will not negotiate below 128-bit encryption for connections using this resource; if the client does not have this level of encryption, CICS will close the connection. It also means that CICS will start by trying to negotiate using the AES-128 and AES_256 cipher suites (35 and 2F) because these are first in the list of cipher suite numbers.
- You cannot include cipher suites that are not in the default values for that level of encryption. For example, if you have a MEDIUM level of encryption specified, you cannot add the AES cipher suites 2F and 35 to the CIPHERS attribute.

Important: If the SSL handshake negotiates down to using cipher suite 01 or 02, there is no encryption and data will be transmitted in the clear. If you require encryption, you may therefore want to remove 01 and 02 from the list of cipher suites.

- SSL(NO | YES | CLIENTAUTH)

  Specifies whether the TCPIPSERVICE is to use SSL for encryption and authentication.

  – NO
    SSL is not to be used
  – YES
    An SSL session is to be used; CICS will send a server certificate to the client.
  – CLIENTAUTH
    An SSL session is to be used; CICS will send a server certificate to the client, and the client must send a client certificate to CICS.

Defining a URIMAP resource for SSL

The following attributes of the URIMAP resource definition relate to using SSL for outbound requests when the transport is HTTP:

- USAGE(CLIENT)
  Specifying CLIENT creates a URIMAP definition for CICS as an HTTP client.

This type of URIMAP definition is used when CICS makes a request for an HTTP resource on a server.

- SCHEME(HTTPS)
- CERTIFICATE(*label*)
  This attribute specifies the label of the X.509 certificate that is to be used as the SSL client certificate during the SSL handshake. It is up to the server to request an SSL client certificate, and if this happens, CICS supplies the certificate identified by the label that is specified in the CERTIFICATE attribute. If this attribute is omitted, the default certificate defined in the key ring for the CICS region user ID is used.
- CIPHERS
  The description of the CIPHERS attribute for the URIMAP resource is the same as the description of the CIPHERS attribute for the TCPIPSERVICE resource; see "Defining a TCPIPSERVICE resource for SSL" on page 244.

Using hardware cryptographic features with System SSL

The use of cryptographic hardware by System SSL is required in order to maximize performance of using SSL/TLS with CICS. During its runtime initialization processing, System SSL checks to see what cryptographic hardware is available. Whenever possible, it will use the hardware rather than its own software algorithms to perform a cryptographic algorithm.

In "Cryptographic hardware" on page 220 we introduced you to IBM cryptographic hardware and to Integrated Cryptographic Service Facility (ICSF). In this section we highlight some of the encryption algorithms that can be used with different cryptographic features by System SSL.

System SSL handshake processing utilizes both RSA encryption and digital signature functions. These functions are very expensive functions when performed in software. For installations that have high volumes of SSL handshake processing, utilizing the capabilities of the hardware will provide maximum performance and throughput, and it will also reduce CPU costs.

Note: The number of full handshakes can be minimized by setting the SOCKETCLOSE attribute of the TCPIPSERVICE definition to No or to a high value (see "Configuring the TCPIPSERVICE definition" on page 78).

For installations that are more concerned with the transfer of encrypted data than with SSL handshakes, moving the encrypt/decrypt processing to hardware will provide maximum performance. The encryption algorithm is determined by the SSL cipher suite. To utilize hardware, the cipher suite's encryption algorithm must be available in hardware. For example, on a z9-109, if you specify a cipher suite that uses TDES to encrypt/decrypt data, then you will benefit from the processing being done in the hardware (using the CPACF). On the other hand, if you specify a cipher suite that uses AES-256 to encrypt/decrypt data, then the processing will be done in software.

In Table 8-1 each row represents a cryptographic algorithm or function that System SSL supports, and each column represents a common server and cryptographic hardware combination. If an X appears at the intersection of a row and a column, then System SSL is able to implement the algorithm or function represented by the row using the hardware represented by the column.

Table 8-1 Hardware cryptographic functions used by System SSL

	z800 z900	**z890**	**z990**	**z9 109**	**z9 BC**	**z9 EC**
Algorithm or function	**CCF**	**CPACF**	**PCIXCC/ CEX2C**	**CPACF**	**CEX2C**	**CEX2A**
RC2						
RC4						
DES	X	X		X		
TDES	X	X		X		
AES-128				X		
AES-256						
MD5						
SHA-1		X		X		
SHA-256				X		
PKA (RSA) Decrypt	X		X		X	X
PKA (RSA) Encrypt	X		X		X	X
Digital Signature Generate	X		X		X	
Digital Signature Verify	X		X		X	X

In order for System SSL to use the hardware support provided through ICSF, the ICSF started task must be running prior to CICS initialization and the CICS user ID must be authorized to the appropriate resources in the CSFSERV class, if defined (see “Controlling who can use cryptographic keys and services” on page 226).

Table 8-2 identifies the required CSFSERV resource class accesses for different cryptographic algorithms and functions.

Table 8-2 CSFSERV resource class access

Function	z800, z900	z890, z990, z9-109, z9 BC, z9 EC
DES	CSFCKI, CSFDEC, CSFENC	
TDES	CSFCKM, CSFDEC, CSFENC	
PKA (RSA) Decrypt	CSFPKD	CSFPKD
PKA (RSA) Encrypt	CSFPKE	CSFPKE
Digital Signature Generate	CSFPKI, CSFDSG	CSFPKI, CSFDSG
Digital Signature Verify	CSFDSV	CSFDSV

The resource classes are the following:

- CSFCKI - Clear key import
- CSFCKM - Multiple clear key import
- CSFDEC - Symmetric key decrypt
- CSFDSG - Digital signature generate
- CSFDSV - Digital signature verify
- CSFENC - Symmetric key encrypt
- CSFPKD - PKA decrypt
- CSFPKE - PKA encrypt
- CSFPKI - PKA key import

In addition to the CSFSERV class, the CICS user ID needs access to the RACF CSFKEYS class when key rings are being used and the certificate keys are stored in an ICSF data set.

Optimizing SSL

Implementing SSL will cause an increase in CPU usage. You should only use SSL for applications that need this level of security. For these applications, you should consider the following techniques for optimizing the performance of SSL in your environment:

- Utilizing the zSeries cryptographic hardware as discussed in Using hardware cryptographic features with System SSL.
- Increasing the value of the CICS system initialization table parameter SSLDELAY so the session IDs remain in the SSLCACHE longer, which will result in only partial SSL handshakes.

- Increasing the value of the CICS system initialization table parameter MAXSSLTCBS so there are more S8 TCBs in the SSL pool for the SSL handshake negotiation.
- Using the CICS SSLCACHE system initialization table parameter to implement SSL caching across a sysplex if Web service requests are being routed across a set of CICS regions. However, if you are using a single CICS region then you should specify SSLCACHE(CICS) as opposed to SSLCACHE(SYSPLEX) in order to avoid the additional cost of making the SSL session ID shareable.
- Keeping the socket open by coding SOCKETCLOSE NO on the TCPIPSERVICE definition for the PIPELINE. This is the default for HTTP 1.1 persistent sessions and removes the need to perform a full SSL handshake on the second or subsequent HTTP request.
- Only using client authentication by specifying SSL(CLIENTAUTH) on the TCPIPSERVICE definition when you really need your clients to identify themselves with a client certificate. Client authentication requires more network transmissions during the SSL handshake, and more processing by CICS to handle the received certificate.

Setting the user ID on the URIMAP

You can specify a user ID on a URIMAP that is defined with USAGE(PIPELINE). You do this by setting the USERID attribute of the URIMAP definition. It specifies the 1 to 8 character user ID under which the Web services pipeline alias transaction is attached. See "Setting the user ID on a URIMAP definition" on page 280 for an example of how to set the user ID on the URIMAP definition.

A user ID that you specify in the URIMAP definition is overridden by any user ID that is obtained directly from the client.

Important: If you use a URIMAP definition to set a user ID, there is no authentication of the client's identity. You should only do this when communicating with your own client system, which has already authenticated its users and communicates with the server in a secure environment.

Order of precedence for determining the user ID when using HTTP

It is possible that for a single Web service request transported by HTTP, multiple methods for setting the user ID will be used at the same time. In this event, CICS uses the following order of precedence for determining the user ID under which the target business logic program runs:

1. A user ID specified by a message handler, or a SOAP header processing program, that is included in the pipeline that processes the SOAP message. For example, a SOAP header processing program could extract a username

from the SOAP message and specify that the CICS task should run with this user ID. See “SOAP message security” on page 253 for more information about how SOAP message security can be used with CICS and HTTP.

2. A user ID obtained from the Web client using basic authentication, or a user ID associated with a client certificate.
3. A user ID specified in the URIMAP definition for the request.
4. The CICS default user ID, if no other can be determined.

8.3.2 WebSphere MQ transport

To control security checking performed by WebSphere MQ (WMQ), you must define *switch profiles*. When a queue manager is started (or when the WMQ REFRESH SECURITY command is issued), WMQ first checks the status of RACF and the MQADMIN class. It sets the subsystem security switch off if it discovers one of these conditions:

- RACF is inactive or not installed.
- The MQADMIN class is not defined to RACF.
- The MQADMIN class has not been activated.

If both RACF and the MQADMIN class are active, WMQ checks the MQADMIN class to see whether any of the switch profiles have been defined. If subsystem security is not required, WMQ sets the internal subsystem security switch off and performs no further checks. The sequence of subsystem security checks is shown in Figure 8-2.

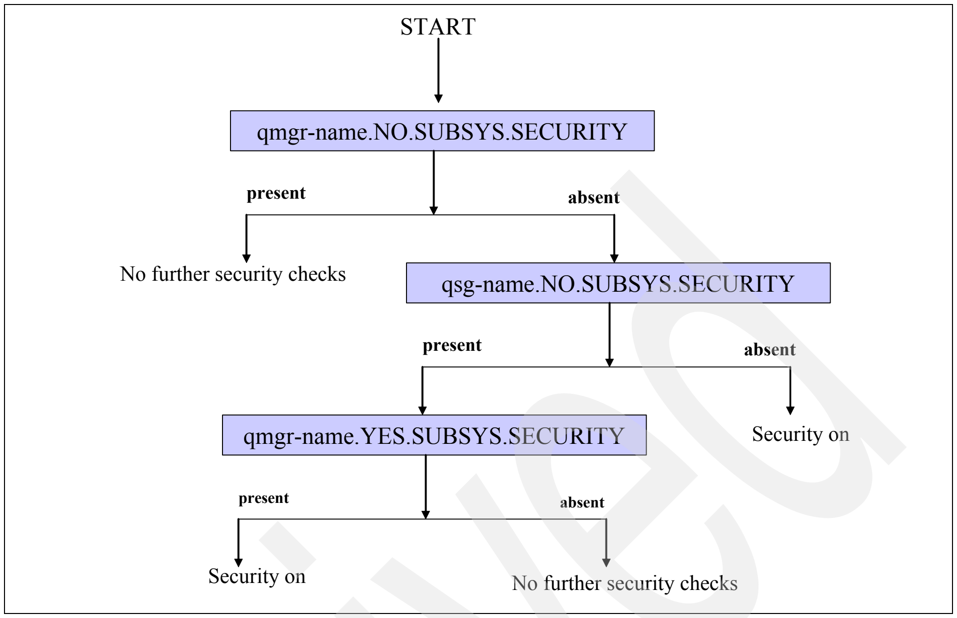

START
qmgr-name.NO.SUBSYS.SECURITY
present
absent
No further security checks
qsg-name.NO.SUBSYS.SECURITY
present
absent
qmgr-name.YES.SUBSYS.SECURITY
Security on
present
absent
Security on
No further security checks

Figure 8-2 Sequence for deciding if security is on for WebSphere MQ

Switch profiles can be set at the queue manager level and at the queue-sharing group level, but the queue manager level is always checked first. If your queue manager is not a member of a queue-sharing group, then no queue-sharing group checks are made. Switch profiles are not subject to any access list checks and are merely used to indicate to WebSphere MQ whether a particular security switch is on or off. A number of switch profiles exist that can be used to control the security checking for your WebSphere MQ environment.

When using WebSphere MQ as the transport mechanism for accessing Web services in CICS, you need to consider the following points:

- The SOAP MQ inbound listener transaction (CPIL) is started by the trigger monitor using the same user ID as the trigger monitor transaction. This user ID must have UPDATE authority to the request queue and the backout queue (if this is specified).
- If AUTH=IDENTIFY is specified in the USERDATA parameter of the WMQ PROCESS definition for CPIL, then the user ID under which CPIL runs must have surrogate authority to allow it to start transactions on behalf of the user IDs in the MQ message descriptors (MQMDs) of the messages.

More information about security for WMQ can be found in *WebSphere MQ Security,* SC34-6588 and in the book *WebSphere MQ Security in an Enterprise Environment*, SG24-6814.

SSL/TLS with WebSphere MQ

SSL/TLS can be used to secure SOAP messages that are transported using WMQ. WMQ supports both the SSL 3.0 and TLS 1.0 protocols. You specify the cryptographic algorithms that are used by the SSL protocol by supplying a CipherSpec as part of the channel definition. WMQ also supports Version 1.0 of the Transport Layer Security (TLS) protocol.

See *WebSphere MQ Security,* SC34-6588 for more information about using SSL/TLS with WMQ.

Order of precedence for determining user ID when using WMQ

It is possible that for a single Web service request transported by WMQ, multiple methods for setting the user ID will be used at the same time. In this event, CICS uses the following order of precedence to determine the user ID under which the target business logic program runs:

1. A user ID specified by a message handler, or a SOAP header processing program, that is included in the pipeline that processes the SOAP message

   For example, a SOAP header processing program could extract a username from the SOAP message and specify that the CICS task should run with this user ID. See "Enabling SOAP message security with WMQ" on page 312 for more information about how SOAP message security can be used with CICS and WMQ.

2. A user ID obtained from the MQ message descriptor

   A message can contain message context information, such as a user ID. This information is held in the message descriptor and can be generated by the queue manager when a message is put on a queue by an application, or by the application itself. This allows the receiving application to run with the same identity as the application that put the message on the queue.

3. The CICS default user ID, if no other can be determined

Note: We found that although URIMAP resources are required for service providers using WMQ, the USERID and TRANSACTION attributes were ignored.

8.4 SOAP message security

The first version of the WS-Security specification was proposed by IBM, Microsoft, and VeriSign in April 2002. After the formalization of the April 2002 specification, the specification was transferred to the OASIS consortium:

```
http://www.oasis-open.org
```

The latest core specification, Web Services Security: SOAP Message Security 1.0 (WS-Security 2004) was standardized in March 2004.

WS-Security provides a foundational set of SOAP message extensions for building secure Web services by defining new elements to be used in the SOAP header for message-level security. It specifies the use of *security tokens, digital signatures*, and *XML encryption* to protect and authenticate SOAP messages. It specifies the use of digital signatures to provide integrity for XML elements in a SOAP message, and it specifies the use of encryption to provide confidentiality for XML elements in a SOAP message. The specification allows you to protect the body of the message or any XML elements within the body or the header. You can give different levels of protection to different elements within the SOAP message.

The advantage of using WS-Security over SSL is that it can provide *end-to-end* message-level security. This means that the message security can be protected even if the message goes through multiple services, called intermediaries. SSL security is considered to be *point-to-point*, and the data may be decrypted prior to reaching the intended recipient.

As illustrated in Figure 8-3, if the service requester identifies itself to the intermediate gateway, and the intermediate gateway identifies itself to the service provider, the target service will normally run with the identity of the intermediate gateway rather than the service requester.

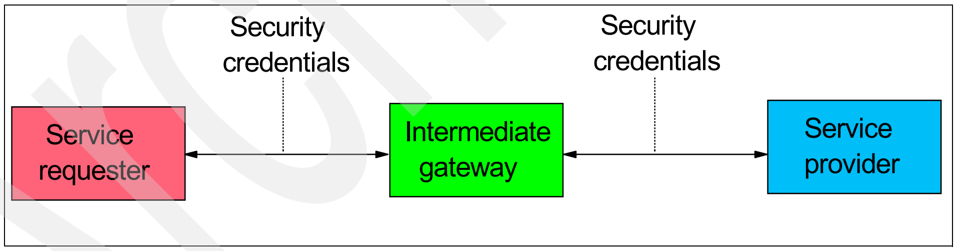


Figure 8-3 Transport-level security with an intermediate gateway

WS-Security addresses this problem by allowing security credentials to be passed within the SOAP message, so that the credentials of the service requester can be passed via an intermediate gateway, and can still be used to identify the requester to the service provider. See Figure 8-4.

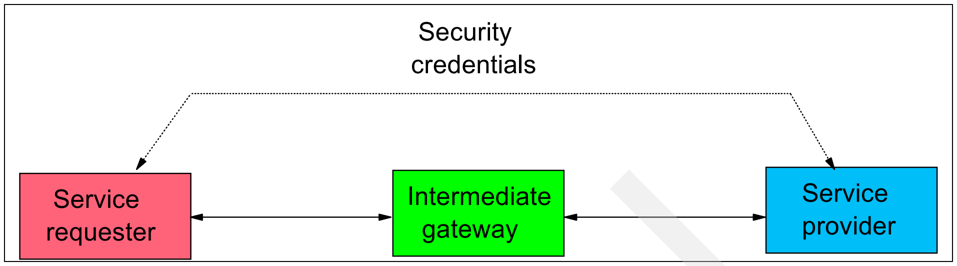


Figure 8-4 SOAP message security with an intermediate gateway

Figure 8-5 shows how a SOAP message can be extended with security data that is used to authenticate the service requester and to protect the message as it passes between the requester and the service provider. The network portion of the diagram could contain any number of intermediate nodes, some of which may not be trusted.

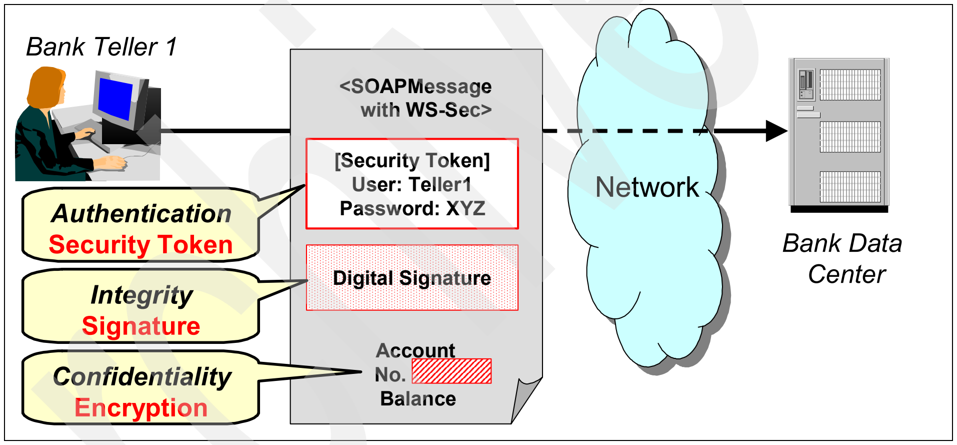

Bank Teller 1
<SOAPMessage with WS-Sec>
[Security Token] User: Teller1 Password: XYZ
Digital Signature
Account No. Balance
Authentication Security Token
Integrity Signature
Confidentiality Encryption
Network
Bank Data Center

Figure 8-5 An example of a typical scenario with WS-Security

The SOAP message shown in Figure 8-5 contains three pieces of security data:

- A security token used to authenticate and identify user Teller1
- A digital signature to ensure that no one modifies the message while it is in transit without the modification being detected
- An account balance XML element that is encrypted to ensure confidentiality

To read more about the Web services security specifications, refer to:

- Specification: Web Services Security (WS-Security) Version 1.0 (April 2002):

  http://www.ibm.com/developerworks/webservices/library/ws-secure/

- Web Services Security Addendum (August 2002):

  http://www.ibm.com/developerworks/webservices/library/ws-secureadd.html
- Web Services Security: SOAP Message Security V1.0 (March 2004):

  http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

Important: When we wrote the first edition of this book, CICS did not provide support for WS-Security. However, it was still possible for a user-written SOAP header processing program to process WS-Security security tokens in CICS. We show how to do this in Chapter 9, “Security scenarios” on page 275.

8.4.1 CICS and SOAP message security

Using WS-Security, you can apply authentication, integrity, and confidentiality at the *message* level. CICS TS V3.1 provides support for WSS: SOAP Message Security through the use of a CICS-supplied message handler, DFHWSSE1, which was shipped by APAR PK22736.

The WS-Security implementation in CICS TS V3.1 contains code derived from the Apache XML Security project. The licensing terms associated with that code mean that the WS-Security implementation in CICS TS V3.1 is not licensed on the same basis as the rest of the CICS TS V3.1 product.

Except for the WS-Security implementation, CICS TS V3.1 is an “ICA Program,” licensed under the relevant terms and conditions of the IBM Customer Agreement (ICA) or IBM International Customer Agreement (IICA). The WS-Security implementation is licensed under the terms and conditions of the IBM International Program License Agreement (IPLA). IPLA licensing is widely used in IBM, especially for products on distributed platforms and for one-time charge products on z/OS.

The WS-Security implementation, known as the CICS WS-Security Component, is packaged as a unique FMID with the identifier JCI640W. FMID JCI640W is licensed under the IPLA.

For this reason, the Licensed Program Specification (LPS) for CICS TS V3.1 includes a section explaining that the CICS WS-Security Component has a different licensing basis from the rest of CICS TS V3.1. Specifically, the licensing of the CICS WS-Security Component is addressed by the following three additional paper items:

- The IPLA (multi-language booklet)
- License Information (LI) document for CICS WS-Security Component (multi-language booklet)

- Proof of Entitlement for CICS WS-Security Component (multi-language sheet of paper)

Deliveries of CICS TS V3.1 after June 2, 2006 contain the additional material. Customers who ordered and received CICS TS V3.1 before that date were sent the material separately, together with the updated LPS.

Important: The WS-Security implementation in CICS Transaction Server for z/OS Version 3.1 is not licensed on the same basis as is used for the rest of the CICS TS V3.1 product.

There are several options available with the CICS WS-Security support, and which ones you choose will depend on the level of security required for the data and the transmission path of the data. The options that you can choose from are:

- Basic authentication

  In service provider mode, CICS can accept a UsernameToken in the SOAP message header for authentication on inbound SOAP messages. The UsernameToken contains a Username element and a Password element. CICS verifies the Username and Password using an external security manager such as RACF. If this is successful, CICS places the Username in container DFHWS-USERID and processes the SOAP message in the pipeline. If CICS is unable to verify the UsernameToken, it returns a SOAP fault message to the service requester.

  Username tokens are not supported on outbound SOAP messages when CICS is the service requester.

- Signing with X.509 certificates

  In service provider and service requester modes, you can provide an X.509 certificate in the SOAP message header to sign the body of the SOAP message for authentication. An X.509 certificate is an example of a binary security token.

  – Inbound SOAP messages

    To accept binary security tokens from inbound SOAP messages you must import the public key associated with the certificate into RACF and associate it with the keyring that is specified in the KEYRING system initialization parameter for the CICS region.

  – Outbound SOAP messages

    For outbound SOAP messages you need to generate and publish the public key to the intended recipients. The Integrated Cryptographic Service Facility (ICSF) is used to generate private keys.

Important: ICSF must be started and configured with cryptographic devices in order to use the CICS WS-Security XML digital signature support.

- Encrypting

  In service provider and service requester modes, you can encrypt the SOAP message body using either the Triple DES algorithm or the AES algorithm. It is then included in the message and encrypted using the intended recipient's public key with the asymmetric key encryption algorithm RSA 1.5.

  CICS does not support inbound SOAP messages that only have an encrypted element in the message header and no encrypted elements in the SOAP body.

  Important: ICSF must be started and configured with cryptographic devices in order to use the CICS WS-Security XML encryption support.

- Signing and encrypting

  In service provider and service requester modes, you can choose to both sign and encrypt a SOAP message. CICS always signs the SOAP message body first and then encrypts it. This provides both message integrity and confidentiality.

Implementing CICS WS-Security support will cause an increase in CPU usage, particularly when using XML digital signatures and XML encryption. The CICS support for these functions is dependent on ICSF services and therefore the configuration and startup of ICSF is a requirement for using this support. See "ICSF services used by CICS WS-Security support" on page 229 for information about what hardware cryptographic devices are required for CICS WS-Security support, and for guidance on how to optimize WS-Security processing.

Authentication

In this section, we provide an example of a SOAP message with WS-Security used for authentication.

WS-Security provides a general purpose mechanism to associate security tokens with messages for single message authentication. It does not require you to use a specific type of security token. Instead it is designed to be extensible and support multiple security token formats to accommodate a variety of authentication mechanisms. For example, a client might provide proof of identity and proof of a particular business certification.

Example 8-1 shows a sample SOAP message without applying WS-Security. The SOAP message is an Order request for our sample catalog application.

Example 8-1 SOAP message without WS-Security

```
<soapenv:Envelope
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header/>
  <soapenv:Body>
    <p635:DFHOXCMN xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
      <p635:ca_request_id>01ORDR</p635:ca_request_id>
      <p635:ca_return_code>0</p635:ca_return_code>
      <p635:ca_response_message></p635:ca_response_message>
      <p635:ca_order_request>
        <p635:ca_userid>srthstrh</p635:ca_userid>
        <p635:ca_charge_dept>hbhhhh</p635:ca_charge_dept>
        <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
        <p635:ca_quantity_req>1</p635:ca_quantity_req>
        <p635:filler1 xsi:nil="true" />
      </p635:ca_order_request>
    </p635:DFHOXCMN>
  </soapenv:Body>
</soapenv:Envelope>
```

As you can see in Example 8-1, the SOAP message does not have any SOAP headers. We will apply WS-Security by inserting a SOAP security header.

WS-Security defines a vocabulary that can be used inside the SOAP envelope. The XML element `<wsse:Security>` is the *container* for security-related information. (Note that *wsse* stands for *Web services security extension*.)

When you use WS-Security for authentication, a security token is embedded in the SOAP header and is propagated from the message sender to the intended message receiver. On the receiving side, it is the responsibility of the server security handler to authenticate the security token and to set up the caller identity for the request.

In Example 8-2 we show the same SOAP message, but this time with authentication. As you can see, we have user name and password information contained in the `<UsernameToken>` element.

Example 8-2 SOAP message with WS-Security

```
<soapenv:Envelope
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.
     xsd">
      <wsse:UsernameToken>
        <wsse:Username>WEBUSER</wsse:Username>
        <wsse:Password
         Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1
         .0#PasswordText">
           REDBOOKS
        </wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <p635:DFHOXCMN xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
      <p635:ca_request_id>01ORDR</p635:ca_request_id>
      <p635:ca_return_code>0</p635:ca_return_code>
      <p635:ca_response_message></p635:ca_response_message>
      <p635:ca_order_request>
        <p635:ca_userid>srthstrh</p635:ca_userid>
        <p635:ca_charge_dept>hbhhhh</p635:ca_charge_dept>
        <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
        <p635:ca_quantity_req>1</p635:ca_quantity_req>
        <p635:filler1 xsi:nil="true" />
      </p635:ca_order_request>
    </p635:DFHOXCMN>
  </soapenv:Body>
</soapenv:Envelope>
```

The `<UsernameToken>` element of the SOAP message in Example 8-2 contains credentials that can be used to authenticate the user `WEBUSER`.

The simplest form of security token is the UsernameToken, which is used to provide a user name and password for basic authentication. In “The WS-Security header processing program” on page 295, we show an example of how a header processing program can extract a UsernameToken from a SOAP header, validate the username and password, and set the user ID of the CICS task to the username passed in the header.

A signed security token is one that is cryptographically signed by a specific authority. For example, an X.509 certificate is a signed security token.

Security token usage for WS-Security is defined in separate profiles such as the Username token profile and the X.509 token profile.

To read more about these security token standards, refer to:

- Web Services Security: UsernameToken Profile V1.0 (March 2004):

  `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`

- Web Services Security: X.509 Token Profile V1.0 (March 2004):

  `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf`

Implementing SOAP message security in CICS

To implement WS-Security in CICS TS for either a service provider or a service requester, you must include a `<wsse_handler>` element in the configuration file for the appropriate pipeline. Use the following sub-elements of `<wsse_handler>` to provide configuration information to DFHWSSE1:

- `<dfhwsse_configuration>`

  This element can be used in either a service provider or a service requester pipeline. It may contain the following optional elements:

  – `<authentication>`

  – `<expect_signed_body/>`

  – `<expect_encrypted_body/>`

  – `<sign_body>`

  – `<encrypt_body>`

  We now discuss each of these optional elements.

- `<authentication>`

  Specifies the use of security tokens in the headers of inbound and outbound SOAP messages. It can be used in either a service provider or a service requester pipeline. In a service provider pipeline, the element specifies whether CICS should use the security tokens in an inbound SOAP message to determine the user ID under which work will be processed. In a service requester pipeline, it specifies that CICS should add an X.509 certificate to the security header for outbound SOAP messages.

  The `<authentication>` element has two attributes: *trust* and *mode*. These attributes determine whether asserted identity is used and the combination of security tokens used in a SOAP message. The *trust* attribute can be set to either none, basic or signature. The *mode* attribute can also be set to either none, basic, or signature. For more information about the meaning and valid combinations of these attributes, refer to the *CICS Transaction Server for z/OS V3.1 Web Services Guide*, SG34-6458.

Asserted identity allows a trusted user to assert, or declare, that work should run under a different identity (the *asserted identity*), without the trusted user having the credentials associated with that identity. Messages contain a *trust token* and an *identity token*. The adjectives *trust* and *identity* indicate how the token is used rather than the kind of token. A trust token, for example, could be a UsernameToken or an X.509 token. The trust token is used to check that the sender has the correct permissions to assert identities, and the identity token holds the asserted identity (user ID) under which the request is to run.

Note: If you use asserted identity, it requires that the service provider trusts the requester to make this assertion. In CICS, the trust relationship is established with security manager surrogate definitions: the requesting identity must have the correct authority to start work on behalf of the asserted identity.

The `<authentication>` element can contain the following elements:

– `<certificate_label>`
  Optional. Specifies the label associated with an X.509 digital certificate. Ignored in a service provider pipeline.

– `<suppress/>`
  Optional. For a service provider, the handler will not use any security tokens in the message to determine under which user ID to run. For a service requester, the handler will not add to the SOAP message any of the security tokens required for authentication.

– `<algorithm>`
  Specifies the URI of the signature algorithm. CICS currently supports the signature algorithms for inbound SOAP messages shown in Table 8-3.

Table 8-3 Signature algorithms for inbound SOAP messages

Algorithm	URI
Digital Signature Algorithm with Secure Hash Algorithm 1 (DSA with SHA1)	http://www.w3.org/2000/09/xmldsig#dsa-sha1
Rivest-Shamir-Adleman algorithm with Secure Hash Algorithm 1 (RSA with SHA1)	http://www.w3.org/2000/09/xmldsig#rsa-sha1

- `<expect_signed_body/>`

  Indicates that the `<body>` of the inbound message must be properly signed. If it is not, CICS rejects the message with a security fault.

- `<expect_encrypted_body/>`

  Indicates that the `<body>` of the inbound message must be properly encrypted. If it is not, CICS rejects the message with a security fault.

- `<sign_body>`

  Directs DFHWSSE1 to sign the body of outbound SOAP messages, and provides information regarding how the messages are to be signed. It can be used in either a service provider or a service requester pipeline. It contains the following elements:

  – `<algorithm>`

    Specifies the URI of the algorithm used to sign the body of the SOAP message.

    CICS currently supports the following signature algorithm for outbound SOAP messages:

    - Rivest-Shamir-Adleman algorithm with Secure Hash Algorithm 1 (RSA with SHA1), which is specified using the URI http://www.w3.org/2000/09/xmldsig#rsa-sha1

  – `<certificate_label>`

    Specifies the label associated with an X.509 digital certificate. The digital certificate should contain the private key since this was used to sign the message. The public key associated with the private key is then sent in the SOAP message, which allows the signature to be validated.

- `<encrypt_body>`

  Directs DFHWSSE1 to encrypt the body of outbound SOAP messages, and provides information regarding how the messages are to be encrypted. It can be used in both a service provider and service requester pipeline. It contains the following elements:

  – `<algorithm>`

    Specifies the URI identifying the algorithm used to encrypt the body of the SOAP message. CICS currently supports the encryption algorithms shown in Table 8-4.

Table 8-4 Encryption algorithms

Algorithm	URI
Triple DES in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#tripledes-cbc
AES with a key length of 128 bits in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#aes128-cbc
AES with a key length of 192 bits in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#aes192-cbc
AES with a key length of 256 bits in cipher block chaining mode	http://www.w3.org/2001/04/xmlenc#aes256-cbc

– `<certificate_label>`

  Specifies the label associated with an X.509 digital certificate. The digital certificate should contain the public key of the intended recipient of the SOAP message so that it can be decrypted with the private key when the message is received.

Example 8-3 shows a `<wsse_handler>` element with all of the optional elements present. You would add this to your configuration file for the pipeline.

Example 8-3 <wsse_handler>

```
<wsse_handler>
 <dfhwsse_configuration version="1">
   <authentication trust="signature" mode="basic">
     <certificate_label>AUTHCERT03</certificate_label>
     <suppress/>
   </authentication>
   <expect_signed_body/>
   <expect_encrypted_body/>
   <sign_body>
     <algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>
     <certificate_label>SIGCERT01</certificate_label>
   </sign_body>
   <encrypt_body>
     <algorithm>http://www.w3.org/2001/04/xmlenc#tripledes-cbc</algorithm>
     <certificate_label>ENCCERT02</certificate_label>
   </encrypt_body>
 </dfhwsse_configuration>
</wsse_handler>
```

Example 8-4 shows the pipeline configuration file `basicsoap11provider.xml` for the EXPIPE01 service provider pipeline associated with the sample catalog application.

Example 8-4 CICS-supplied sample pipeline configuration file basicsoap11provider.xml

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<provider_pipeline
   xmlns="http://www.ibm.com/software/htp/cics/pipeline"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline provider.xsd">
  <service>
    <terminal_handler>
      <cics_soap_1.1_handler/>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

Example 8-5 shows how you would modify the pipeline configuration file to add the `<service_handler_list>` and `<wsse_handler>` elements to implement WS-Security. CICS will read the pipeline configuration file and when it finds the `<wsse_handler>` element it will load program DFHWSSE1 from library SDFHWSLD in your DFHRPL concatenation to process the security information. For more information about the elements for the pipeline configuration file, and which ones are contained by other elements (high-level structure diagrams), refer to the most current version of *CICS Transaction Server for z/OS V3.1 Web Services Guide*, SG34-6458.

Example 8-5 <wsse_handler> element added to configuration file basicsoap11provider.xml

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<provider_pipeline
    xmlns="http://www.ibm.com/software/htp/cics/pipeline"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipelineprovider.xsd">
  <service>
    <service_handler_list>
      <wsse_handler>
        <dfhwsse_configuration version="1">
          <authentication trust="signature" mode="basic">
            <certificate_label>AUTHCERT03</certificate_label>
            <suppress/>
          </authentication>
          <expect_signed_body/>
          <expect_encrypted_body/>
          <sign_body>
            <algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>
            <certificate_label>SIGCERT01</certificate_label>
          </sign_body>
          <encrypt_body>
            <algorithm>http://www.w3.org/2001/04/xmlenc#tripledes-cbc</algorithm>
            <certificate_label>ENCCERT02</certificate_label>
          </encrypt_body>
        </dfhwsse_configuration>
      </wsse_handler>
    </service_handler_list>
    <terminal_handler>
      <cics_soap_1.1_handler/>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

If CICS is the service provider, CICS will decrypt any inbound encrypted SOAP message automatically when it processes the message, provided you have the `<wsse_handler>` element in the pipeline configuration file. The security header in the received message provides all of the information needed for CICS to decrypt it. In other words, the `<encrypt_body>` and `<sign_body>` elements do not need to be specified in the provider pipeline configuration file in order to decrypt the inbound SOAP message. But you can (and probably will want to) include the `<encrypt_body>` or `<sign_body>`, or both, in the provider pipeline configuration file if you want to encrypt or sign the reply body sent back to the requester. This is what we have shown in Example 8-5.

8.4.2 WebSphere and SOAP message security

WebSphere Application Server V6.1 is based on the implementation of WS-Security in the following OASIS specification and profiles:

- WS-I Basic Security Profile

  `http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html`

- Web Services Security: SOAP Message Security 1.0 (March 2004):

  `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf`

- Web Services Security: UsernameToken Profile 1.0 (March 2004):

  `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf`

- Web Services Security: X.509 Certificate Token Profile V1.0 (March 2004):

  `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0.pdf`

WebSphere Application Server uses the J2EE 1.4 Web services deployment model to implement WS-Security. The Web services security constraints are specified in the IBM extension of the Web services deployment descriptors and bindings. The Web services security runtime enforces the security constraints specified in the deployment descriptors. One of the advantages of this deployment model is that you can define the Web services security requirements outside of the application business logic. With the separation of roles, the application developer can focus on the business logic, and the security expert can specify the security requirement.

Figure 8-6 shows the high-level architecture model that is used to secure Web services in WebSphere Application Server.

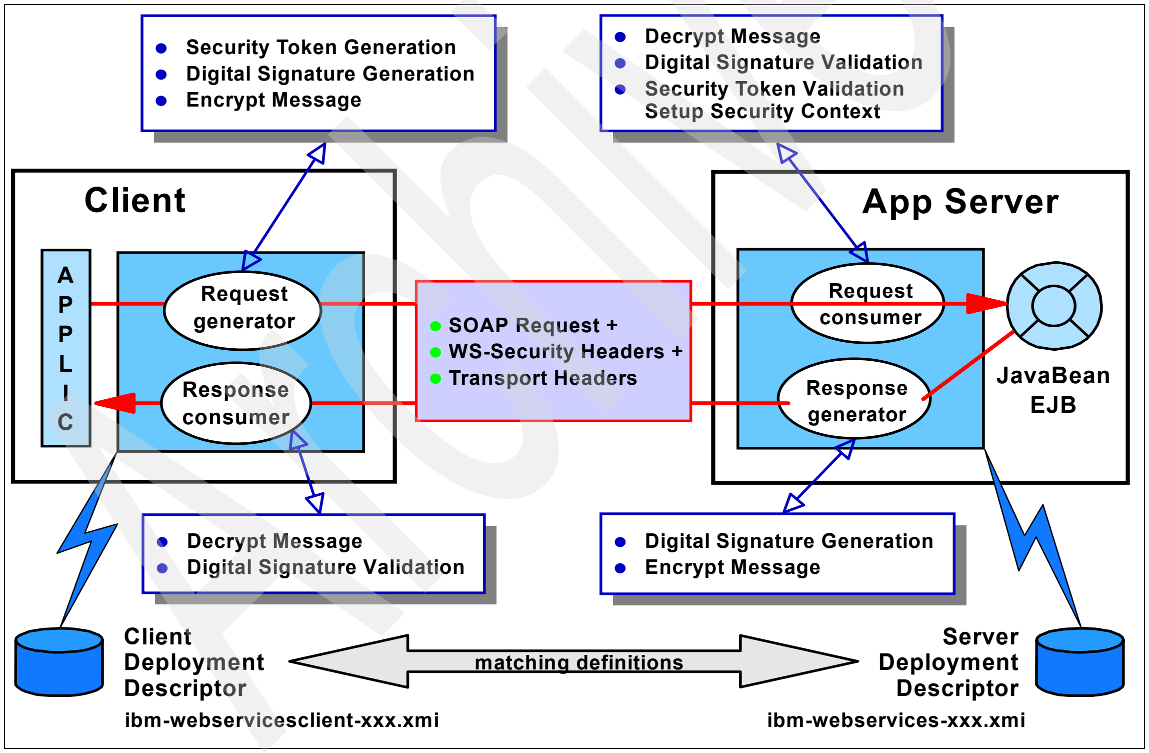


Figure 8-6 WebSphere Application Server support for WS-Security

As shown in the figure, there are two sets of configurations on both the client side and the server side.

- **Request generator**

  This client-side configuration defines the Web services security requirements for the outgoing SOAP message request. These requirements might involve generating a SOAP message request that uses a digital signature, incorporates encryption, and attaches a security token.

- **Request consumer**

  This server-side configuration defines the Web services security requirements for the incoming SOAP message request. These requirements might involve verifying that the required integrity parts are digitally signed, verifying the digital signature, verifying that the required confidential parts were encrypted by the request generator, decrypting the required confidential parts, validating the security token, and verifying that the security context is set up with the appropriate identity.

- **Response generator**

  This server-side configuration defines the Web services security requirements for the outgoing SOAP message response. These requirements might involve generating the SOAP message response with Web services security, including digital signature; and encrypting and attaching a security token, if necessary.

- **Response consumer**

  This client-side configuration defines the Web services security requirements for the incoming SOAP response. The requirements might involve verifying that the integrity parts are signed and the signature is verified; verifying that the required confidential parts are encrypted and that the parts are decrypted; and validating the security token, if necessary.

The Web services security requirements that are defined in the request generator must match the request consumer. The requirements that are defined in the response generator must match the response consumer. Otherwise, the request or response is rejected because the Web services security constraints cannot be met by the request consumer and response consumer.

The format of the Web services security deployment descriptors and bindings are IBM proprietary. However, the following tools are available to edit the deployment descriptors and bindings:

- Rational Application Developer V6.0 (RAD)

  In Chapter 9, "Security scenarios" on page 275 we show how we configured WS-Security for our sample catalog manager application using RAD.

- Application Server Toolkit V6.1 (AST)

In Chapter 10, “Security scenarios using CICS WS-Security support” on page 323 we show how we configured WS-Security using the AST.

8.5 Comparison of transport level and SOAP message security

We have shown in this chapter that it is possible to implement Web services security at two levels: the transport level and the SOAP message level. If your Web services environment is simple (for example, it does not span multiple nodes) a security solution based on transport-level security alone may be all that you need. For more complex scenarios, however, it may not be enough on its own.

In this section, we provide general guidelines to help you decide what type of security solution to implement.

- You might chose to use only transport-level security to secure your CICS Web services environment when:
  - No intermediaries are used in the Web service environment or, if there are intermediaries, then you can guarantee that once the data is decrypted, it cannot be accessed by an untrusted node or process.
  - The transport is only based on HTTP.
  - Performance is your primary concern

    SSL/TLS is a mature technology that has been optimized over a long period of time and there are ways of optimizing performance such as persistent TCP/IP connections and SSL session ID reuse. These optimizations mean that expensive security functions, such as SSL handshaking, can be avoided for service requests following the initial handshake.

    WS-Security support, in comparison, is completely stateless, and expensive security functions, such as XML digitital signature validation, are repeated for each service request.
  - The Web services client is a stand-alone Java program.

    WS-Security can only be applied to clients that run in a Web services environment that supports the WS-Security specification (for example, WebSphere Application Server).
- You might chose to use WS-Security (possibly in addition to transport-level security) when:
  - Intermediaries are used, some of which may be untrusted.

Security credentials that flow in the SOAP message can pass through any number of intermediaries. Protecting confidential information in the actual SOAP message may avoid the overhead of encrypting and decrypting via SSL at every intermediary node.

Furthermore, an intermediary may be able to provide an authentication service to CICS, such that the intermediary server authenticates the Web service client and then flows an *asserted* identity to CICS.

– Multiple transport protocols are used.

  WS-Security works across multiple transports and is independent of the underlying transport protocol.

– The Web service partners support WS-Security and a general decision has been taken to flow security tokens in accordance with the WS-Security specification.

– You choose to implement your own security procedures and processing by writing a custom message handler program that can process secure SOAP messages in the pipeline.

8.6 Securing CICS Web services using the service integration bus

WebSphere Application Server provides the ability to use the *service integration bus* as an intermediary between service requesters and service providers, allowing you to control the flow, routing, and transformation of messages through mediations and JAX-RPC handlers (see Chapter 5, “Connecting CICS to the service integration bus” on page 129).

The bus provides a flexible way to expose and call services located in an intranet from the Internet (and vice versa), while also providing mechanisms for protocol switching and security, including extensive support for the WS-Security specification.

Figure 8-7 illustrates how the bus could be used to enable service requesters to access CICS Web services via a *gateway service*. See “Creating a gateway service on the bus” on page 139 for information about how we enabled the inquireSingle Web service of the CICS sample catalog application to be invoked via a gateway service.

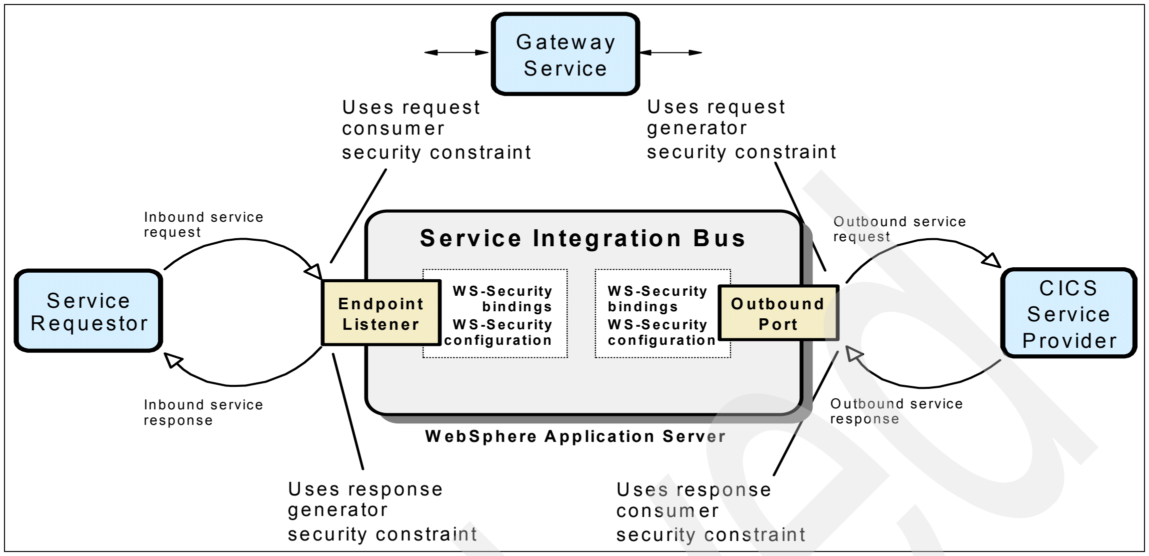

Gateway Service
Uses request consumer security constraint
Uses request generator security constraint
Inbound service request
Outbound service request
Service Integration Bus
Service Requestor
Endpoint Listener
WS-Security bindings
WS-Security configuration
WS-Security bindings
WS-Security configuration
Outbound Port
CICS Service Provider
Inbound service response
Outbound service response
WebSphere Application Server
Uses response generator security constraint
Uses response consumer security constraint

Figure 8-7 Securing a CICS Web service using the service integration bus

One advantage of such a configuration is that you can configure the bus for secure transmission of SOAP messages using tokens, digital signatures, and encryption, in accordance with the WS-Security specification. For each service, you can select the security settings that are applied between the service requester and the gateway service, and between the gateway service and the service provider (in this case, CICS).

WS-Security security constraints can be configured for the following:

- Request consumer - used on inbound requests from a client to a service.
- Request generator - used when generating outbound requests from a service to a target Web service.
- Response consumer - used on outbound responses from a target Web service to a service.
- Response generator - used when generating inbound responses from a service to a client.

The bus can also invoke Web services that include https:// in their addresses. This provides message integrity and confidentiality while the message is transmitted between the bus and the CICS region.

8.7 WebSphere Datapower SOA appliances

IBM WebSphere Datapower SOA appliances are purpose-built, easy-to-deploy network devices that simplify and accelerate XML and Web services deployments.

There are three types of DataPower® appliances available, each building on the features of the last:

- IBM WebSphere DataPower XML Accelerator XA35

  Accelerates common types of XML processing by offloading this processing from servers and networks. It can perform XML parsing, XML Schema validation, XPath routing, Extensible Stylesheet Language Transformations (XSLT), XML compression, and other essential XML processing with wirespeed XML performance.

- IBM WebSphere DataPower XML Security Gateway XS40

  Provides a security-enforcement point for XML and Web services transactions, including encryption, firewall filtering, digital signatures, schema validation, WS-Security, XML access control, XPath, and detailed logging.

- IBM WebSphere DataPower Integration Appliance XI50

  Transport-independent transformations between binary, flat text files, and XML message formats. Visual tools are used to describe data formats, create mappings between different formats, and define message choreography.

For full product information about IBM WebSphere DataPower SOA Appliances see:

```
http://www-306.ibm.com/software/integration/datapower/index.html
```

A Datapower SOA appliance can be used in conjunction with CICS Web services to help secure the services and to offload expensive operations by processing the complex part of XML messages (such as an XML signature) at wirespeed (see Figure 8-8).

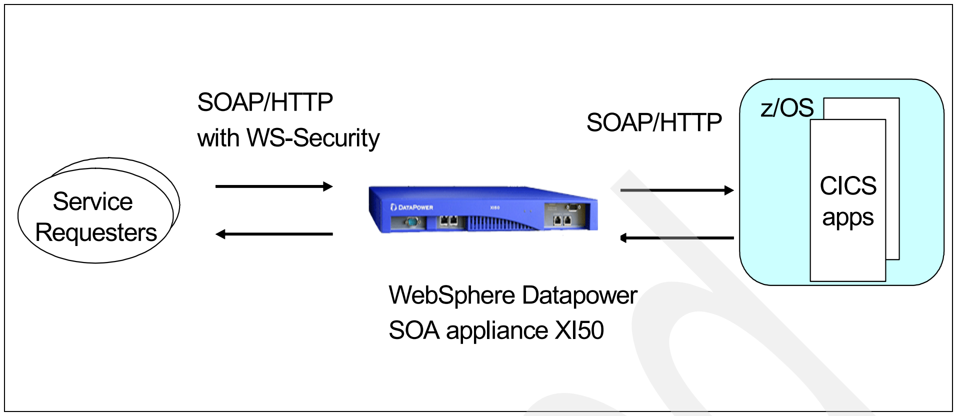


Figure 8-8 Using a WebSphere Datapower SOA Appliance with CICS Web services

8.8 Identity assertion

Identity assertion is an authentication mechanism that is applied among three parties: a client, an intermediary server, and a target server:

- ► A request message is sent to an intermediary server with a client's security token.
- ► The intermediary server (for example, a service integration bus or a Datapower SOA appliance) authenticates the client and transfers the client's request message and identity to the target server with the *intermediary's* security token.

Identity assertion is an extended security mechanism supported by WebSphere Application Server Version V6 and the service integration bus. There are several options for sending the client's identity with the intermediary's token to the target server.

It is possible to use identity assertion to secure a CICS Web service via the bus. For example, for the Gateway service shown in Figure 8-7, we could:

- ► Define a security requirement for the request consumer such that the requester must provide a binary security token, such as an X.509 certificate.
- ► Configure the bus (WebSphere Application Server) to map the certificate to a user identity.
- ► Define a security requirement for the request generator such that the bus propagates the service requester's identity to CICS in a UsernameToken.

The bus must establish a trust relationship with the CICS region by authenticating itself and then by being recognized as a trusted partner of the CICS region. This can be done using one of two different models:

Trust token	The bus sends a trust token to CICS.
Presumed trust	Trust is established at the transport level rather than at the SOAP message level.

8.8.1 Trust token model

In this model, SOAP messages that flow between the bus and CICS contain a trust token and an identity token. The trust token is used to check that the sender has the correct permissions to assert identities, and the identity token holds the asserted identity (user ID) under which the request is to run. When using the CICS-provided support for WS-Security, a trust token can be a UsernameToken or an X.509 token.

The advantage of the trust token model is that it is independent of the mechanism used to transport the SOAP messages. The disadvantage, however, is the overhead of validating the trust token for each SOAP request.

8.8.2 Presumed trust model

In this model, SOAP messages that flow between the bus and CICS contain only an identity token. The trust relationship between the bus and CICS must be established using a transport-based mechanism such as SSL client authentication.

The CICS-provided support for WS-Security does not support the presumed trust model. To implement such a model, you need to write a custom handler. The user-written header processing program CIWSSECS described in "Configuring the service provider" on page 316 is an example of such a custom handler.

The advantage of the presumed trust model is that the trust established between the bus and CICS can be persistent (for example, by using SSL persistent connections) and does not need to re-established for each SOAP message. The disadvantage, however, is that it is dependent on the mechanism used to transport the SOAP messages.

Recommendation: The presumed trust model offers significant performance advantages over the trust token model.

Security scenarios

In this chapter we outline several security scenarios that demonstrate how you can secure a CICS Web services environment. We start with an explanation of how we prepared the system and the settings that we used for the basic security configuration of our CICS system. We then provide step-by-step security configuration for a number of scenarios, including:

- Setting the user ID on a URIMAP resource definition
- Enabling SOAP message security with HTTP
- Enabling SSL/TLS
- Enabling SOAP message security with WebSphere MQ

Important: The SOAP message security scenarios documented in this chapter were tested on a version of CICS TS V3.1 which did not have the CICS WS-Security PTFs installed. See Chapter 10, “Security scenarios using CICS WS-Security support” on page 323 for scenarios using the CICS supplied WS-Security support.

We show how we configured both CICS and WebSphere Application Server for these security scenarios.

9.1 Preparation

Security techniques for securing Web services were discussed in Chapter 8, “Securing Web services” on page 235. In this chapter, we describe the following security scenarios:

- Setting the user ID on a URIMAP definition

  The user ID is then used for Web service requests that match the URIMAP. No authentication of the clint’s credentials takes place. See “Setting the user ID on a URIMAP definition” on page 280 for details of this scenario.

- Enabling SOAP message security with HTTP

  In this scenario, we show how a header processing program in a service provider pipeline can extract a WS-Security UsernameToken from a SOAP header, validate the username and password, and set the user ID of the CICS task to the username passed in the header. This scenario is described in “Enabling SOAP message security with HTTP” on page 285.

- Enabling SSL/TLS

  In this scenario, we show how SSL/TLS can be used in combination with SOAP message security to secure a connection between WebSphere Application Server and CICS. See “Enabling SSL/TLS” on page 301.

- Enabling SOAP message security with WebSphere MQ (WMQ)

  In this scenario, we show how a header processing program in a service requester pipeline can attach a WS-Security header to a SOAP message, which is then sent, using WMQ, to a CICS service provider. This scenario is described in “Enabling SOAP message security with WMQ” on page 312.

9.1.1 Software checklist

The software we used is listed in Table 9-1.

Table 9-1 Software used in the security scenarios

Windows	z/OS
Internet Explorer V6.0	z/OS V1.6
Windows 2000 SP4	CICS Transaction Server V3.1[a]
IBM WebSphere Application Server - ND V6.0.2.0	RACF V1.6

Windows	z/OS
Our J2EE applications ► CatalogSec.ear Catalog manager service requester application with no security enabled ► CatalogSec_WS-Security.ear Catalog manager service requester application with WS-security enabled ► CatalogSec_WS-Security_HTTPS.ear Catalog manager service requester application with WS-security and HTTPS enabled	Our user-written CICS programs ► CIWSMSGH (message handler program) ► CIWSSECH (header processing program) ► CIWSSECR (header processing program) ► CIWSSECS (header processing program) ► MYPARSER (COBOL XML parsing program)

a. The version of CICS TS V3.1 used in these security scenarios does not include the CICS support for WS-Security which was introduced later in PTFs UK15271 and UK15261.

9.1.2 Definition checklist

The definitions we used are listed in Table 9-2.

Table 9-2 Settings used in the security scenarios

Value	CICS TS	WebSphere Application Server
IP name	mvsg3.mop.ibm.com	cam21-pc3.mop.ibm.com
IP address	9.100.193.167	9.100.199.171
TCP/IP port	14301	9080
SSL/TLS port	14303	
Jobname	CIWSS3C1	
APPLID	A6POS3C1	
TCPIPSERVICE	S3C1	
Provider PIPELINE	PIPE1	
Configuration files	ITSO_7206_secprovider.xml ITSO_7206_secrequester.xml ITSO_7206_secprovider_dispatch.xml	
MQ queue manager	MQS3	

Value	CICS TS	WebSphere Application Server
URIMAP	SECPORDR SECICATA SECISING	

The user IDs we used in our configuration are listed in Table 9-3.

Table 9-3 User IDs

Value	CICS TS
CICS region user ID	CIWS3D
CICS default user ID	CICSUSER
User IDs to which we wish to permit access	CIWSNW WEBAS3

Tip: For the examples described in this book, we permit access to single user IDs. In a production environment, you will probably create a group of users requiring common access. Once a group is built, you can permit access to the group. This allows users' access to be controlled by the group to which they belong, rather than by individual permissions. This simplifies the security definitions required.

9.2 Basic security configuration

First we discuss our basic security configuration, taking a CICS region with no security and configuring it to enable transaction security (we do not implement other types of CICS security such as resource security and command security). We document the system initialization table parameters necessary to set up basic CICS security and then we test our basic security configuration.

For detailed information about CICS security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454.

Because the INQC and INQS transactions are browse-only transactions, we chose not to secure them. The ORDR transaction updates the database, so we decided to protect it from unauthorized use.

Our starting point is a CICS region with no security configured. This means that all transactions run under the CICS default user ID, as shown in Figure 3-12 on page 93.

9.2.1 Setting up basic security configuration

We configured our CICS region with security prefixing, transaction security, and surrogate user security active using the following SIT parameters:

- `SEC=YES`
- `SECPRFX=YES`
- `XTRAN=YES`
- `XUSER=YES`

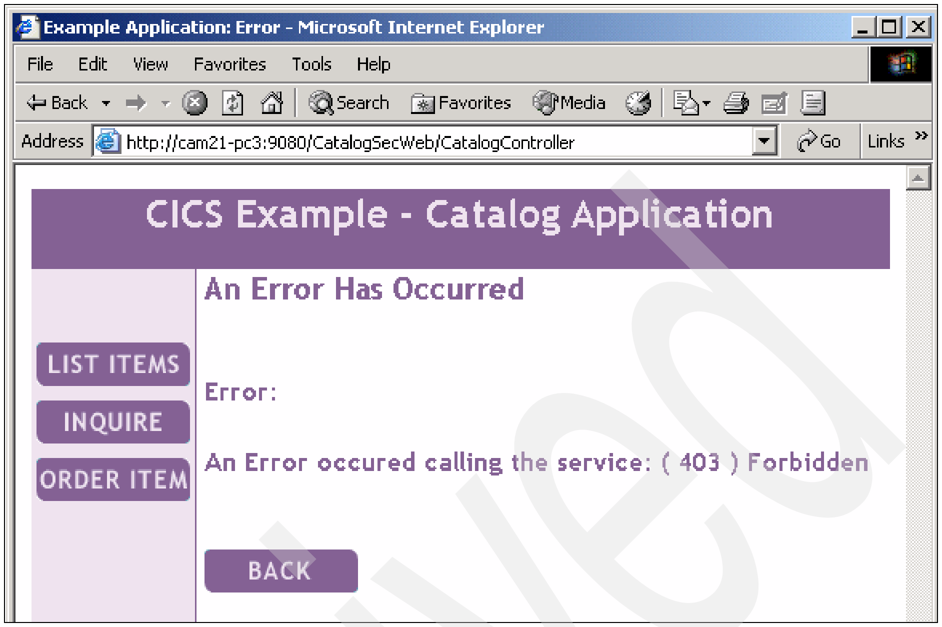
SEC=YES was specified to indicate that we wanted RACF services to control access to CICS resources.

We used security prefixing (SECPRFX=YES) in our CICS region, which prevents our RACF security profiles from affecting other CICS regions. This is useful in a production environment since it means all security profiles are unique to an individual region; however, it can mean more work for the security administrator because more profiles must be defined.

XTRAN=YES was specified so CICS would control who could start transactions and XUSER=YES specifies that CICS is to perform surrogate user checking.

9.2.2 Testing the basic security configuration

After enabling security in our CICS region, we attempted to place an order with the sample catalog application. The Web browser received an error message indicating that the use of some transaction is forbidden (Figure 9-1).



Example Application: Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://cam21-pc3:9080/CatalogSecWeb/CatalogController
Go Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM
BACK

An Error Has Occurred

Error:

An Error occured calling the service: (403) Forbidden

Figure 9-1   Browser - Service call forbidden

Example 9-1 shows the security violation highlighting that the CICS default user ID does not have access to the CPIH transaction.

Example 9-1   CICSUSER security violation messages running CPIH

```
DFHXS1111 11/24/2005 12:33:04 A6POS3C1 CPIH Security violation by user CICSUSER for resource
            CIWS3D.CPIH in class TCICSTRN. SAF codes are (X'00000008',X'00000000'). ESM codes
            are (X'00000008',X'00000000').
DFHWB0361 11/24/2005 12:33:04 A6POS3C1 An attempt to attach a CICS Web alias transaction for
            userid CICSUSER has failed because the user is not authorized to execute transaction
            CPIH. Host IP address: 9.100.193.167. Client IP address: 9.100.199.171.
            TCPIPSERVICE: S3C1.
DFHAC2003 11/24/2005 12:33:04 A6POS3C1 Security violation has been detected term id = ????,
            trans id = CPIH, userid = CICSUSER.
```

9.3 Setting the user ID on a URIMAP definition

With this option the security identity used for the transaction is based on a statically defined user ID. We achieved this preset identification by using a URIMAP resource definition in which we specified a valid user ID (CIWSNW) in the USERID parameter. This is shown in Figure 9-2.

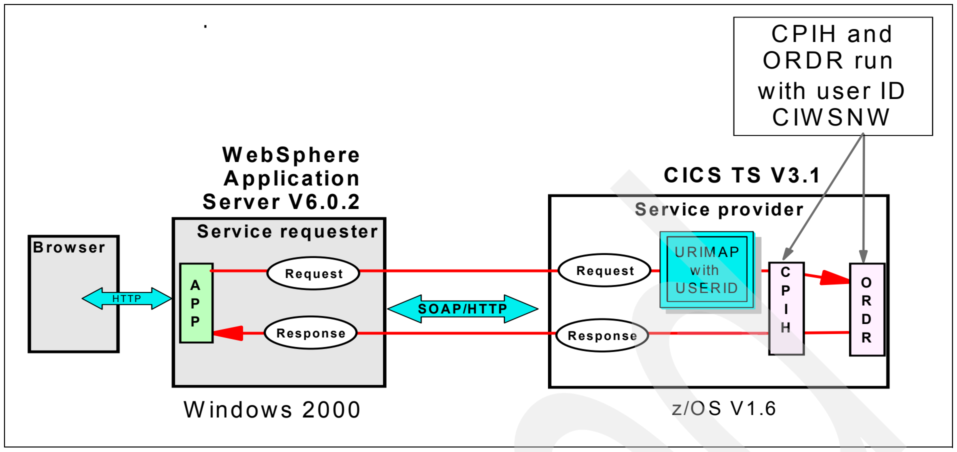


Figure 9-2 Setting user ID on a URIMAP scenario

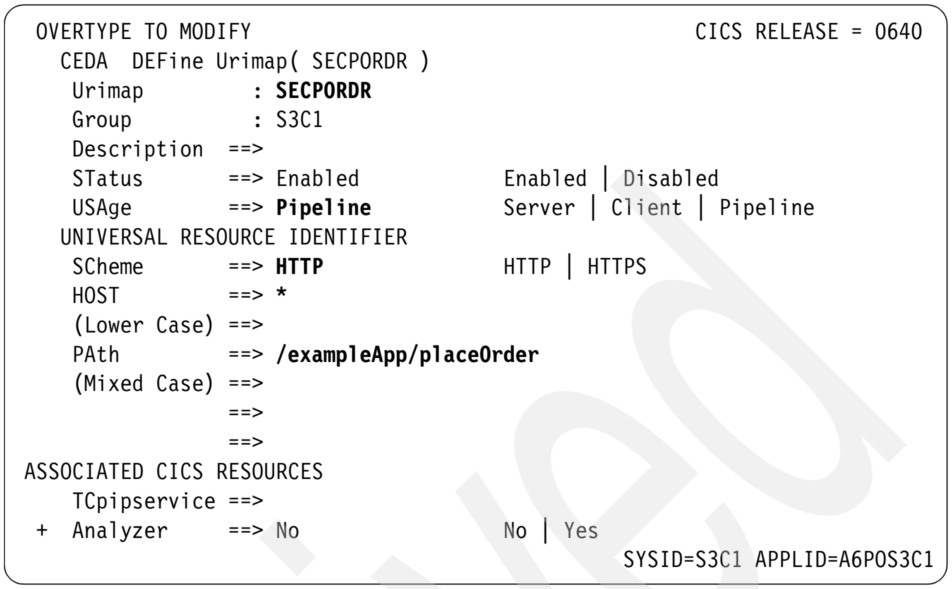
At this stage, the only security check is to validate whether or not the user ID specified in the URIMAP attribute (CIWSNW) has permission to execute both the CPIH and ORDR transactions.

9.3.1 Defining the URIMAP

We used the following command to define a URIMAP resource:

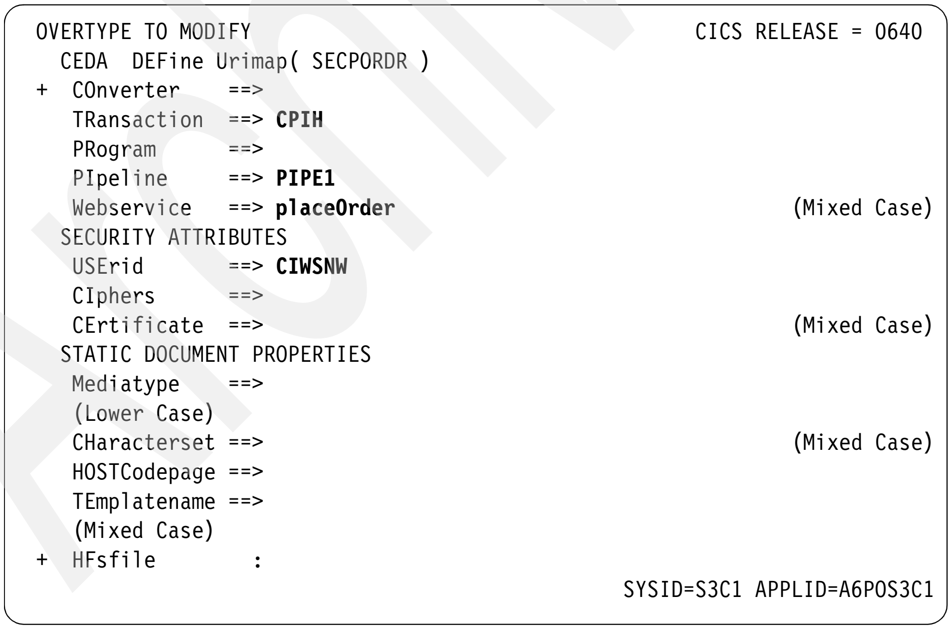
```
CEDA DEFINE URIMAP(SECPORDR) GROUP(S3C1)
```

We then set the attributes as shown in Figure 9-3 and Figure 9-4.



```
 OVERTYPE TO MODIFY                                          CICS RELEASE = 0640
   CEDA  DEFine Urimap( SECPORDR )
    Urimap         : SECPORDR
    Group          : S3C1
    Description  ==>
    STatus       ==> Enabled            Enabled | Disabled
    USAge        ==> Pipeline           Server | Client | Pipeline
   UNIVERSAL RESOURCE IDENTIFIER
    SCheme       ==> HTTP               HTTP | HTTPS
    HOST         ==> *
    (Lower Case) ==>
    PAth         ==> /exampleApp/placeOrder
    (Mixed Case) ==>
                 ==>
                 ==>
ASSOCIATED CICS RESOURCES
    TCpipservice ==>
 +  Analyzer     ==> No                 No | Yes
                                                        SYSID=S3C1 APPLID=A6POS3C1
```

Figure 9-3 CEDA DEFINE URIMAP (page 1 of 2)



```
 OVERTYPE TO MODIFY                                          CICS RELEASE = 0640
   CEDA  DEFine Urimap( SECPORDR )
 +  COnverter    ==>
    TRansaction  ==> CPIH
    PRogram      ==>
    PIpeline     ==> PIPE1
    Webservice   ==> placeOrder                                     (Mixed Case)
   SECURITY ATTRIBUTES
    USErid       ==> CIWSNW
    CIphers      ==>
    CErtificate  ==>                                                (Mixed Case)
   STATIC DOCUMENT PROPERTIES
    Mediatype    ==>
    (Lower Case)
    CHaracterset ==>                                                (Mixed Case)
    HOSTCodepage ==>
    TEmplatename ==>
    (Mixed Case)
 +  HFsfile        :
                                                        SYSID=S3C1 APPLID=A6POS3C1
```

Figure 9-4 CEDA DEFINE URIMAP (page 2 of 2)

We used the following values:

- URIMAP name was set to SECPORDR.
- USAGE was set to PIPELINE, indicating that it is to be used for incoming Web service requests.
- SCHEME was set to HTTP because no transport security is being used.
- HOST attribute was set to the wildcard (*), meaning that the URIMAP definition matches on any host name.
- PATH was set as /exampleApp/placeOrder. Together with the HOST and SCHEME values this means that the URIMAP resource caters to any Web service request originating from `http://*/exampleApp/placeOrder`.
- TRANSACTION is automatically defaulted to CPIH by CICS when USAGE(PIPELINE) is specified.
- PIPELINE was set to PIPE1, the name of the pipeline for which this URIMAP definition is being made.
- WEBSERVICE can only be specified when USAGE(PIPELINE) is in effect and specifies the name of the Web service to be invoked. We set the value to placeOrder.
- USERID was set with the value of CIWSNW, the user ID under which we have decided to execute the ORDR transaction.
- All other values were allowed to default.

Two other URIMAP resources were defined with similar definitions. First SECPORDR was copied as SECICATA and as SECISING, then these resource definitions were altered as shown in Example 9-2.

Example 9-2 Creating URIMAP definitions for SECICATA and SECISING

```
CEDA COPY URIMAP(SECPORDR) GROUP(S3C1) AS(SECICATA)
CEDA COPY URIMAP(SECPORDR) GROUP(S3C1) AS(SECISING)
CEDA ALTER URIMAP(SECICATA) GROUP(S3C1) PATH(/exampleApp/inquireCatalog)
       WEBSERVICE(inquireCatalog)
CEDA ALTER URIMAP(SECISING) GROUP(S3C1) PATH(/exampleApp/inquireSingle)
       WEBSERVICE(inquireSingle)
```

This ensures that the transactions INQC, INQS, and ORDR all execute with the same user ID CICSNW.

Example 9-3 shows the RACF commands that we used to permit the CICS region user ID (CIWS3D) to start transactions (such as ORDR) with the user ID CIWSNW.

Example 9-3 RACF commands to allow CIWS3D to act as surrogate for CIWSNW

```
RDEFINE SURROGAT CIWSNW.DFHSTART UACC(NONE) OWNER(CIWSNW)
PERMIT CIWSNW.DFHSTART CLASS(SURROGAT) ID(CIWS3D) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information regarding surrogate user security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454.

9.3.2 Permitting access to user ID CICSNW

In “Writing the message handler program” on page 80 we explained how the message handler program CIWSMSGH is used to change the default transaction ID (CPIH) to a transaction ID based on the Web service request in the DFHWS-WEBSERVICE container. Example 9-4 shows the RACF commands that we used to limit access to the ORDR transaction to a single user ID (CIWSNW).

Example 9-4 RACF commands to allow access to CPIH and ORDR transactions

```
RDEFINE GCICSTRN CIWS3D ADDMEM(CIWS3D.CPIH) UACC(NONE) OWNER(IBMUSER)
RALTER  GCICSTRN CIWS3D ADDMEM(CIWS3D.ORDR)
PERMIT CIWS3D CLASS(GCICSTRN) ID(CIWSNW) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

In Example 9-4 we:

- Define the transaction group CIWS3D and add the CPIH and ORDR transactions to the group
- Permit access to the group for user ID CIWSNW
- Refresh the RACF CICS transaction class TCICSTRN

9.3.3 Testing user ID on URIMAP resource definition

With these definitions installed, both the CPIH and ORDR transactions now execute with the user ID as supplied in the URIMAP. This is shown in Figure 9-5.

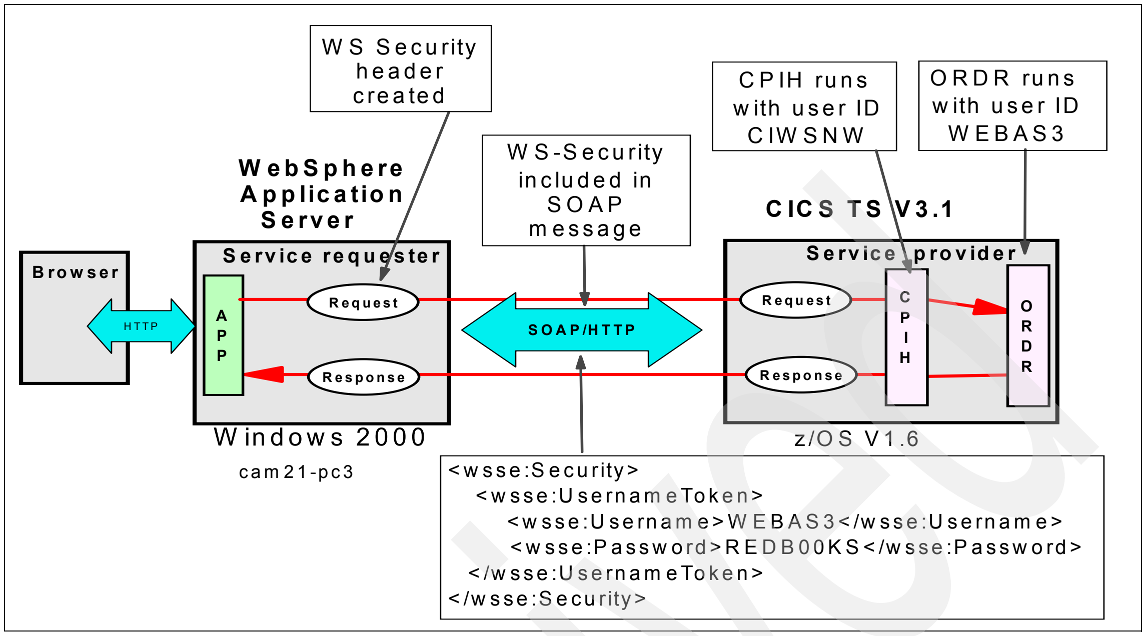
```
INQUIRE TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000050) Tra(CEMT) Fac(G312) Run Ter Pri( 255 )
    Sta(TO) Use(CIWSGA  ) Uow(BDF617FA8ADB8006)
 Tas(0000052) Tra(CPIH)           Sus Tas Pri( 001 )
    Sta(U ) Use(CIWSNW  ) Uow(BDF617FF222CDF0D) Hty(RZCBNOTI)
 Tas(0000053) Tra(ORDR)           Sus Tas Pri( 001 )
    Sta(U ) Use(CIWSNW  ) Uow(BDF617FF55DCA22A) Hty(EDF     )
 Tas(0000055) Tra(CEDF) Fac(G343) Sus Ter Pri( 001 )
    Sta(SD) Use(CIWSGA  ) Uow(BDF617FF7D31E968) Hty(ZCIOWAIT)
```

Figure 9-5 ORDR executing with preset user ID

9.4 Enabling SOAP message security with HTTP

In Section 8.4, “SOAP message security” on page 253, we explained how WS-Security extends the SOAP specification by defining new elements to be used in the SOAP header for message-level security. We also provided an outline of how WebSphere Application Server supports WS-Security, including support for generating security tokens that are passed in the WS-Security SOAP header. This scenario is shown in Figure 9-6 on page 286.

In this section we show how a simple UsernameToken security token, passed in a Web service call from WebSphere Application Server, can be processed in CICS by a user-written header processing program. The header processing program extracts the UsernameToken from the SOAP header, validates the username and password, and sets the user ID of the CICS task to the username passed in the header.


WS Security header created
WebSphere Application Server
WS-Security included in SOAP message
CPIH runs with user ID CIWSNW
ORDR runs with user ID WEBAS3
CICS TS V3.1
Browser
HTTP
Service requester
APP
Request
Response
SOAP/HTTP
Service provider
Request
Response
CPIH
ORDR
Windows 2000
cam21-pc3
z/OS V1.6

```
<wsse:Security>
   <wsse:UsernameToken>
      <wsse:Username>WEBAS3</wsse:Username>
      <wsse:Password>REDB00KS</wsse:Password>
  </wsse:UsernameToken>
</wsse:Security>
```

Figure 9-6 Enabling SOAP message security scenario

9.4.1 Configuring the service requester

In “Installing the service requester” on page 87 we showed how we installed our sample Web service client in WebSphere Application Server. The service requester setup tasks that we cover here are as follows:

- Defining the WebSphere WS-Security constraint for the Web service client
- Re-deploying the Web service client application

WebSphere WS-Security configuration files

The WebSphere WS-Security constraints are defined in the IBM extension of the J2EE Web services deployment descriptor. There are four configuration files: application-level deployment descriptor extensions for a client and a server, and binding files for a client and a server (Figure 9-7).

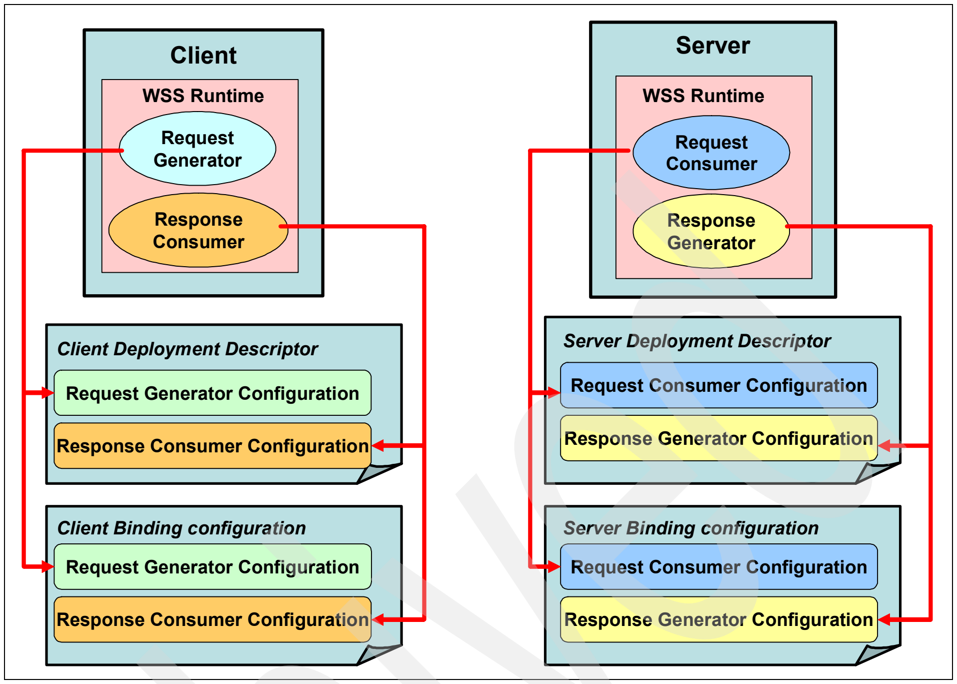

Client
WSS Runtime
Request Generator
Response Consumer
Client Deployment Descriptor
Request Generator Configuration
Response Consumer Configuration
Client Binding configuration
Request Generator Configuration
Response Consumer Configuration
Server
WSS Runtime
Request Consumer
Response Generator
Server Deployment Descriptor
Request Consumer Configuration
Response Generator Configuration
Server Binding configuration
Request Consumer Configuration
Response Generator Configuration

Figure 9-7 Structure of WS-Security configuration files

The configuration files are:

- Client deployment descriptor extension file - includes request generator and response consumer constraints:

  ibm-webservicesclient-ext.xmi

- Client binding configuration file - includes how to apply request generator and response consumer constraints:

  ibm-webservicesclient-bnd.xmi

- Server deployment descriptor extension file - includes request consumer and response generator constraints:

  ibm-webservices-ext.xmi

- Server binding configuration file - includes how to apply request consumer and response generator constraints:

  ibm-webservices-bnd.xmi

Note: The *.xmi* in the file names stands for *XML metadata interchange*.

The deployment descriptor extension files specify *what* security constraints are required, for example, what type of security token and whether to sign the message. The binding files specify *how* to apply the required security constraints defined in the deployment descriptor extension, for example, which security token is inserted and which keys are used for signing.

These deployment descriptor extension and binding files define the application-level security constraints and they belong to each application.

For our scenario, we needed to configure the client deployment descriptor and the client binding only, since our service provider is running in CICS and not in WebSphere Application Server.

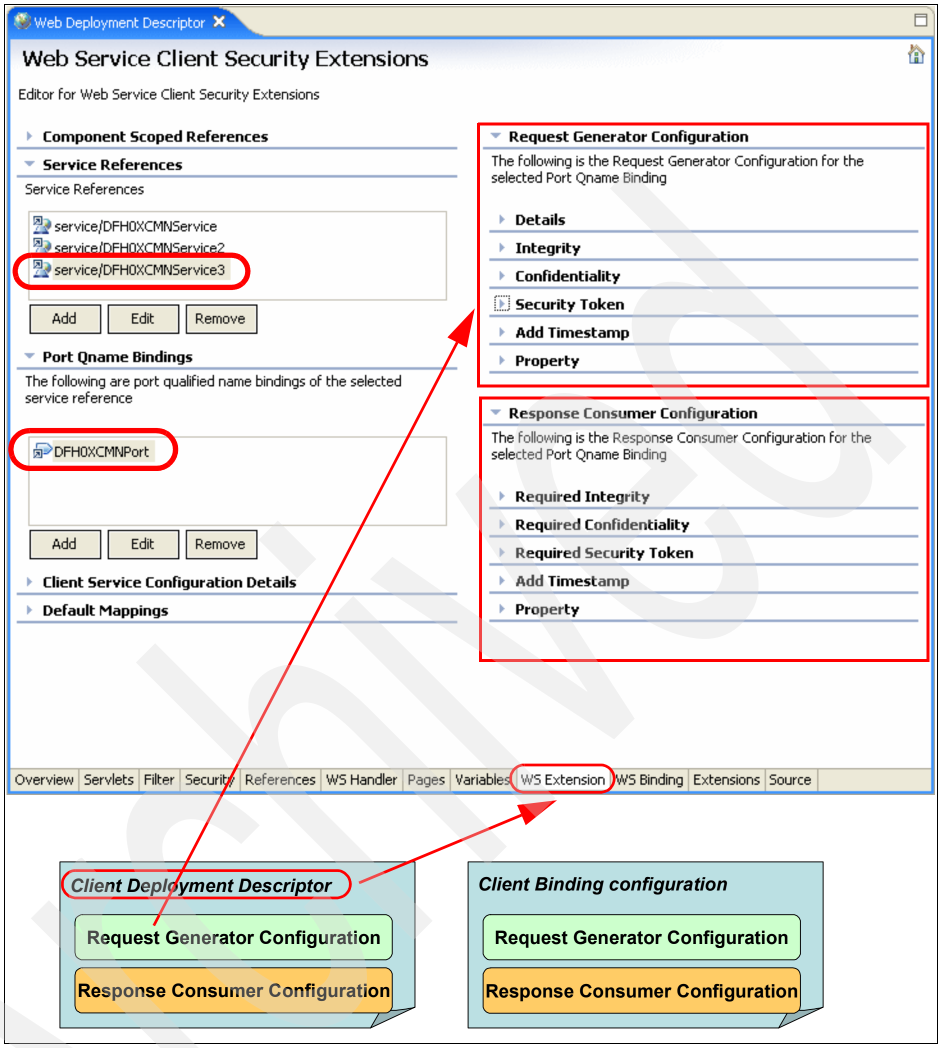
Editing the client configuration

To configure our Web service client to use WS-Security, we used Rational Application Developer V6.0 (RAD).

Note: You can also use the Application Server Toolkit to configure WS-Security for WebSphere applications.

We imported the client application archive CatalogSec.ear into RAD. We then expanded the Dynamic Web Project (CatalogSecWeb) in the Project Explorer and opened (double-clicked) the deployment descriptor.

After the Deployment Descriptor Editor opened, we selected the **WS Extension** page. The WS Extension page is used for editing the client's deployment descriptor extension file, so you can specify what security is required. Figure 9-8 shows the WS Extension page in the Deployment Descriptor Editor.


Web Deployment Descriptor

Web Service Client Security Extensions

Editor for Web Service Client Security Extensions

- Component Scoped References
- Service References

Service References

- service/DFH0XCMNService
- service/DFH0XCMNService2
- service/DFH0XCMNService3

Add | Edit | Remove

- Port Qname Bindings

The following are port qualified name bindings of the selected service reference

- DFH0XCMNPort

Add | Edit | Remove

- Client Service Configuration Details
- Default Mappings

- Request Generator Configuration

The following is the Request Generator Configuration for the selected Port Qname Binding

- Details
- Integrity
- Confidentiality
- Security Token
- Add Timestamp
- Property

- Response Consumer Configuration

The following is the Response Consumer Configuration for the selected Port Qname Binding

- Required Integrity
- Required Confidentiality
- Required Security Token
- Add Timestamp
- Property

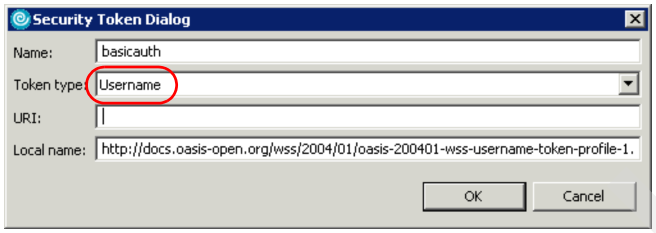
Overview | Servlets | Filter | Security | References | WS Handler | Pages | Variables | WS Extension | WS Binding | Extensions | Source

Client Deployment Descriptor
Request Generator Configuration
Response Consumer Configuration
Client Binding configuration
Request Generator Configuration
Response Consumer Configuration

Figure 9-8 WS Extension page in Web service client Deployment Descriptor Editor

- We clicked the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPort** (the port binding for the service reference).
- In the Request Generator Configuration, we expanded **Security Token** and clicked **Add**. We entered the name `basicauth`.

  We selected the **Username** Token type from the drop-down list. When you select a Token type, the Local name is filled in automatically (Figure 9-9 on page 290). We left the URI empty.


Security Token Dialog

Name: basicauth

Token type: Username

URI:

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.

Figure 9-9   Security Token Dialog for specifying basic authentication

We clicked **OK** and a security token was created. We then saved the configuration.

After specifying the security token, a corresponding token generator must be specified in the binding configuration. The WS Binding page is for editing the client's binding file, so you can specify how to apply the required security. We clicked the **WS Binding** tab. Figure 9-10 shows the WS Binding page in the Deployment Descriptor Editor.

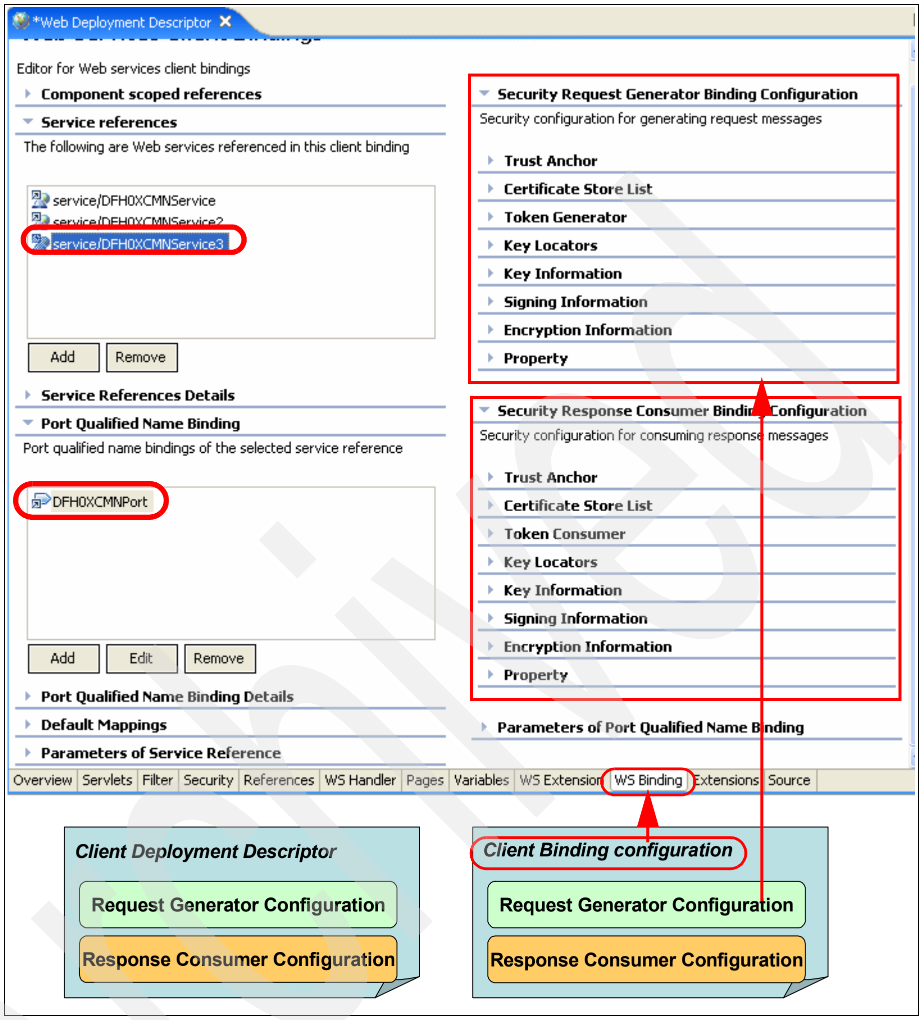
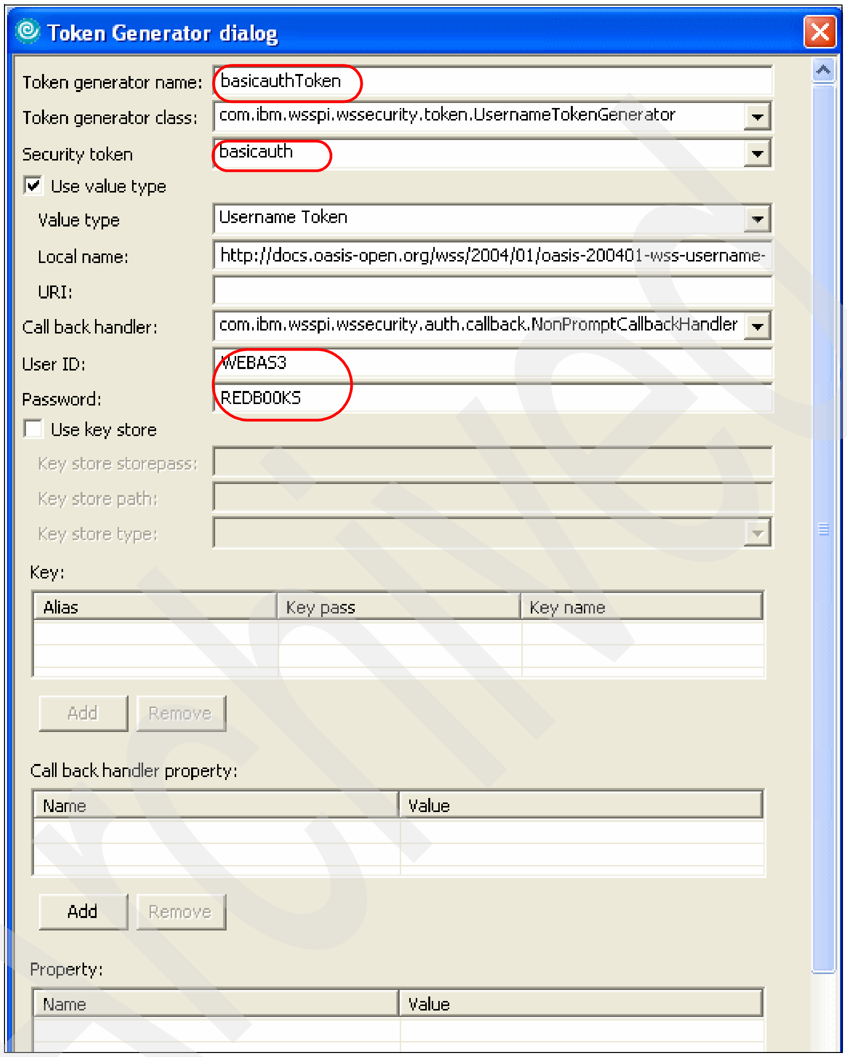

*Web Deployment Descriptor
Editor for Web services client bindings
Component scoped references
Service references
The following are Web services referenced in this client binding
service/DFHOXCMNService
service/DFHOXCMNService2
service/DFHOXCMNService3
Add
Remove
Service References Details
Port Qualified Name Binding
Port qualified name bindings of the selected service reference
DFHOXCMNPort
Add
Edit
Remove
Port Qualified Name Binding Details
Default Mappings
Parameters of Service Reference
Security Request Generator Binding Configuration
Security configuration for generating request messages
Trust Anchor
Certificate Store List
Token Generator
Key Locators
Key Information
Signing Information
Encryption Information
Property
Security Response Consumer Binding Configuration
Security configuration for consuming response messages
Trust Anchor
Certificate Store List
Token Consumer
Key Locators
Key Information
Signing Information
Encryption Information
Property
Parameters of Port Qualified Name Binding
Overview
Servlets
Filter
Security
References
WS Handler
Pages
Variables
WS Extension
WS Binding
Extensions
Source
Client Deployment Descriptor
Request Generator Configuration
Response Consumer Configuration
Client Binding configuration
Request Generator Configuration
Response Consumer Configuration

Figure 9-10 WS Binding page in the Deployment Descriptor Editor

- We selected **Token Generator** and clicked **Add**. We entered a Token generator name `basicauthToken` and allowed the Token generator class to default as `com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator`
- To select the security token, we expanded the drop-down list and selected the **basicauth** security token that we defined previously on the WS Extension page.
- We checked **Use value type** and selected the **Username Token** value type from the drop-down list. This selection filled the local name and callback handler.

- We entered the user ID WEBAS3 and password REDBOOKS that we wanted to use to identify the Web service client (Figure 9-11).



Token generator name: basicauthToken
Token generator class: com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator
Security token: basicauth
☑ Use value type
Value type: Username Token
Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-
URI:
Call back handler: com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
User ID: WEBAS3
Password: REDBOOKS
☐ Use key store
Key store storepass:
Key store path:
Key store type:

Key:

Alias	Key pass	Key name

Call back handler property:

Name	Value

Property:

Name	Value

Figure 9-11 Token Generator dialog

- We clicked **OK**, and the token generator was created. We then saved the configuration.

Note: The User ID and Password in Figure 9-11 represent the credentials of the application server, *not* the browser user. The same credentials are passed to CICS in the SOAP message irrespective of the identity of the end user.

Redeploying the Web service application

After configuring a security constraint for the service requester application, we exported a new EAR file called CatalogSec_WS-Security.ear. We followed the same process that is described in “Installing the service requester” on page 87 to deploy the CatalogSec_WS-Security.ear.

9.4.2 Configuring CICS

The credentials configured using Rational Application Developer are used to form the WS-Security header that is contained in the SOAP message. An example of this message is shown in Example 9-5.

Example 9-5 SOAP security header extracted from a SOAP message

```
<wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd">
    <wsse:UsernameToken>
      <wsse:Username>WEBAS3</wsse:Username>
      <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       wss-username-token-profile-1.0#PasswordText">REDBOOKS</wsse:Password>
    </wsse:UsernameToken>
</wsse:Security>
```

This header includes the `mustUnderstand="1"` attribute, which indicates that either this header must be processed or a SOAP fault thrown.

Note: See *CICS Transaction Server for z/OS CICS Web Services Guide*, SC34-6458, for details on `mustUnderstand` and other SOAP-defined header block attributes.

In order for CICS to process the security token in the SOAP header, the pipeline configuration file must be updated to include a SOAP header processing program. Example 9-6 shows the pipeline configuration file that activates our sample security header processing program CIWSSECH.

Example 9-6 Pipeline config file, ITSO_7206_secprovider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline provider.xsd ">
  <transport>
    <default_transport_handler_list>
      <handler>
        <program>CIWSMSGH</program>
        <handler_parameter_list/>
      </handler>
    </default_transport_handler_list>
  </transport>
  <service>
    <terminal_handler>
      <cics_soap_1.2_handler>
        <headerprogram>
          <program_name>CIWSSECH</program_name>
          <namespace>http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-
                     1.0.xsd</namespace>
          <localname>Security</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.2_handler>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

The `<headerprogram>` element shown in Example 9-6 introduces four new elements, `<program name>`, `<namespace>`, `<localname>` and `<mandatory>`:

- The `<program name>` element contains the name of the header processing program CIWSSECH, which is to be invoked to process the security token.
- The `<namespace>` element is used with the `<localname>` element to determine which header blocks in a SOAP message should be processed by the header processing program.
- A `<localname>` element is also used to determine which header blocks in a SOAP message should be processed by the header processing program. The `<localname>` contains the element name of the header block, in our case, `Security` (see Example 9-5 on page 293).
- The `<mandatory>` element indicates that the header processing program must be invoked at least once, regardless of whether or not such a header has been found.

  `<mandatory>true</mandatory>` has the following meaning:

  During request processing in a service provider pipeline, and response processing in a service requester pipeline, the header processing program is to be invoked at least once, even if none of the headers in the SOAP message matches the `<namespace>` and `<localname>` elements:

  – If none of the headers matches, the header processing program is invoked once.

  – If any of the headers match, the header processing program is invoked once for each matching header.

During request processing in a service requester pipeline, and response processing in a service provider pipeline, the header processing program is to be invoked at least once, even though the SOAP message that CICS creates has no headers initially. If you want to add headers to your message, you must ensure that at least one header processing program is invoked, by specifying `<mandatory>true</mandatory>` or `<mandatory>1</mandatory>`.

Important: If a Web service request that contains the attribute `mustUnderstand="1"` (or `="true"`) is received by CICS, and you have not coded a `<headerprogram>` element in your configuration file for this header, a SOAP fault message will be created by CICS.

Example 9-7 shows the fault message that CICS created when we had not specified a header processing program for the Security header.

Example 9-7 SOAP fault message created when no header program provided

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <SOAP-ENV:NotUnderstood qname="wsse:Security"
     xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecu
     rity-secext-1.0.xsd" />
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault xmlns="">
      <faultcode>SOAP-ENV:MustUnderstand</faultcode>
      <faultstring>Header block local name 'Security' is not defined to CICS.
       Mustunderstand check failed for the header block.</faultstring>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The WS-Security header processing program

The header processing program CIWSSECH is invoked at every invocation of this pipeline because of the `<mandatory>true</mandatory>` element, but only executes entirely if the DFHFUNCTION container holds the value `RECEIVE-REQUEST`. Example 9-8 gives an overview of the program.

Example 9-8 Pseudo code overview of the CIWSSECH program

```
Check if invoked for RECEIVE-REQUEST, else exit
Check for correct URI, else exit
Obtain WS-Security header from DFHHEADER container
Invoke MYPARSER to parse the header
Verify the credentials extracted by MYPARSER
Put user ID into DFHWS-USERID container
```

Example 9-9 shows how this is done. The full program is shown in Section A.2, "Sample header processing program - CIWSSECH" on page 533.

Note: You should *not* install CIWSSECH into a production environment in its present form. It has only been coded as a example and does not contain sufficient error checking. If you wish to use it in production, you *must* review the code and make whatever changes are required for your environment.

Example 9-9 Determining if invoked to process an inbound request message

```
       GET-DFHFUNCTION.
           EXEC CICS
              GET CONTAINER('DFHFUNCTION')
              INTO(WS-FUNC-AREA)
              FLENGTH(WS-FUNC-LEN)
              RESP(WS-RESP)
           END-EXEC.

      * Copy the input container to our storage
           IF WS-FUNC-LEN NOT = 16
              MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
              PERFORM FAULT-MESSAGE
           ELSE
              IF WS-FUNC-AREA NOT = 'RECEIVE-REQUEST '
                 EXEC CICS RETURN END-EXEC
              END-IF
           END-IF.
```

Example 9-10 shows how we determined that a specific request is for the placeOrder service.

Example 9-10 Determining the requester URI

```
           PERFORM GET-DFHWS-URI.
           IF WS-URI-AREA(1:WS-URI-LEN) NOT = '/exampleApp/placeOrder'
              EXEC CICS RETURN END-EXEC
           END-IF
```

Note: The value in the DFHWS-URI container is the same value that we specified previously for the PATH attribute of the SECPORDR URIMAP.

After verifying that we are processing a placeOrder request, the Security header (see Example 9-5 on page 293) is parsed by calling the MYPARSER program. MYPARSER is a user-written COBOL program that uses the COBOL XML PARSE statement to parse through the Security header, extracting the user ID and password. This extracted data is then returned in a COMMAREA for the calling program to use. Example 9-11 shows the procedure division for the MYPARSER program.

Example 9-11 MYPARSER COBOL program procedure division

```
PROCEDURE DIVISION.
       MAIN-PROCESSING SECTION.
      *    Validate the commarea
           PERFORM INIT-AND-VALIDATE.
      * Received a valid COMMAREA so invoke the XML parser
      * passing it the XML message
           XML PARSE XML-DOCUMENT
              PROCESSING PROCEDURE XML-HANDLER
           END-XML.
           PERFORM RETURN-RESPONSE.
           EXEC CICS RETURN END-EXEC.
       MAIN-PROCESSING-END. EXIT.
```

The XML PARSE statement parses the data in XML-DOCUMENT, using the XML-HANDLER procedure. The XML-EVENT field is evaluated and if a match is found, the code for that event is executed. The START-OF-ELEMENT and CONTENT-CHARACTERS events are handled as shown in Example 9-12.

Example 9-12 Extract of code from MYPARSER program

```
       XML-HANDLER SECTION.
           EVALUATE XML-EVENT
              WHEN 'START-OF-ELEMENT'
      * check if we have an element of interest
      * i.e. Username and Password
                 IF XML-TEXT = 'wsse:Username'
                    MOVE 'Y' TO IN-ELEM USERNAME-XMLTAG-FOUND
                 ELSE
                    if XML-TEXT = 'wsse:Password'
                       move 'Y' to in-ref-req
                    end-if
                 END-IF
              WHEN 'CONTENT-CHARACTERS'
      * If we are in an element we are interested in,
      * then extract its value
                 IF IN-ELEM = 'Y'
                    PERFORM EXTRACT-USER-ID
                 ELSE
                    IF IN-REF-REQ = 'Y'
                        PERFORM EXTRACT-PASSWORD
                    END-IF
                 END-IF
              WHEN 'END-OF-ELEMENT'
                 CONTINUE
              WHEN 'START-OF-DOCUMENT'
                 CONTINUE
```

Most of the other XML event types are handled with the CONTINUE statement because we have no need to process them. The full MYPARSER program is shown in Section A.4, "Sample XML parser program - MYPARSER" on page 550.

The parsed credentials are then verified, as shown in Example 9-13. If the user ID and password verify successfully, then the user ID is stored into the DFHWS-USERID container. This causes the business logic transaction ORDR to be executed with this user ID.

Example 9-13 Verifying the flowed credentials

```
               EXEC CICS VERIFY
                  PASSWORD(CA-PASSWORD)
                  USERID(CA-USER-ID)
                  RESP(WS-RESP)
               END-EXEC
*        Succesful ?
               IF WS-RESP = DFHRESP(NORMAL)
                  PERFORM SET-USER-ID
               ELSE
                  MOVE WS-NOT-AUTH TO WS-FAULT-STRING
                  PERFORM FAULT-MESSAGE
               END-IF
```

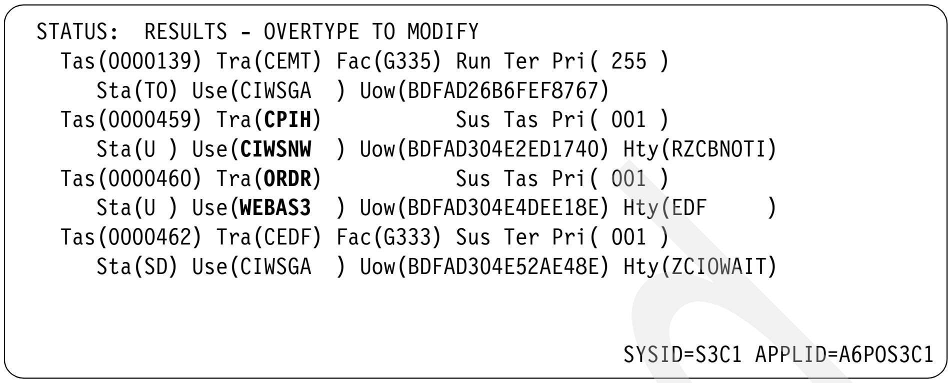
Note: In the same way that we are using the header processing program for validating security credentials, it is quite feasible to use such a program to provide some form of audit trail.

We completed the CICS setup for this scenario by permitting access to the ORDR transaction to the user ID WEBAS3:

```
PERMIT CIWS3D.ORDR CLASS(GCICSTRN) ID(WEBAS3) ACCESS(READ)
```

9.4.3 Testing SOAP message security

Now when we submit an order request on the Web browser, the credentials are added to the SOAP Security header and are extracted by the header processing program CIWSSECH. Figure 9-12 shows the ORDR transaction running with the user ID WEBAS3, while the CPIH transaction continues to execute with the CIWSNW user ID.



```
STATUS:  RESULTS - OVERTYPE TO MODIFY
  Tas(0000139) Tra(CEMT) Fac(G335) Run Ter Pri( 255 )
     Sta(TO) Use(CIWSGA  ) Uow(BDFAD26B6FEF8767)
  Tas(0000459) Tra(CPIH)           Sus Tas Pri( 001 )
     Sta(U ) Use(CIWSNW  ) Uow(BDFAD304E2ED1740) Hty(RZCBNOTI)
  Tas(0000460) Tra(ORDR)           Sus Tas Pri( 001 )
     Sta(U ) Use(WEBAS3  ) Uow(BDFAD304E4DEE18E) Hty(EDF     )
  Tas(0000462) Tra(CEDF) Fac(G333) Sus Ter Pri( 001 )
     Sta(SD) Use(CIWSGA  ) Uow(BDFAD304E52AE48E) Hty(ZCIOWAIT)



                                                    SYSID=S3C1 APPLID=A6POS3C1
```

Figure 9-12 ORDR executing with SOAP security header user ID

9.4.4 SOAP fault messages

The `<Fault>` element in a SOAP message is used for reporting errors. CICS provides an application programming interface for creating, adding, or deleting SOAP fault messages. See the *CICS Transaction Server for z/OS CICS Application Programming Reference,* SC34-6434, and *CICS Transaction Server for z/OS CICS Web Services Guide,* SC34-6458, for further details on these commands.

If the credentials extracted do not validate, then CIWSSECH creates a SOAP fault message (Example 9-14) which is returned to the service requester.

Example 9-14 Generating a SOAP fault

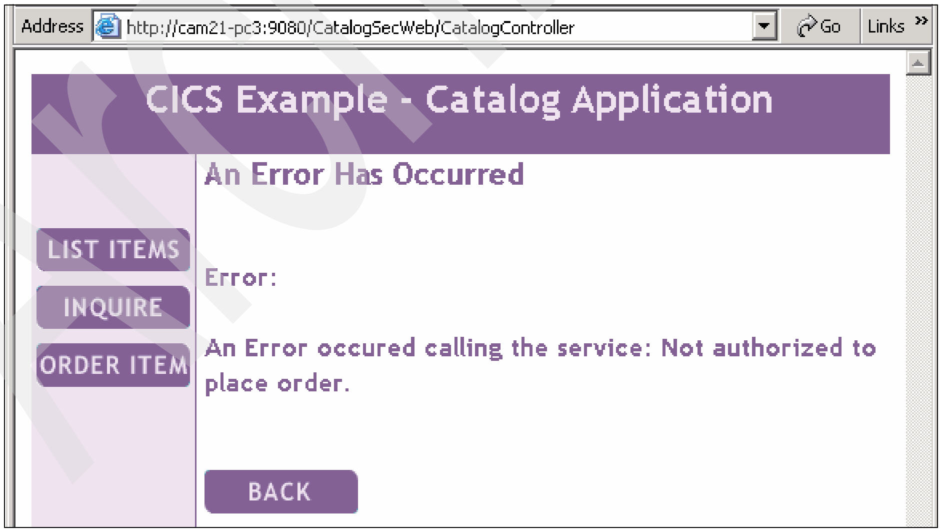
```
                EXEC CICS VERIFY
                   PASSWORD(CA-PASSWORD)
                   USERID(CA-USER-ID)
                   RESP(WS-RESP)
                END-EXEC
       *        Successful ?
                IF WS-RESP = DFHRESP(NORMAL)
                   PERFORM SET-USER-ID
                ELSE
                   MOVE WS-NOT-AUTH TO WS-FAULT-STRING
                   PERFORM FAULT-MESSAGE
                END-IF
              .
              .
              .
```

```
 FAULT-MESSAGE SECTION.
*----------------------------------------------------------------*
* Generate a SOAP Fault
*----------------------------------------------------------------*
     EXEC CICS
         GET CONTAINER('DFHWS-SOAPLEVEL')
         INTO(WS-SOAP-LEVEL)
         FLENGTH(WS-HEAD-LEN)
         RESP(WS-RESP)
     END-EXEC.

* MOVE CORRECT VERSION OF FAULTCODE
     IF WS-SOAP-11 MOVE DFHVALUE(CLIENT) to WS-FAULT-CODE
     ELSE
        MOVE DFHVALUE(SENDER) to WS-FAULT-CODE
     END-IF

     EXEC CICS SOAPFAULT CREATE
               FAULTSTRING(WS-FAULT-STRING)
               FAULTSTRLEN(length of WS-FAULT-STRING)
               FAULTCODE(WS-FAULT-CODE)
     END-EXEC.
 FAULT-MESSAGE-END. EXIT.
```

Figure 9-13 shows the fault string `Not authorized to place order`, which occurs when the user's credentials are invalid.


Address
http://cam21-pc3:9080/CatalogSecWeb/CatalogController
Go
Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM
BACK

An Error Has Occurred

Error:

An Error occured calling the service: Not authorized to place order.

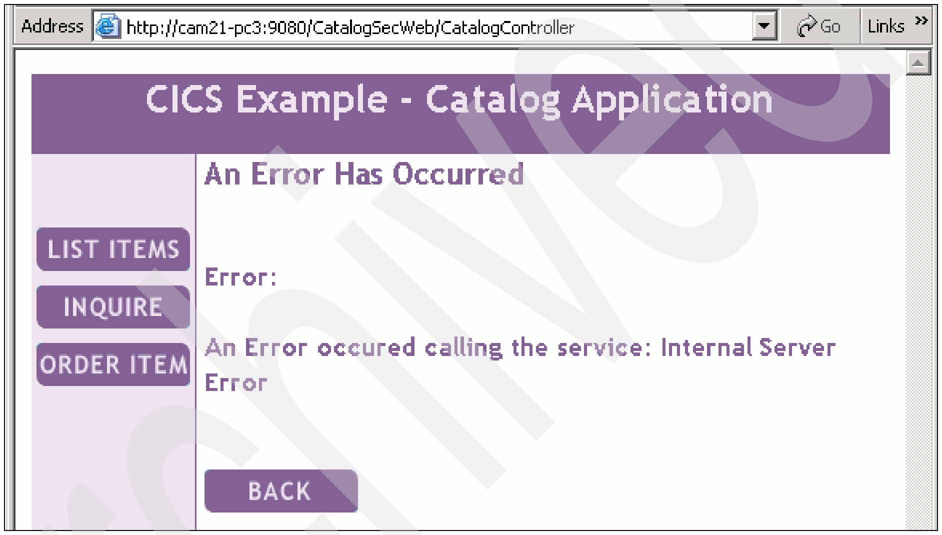
Figure 9-13 Browser - SOAP fault message for 'not authorized' error

If the user ID and password supplied in the SOAP Security header validate correctly, but the user ID has no access to the ORDR transaction, then a RACF error message is issued, as seen in Example 9-15.

Example 9-15 RACF messages for insufficient access

```
ICH408I USER(CIWSPC  ) GROUP(SYS1    ) NAME(PAOLO )
   CIWS3D.ORDR CL(TCICSTRN)
   INSUFFICIENT ACCESS AUTHORITY
   ACCESS INTENT(READ   )  ACCESS ALLOWED(NONE   )
```

Figure 9-14 shows the fault string that is generated by CICS for this error.


Address
http://cam21-pc3:9080/CatalogSecWeb/CatalogController
Go
Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM
BACK

An Error Has Occurred

Error:

An Error occured calling the service: Internal Server Error

Figure 9-14 Browser - Internal server error

9.5 Enabling SSL/TLS

By default, SOAP messages are flowed in clear text. This is likely to be unacceptable in many cases. In order to address this, some form of encryption is required. SSL/TLS is a well understood and popular way of encrypting message flows. When the client connects with SSL/TLS, privacy of the data is obtained by encrypting the data.

In this section we show how we configured both the CICS and WebSphere environments to use an SSL/TLS connection (Figure 9-15).

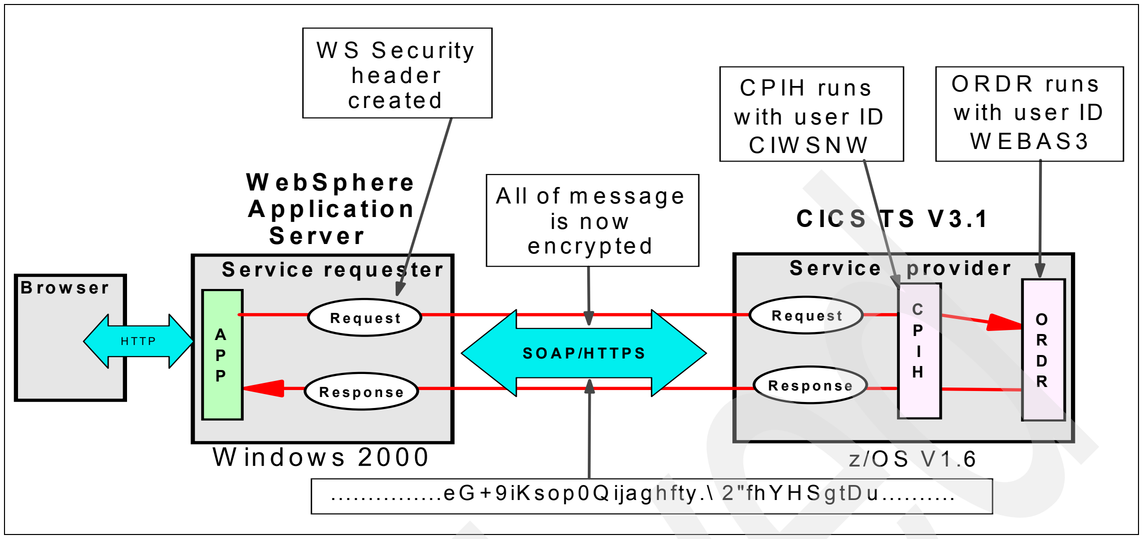

WS Security header created
CPIH runs with user ID CIWSNW
ORDR runs with user ID WEBAS3
WebSphere Application Server
All of message is now encrypted
CICS TS V3.1
Browser
Service requester
Service provider
HTTP
APP
Request
Response
SOAP/HTTPS
CPIH
ORDR
Windows 2000
z/OS V1.6
..............eG+9iKsop0Qijaghfty.\ 2"fhYHSgtDu..........

Figure 9-15 Enabling SSL/TLS scenario

We document the following steps that we followed to enable SSL/TLS:

- Creating a key ring and certificates on z/OS for CICS
- Enabling an SSL/TLS connection from WebSphere
- Configuring CICS support for SSL/TLS

9.5.1 Creating a key ring and certificates on z/OS for CICS

In order to create the certificates required for using SSL/TLS, we first needed to prepare RACF. We issued the commands shown in Example 9-16. For more information, see the chapter titled “Configuring CICS to use SSL” in *CICS Transaction Server for z/OS RACF Security Guide, SC34-6454.*

Example 9-16 RACF commands to prepare for running the DFH$RING exec

```
RDEFINE FACILITY (IRR.DIGTCERT.ADD)
RDEFINE FACILITY (IRR.DIGTCERT.CONNECT)
RDEFINE FACILITY (IRR.DIGTCERT.GENCERT)

SETROPTS RACLIST(FACILITY) REFRESH
SETROPTS GENERIC(FACILITY) REFRESH

PERMIT IRR.DIGTCERT.*       CLASS(FACILITY) ID(CIWS3D) ACCESS(READ)
PERMIT IRR.DIGTCERT.CONNECT CLASS(FACILITY) ID(CIWS3D) ACCESS(CONTROL)
PERMIT IRR.DIGTCERT.GENCERT CLASS(FACILITY) ID(CIWS3D) ACCESS(CONTROL)
```

The sample clist DFH$RING is provided by CICS TS to help you build a RACF KEYRING for CICS SSL use. We copied the DFH$RING clist from hlq.SDFHSAMP into our own private library and modified the clist using the customization values shown in Table 9-4.

Table 9-4 Customizing the DFH$RING clist - Site settings

Name	Value
organization	ITSO
department	PSSC
certifier	CICS-Sample-Certification
city	Montpellier
state	Herault
country	FR

These values represent the entity for which we are creating the certificates and key ring. As can be seen, these were chosen to indicate the IBM site where the Redbook residency was held.

Further parameters were supplied as input to the clist for the specific key ring to be created (Table 9-5).

Table 9-5 Customizing the DFH$RING clist - Key ring settings

Name	Value
firstname	Ciws
lastname	Ciwss3c1
ipname	mvsg3.mop.ibm.com
userid	CIWS3D

The name of the key ring to be created is `Ciws.Ciwss3c1`. The userid parameter was supplied as `CIWS3D` (the CICS region user ID). The ipname is the address of the z/OS on which our CICS region CIWSS3C1 runs.

Further modifications were made to the clist because we chose not to generate certificates for a Web server or for an EJB container. Instead we generated certificates for a Web service Client, Web service Server, and a default certificate. These changes are shown in Example 9-17.

Example 9-17 Customizing the DFH$RING EXEC - certificate labels

```
"RACDCERT ID("foruser") DELETE(LABEL('"lastname"-Web-service-Client'))"
"RACDCERT ID("foruser") DELETE(LABEL('"lastname"-Web-service-Server'))"
"RACDCERT ID("foruser") DELETE(LABEL('"lastname"-Default-Certificate'))"
```

We used the batch job shown in Example 9-18 to execute the DFH$RING clist in order to create the required RACF profiles.

Example 9-18 Batch job to invoke the DFH$RING clist

```
//REXXJOB  JOB 1,CIWS,TIME=1440,NOTIFY=&SYSUID,REGION=4M,
//    CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//*
//REXX EXEC PGM=IKJEFT01
//SYSPROC DD DISP=SHR,DSN=CIWS.CICS.JCL
//SYSTSPRT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSTSIN DD *
 %DFH$RING CIWS CIWSS3C1 mvsg3.mop.ibm.com FORUSER(CIWS3D)
/*
```

To complete this part of the configuration we exported the Certificate Authority (CA) certificate in Base64 X.509 format using the RACF RACDCERT command:

```
RACDCERT CERTAUTH EXPORT(LABEL(‘CICS-Sample-Certification’)) +
DSN(CIWSGA.CERTAUTH.X509) FORMAT(CERTB64)
```

9.5.2 Enabling an SSL/TLS connection from WebSphere

To enable an SSL/TLS connection from WebSphere we did the following:

- FTPed the certificate from z/OS to our client machine
- Imported the certificate into the WebSphere Application Server keystore
- Enabled the service requester to use HTTPS

FTP the certificate from z/OS

We used FTP as shown in Example 9-19 to send the exported certificate in ASCII format to our client machine.

Example 9-19 Using FTP to send an exported CA certificate to a client workstation

```
C:\>ftp mvsg3.mop.ibm.com
Connected to mvsg3.mop.ibm.com.
220-FTPD1 IBM FTP CS V1R6 at MVSG3.pssc.mop.ibm.com, 13:23:10 on 2005-11-30.
220 Connection will close if idle for more than 5 minutes.
User (mvsg3.mop.ibm.com:(none)): ciwsga
```

```
331 Send password please.
Password:
230 CIWSGA is logged on.  Working directory is "CIWSGA.".
ftp> ascii
200 Representation type is Ascii NonPrint
ftp> get CERTAUTH.X509
200 Port request OK.
125 Sending data set CIWSGA.CERTAUTH.X509
250 Transfer completed successfully.
ftp: 1022 bytes received in 0.00Seconds 1022000.00Kbytes/sec.
ftp> quit
221 Quit command received. Goodbye.
```

Importing the certificate

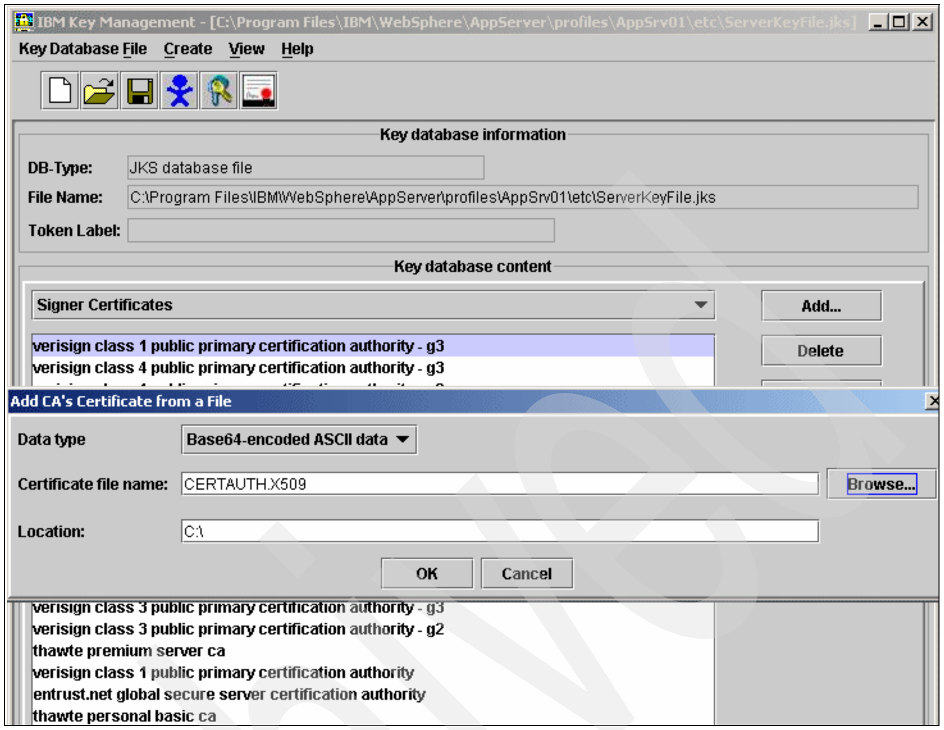
Next, we needed to make the CERTAUTH.X509 certificate available for use by WebSphere Application Server. To do this, we imported the CERTAUTH.X509 certificate to the WebSphere Application Server keystore (DummyServerKeyFile.jks) using iKeyMan.

Note: The iKeyMan tool is provided by the IBM Java SDK. On our workstation the tool was located in directory C:\Program Files\IBM\Java142\jre\bin.

After opening a command prompt window we set the PATH variable and typed `ikeyman` to invoke the iKeyMan GUI.

```
set PATH=C:\Program Files\IBM\Java142\jre\bin\;%PATH%
ikeyman
```

We used iKeyMan to add the CERTAUTH.X509 file to the WebSphere Application Server keystore (ServerKeyFile.jks) as shown in Figure 9-16.


IBM Key Management - [C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\etc\ServerKeyFile.jks]

Key database information

DB-Type: JKS database file

File Name: C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\etc\ServerKeyFile.jks

Token Label:

Key database content

Signer Certificates

verisign class 1 public primary certification authority - g3
verisign class 4 public primary certification authority - g3

Add CA's Certificate from a File

Data type: Base64-encoded ASCII data

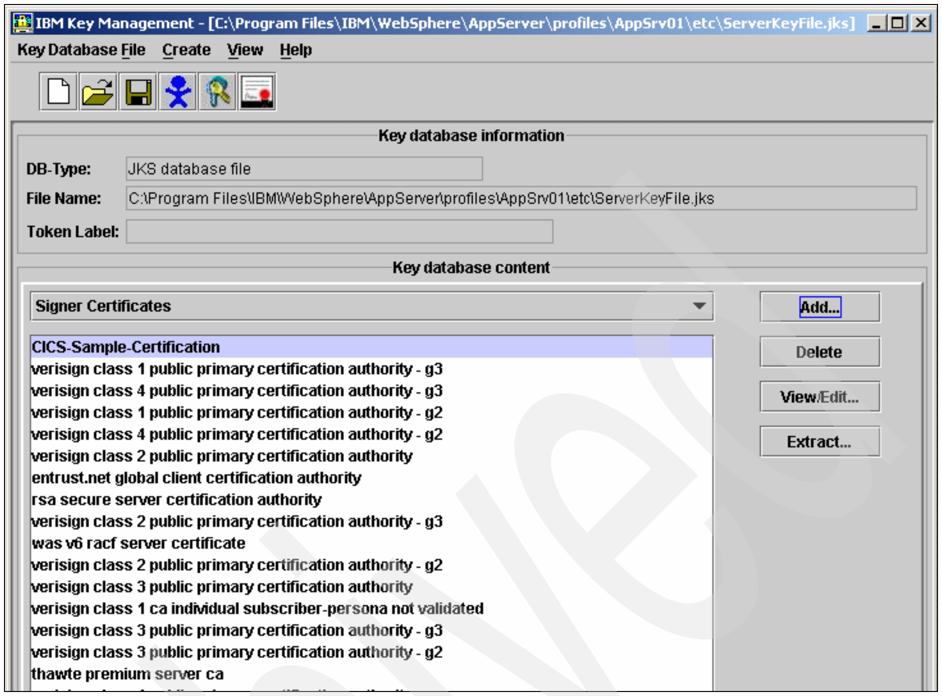
Certificate file name: CERTAUTH.X509

Location: C:\

verisign class 3 public primary certification authority - g3
verisign class 3 public primary certification authority - g2
thawte premium server ca
verisign class 1 public primary certification authority
entrust.net global secure server certification authority
thawte personal basic ca

Figure 9-16 IKeyman - Add CICS CA certificate from an X509 file

After we clicked **OK** to proceed with this action, we were prompted to enter a label for this certificate. Figure 9-17 shows the CICS CA certificate added to the WebSphere keystore.


IBM Key Management - [C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\etc\ServerKeyFile.jks]

Key database information

DB-Type: JKS database file

File Name: C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\etc\ServerKeyFile.jks

Token Label:

Key database content

Signer Certificates

CICS-Sample-Certification
verisign class 1 public primary certification authority - g3
verisign class 4 public primary certification authority - g3
verisign class 1 public primary certification authority - g2
verisign class 4 public primary certification authority - g2
verisign class 2 public primary certification authority
entrust.net global client certification authority
rsa secure server certification authority
verisign class 2 public primary certification authority - g3
was v6 racf server certificate
verisign class 2 public primary certification authority - g2
verisign class 3 public primary certification authority
verisign class 1 ca individual subscriber-persona not validated
verisign class 3 public primary certification authority - g3
verisign class 3 public primary certification authority - g2
thawte premium server ca

Figure 9-17 IKeyMan - After adding CICS CA certificate to keystore

Next we created a new JSSE repertoire by logging onto the admin console and clicking **Security** → **SSL** → **New JSSE repertoire**. We entered the Alias name as `CICS-SSLSettings` and defaulted the Protocol to SSL_TLS. The Key file and Trust file names were both specified as `c:\Program Files\IBM\WebSphere \Appserver\profiles\AppSrv01\bin\ServerKeyFile.jks` and the password specified was `WebAS`. Changed fields are shown in Table 9-6, all other values were allowed to default.

Table 9-6 Altered values for new JSSE repertoire

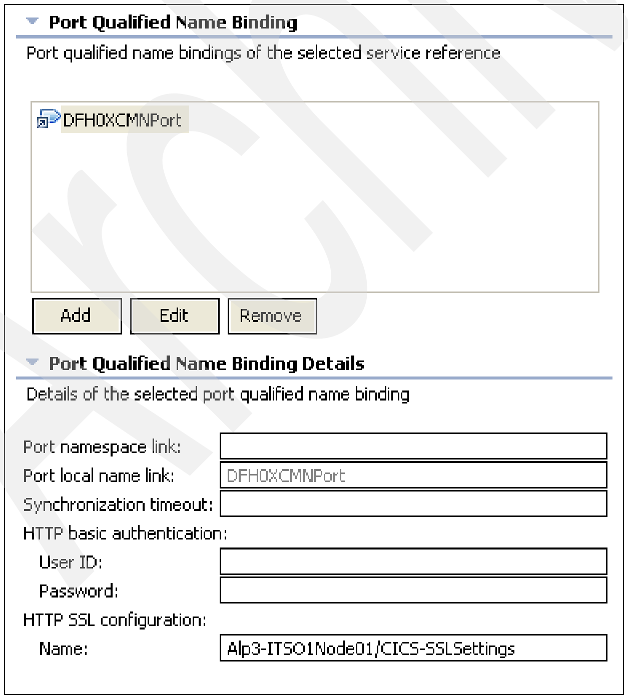
Field name	Value
Alias	CICS-SSLSettings
Key file name	C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\bin\ServerKeyFile.jks
Key file password	WebAS
Trust file name	C:\Program Files\IBM\WebSphere\AppServer\profiles\AppSrv01\bin\ServerKeyFile.jks
Trust file password	WebAS

We clicked **OK** → **Save** → **Save** to save the configuration. Next we stopped and restarted the application server, in order to have the new repertoire recognized.

Enabling the service requester to use HTTPS

In Section 9.4.1, "Configuring the service requester" on page 286 we showed that to configure our Web service client to use WS-Security, we used Rational Application Developer V6.0. We next used a similar process to enable the service requester to use HTTPS:

1. We imported the client application archive CatalogSec_WS-Security.ear into RAD. We then expanded the Dynamic Web Project (`CatalogSecWeb`) in the Project Explorer and opened (double-clicked) the deployment descriptor.
2. After the Deployment Descriptor Editor opened, we selected the **WS Binding** page. The WS Binding page is for editing the client's binding file, so you can specify how to apply the required security.
3. We clicked the service reference **service/DFH0XCMNService** and then the Port Qualified Name Binding **DFH0XCMNPort.**
4. In the Port Qualified Name Binding Details section we specified our SSL repertoire name `Alp3-ITSO1Node01/CICS-SSLSettings` for the name of the HTTP SSL Configuration (Figure 9-18).


Port Qualified Name Binding
Port qualified name bindings of the selected service reference
DFH0XCMNPort
Add
Edit
Remove
Port Qualified Name Binding Details
Details of the selected port qualified name binding

Port namespace link:
Port local name link: DFH0XCMNPort
Synchronization timeout:
HTTP basic authentication:
User ID:
Password:
HTTP SSL configuration:
Name: Alp3-ITSO1Node01/CICS-SSLSettings

Figure 9-18 RAD - Enabling HTTPS for service requester

5. We repeated steps 3 and 4 for our other CICS Web services `service/DFHOXCMNService2` and **`service/DFHOXCMNService3`**.
6. We then saved the configuration and exported a new EAR file called CatalogSec_WS-Security_HTTPS.ear.

Note: The fully qualified name, including the node name, must be specified for HTTP SSL configuration: Name in order for this to be recognized by WebSphere Application Server when the application is deployed.

After enabling the service requester to use HTTPS, we needed to re-deploy the application using the WebSphere admin console. We followed the same process that is described in "Installing the service requester" on page 87 to deploy the CatalogSec_WS-Security_HTTPS.ear.

See "Testing SSL/TLS" on page 310 for information about how we tested SSL/TLS connectivity from WebSphere Application Server to CICS.

9.5.3 Configuring CICS support for SSL/TLS

To activate SSL/TLS support in our CICS TS V3.1 region, we specified the following system initialization parameter:

- `KEYRING=Ciws.Ciwss3c1`

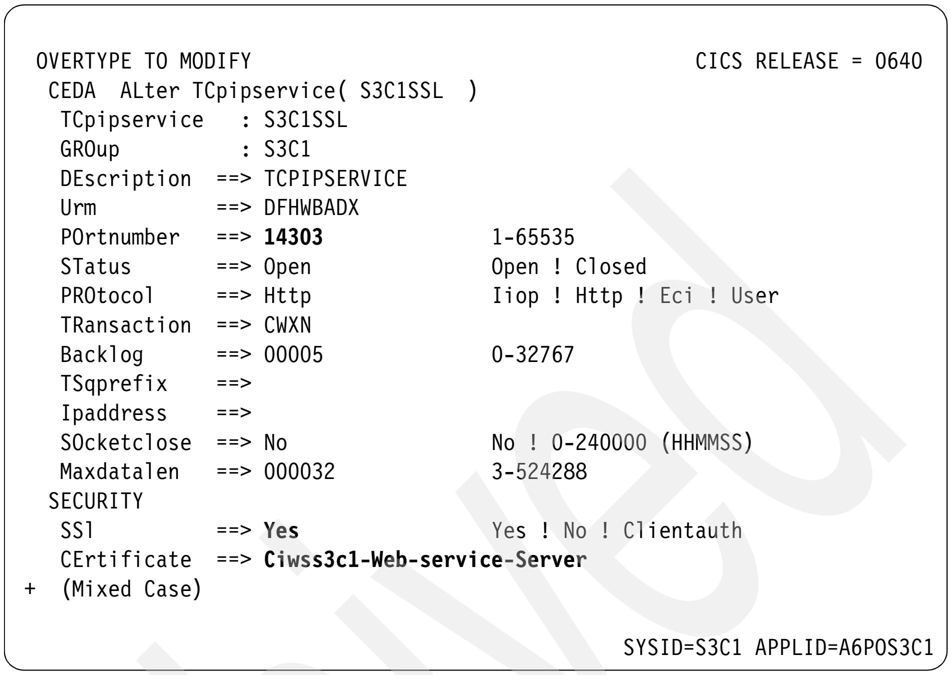
  This specifies the name of a key ring we created in the RACF database that contains keys and certificates used by this CICS region.

We allowed the ENCRYPTION, MAXSSLTCBS, SSLCACHE and SSLDELAY system initialization parameters to default. We then restarted the CICS region.

Tip: You should restart your CICS region after updating the system initialization parameter and before creating the TCPIPSERVICE for SSL. The CIPHERS attribute of the TCPIPSERVICE resource is only automatically updated by CICS if CICS has been started with SSL support active.

Changing the TCPIPSERVICE

To enable SSL connections in CICS we created a new TCPIPSERVICE with the PORTNUMBER, CERTIFICATE and SSL attributes set as shown in Figure 9-19.



```
 OVERTYPE TO MODIFY                                     CICS RELEASE = 0640
  CEDA  ALter TCpipservice( S3C1SSL  )
   TCpipservice   : S3C1SSL
   GROup          : S3C1
   DEscription  ==> TCPIPSERVICE
   Urm          ==> DFHWBADX
   POrtnumber   ==> 14303             1-65535
   STatus       ==> Open              Open ! Closed
   PROtocol     ==> Http              Iiop ! Http ! Eci ! User
   TRansaction  ==> CWXN
   Backlog      ==> 00005             0-32767
   TSqprefix    ==>
   Ipaddress    ==>
   SOcketclose  ==> No                No ! 0-240000 (HHMMSS)
   Maxdatalen   ==> 000032            3-524288
  SECURITY
   SSl          ==> Yes               Yes ! No ! Clientauth
   CErtificate  ==> Ciwss3c1-Web-service-Server
+  (Mixed Case)

                                                      SYSID=S3C1 APPLID=A6POS3C1
```

Figure 9-19 TCPIPSERVICE - Updated for SSL support

We noticed that CICS automatically initialized the CIPHERS attribute with a default list of acceptable codes (Example 9-20).

Example 9-20 TCPIPSERVICE CIPHERS attribute

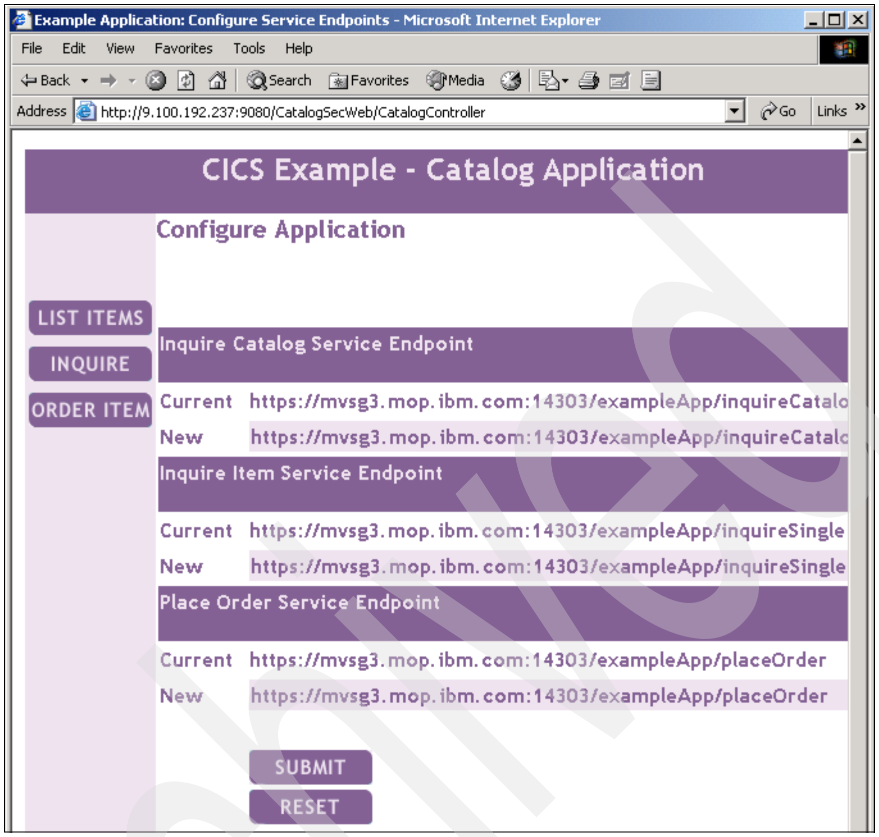
```
CIphers      ==> 050435363738392F303132330A1613100D0915120F0C03060201
```

9.5.4 Testing SSL/TLS

To test the SSL/TLS scenario, we needed to configure the catalog manager J2EE application so that it accesses the catalog services using HTTPS. We started a Web browser session and entered the URL:

```
http://cam21-pc3:9080/CatalogWeb/Welcome.jsp
```

We clicked **CONFIGURE** and specified the endpoint addresses of the Catalog services (Figure 9-20). We clicked **SUBMIT** to complete the update.


Example Application: Configure Service Endpoints - Microsoft Internet Explorer

Address http://9.100.192.237:9080/CatalogSecWeb/CatalogController

CICS Example - Catalog Application

Configure Application

Inquire Catalog Service Endpoint

Current https://mvsg3.mop.ibm.com:14303/exampleApp/inquireCatalo

New https://mvsg3.mop.ibm.com:14303/exampleApp/inquireCatalo

Inquire Item Service Endpoint

Current https://mvsg3.mop.ibm.com:14303/exampleApp/inquireSingle

New https://mvsg3.mop.ibm.com:14303/exampleApp/inquireSingle

Place Order Service Endpoint

Current https://mvsg3.mop.ibm.com:14303/exampleApp/placeOrder

New https://mvsg3.mop.ibm.com:14303/exampleApp/placeOrder

File Edit View Favorites Tools Help
Back Search Favorites Media
Go Links
LIST ITEMS
INQUIRE
ORDER ITEM
SUBMIT
RESET

Figure 9-20 Web browser - Updating the Catalog application to use https

We then tested each of the three services (inquireCatalog, inquireSingle, and placeOrder) to ensure that all worked correctly. A display of TCP/IP connections on z/OS showed a connection to CICS region CIWSS3C1 on port 14303 (Example 9-21).

Example 9-21 Display of IP connections

```
Display  Filter  View  Print  Options  Help
-------------------------------------------------------------------------------
SDSF ULOG  CONSOLE CIWSGA                        LINE 42      COLUMNS 42- 121
COMMAND INPUT ===>                                           SCROLL ===> CSR
 -D TCPIP,,NETSTAT,CONN
  EZZ2500I NETSTAT CS V1R6 TCPIP 328
  USER ID  CONN     LOCAL SOCKET           FOREIGN SOCKET         STATE
  BPXOINIT 00000011 0.0.0.0..10007         0.0.0.0..0             LISTEN
  CIWSR3C1 00008C23 0.0.0.0..13301         0.0.0.0..0             LISTEN
  CIWSR3C1 00008C87 9.100.193.167..13301   9.100.199.156..4338    ESTBLSH
```

```
CIWSR3C2 00008C25 0.0.0.0..13302        0.0.0.0..0            LISTEN
CIWSS3C1 00008B96 9.100.193.167..14303  9.100.192.237..1086   ESTBLSH
CIWSS3C1 00008B71 0.0.0.0..14303        0.0.0.0..0            LISTEN
```

A `CEMT INQ TCPIPSERVICE` command shows that this port has the SSL attribute and that one connection has been established (Figure 9-21).

```
 INQUIRE TCPIPSERVICE
  STATUS:  RESULTS - OVERTYPE TO MODIFY
   Tcpips(S3C1SSL ) Ope Por(14303) Http Ssl Tra(CWXN)
      Con(00001) Bac( 00005 ) Max( 000032 ) Urm( DFHWBADX ) Sup






                                                   SYSID=S3C1 APPLID=A6POS3C1
```

Figure 9-21 TCPIPSERVICE - SSL support installed and active

9.6 Enabling SOAP message security with WMQ

In this scenario, we show how SOAP Security headers can be used with WMQ. We customize the catalog application so that when the ORDR transaction executes, it calls a Web service to dispatch the order. This Web service is located in a separate CICS region and the transport mechanism used is WMQ. To make the dispatchOrder service (CICS transaction DISP) run with the same user ID as the calling transaction (ORDR), we transfer the caller's identity in a SOAP Security header.

This scenario was chosen for two reasons:

- To demonstrate the use of a SOAP Security header in a CICS service request
- To demonstrate the use of SOAP Security headers with WMQ

Figure 9-22 shows how we configured the environment.

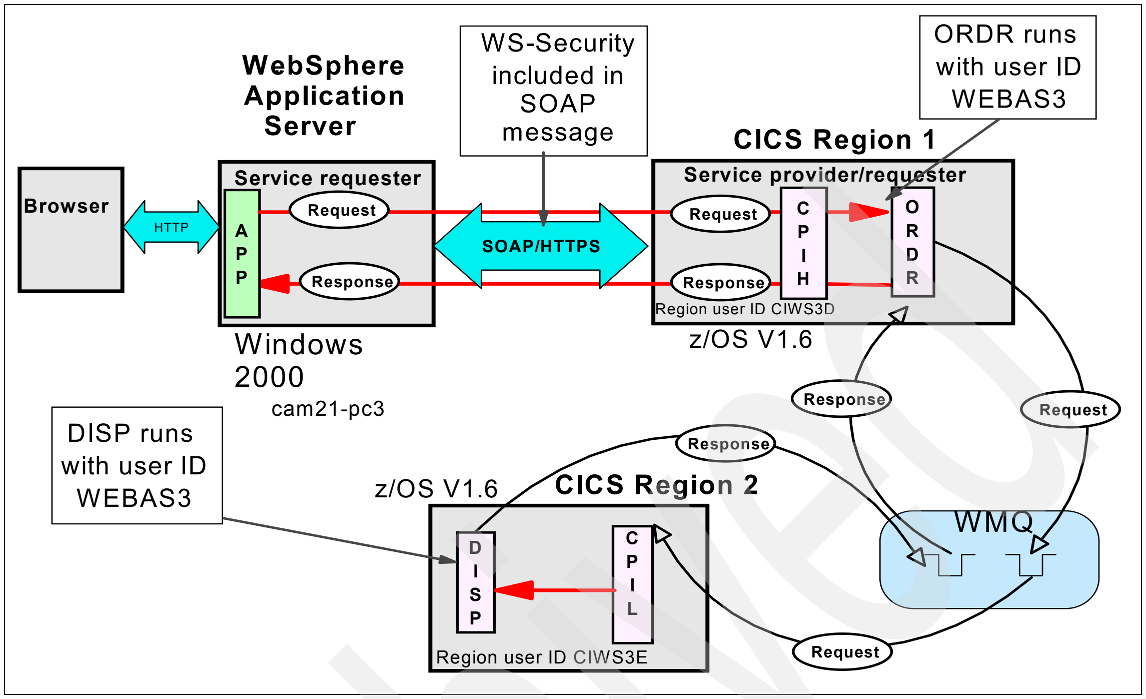


Figure 9-22 SOAP message security with WebSphere MQ scenario

Note: This scenario is an example of *identity assertion*, in which an intermediary server (CICS region 1) authenticates the client and transfers the request message and identity to the target server (CICS region 2). See "Identity assertion" on page 273 for more information about the concept of identity assertion.

We document the following steps necessary to set up this security scenario:

- Configuring CICS to use WMQ
- Configuring the service requester
- Configuring the service provider
- Configuring security for WMQ

9.6.1 Configuring CICS to use WMQ

For this, we rely heavily upon the work already done and described in Chapter 4, "Web services using WebSphere MQ" on page 111. We note only where differences occur, otherwise we used the same values.

We added WebSphere MQ support to our two CICS regions, CIWSS3C1 and CIWSS3C2. For the dispatchOrder service, the service provider is CIWSS3C2 and the service requester is CIWSS3C1. Table 9-7 shows the different settings, parameters, and values used for these two CICS regions.

Table 9-7 Different settings for the two CICS regions

	CICS Region 1	**CICS Region 2**
WMQ queue manager	MQS3	MQS3
INITQ	VSG3.S3C1.INITQ	VSG3.S3C2.INITQ
Pipeline	PIPE2	PIPE3
Configuration file	/CIWS/S3C1/config/ITSO_7206_secrequester.xml	/CIWS/S3C2/config/ITSO_7206_secprovider_dispatch.xml
HFS structure	/CIWS/S3C1/....	/CIWS/S3C2/....
Header processing program	CIWSSECR	CIWSSECS
Region user ID	CIWS3D	CIWS3E
Jobname	CIWSS3C1	CIWSS3C2
Applid	A6POS3C1	A6POS3C2

The WebSphere MQ queuenames used were `VSG3.S3C2.PIPE3.REQUEST` and `VSG3.S3C2.PIPE3.RESPONSE` while the process name was `VSG3.S3C2.PROCESS`.

The security of the CICS transactions supplied by WebSphere MQ resources was protected by issuing the RACF commands shown in Example 9-22.

Example 9-22 RACF commands to protect WebSphere MQ CICS transactions

```
RDEFINE GCICSTRN WS3MQUSR UACC(NONE) +
        ADDMEM(CIWS3E.CKBM, CIWS3E.CKCN, CIWS3E.CKDL, +
               CIWS3E.CKDP, CIWS3E.CKQC, CIWS3E.CKRS, +
               CIWS3E.CKRT, CIWS3E.CKSD, CIWS3E.CKSQ, +
               CIWS3E.CKMC, CIWS3E.CKMH, CIWS3E.CKRC, +
               CIWS3E.CKRQ, CIWS3E.CKSG, CIWS3E.CKSV) +
        OWNER(IBMUSER)

PERMIT WS3MQUSR CLASS(GCICSTRN) ID(SYS1)
RALTER GCICSTRN CIWS3E ADDMEM(CIWS3E.CKTI, CIWS3E.CKAM) OWNER(IBMUSER)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

In Example 9-22 we:

- Define the transaction group WS3MQUSR and add transactions CKBM, CKCN, CKDL, CKDP, CKQC, CKRS, CKRT, CKSD, CKSQ, CKMC, CKMH, CKRC, CKRQ, CKSG and CKSV to the group.
- Permit access to the transaction group WS3MQUSR to the RACF user group SYS1 (SYS1 is the group of administration user IDs on our system).
- Alter the transaction group CIWS3E (used for CICS region CIWSS3C2) by adding transactions CKTI and CKAM to the group.
- Refresh the RACF CICS transaction class TCICSTRN.

These commands were repeated with the prefix of CIWS3D for the CIWSS3C1 CICS region.

9.6.2 Configuring the service requester

In this scenario the service requester is the ORDR transaction itself. This demonstrates that a transaction can act as both a service provider and a service requester.

Configuring the sample application

We configured the catalog application in the same way as in "Configuring the Catalog application" on page 122, except that we specified the Outbound WebService URI as:

```
jms:/queue?destination=VSG3.S3C2.PIPE3.REQUEST@MQS3&targetService=/e
xampleApp/dispatchOrder&replyDestination=VSG3.S3C2.PIPE3.RESPONSE
```

This means that the SOAP request for the dispathOrder service is sent using WMQ, via the WMQ queue VSG3.S3C2.PIPE3.REQUEST, to an application servicing that queue. The response is returned to the ORDR transaction via the queue VSG3.S3C2.PIPE3.RESPONSE.

9.6.3 Header processing program

In order to add a Security header to the orderDispatch SOAP message, we wrote a new header processing program CIWSSECR that inserts a `<wsse:Security>` header into the message. A synopsis of this program is shown in Example 9-23.

Example 9-23 Header processing program to insert the SOAP security header

```
Check if invoked for SEND-REQUEST, else exit
Check for correct URI, else exit
Obtain user ID from DFHWS-USERID container and insert into Security header
Put Security header to DFHHEADER container
```

The full CIWSSECR program is shown in A.5, “Sample header processing program - CIWSSECR” on page 555.

Pipeline configuration file

A new requester pipeline configuration file was created by copying the CICS-supplied sample from basicsoap11requester.xml to /CIWS/S3C2/config/ITSO_7206_secrequester.xml. The header processing program was then added to the configuration file as shown in Example 9-24. The `<program name>` element contains the name of the header processing program CIWSSECR, which is to be invoked to create the security token.

Example 9-24 Configuration file for pipeline PIPE2

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<requester_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline”
                      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance”
                      xsi:schemaLocation="http://www.ibm.com/software/htp/cics/
                      pipeline requester.xsd ">
 <service>
  <service_handler_list>
      <cics_soap_1.1_handler>
        <headerprogram>
          <program_name>CIWSSECR</program_name>
          <namespace>http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
           security-secext-1.0.xsd</namespace>
          <localname>Security</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.1_handler>
  </service_handler_list>
 </service>
</requester_pipeline>
```

In CICS, the new PIPELINE resource definition was created by copying PIPE1 as PIPE2 and modifying the CONFIGFILE attribute to the name of the new pipeline configuration file.

9.6.4 Configuring the service provider

A new header processing program CIWSSECS and pipeline configuration file were required.

Header processing program

Program CIWSSECS is similar to CIWSSECH except that no password verification is done because no password has been sent. A short description of this program is shown in Example 9-25.

Example 9-25 Header processing program logic to change user ID

```
Check if invoked for RECEIVE-REQUEST, else exit
Check for correct URI, else exit
Obtain Security header from DFHHEADER container
Parse the header and obtain user ID
Put user ID into DFHWS-USERID container
```

Pipeline configuration file

The new configuration file /CIWS/S3C2/config/ITSO_7206_secprovider_dispatch.xml, was created by copying /CIWS/S3C1/config/ITSO_7206_secprovider.xml (see Example 9-6 on page 293). The CIWSSECS program was specified in place of CIWSSECH.

Example 9-26 Configuration file for pipeline PIPE3

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
  <transport>
      <default_transport_handler_list>
          <handler>
             <program>CIWSMSGH</program>
             <handler_parameter_list/>
          </handler>
      </default_transport_handler_list>
  </transport>
  <service>
    <terminal_handler>
      <cics_soap_1.1_handler>
        <headerprogram>
          <program_name>CIWSSECS</program_name>
          <namespace>http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-ws
          security-secext-1.0.xsd</namespace>
          <localname>Security</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.1_handler>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

9.6.5 Configuring WebSphere MQ for security

Since in this scenario we are passing a user ID from CICS region 1 to CICS region 2 without re-authenticating, it is necessary to establish a trust relationship between the CICS regions. We can do this by restricting access to the specific WMQ queues used to transport the dispatchOrder request.

We secured the WMQ connection between CICS region 1 and CICS region 2 as follows:

- Configure general security for WMQ.

  The RACF commands that we issued in order to configure general security for WMQ are listed in Example 9-27.

  Example 9-27 RACF commands to configure WebSphere MQ security

  ```
  SETROPTS CLASSACT(MQADMIN,MQQUEUE,MQPROC,MQNLIST,MQCONN,MQCMDS)
  SETROPTS  GENERIC(MQADMIN,MQQUEUE,MQPROC,MQNLIST,MQCONN,MQCMDS)
  RDEFINE MQADMIN MQS3.NO.CMD.CHECKS
  RDEFINE MQADMIN MQS3.NO.CMD.RESC.CHECKS
  RDEFINE MQADMIN MQS3.NO.PROCESS.CHECKS
  RDEFINE MQADMIN MQS3.NO.NLIST.CHECKS
  RDEFINE MQADMIN MQS3.NO.CONTEXT.CHECKS
  RDEFINE MQADMIN MQS3.NO.ALTERNATE.USER.CHECKS
  RDEFINE MQADMIN MQS3.YES.SUBSYS.SECURITY
  RDEFINE MQCONN  MQS3.CICS UACC(NONE)
  PERMIT MQS3.CICS CLASS(MQCONN) ID(CIWS3D,CIWS3E) ACCESS(READ)
  SETROPTS RACLIST(MQQUEUE) REFRESH
  SETROPTS GENERIC(MQQUEUE) REFRESH
  SETROPTS RACLIST(MQCONN) REFRESH
  SETROPTS GENERIC(MQCONN) REFRESH
  SETROPTS RACLIST(MQADMIN) REFRESH
  SETROPTS GENERIC(MQADMIN) REFRESH
  ```

  The SETROPTS CLASSACT and SETROPTS GENERIC commands activate the specific and generic WMQ security classes. All classes were activated.

- Turn security on for the queue manager.

  We set the switch qmgr-name.YES.SUBSYS.SECURITY to `ON` for our queue manager MQS3 (see "WebSphere MQ transport" on page 251).

- Grant access to the WMQ queues.

  We granted both CICS region user IDs `connect` access to the queue manager and then refreshed the RACF profiles. Once these definitions were active, we then secured access to the queues, using the commands shown in Example 9-28.

Example 9-28 RACF commands to secure WebSphere MQ queues

```
RDEFINE GMQQUEUE ORDR.DISPATCH UACC(NONE) +
       ADDMEM(MQS3.VSG3.S3C2.PIPE3.REQUEST, +
              MQS3.VSG3.S3C2.PIPE3.RESPONSE)
PERMIT ORDR.DISPATCH CLASS(GMQQUEUE) ID(CIWS3D,CIWS3E) ACCESS(UPDATE)
RDEFINE GMQQUEUE S3C1.INITQ UACC(NONE) +
       ADDMEM(MQS3.VSG3.S3C1.INITQ)
RDEFINE GMQQUEUE S3C2.INITQ UACC(NONE) +
       ADDMEM(MQS3.VSG3.S3C2.INITQ)
PERMIT S3C1.INITQ    CLASS(GMQQUEUE) ID(CIWS3D) ACCESS(UPDATE)
PERMIT S3C2.INITQ    CLASS(GMQQUEUE) ID(CIWS3E) ACCESS(UPDATE)
PERMIT ORDR.DISPATCH CLASS(GMQQUEUE) ID(CIWS3D) ACCESS(UPDATE)
PERMIT ORDR.DISPATCH CLASS(GMQQUEUE) ID(CIWS3E) ACCESS(UPDATE)
PERMIT ORDR.DISPATCH CLASS(GMQQUEUE) ID(WEBAS3) ACCESS(UPDATE)
```

In Example 9-28 we add the response and request queues to the generic group ORDR.DISPATCH. By default, when CICS makes an API-resource security check on a CICS connection, it checks to see if two user IDs have access to the resource. The first user ID checked is the CICS region user ID. The second user ID checked is the user ID associated with the CICS transaction. The CICS region user IDs (CIWS3D and CIWS3E) are permitted UPDATE access to the group. The user ID WEBAS3 is also granted UPDATE access.

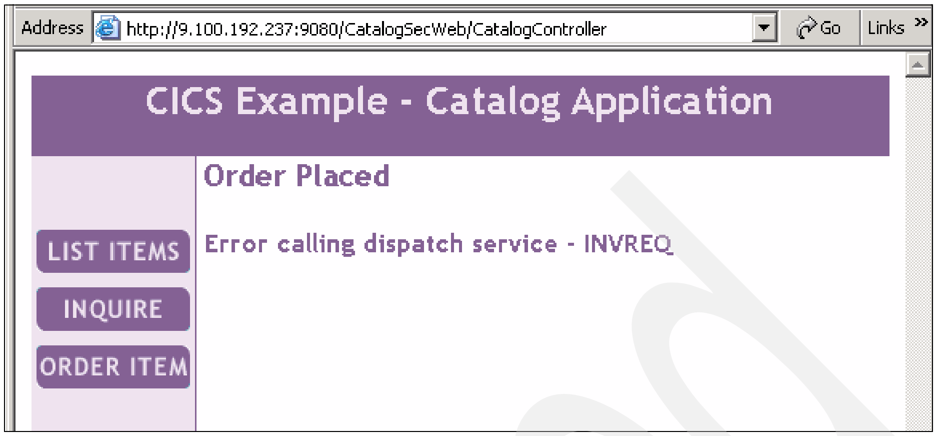
For full details on securing access to WMQ queues, see *WebSphere MQ for z/OS System Setup Guide V6.0,* SC34-6583.

9.6.6 Testing security with WMQ

To test the WMQ security scenario, we started a Web browser session and entered the URL:

```
http://cam21-pc3:9080/CatalogWeb/Welcome.jsp
```

Figure 9-23 shows the result of the initial ORDER ITEM request.


Address http://9.100.192.237:9080/CatalogSecWeb/CatalogController

CICS Example - Catalog Application

Order Placed

Error calling dispatch service - INVREQ

Figure 9-23 Browser - INVREQ calling dispatch server

We noticed the following RACF message in the CICS log of region CIWSS3C2 (Example 9-29).

Example 9-29 RACF message when WEBAS3 has no access to DISP

```
ICH408I USER(WEBAS3  ) GROUP(CIWS    ) NAME(WEB USER1
   CIWS3E.DISP CL(TCICSTRN)
   INSUFFICIENT ACCESS AUTHORITY
   ACCESS INTENT(READ   )  ACCESS ALLOWED(NONE   )
```

This error shows that user ID WEBAS does not have access to the DISP transaction. We permitted access to the DISP transaction to the user ID WEBAS3 using the RACF commands shown in Example 9-30.

Example 9-30 RACF commands to define DISP and grant access to it

```
RDEFINE GCICSTRN WS3DISP ADDMEM(CIWS3E.DISP) UACC(NONE) OWNER(IBMUSER)
PERMIT WS3DISP CLASS(GCICSTRN) ID(WEBAS3) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
RDEFINE SURROGAT WEBAS3.DFHSTART UACC(NONE) OWNER(WEBAS3)
PERMIT WEBAS3.DFHSTART CLASS(SURROGAT) ID(CIWS3E) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

We then repeated the ORDER ITEM request successfully. Figure 9-24 on page 321 shows the DISP transaction executing with the user ID of WEBAS3 on CIWSS3C2, with CPIQ and CPIL executing with the user ID of CIWS3E.

```
I TAS
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0001242) Tra(CPIL)           Sus Tas Pri( 001 )
    Sta(SD) Use(CIWS3E  ) Uow(BE07E790763F2B65) Hty(MQSeries)
 Tas(0001243) Tra(CPIQ)           Sus Tas Pri( 001 )
    Sta(S ) Use(CIWS3E  ) Uow(BE07E79075898505) Hty(RZCBNOTI)
 Tas(0001244) Tra(DISP)           Sus Tas Pri( 001 )
    Sta(U ) Use(WEBAS3  ) Uow(BE07E79076A61846) Hty(EDF     )



                                                   SYSID=S3C2 APPLID=A6POS3C2
```

Figure 9-24 DISP transaction running in CIWSS3C2

Security scenarios using CICS WS-Security support

In this chapter we outline several security scenarios that demonstrate how you can secure CICS Web services using the CICS-supplied message handler, DFHWSSE1.

We provide step-by-step security configuration for a number of scenarios, including:

- Basic authentication
- Signing a SOAP message
- Encrypting a SOAP message

Note: The security scenarios documented in this chapter use the CICS-supplied WS-Security support. See Chapter 9, “Security scenarios” on page 275 for WS-Security scenarios using custom message handlers.

We show how we configured both CICS and WebSphere Application Server for these security scenarios.

10.1 Preparation

Security techniques for securing Web services were discussed in Chapter 8, “Securing Web services” on page 235. In this chapter, we describe the following security scenarios:

- Basic authentication

  In this scenario we show how the CICS-supplied message handler DFHWSSE1 can extract a WS-Security UsernameToken from a SOAP header, validate the username and password, and set the user ID of the CICS task to the username passed in the header. This scenario is described in “Basic authentication” on page 330.

- Signing a SOAP message

  In this scenario we show how CICS and WebSphere Application Server can be configured to exchange signed SOAP messages. The SOAP message sent by WebSphere Application Server contains an X.509 certificate that allows the CICS transaction to run under a user ID associated with the X.509 certificate. See “Signing a SOAP message” on page 358 for details of this scenario.

- Encrypting a SOAP message

  In this scenario we show how CICS and WebSphere Application Server can be configured to exchange encrypted SOAP messages. The SOAP message contains an encrypted body so that the content of the message cannot be understood by an unauthorized party. This scenario is described in 10.6, “Encrypting a SOAP message” on page 385.

10.1.1 Software checklist

The software we used is listed in Table 10-1.

Table 10-1 Software used in the WS-Security security scenarios

Windows	z/OS
Internet Explorer V6.0	z/OS V1.7
Windows XP V5.1 SP2	CICS Transaction Server V3.1 (with PTFs UK15271 and UK15261 installed)
IBM WebSphere Application Server - ND V6.1.0.0	RACF V1.7

Windows	z/OS
Our J2EE applications ▶ CatalogSec2.ear Catalog manager service requester application with no security enabled ▶ CatalogSec2_WS-Security_BasicAuth.ear Catalog manager service requester application with WS-security basic authentication enabled ▶ CatalogSec2_WS-Security_Signature.ear Catalog manager service requester application with WS-security signature enabled ▶ CatalogSec2_WS-Security_Encryption.ear Catalog manager service requester application with WS-security encryption enabled	Our user-written CICS programs ▶ CIWSMSGH (message handler program) This program changes the transaction ID for the Web service requests received in the pipeline. ▶ SNIFFER (message handler program) This program browses through the containers available in the pipeline.

For this chapter we use a slightly different version of the Catalog application than the version used in Chapter 9. In the wsdl created by CICS the names of the Web service binding ports are the same for each of the three Web services. We noticed that this caused some problems in AST when editing the WebSphere WS-Security configuration files. We changed the name of the binding ports in the corresponding wsdl files from the default DFH0XCMNPort to DFH0XCMNPortPO for the placeOrder service, DFH0XCMNPortIS for inquireSingle and DFH0XCMNPortIC for inquireCatalog. We then recreated the Web services using our development tool (Rational Application Developer),

10.1.2 Definition checklist

The definitions we used are listed in Table 10-2.

Table 10-2 Settings used in the WS-Security security scenarios

Value	CICS TS	WebSphere Application Server
IP name	wtsc.itso.ibm.com	mikee01
IP address	9.12.4.75	9.12.4.217
TCP/IP port	14301	9081
Jobname	CIWSS3C1	
APPLID	A6POS3C1	
TCPIPSERVICE	S3C1	

Value	CICS TS	WebSphere Application Server
Provider PIPELINEs	PIPE1 used for inquireSingle and inquireCatalog services PIPEWSSE used for placeOrder service	
Configuration files	ITSO_7206_basicsoap12provider.xml ITSO_7206_wssec_basicauth_provider.xml ITSO_7206_wssec_signature_provider.xml ITSO_7206_wssec_encryption_provider.xml	
URIMAP	SECPORDR SECICATA SECISING	

The user IDs we used in our configuration are listed in Table 10-3.

Table 10-3 User IDs

Value	CICS TS
CICS region user ID	CIWS3D
CICS default user ID	CICSUSER
User IDs for which we wish to permit access	CIWSNW WEBAS1

Because the Catalog application inquireSingle and inquireCatalog services are read-only services, whereas the placeOrder service updates the database, we show here how to secure the placeOrder service. We run the inquireSingle and inquireCatalog services under a predefined user D (CIWSNW). The placeOrder service is run under the caller's user ID (WEBAS1).

Tip: For the examples described in this book, we permit access for single user IDs. In a production environment, you will probably create a group of users requiring common access. Once a group is built, you can permit access for the group. This allows users' access to be controlled by the group to which they belong, rather than by individual permissions. This simplifies the security definitions required.

10.2 Basic security configuration

First we discuss our basic security configuration, taking a CICS region with no security and configuring it to enable transaction security (we do not implement other types of CICS security such as resource security and command security).

Note: The security scenarios in this chapter are based on a different MVS system that those documented in Chapter 9, “Security scenarios” on page 275. We therefore repeat some of the security setup steps for CICS region CIWSS3C1 here.

In this section, we document the following steps:

- Creating a RACF key ring
- System initialization table parameters necessary to set up basic CICS security
- Testing the basic security configuration
- Configuring the pipeline
- Setting a user ID on a URIMAP definition

10.2.1 Creating a RACF key ring

In order to use the CICS WS-Security support for signing and encryption, it is necessary to create a RACF key ring, public-private key pairs, and X.509 certificates. In 10.4, “Certificate and key pair generation” on page 347 we show how to create certificates and key pairs for the signing and encryption scenarios. In the basic CICS security setup, however, we created a key ring for use by CICS region CIWSS3C1.

Note: In our tests we found that the CICS region must be connected to a RACF key ring even if the only CICS WS-Security support being used is basic authentication.

The sample clist DFH$RING is provided by CICS TS to help you build a RACF key ring and sample X.509 certificates for SSL usage. We used the DFH$RING clist to create the key ring `Ciws.Ciwss3c1`. See 9.5.1, “Creating a key ring and certificates on z/OS for CICS” on page 302 for information about using the DFH$RING sample clist.

10.2.2 Specifying the security SIT parameters

We specified the following security SIT parameters for our CICS region:

- `KEYRING=Ciws.Ciwss3c1`

  This specifies the name of a key ring we created in the RACF database that contains keys and certificates used by this CICS region.

- `SEC=YES`

  SEC=YES was specified to indicate that we wanted RACF services to control access to CICS resources.

- `SECPRFX=YES`

  We used security prefixing (SECPRFX=YES) in our CICS region, which prevents our RACF security profiles from affecting other CICS regions. This is useful in a production environment since it means all security profiles are unique to an individual region; however, it can mean more work for the security administrator because more profiles must be defined.

- `XTRAN=YES`

  XTRAN=YES was specified so CICS would control who could start transactions.

- `XUSER=YES`

  XUSER=YES specifies that CICS is to perform surrogate user checking.

For detailed information about CICS security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454.

10.2.3 Testing the basic security configuration

In our testing, we followed the same process that is described in “Installing the service requester” on page 87 to deploy the CatalogSec2.ear.

When attempting to place an order with the catalog application, the Web browser received an error message and the cause of the error was reported by CICS as a security violation (Example 10-1).

Example 10-1 CICSUSER security violation messages running CPIH

```
DFHXS1111 08/09/2006 18:05:38 A6POS3C1 CPIH Security violation by user CICSUSER for resource
          CIWS3D.CPIH in class TCICSTRN. SAF codes are (X'00000008',X'00000000'). ESM codes
          are (X'00000008',X'00000000').
DFHWB0361 08/09/2006 18:05:38 A6POS3C1 An attempt to attach a CICS Web alias transaction for
          userid CICSUSER has failed because the user is not authorized to execute transaction
          CPIH. Host IP address: 9.12.4.75. Client IP address: 9.12.4.217.
          TCPIPSERVICE: S3C1.
```

```
DFHAC2003 08/09/2006 18:05:38 A6POS3C1 Security violation has been detected term id = ????,
          trans id = CPIH, userid = CICSUSER.
```

The CICS default user ID does not have access to the CPIH transaction. This shows that the CICS default user ID is not authorized to access the placeOrder service. In the following scenarios, we enable placeOrder service access to a set of authorized user IDs.

10.2.4 Configuring the pipeline

Example 10-2 shows the initial pipeline configuration file used for our CICS WS-Security test setup.

Example 10-2 Pipeline configuration file ITSO_7206_basicsoap12provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
    <transport>
      <default_transport_handler_list>
          <handler>
             <program>CIWSMSGH</program>
             <handler_parameter_list/>
          </handler>
      </default_transport_handler_list>
    </transport>
    <service>
      <service_handler_list>
        <handler>
          <program>SNIFFER</program>
          <handler_parameter_list/>
        </handler>
      </service_handler_list>
      </terminal_handler>
        <cics_soap_1.2_handler/>
      </terminal_handler>
    </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

- The message handler program CIWSMSGH is used to change the transaction ID to INQS, INQC, or ORDR based on the service request. See “Customizing the pipeline configuration file” on page 80 for more information about the CIWSMSGH program.

- The message handler program SNIFFER is a simple program that browses through the containers available in the pipeline. See “Using SNIFFER” on page 106 for more information about the SNIFFER program.

10.2.5 Setting a user ID on a URIMAP definition

We initially want to run the CPIH, INQS, INQC, and ORDR transactions under a predefined user ID, other than the default CICS user ID. In the security scenarios that follow, we then show how to run ORDR transactions for specific placeOrder service requests under user IDs which flow in the SOAP messages.

To set the predefined user ID for the CPIH and ORDR transactions, we created a URIMAP definition for the placeOrder service such that the transactions run with the user ID CICSNW. See Section 9.3, “Setting the user ID on a URIMAP definition” on page 280 for more information about defining URIMAPs. We then permitted RACF access to the CPIH and ORDR transactions for the user ID CIWSNW.

Figure 10-1 shows the CPIH and ORDR transactions running under user ID CIWSNW.

.

```
INQUIRE TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000081) Tra(CPIH)          Sus Tas Pri( 001 )
    Sta(U ) Use(CIWSNW  ) Uow(BF3BFA8462A2600C) Hty(RZCBNOTI)
 Tas(0000082) Tra(ORDR)          Sus Tas Pri( 001 )
    Sta(U ) Use(CIWSNW  ) Uow(BF3BFA8474FD05C9) Hty(EDF     )
 Tas(0000084) Tra(CEDF) Fac(E022) Sus Ter Pri( 001 )
    Sta(SD) Use(NIGELW  ) Uow(BF3BFA847A218343) Hty(ZCIOWAIT)
 Tas(0000085) Tra(CEMT) Fac(E021) Run Ter Pri( 255 )
    Sta(TO) Use(NIGELW  ) Uow(BF3BFA90A97D2601)
```

Figure 10-1 ORDR executing with preset user ID

We also created URIMAP resources for the inquireSingle and inquireCatalog services such that the INQS and INQC transactions also run with user ID CIWSNW, and we permitted RACF access to the INQS and INQC transactions for the user ID CIWSNW.

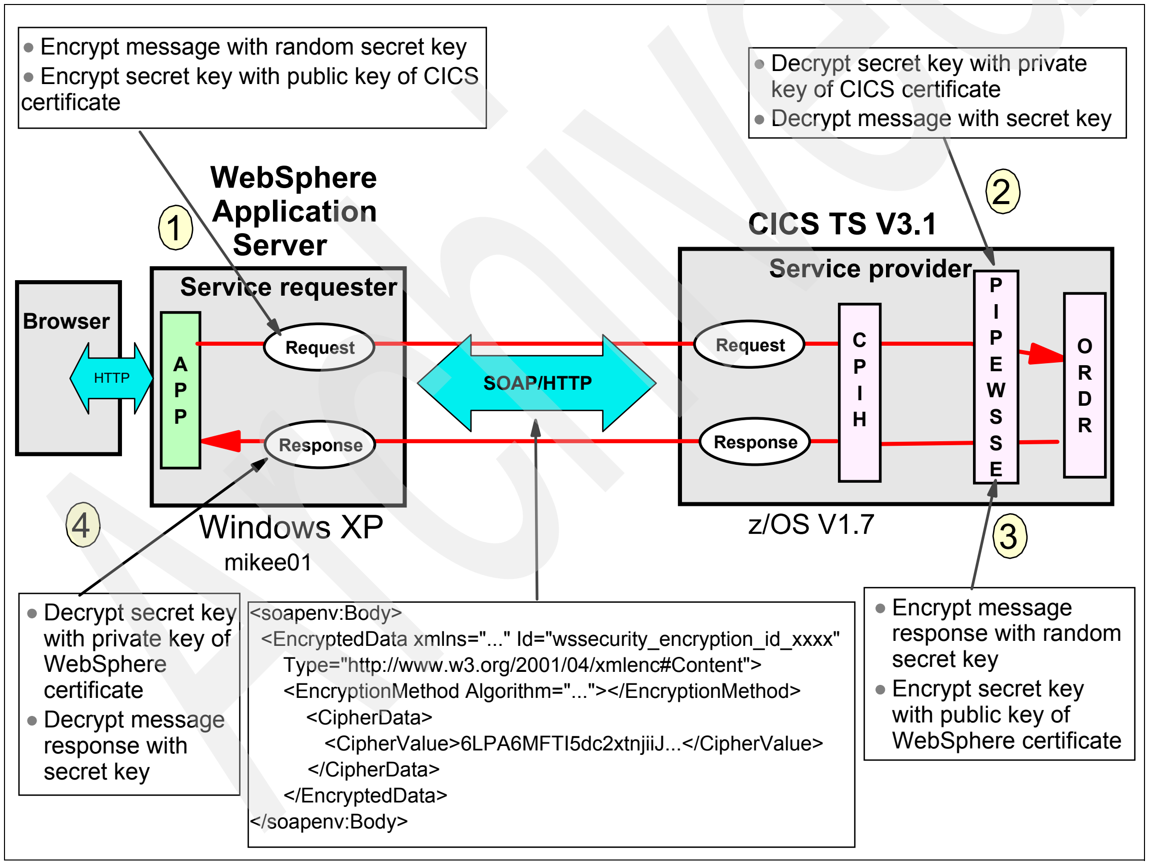
10.3 Basic authentication

WebSphere Application Server supports the generation of security tokens that are passed in the WS-Security SOAP header. A UsernameToken is a token for basic authentication; it has a user name and password. In 9.4, “Enabling SOAP message security with HTTP” on page 285 we show how we wrote the message handler program CIWSSECS, which we then used to process a UsernameToken passed in a Web service call from WebSphere Application Server.

In this section we show how a UsernameToken can also be processed by the CICS-supplied message handler, DFHWSSE1. DFHWSSE1 extracts the UsernameToken from the SOAP header, validates the username and password, and sets the user ID of the CICS task to the username passed in the header.

Note: CICS supports UserNameToken in service provider mode only.

This scenario is shown in Figure 10-2.



Encrypt message with random secret key
Encrypt secret key with public key of CICS certificate
Decrypt secret key with private key of CICS certificate
Decrypt message with secret key
1
2
3
4
WebSphere Application Server
Service requester
Browser
HTTP
APP
Request
Response
SOAP/HTTP
CICS TS V3.1
Service provider
Request
Response
CPIH
PIPEWSSE
ORDR
Windows XP
mikee01
z/OS V1.7
Decrypt secret key with private key of WebSphere certificate
Decrypt message response with secret key
Encrypt message response with random secret key
Encrypt secret key with public key of WebSphere certificate

```
<soapenv:Body>
  <EncryptedData xmlns="..." Id="wssecurity_encryption_id_xxxx"
     Type="http://www.w3.org/2001/04/xmlenc#Content">
     <EncryptionMethod Algorithm="..."></EncryptionMethod>
        <CipherData>
          <CipherValue>6LPA6MFTI5dc2xtnjiiJ...</CipherValue>
        </CipherData>
     </EncryptedData>
</soapenv:Body>
```

Figure 10-2 Enabling CICS WS-Security basic authentication

Recommendation: Internal tests have shown that the performance of a user-written message handler program such as CIWSSECS for processing a UsernameToken is significantly better than DFHWSSE1. If throughput is expected to be high and performance is a priority, we recommend that you use a user-written message handler program for basic authentication. If, however, you do not want to write custom handlers, or if performance is not a priority, then enabling basic authentication using the CICS-supplied DFHWSSE1 is a simpler solution.

We document the following steps that we performed to enable support for basic authentication:

- Configuring the service requester for basic authentication
- Configuring CICS for basic authentication
- Testing the basic authentication scenario

10.3.1 Configuring the service requester for basic authentication

The service requester setup tasks that we cover here are as follows:

- Defining the WebSphere WS-Security constraint for the Web service client
- Re-deploying the Web service client application

WebSphere WS-Security configuration files

The WebSphere WS-Security constraints are defined in the IBM extension of the J2EE Web services deployment descriptor. There are four configuration files:

- Client deployment descriptor extension file - includes request generator and response consumer constraints:

  ibm-webservicesclient-ext.xmi

- Client binding configuration file - includes how to apply request generator and response consumer constraints:

  ibm-webservicesclient-bnd.xmi

- Server deployment descriptor extension file - includes request consumer and response generator constraints:

  ibm-webservices-ext.xmi

- Server binding configuration file - includes how to apply request consumer and response generator constraints:

  ibm-webservices-bnd.xmi

The deployment descriptor extension files specify *what* security constraints are required, for example, what type of security token and whether to sign the message. The binding files specify *how* to apply the required security constraints defined in the deployment descriptor extension, for example, which security token is inserted and which keys are used for signing.

These deployment descriptor extension and binding files define the application-level security constraints and they belong to each application. See "WebSphere and SOAP message security" on page 266 for more information about configuring WebSphere WS-Security.

For our scenario, we needed to configure the client deployment descriptor and the client binding only, since our service provider is running in CICS and not in WebSphere Application Server.

Importing the base application

To configure our Web Service client to use WS-Security, we used the IBM WebSphere Application Server Toolkit V6.1 (AST).

Note: You can also use Rational Application Developer V6.0 (RAD) to configure WS-Security for WebSphere applications.

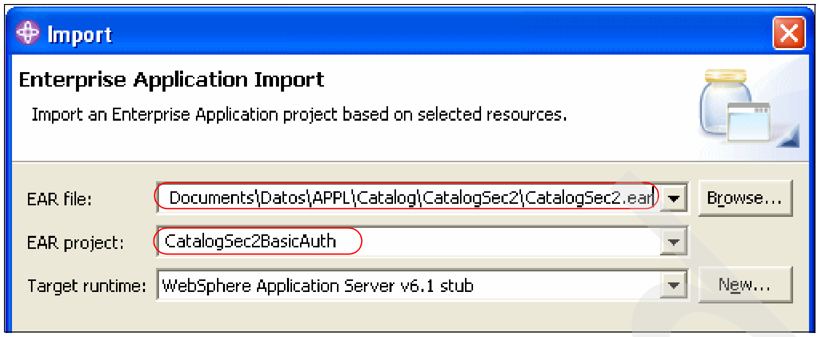
We performed the following steps:

1. We opened the AST in a new workspace. In Windows we clicked:

   Start → Programs → IBM WebSphere → Application Server Toolkit V6.1 → Application Server Toolkit

   We specified a new workspace for this scenario called

   `C:\AST\CatalogSec2\CatalogSec2BasicAuth`
2. To import the CatalogSec2 base application we clicked the **File** menu and then **Import**.
3. We selected **EAR file** in the list of options and clicked **Next**.
4. We clicked **Browse** to select the CatalogSec2.ear file. This file contains the Catalog base application.
5. We also changed the EAR project name to CatalogSec2BasicAuth as shown in Figure 10-3.


Import

Enterprise Application Import

Import an Enterprise Application project based on selected resources.

EAR file: Documents\Datos\APPL\Catalog\CatalogSec2\CatalogSec2.ear

EAR project: CatalogSec2BasicAuth

Target runtime: WebSphere Application Server v6.1 stub

Figure 10-3 Importing the CatalogSec2 ear file

6. Finally, we clicked **Next** → **Next** → **Finish**.

Editing the client configuration

We performed the following steps to edit the client configuration:

1. We expanded the Dynamic Web Project (CatalogSec2Web) in the Project Explorer and opened (double-clicked) the deployment descriptor as shown in Figure 10-4.

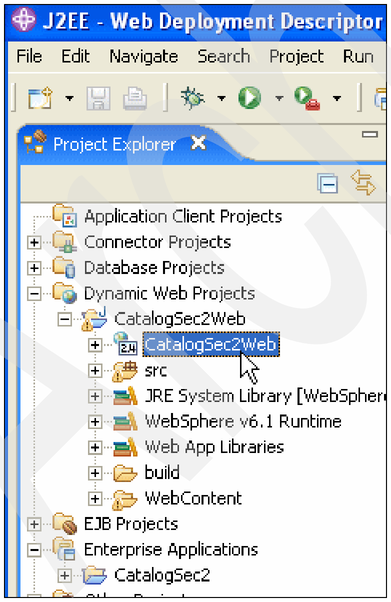
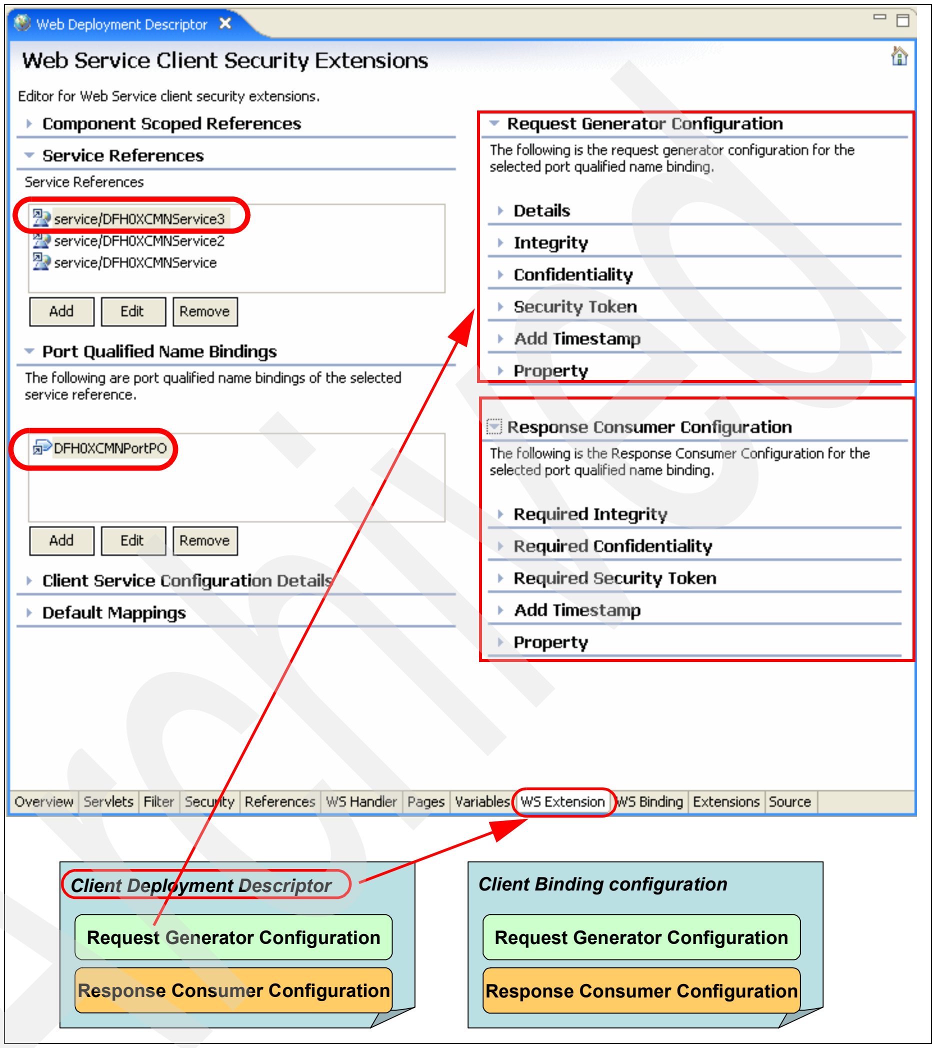


Figure 10-4 Opening the CatalogSec2Web Deployment Descriptor.

2. After the Deployment Descriptor Editor opened, we selected the **WS Extension** page. The WS Extension page is used for editing the client's

deployment descriptor extension file (ibm-webservicesclient-ext.xmi), so you can specify what security is required. Figure 10-5 shows the WS Extension page in the Deployment Descriptor Editor.



Editor for Web Service client security extensions.

Service References

The following are port qualified name bindings of the selected service reference.

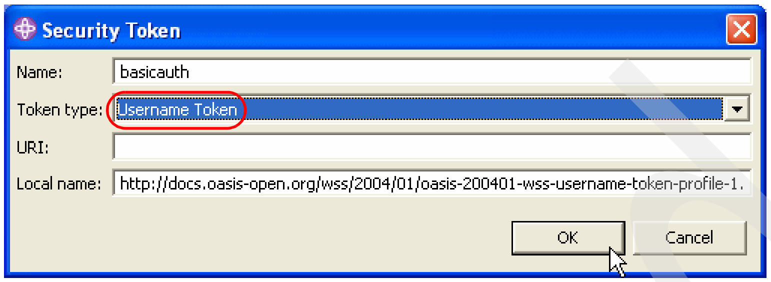
The following is the request generator configuration for the selected port qualified name binding.

The following is the Response Consumer Configuration for the selected port qualified name binding.

Figure 10-5 WS Extension page in Web service client Deployment Descriptor Editor

3. We clicked the **service/DFH0XCMNService3** reference (for the placeOrder service) and **DFH0XCMNPortPO** (the port binding for the service reference).
4. In the Request Generator Configuration, we expanded **Security Token** and clicked **Add**. We entered the name `basicauth`.

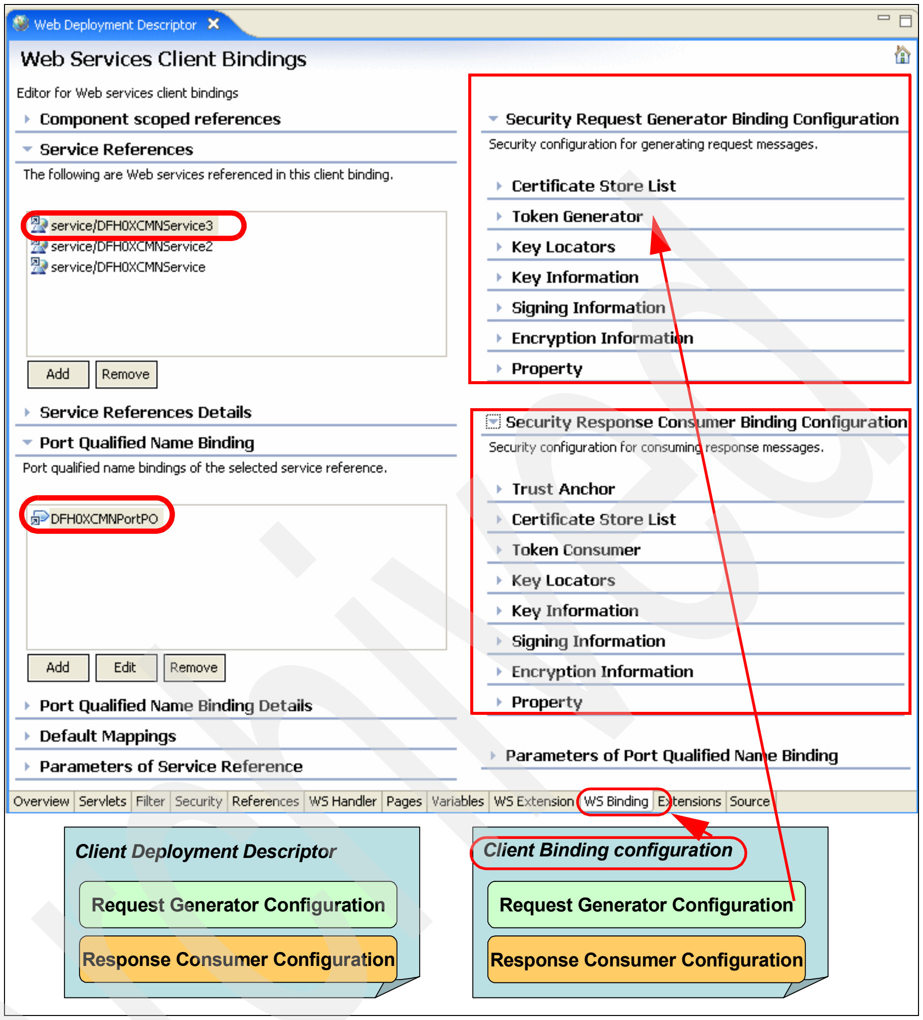
5. We selected the **Username Token** token type from the drop-down list. When you select a Token type, the Local name is filled in automatically (Figure 10-6). We left the URI empty.



Name: basicauth
Token type: Username Token
URI:
Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.

Figure 10-6 Security Token Dialog for specifying basic authentication

6. We clicked **OK** and a security token was created. We then saved the configuration by pressing Ctrl+s.
7. After specifying the security token, a corresponding token generator must be specified in the binding configuration. The WS Binding page is for editing the client’s binding file, so you can specify how to apply the required security. We clicked the **WS Binding** tab. Figure 10-7 shows the WS Binding page in the Deployment Descriptor Editor.


Web Deployment Descriptor
Web Services Client Bindings
Component scoped references
Service References
service/DFHOXCMNService3
service/DFHOXCMNService2
service/DFHOXCMNService
Add
Remove
Service References Details
Port Qualified Name Binding
DFHOXCMNPortPO
Add
Edit
Remove
Port Qualified Name Binding Details
Default Mappings
Parameters of Service Reference
Security Request Generator Binding Configuration
Certificate Store List
Token Generator
Key Locators
Key Information
Signing Information
Encryption Information
Property
Security Response Consumer Binding Configuration
Trust Anchor
Certificate Store List
Token Consumer
Key Locators
Key Information
Signing Information
Encryption Information
Property
Parameters of Port Qualified Name Binding
Overview
Servlets
Filter
Security
References
WS Handler
Pages
Variables
WS Extension
WS Binding
Extensions
Source
Client Deployment Descriptor
Request Generator Configuration
Response Consumer Configuration
Client Binding configuration
Request Generator Configuration
Response Consumer Configuration

Editor for Web services client bindings

The following are Web services referenced in this client binding.

Port qualified name bindings of the selected service reference.

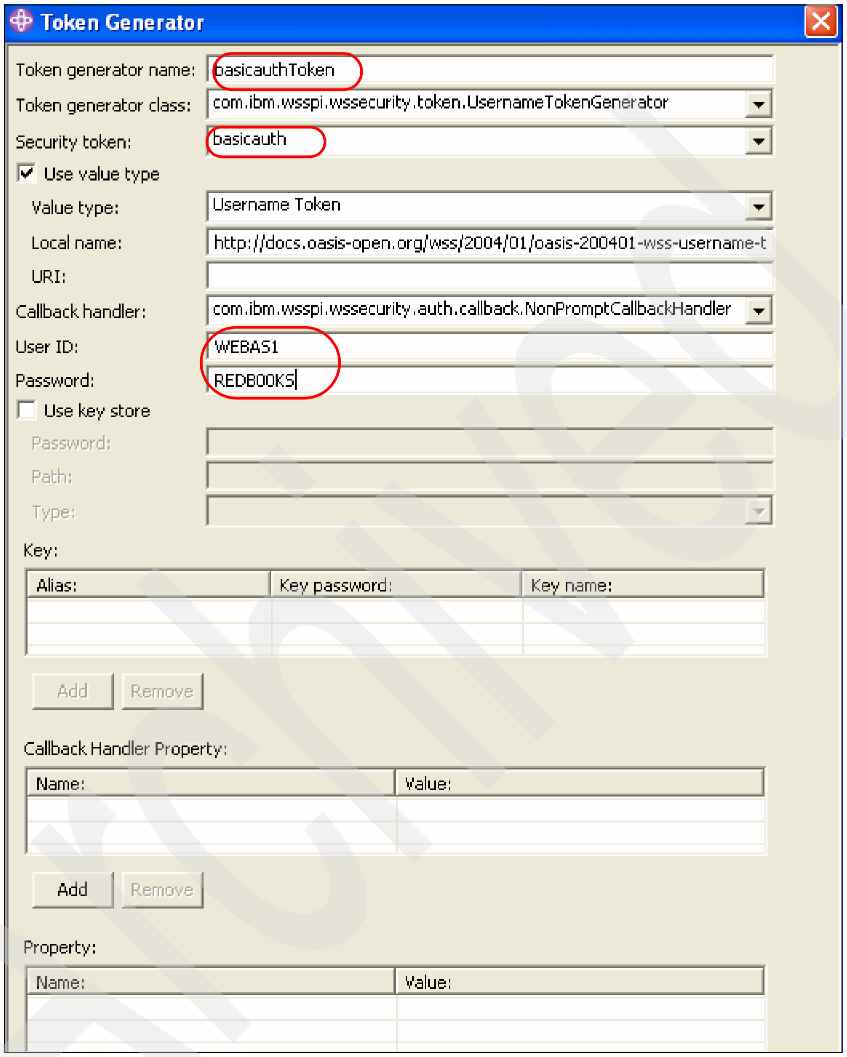
Security configuration for generating request messages.

Security configuration for consuming response messages.

Figure 10-7 WS Binding page in the Deployment Descriptor Editor

8. We selected **Token Generator** and clicked **Add**. We entered a Token generator name `basicauthToken` and allowed the Token generator class to default as `com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator`
9. To select the security token, we expanded the drop-down list and selected the **basicauth** security token that we defined previously on the WS Extension page.
10. We checked **Use value type** and selected the **Username Token** value type from the drop-down list. This selection filled the local name and callback handler.

11. We entered the user ID WEBAS1 and password REDBOOKS that we wanted to use to identify the Web service client (Figure 10-8).



Token generator name: basicauthToken
Token generator class: com.ibm.wsspi.wssecurity.token.UsernameTokenGenerator
Security token: basicauth
☑ Use value type
Value type: Username Token
Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-t
URI:
Callback handler: com.ibm.wsspi.wssecurity.auth.callback.NonPromptCallbackHandler
User ID: WEBAS1
Password: REDBOOKS
☐ Use key store
Password:
Path:
Type:

Key:

Alias:	Key password:	Key name:

Callback Handler Property:

Name:	Value:

Property:

Name:	Value:

Figure 10-8 Token Generator dialog

12. We clicked **OK**, and the token generator was created. We then saved the configuration.

Note: The User ID and Password in Figure 10-8 represent the credentials of the application server, *not* the browser user. The same credentials are passed to CICS in the SOAP message irrespective of the identity of the end user.

Redeploying the Web service application

After configuring a basic authentication security constraint for the service requester application, we exported a new EAR file called CatalogSec2_WS-Security_BasicAuth.ear. We followed the same process that is described in "Installing the service requester" on page 87 to deploy the CatalogSec2_WS-Security_BasicAuth.ear.

For further information about configuring WS-Security in WebSphere, refer to *Web Services Handbook for WebSphere Application Server 6.1,* SG24-7257.

10.3.2 Configuring CICS

The credentials configured using the Application Server Toolkit are used to form the WS-Security header that is contained in the SOAP message. An example of this message is shown in Example 10-3.

Example 10-3 SOAP security header extracted from a SOAP message

```
<wsse:Security soapenv:mustUnderstand="1"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd">
    <wsse:UsernameToken>
      <wsse:Username>WEBAS1</wsse:Username>
      <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
       wss-username-token-profile-1.0#PasswordText">REDBOOKS</wsse:Password>
    </wsse:UsernameToken>
</wsse:Security>
```

This header includes the `mustUnderstand="1"` attribute, which indicates that either this header must be processed or a SOAP fault thrown.

In order for CICS to process the security token in the SOAP header of the placeOrder service request, we made the following changes to the CICS configuration:

- Created a pipeline configuration file for basic authentication
- Created a new pipeline to process placeOrder requests and altered the URIMAP for the placeOrder service to use the new pipeline
- Permitted access to the ORDR transaction for user ID WEBAS1
- Permitted surrogate access for the user ID WEBAS1

The details for these procedures are presented in the following sections.

Creating a pipeline configuration file for basic authentication

Example 10-4 shows the basic authentication pipeline configuration file that includes the message handler DFHWSSE1, and the configuration information for the handler.

Example 10-4 Pipeline config file, ITSO_7206_wssec_basicauth_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
    <transport>
      <default_transport_handler_list>
          <handler>
             <program>CIWSMSGH</program>
             <handler_parameter_list/>
          </handler>
      </default_transport_handler_list>
    </transport>
    <service>
      <service_handler_list>
        <handler>
          <program>SNIFFER</program>
          <handler_parameter_list/>
        </handler>
        <wsse_handler>
          <dfhwsse_configuration version="1">
            <authentication mode="basic">
            </authentication>
          </dfhwsse_configuration>
        </wsse_handler>
      </service_handler_list>
      </terminal_handler>
        <cics_soap_1.2_handler/>
      </terminal_handler>
    </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

The `<wsse_handler>` element shown in Example 10-4 contains a `<dfhwsse_configuration>` element that specifies configuration information for DFHWSSE1.

The `<authentication mode="basic">` specifies that inbound messages must contain a UserNameToken.

Creating a new pipeline to process placeOrder requests

Since the inquireCatalog and inquireSingle service requests will not be subject to basic authentication, we need to create a new pipeline specifically for the placeOrder requests. We also need to change the SECPORDR URIMAP definition to link the placeOrder service to the new pipeline. We do this using the RDO commands in Example 10-5 and then re-installing the PIPELINE and URIMAP resource definitions.

Example 10-5 Create pipeline for WS-Security basic authentication

```
CEDA COPY PIPELINE(PIPE1) GROUP(S3C1EXWS) AS(PIPEWSSE)
CEDA ALTER PIPELINE(PIPEWSSE) GROUP(S3C1EXWS)
Configfile(/CIWS/S3C1/config/ITSO_7206_wssec_basicauth_provider.xml)
CEDA ALTER URIMAP(SECPORDR) GROUP(S3C1EXWS) PIPELINE(PIPEWSSE)
```

Permitting access to the ORDR transaction

We permitted access to the ORDR transaction for the user ID WEBAS1 with the following:

```
PERMIT CIWS3D.ORDR CLASS(TCICSTRN) ID(WEBAS1) ACCESS(READ)
SETROPTS RACLIST(TCICSTRN) REFRESH
```

Permitting surrogate access

Example 10-6 shows the RACF commands that we used to permit the predefined user ID (CIWSNW) to start transactions (such as ORDR) with the user ID WEBAS1.

Example 10-6 RACF commands to allow CIWSNW to act as surrogate for WEBAS1

```
RDEFINE SURROGAT WEBAS1.DFHSTART UACC(NONE) OWNER(NIGELW)
PERMIT WEBAS1.DFHSTART CLASS(SURROGAT) ID(CIWSNW) ACCESS(READ)
SETROPTS RACLIST(SURROGAT) REFRESH
```

For more information regarding surrogate user security, see *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454.

10.3.3 Testing basic authentication

Now when we submit an order request on the Web browser, the credentials are added to the SOAP Security header and are extracted by the DFHWSSE1 program.

Figure 10-9 shows the ORDR transaction running with the user ID WEBAS1, while the CPIH transaction continues to execute with the CIWSNW user ID.

```
INQUIRE TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000055) Tra(CPIH)           Sus Tas Pri( 001 )
    Sta(U ) Use(CIWSNW  ) Uow(BF3C2E9E2114284E) Hty(RZCBNOTI)
 Tas(0000056) Tra(ORDR)           Sus Tas Pri( 001 )
    Sta(U ) Use(WEBAS1  ) Uow(BF3C2E9E24801B8B) Hty(EDF     )
 Tas(0000058) Tra(CEDF) Fac(E024) Sus Ter Pri( 001 )
    Sta(SD) Use(NIGELW  ) Uow(BF3C2E9E28D4A149) Hty(ZCIOWAIT)
 Tas(0000059) Tra(CEMT) Fac(E025) Run Ter Pri( 255 )
    Sta(TO) Use(NIGELW  ) Uow(BF3C2EA64198CCCE)

                                                  SYSID=S3C1 APPLID=A6POS3C1
```

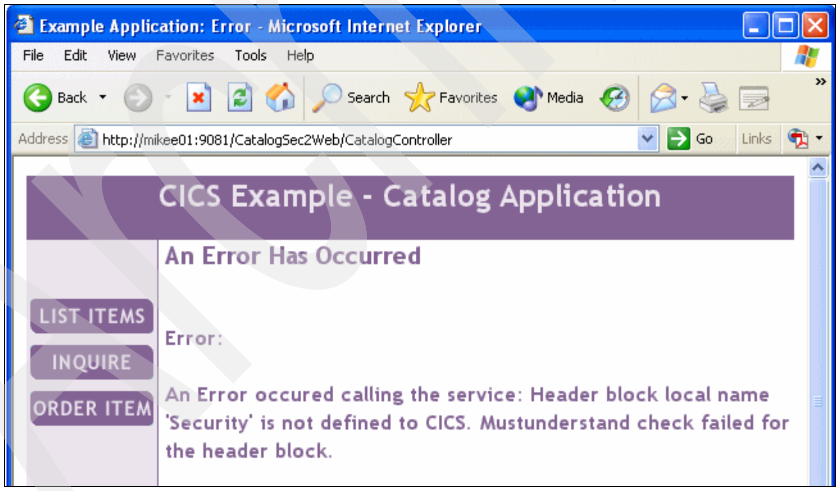
Figure 10-9   ORDR executing with basic authentication user ID

SOAP fault messages

In this section we show some of the SOAP faults that you may see after enabling CICS WS-Security basic authentication.

- Service requester does not send UserNameToken

  If the service requester does not send a UsernameToken, DFHWSSE1 returns a SOAP fault as shown in Figure 10-10.


Example Application: Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mikee01:9081/CatalogSec2Web/CatalogController
Go Links
LIST ITEMS
INQUIRE
ORDER ITEM

CICS Example - Catalog Application

An Error Has Occurred

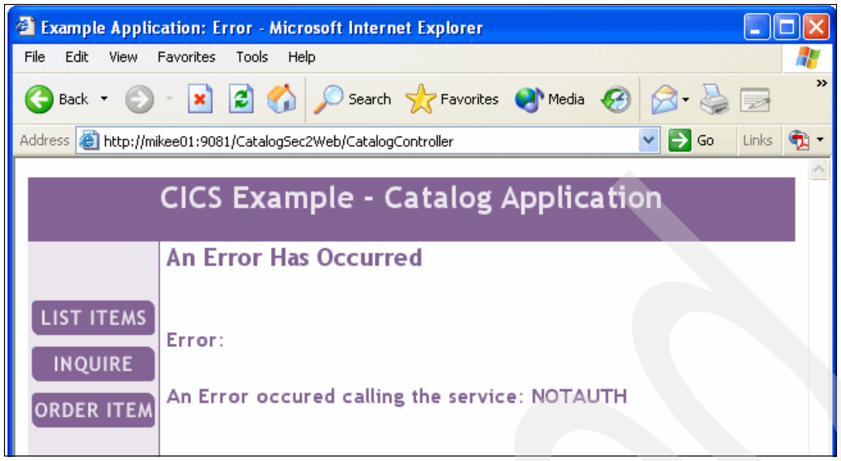
Error:

An Error occured calling the service: Header block local name 'Security' is not defined to CICS. Mustunderstand check failed for the header block.

Figure 10-10   DFHWSSE1 SOAP fault: No UsernameToken

- UserNameToken credentials are invalid

  If the credentials extracted do not validate, then DFHWSSE1 creates a SOAP fault message as shown in Figure 10-11.


Example Application: Error - Microsoft Internet Explorer

http://mikee01:9081/CatalogSec2Web/CatalogController

CICS Example - Catalog Application

An Error Has Occurred

Error:

An Error occured calling the service: NOTAUTH

Figure 10-11 DFHWSSE1 SOAP fault: UsernameToken credentials invalid

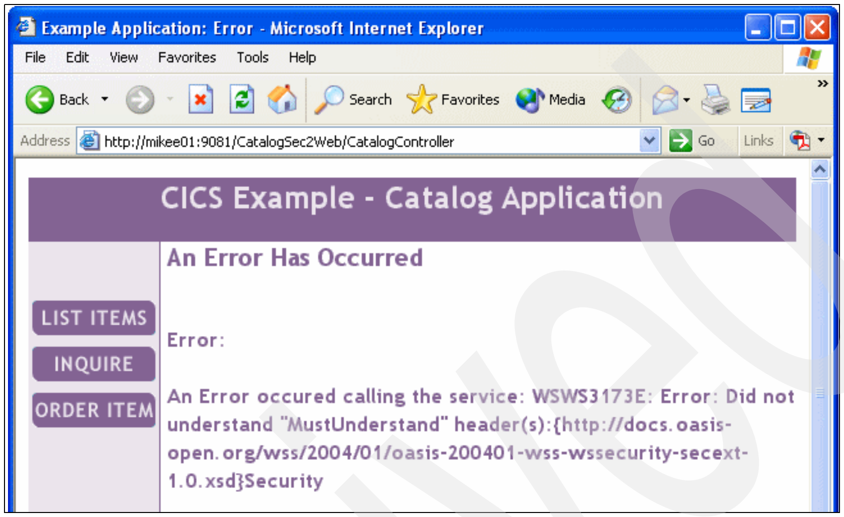
The SOAP fault for this message, shown in Example 10-7, highlights that the user credentials are invalid and that the RACF revoke count is incremented for this user ID.

Example 10-7 SOAP fault for invalid basic authentication credentials

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>wsse:FailedAuthentication
      </faultcode>
      <faultstring>NOTAUTH
      </faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
          <message>The supplied password is wrong. If the external security
          manager is RACF, the revoke count maintained by RACF is incremented
          </message>
          <errorcode>3
          </errorcode>
        </e:myfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- WebSphere rejects SOAP response sent by CICS due to empty security header.

  During our testing we experienced the problem shown in Figure 10-12.


Example Application: Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mikee01:9081/CatalogSec2Web/CatalogController
Go Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM

An Error Has Occurred

Error:

An Error occured calling the service: WSWS3173E: Error: Did not understand "MustUnderstand" header(s):{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd}Security

Figure 10-12 DFHWSSE1 SOAP fault: Empty security token

In this case CICS has returned the empty security header shown in Example 10-8.

Example 10-8 Empty security header returned from CICS

```
<wsse:Security
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
secext-1.0.xsd" SOAP-ENV:mustUnderstand="1"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-u
tility-1.0.xsd" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"/>
```

We found that, depending on the version of WebSphere Application Server and the maintenance level of CICS TS V3.1, it is possible that WebSphere will reject this SOAP response from CICS due to it containing an empty security header.

The WebSphere exception for this error, shown in Example 10-9, highlights that WebSphere is not expecting a security header and is rejecting the empty one sent by CICS because it contains the `mustUnderstand="1"` attribute.

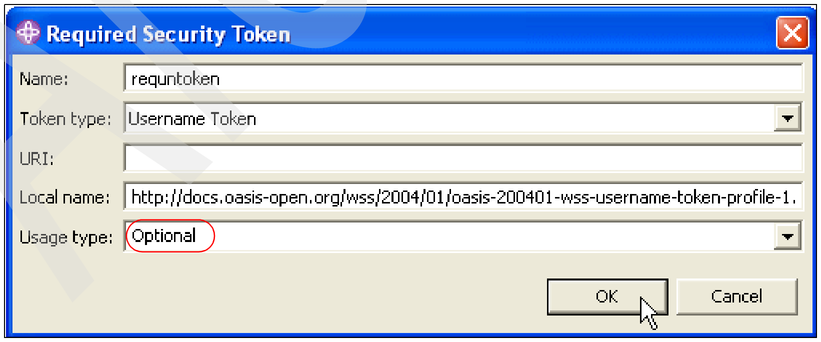
Example 10-9 WebSphere exception for empty security header

```
Exception: WSWS3173E: Error: Did not understand "MustUnderstand"
header(s):{http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-s
ecext-1.0.xsd}Security:
```

We were able to bypass this problem by configuring WebSphere to optionally receive a security token sent in the CICS response.

Perform the following steps to configure WebSphere to tolerate an empty security header sent in the CICS response:

1. Open the CatalogSec2Web Deployment Descriptor.
2. Select the WS Extension page.
3. In Service References select the **service/DFH0XCMNService3** reference. then click **DFH0XCMNPortPO** in Port Qualified Name Bindings.
4. Expand **Response Consumer Configuration**.
5. Expand **Required Security Token** and click **Add**.
6. In the Required Security Token window returned (Figure 10-13), make the following entries and selections:
   - Enter a name for this required security token, for example, `requntoken`.
   - Select **Username Token** from the Token type drop-down list.
   - Select **Optional** for the Usage type from the drop-down list.
     The available choices are Required or Optional. If you select Required, a SOAP fault is thrown if a required security token is not included in a client's request message. If you select Optional, the process of consuming a request message continues without throwing a SOAP fault.
7. Click **OK**.

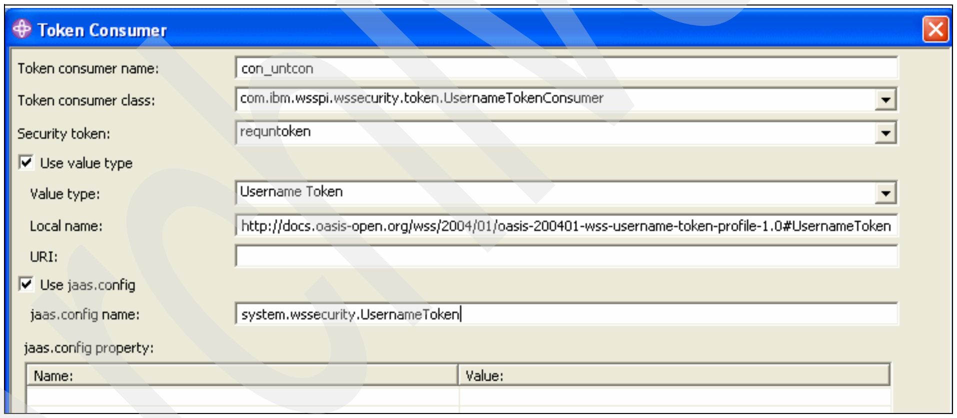


Name: requntoken
Token type: Username Token
URI:
Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.
Usage type: Optional

Figure 10-13 WS Extension page Required Security Token dialog box

8. Save the configuration by pressing Ctrl+s.

9. Open the WS Binding page.
10. In Service References select the **service/DFH0XCMNService3** reference, then click **DFH0XCMNPortPO** in Port Qualified Name Bindings.
11. Expand **Security Response Consumer Binding Configuration.**
12. Expand **Token Consumer** and click **Add.**
13. In the Token Consumer dialog box returned (Figure 10-14) make the following entries and selections:
    - Enter a Token consumer name, for example `con_untcon`.
    - Select **com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer** as the Token consumer class.
    - From the Security token drop-down list select the security token specified in the WS Extension page. In our case this was **requntoken**.
    - Select the **Use jaas.config** option and enter `system.wssecurity.UsernameToken` as the jaas.config name.
14. Click **OK**.


Token Consumer

Token consumer name: con_untcon
Token consumer class: com.ibm.wsspi.wssecurity.token.UsernameTokenConsumer
Security token: requntoken
[✓] Use value type
Value type: Username Token
Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken
URI:
[✓] Use jaas.config
jaas.config name: system.wssecurity.UsernameToken
jaas.config property:

Name:	Value:

Figure 10-14 Token Consumer dialog box in WS Binding page

15. Save the Deployment Descriptor.
16. Export the application as an ear file and install it in the WebSphere Application Server.

Important: APAR PK31924 has been raised for this problem. If you experience this problem and no fix is available for PK31924 at the time, you will need to configure WebSphere to tolerate an empty security header sent by CICS. This problem can also occur for digital signature processing.

10.4 Certificate and key pair generation

In this section we show how we created the certificates and key pairs that are used in the WS-Security signature and encryption test scenarios that follow.

When setting up a WS-Security environment you have a choice between self-signed certificates and CA-signed certificates:

- Self-signed certificates

  A self-signed certificate is an identity certificate that is signed by its creator, so the creator is verifying that the certificate is valid.

- Certificating authority (CA) signed certificates

  CA-signed certificates are created by a user organization and sent to a CA to be signed. Before signing a certificate the CA verifies that the organization requesting the certificate is actually who they claim to be, so the CA is verifying that the certificate is valid.

In the examples in this chapter, we use self-signed certificates because they are more easily and quickly generated. In a production environment, it is expected that CA-signed certificates will be used because they provide a more secure solution.

Figure 10-15 shows the certificates and key pairs used in the signature and encryption test scenarios.

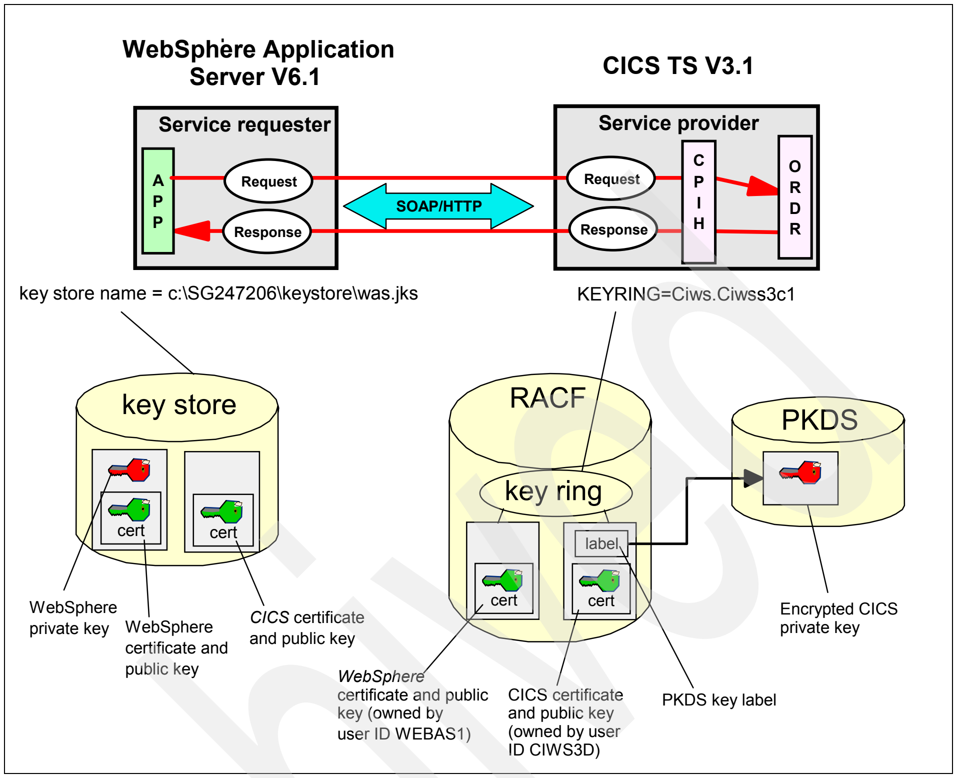

WebSphere Application Server V6.1
CICS TS V3.1
Service requester
A P P
Request
Response
SOAP/HTTP
Service provider
Request
Response
C P I H
O R D R
key store name = c:\SG247206\keystore\was.jks
KEYRING=Ciws.Ciwss3c1
key store
cert
cert
RACF
key ring
label
cert
cert
PKDS
WebSphere private key
WebSphere certificate and public key
CICS certificate and public key
WebSphere certificate and public key (owned by user ID WEBAS1)
CICS certificate and public key (owned by user ID CIWS3D)
PKDS key label
Encrypted CICS private key

Figure 10-15 Certificates and key pairs used in signature and encryption scenarios

Figure 10-15 shows two stores for our certificates and key pairs:

- The WebSphere key store
  - The WebSphere key store contains the WebSphere certificate and holds the WebSphere public and private key pair.
  - The key store also contains the CICS certificate with the public key only.
- RACF
  - RACF contains the CICS certificate and holds the CICS public key. The associated private key is stored in a Private Key Data Set (PKDS).

    In a public key cryptographic system, it is a priority to maintain the security of the private key. It is vital that only the intended user or application have access to the private key.

    ICSF and the cryptographic hardware features ensure this by enciphering PKA private keys under a unique PKA object protection key. The PKA object protection key has itself been enciphered under a PKA *master key*. Each PKA private key also has a name that is cryptographically bound to the private key and cannot be altered. ICSF uses the private key name or the *PKDS key label* to control access to the private key. This combination of hardware-enforced coupling of cryptographic protection and access control, through the use of RACF, is unique to ICSF. It provides a significant level of security and integrity for PKA applications.

– RACF also contains the WebSphere certificate with the public key only.

Note: If you use CA-signed certificates, you also need to add the CA certificates to the WebSphere key store and RACF key ring.

In this section, we document the following steps:

- Validating the cryptography hardware environment
- Creating the CICS certificate and key pair
- Creating the key store, WebSphere certificate, and key pair
- Adding the CICS certificate to the key store
- Adding the WebSphere certificate to RACF

Validating the cryptography hardware environment

CICS support for WS-Security signature and encryption processing benefits from the hardware acceleration capabilities of System z. It is a requirement, therefore, to run CICS on a system that has the appropriate cryptographic hardware configured. The specific cryptographic hardware requirement is dependent on the server type and z/OS level. See Section 7.4, “ICSF services used by CICS WS-Security support” on page 229 for more information about cryptographic hardware requirements.

The SC66 host system used for our signature and encryption test scenarios was a z9 EC (Enterprise Class) with 18 CPs. The SC66 had the following cryptographic hardware functions enabled:

- CP Assist for Cryptographic Function (CPACF)
- Crypto Express2 (CEX2) feature

Three CEX2C coprocessors and one CEX2A accelerator were configured.

Important: The CEX2 feature requires ICSF to be active.

Integrated Cryptographic Service facility

The Integrated Cryptographic Service Facility (ICSF) is a software element of z/OS that works with hardware cryptographic features and RACF to provide secure, high-speed cryptographic services in the z/OS environment. See Section 7.2, “ICSF” on page 225 for more information.

Example 10-10 shows the startup messages of ICSF on the SC66 system. It shows that 3 CEX2C coprocessors and 1 CEX2A coprocessor are configured.

Example 10-10 ICSF startup messages

```
S CSF
IEF695I START CSF WITH JOBNAME CSF IS ASSIGNED TO USER STC , GROUP SYS1
IEF403I CSF - STARTED - ASID=009A - SC66
CSFM441I CRYPTO EXPRESS2 COPROCESSOR E01, SERIAL NUMBER 94000264, ACTIVE
CSFM441I CRYPTO  EXPRESS2 COPROCESSOR E02, SERIAL NUMBER 95001434, ACTIVE
CSFM441I CRYPTO EXPRESS2 COPROCESSOR E03, SERIAL NUMBER 95001437, ACTIVE
CSFM435I CRYPTO EXPRESS2 ACCELERATOR F00 IS ACTIVE
CSFM001I ICSF INITIALIZATION COMPLETE
CSFM400I CRYPTOGRAPHY - SERVICES ARE NOW AVAILABLE.
```

ICSF must be configured and started in order to create public/private key pairs to be used by CICS for XML signature or encryption processing.

Creating the CICS certificate and key pair

In order to enable signature and encryption processing, we need to create a certificate and key pair to be used by CICS, and then connect the certificate to the CICS key ring.

Example 10-11 shows the RACF command that we used to create the CICS X.509 certificate.

Example 10-11 Create the CICS X.509 certificate (CICSCERT)

```
RACDCERT ID(CIWS3D) GENCERT
 SUBJECTSDN(CN('CICSCERT')
            T ('Ciwss3c1-cert')
            OU('PSSC')
            O ('ITSO')
            L ('Endicott')
            SP('New York')
            C ('US'))
 NOTBEFORE(DATE(2005-01-01) TIME(00:00:00))
 NOTAFTER (DATE(2014-12-31) TIME(23:59:59))
 KEYUSAGE (DOCSIGN DATAENCRYPT)
 WITHLABEL('CICSCERT')
 SIZE(1024)
 ICSF
```

- The RACDCERT GENCERT command creates the certificate and a public/private key pair.
- ID specifies that the CICS region user ID CIWS3D is to be associated with the certificate.
- SUBJECTDSN specifies the subject's X.509 distinguished name.
- NOTBEFORE and NOTAFTER specify the certificate validity range.
- KEYUSAGE specifies how the keys associated with the certificate are to be used. We specify:
  - DOCSIGN specifies that the certificate will be used for signing.
  - DATAENCRYPT specifies that the certificate will be used for encryption.
- WITHLABEL specifies the label name to be associated with the certificate.
- SIZE specifies the size of the private key expressed in decimal bits. We specified a high strength key length of 1024.
- ICSF specifies that the private key to be generated is an RSA key to be stored in the ICSF PKDS (public key data set).

  A certificate that CICS uses to sign SOAP messages (or decrypt encrypted SOAP messages) must be created with the ICSF option and ICSF must be configured and started in order for the certificate to be created or used by CICS. If ICSF is not operational, or does not have access to any cryptographic coprocessors or accelerators, you will receive the message `ICSF is not operational` when attempting to create the X.509 certificate.

  Important: A certificate that CICS uses to sign SOAP messages, or to decrypt encrypted SOAP messages, must be created with the ICSF option.

Example 10-12 shows the RACF commands that we used to connect the CICS X.509 certificate to the key ring and to list the certificate.

Example 10-12 RACF command to connect CICS certificate to RACF key ring

```
RACDCERT ID(CIWS3D) CONNECT(ID(CIWS3D) LABEL('CICSCERT') RING(Ciws.Ciwss3c1))
RACDCERT ID(CIWS3D) LIST
```

Example 10-13 shows the output of the RACDCERT LIST command.

Example 10-13 Listing of CICS X.509 certificate (CICSCERT)

```
Digital certificate information for user CIWS3D:

  Label: CICSCERT
  Certificate ID: 2QbDyebi88TDycPiw8XZ40BA
```

```
Status: TRUST
Start Date: 2005/01/01 00:00:00
End Date:   2014/12/31 23:59:59
Serial Number:
     >00<
Issuer's Name:
     >CN=CICSCERT.T=Ciwss3c1-cert.OU=PSSC.O=ITSO.L=ENDICOTT.SP=NEW YORK.C=U<
     >S<
Subject's Name:
     >CN=CICSCERT.T=Ciwss3c1-cert.OU=PSSC.O=ITSO.L=ENDICOTT.SP=NEW YORK.C=U<
     >S<
Key Usage: DATAENCRYPT, DOCSIGN
Private Key Type: ICSF
Private Key Size: 1024
PKDS Label: IRR.DIGTCERT.CIWS3D.SC66.BF386E434BD53FC9
Ring Associations:
  Ring Owner: CIWS3D
  Ring:
     >Ciws.Ciwss3c1<
```

Creating the key store and WebSphere certificate

In order to enable signature and encryption processing, we need to create a certificate and key pair to be used by WebSphere. The certificate and key pair are stored in a key store.

Table 10-4 shows the data used to create the key store and the WebSphere certificate.

Table 10-4 Client key store and certificate values

Field	Value
Key store name	`wasks`
Filename	`was.jks`
Storepass	`itso`
Storetype	`JKS`
Distinguished Name	`CN=wascert, O=IBM, C=US`
Key size	`1024`
Alias	`wascert_rsa`
Certificate file	`wascert_rsa.der`

Both the WebSphere public and private keys are associated to the certificate.

We performed the following tasks using the WebSphere Administration Console:

- Generating the client key store
- Creating the client certificate
- Extracting the client certificate file

Note: The capability to create certificates and key stores using the WebSphere admin console is new in WebSphere Application Server V6.1. If you are using an earlier version of WebSphere, you can use a key management tool such as *iKeyMan* or *keytool*. See “Importing the certificate” on page 305 for an example of using iKeyMan.

Creating the client keystore

Perform the following steps to create the client key store:

1. Select **Security** → **SSL certificate and key management**.
2. Under Related Items select **Key stores and certificates**.
3. The resulting panel shows you all the keystores that you have configured through the administrative console. Click **New** and enter these values:

Name:	`wasks`
Path:	`c:\SG247206\keystore\was.jks`
Password:	`itso`
Type:	`JKS`

4. Click **OK**.

Creating the client certificate

Use the following steps to create the personal certificate for the client key store:

- Select the **wasks** key store.
- Under Additional Properties click **Personal certificates**.
- Click **Create self-signed certificate** and enter these values:

Alias:	`wascert_rsa`
Key size:	`1024`
Common name:	`wascert`
Validity Period:	`365 (days)`
Organization:	`IBM`
Country:	`US`

- Click **OK**.

Be sure to save the configuration changes in the administrative console.

The Java **keytool** command can be used to list the contents of a key store:

```
keytool -list -keystore was.jks -storepass itso -v
```

The keytool command comes with the Java runtime environment. You might need to add the path to your java jre using the command:

```
set PATH=<WAS_HOME>\java\jre\bin;%PATH%
```

In this command, <WAS_HOME> is the directory where your WebSphere is installed (usually C:\Program Files\IBM\WebSphere\AppServer).

Adding the CICS certificate to the WebSphere key store

To make the public key of the CICS certificate available to WebSphere, we performed the following steps:

1. We used the RACF command shown in Example 10-14 to extract the certificate from the RACF database to a sequential data set.

Example 10-14 RACF command to extract CICS certificate from RACF

```
RACDCERT ID(CIWS3D) EXPORT (LABEL('CICSCERT')) DSN('CIWS.CICSCERT.DER')
FORMAT(CERTDER)
```

   The RACDCERT EXPORT command exports the certificate with label 'CICSCERT' to the MVS data set CIWS.CICSCERT.DER as a DER encoded X.509 certificate.

2. We FTPed the CICS certificate to the WebSphere machine.

   Example 10-15 shows how we used FTP to send the exported certificate (in binary format) to our WebSphere machine.

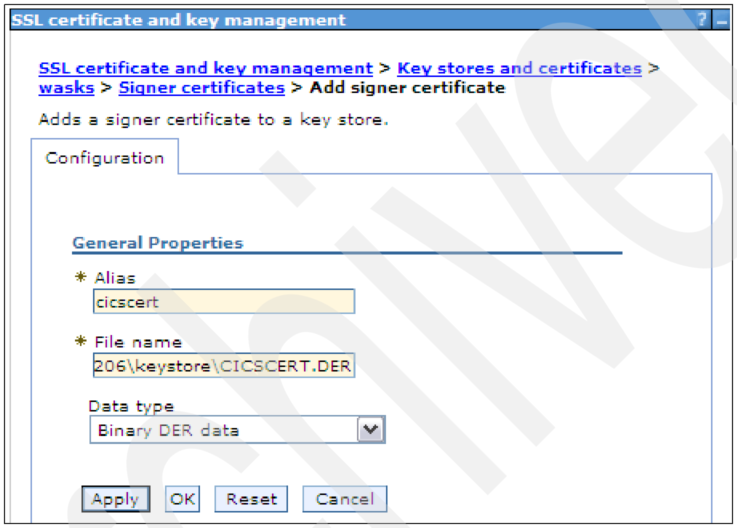
Example 10-15 FTP CICS certificate to WebSphere machine

```
C:\Documents and Settings\Resident>ftp wtsc66.itso.ibm.com
Connected to wtsc66.itso.ibm.com.
220-FTPMVS1 IBM FTP CS V1R7 at wtsc66.itso.ibm.com, 21:50:57 on 2006-08-15.
220 Connection will close if idle for more than 5 minutes.
User (wtsc66.itso.ibm.com:(none)): NIGELW
331 Send password please.
Password:
230 NIGELW is logged on.  Working directory is "NIGELW.".
ftp> lcd c:\SG247206\keystore
Local directory now C:\SG247206\keystore.
ftp> bin
200 Representation type is Image
ftp> cd 'CIWS'
250 "CIWS." is the working directory name prefix.
ftp> get CICSCERT.DER
```

```
200 Port request OK.
125 Sending data set CIWS.CICSCERT.DER
250 Transfer completed successfully.
ftp: 740 bytes received in 0.00Seconds 740000.00Kbytes/sec.
ftp> bye
221 Quit command received.
Goodbye.
```

3. We imported the CICS certificate into the WebSphere key store using the admin console.

   Figure 10-16 shows the certificate being imported using the admin console.



SSL certificate and key management

SSL certificate and key management > Key stores and certificates > wasks > Signer certificates > Add signer certificate

Adds a signer certificate to a key store.

Configuration

General Properties

* Alias: cicscert

* File name: 206\keystore\CICSCERT.DER

Data type: Binary DER data

Figure 10-16 Import CICS certificate into key store

Adding the WebSphere certificate to RACF

Perform the following steps to make the public key of the WebSphere certificate available to CICS:

1. Extract the certificate from the key store using the WebSphere administration console.

   In “Creating the key store and WebSphere certificate” on page 352 we show how we created the WebSphere certificate. We now extract the certificate from the jks to file wascert.der (a Binary DER file).

   The certificate has both public and private keys. However, if you distribute the certificate, you must ensure that you only send the public key and not the private key.

Follow these steps to export the public key from the wasks key store:

a. Locate the client personal certificates by selecting **SSL certificate and key management** → **Key stores and certificates** → **wasks** → **Personal certificates**.

b. Select the check box next to the newly created certificate `wascert_rsa`.

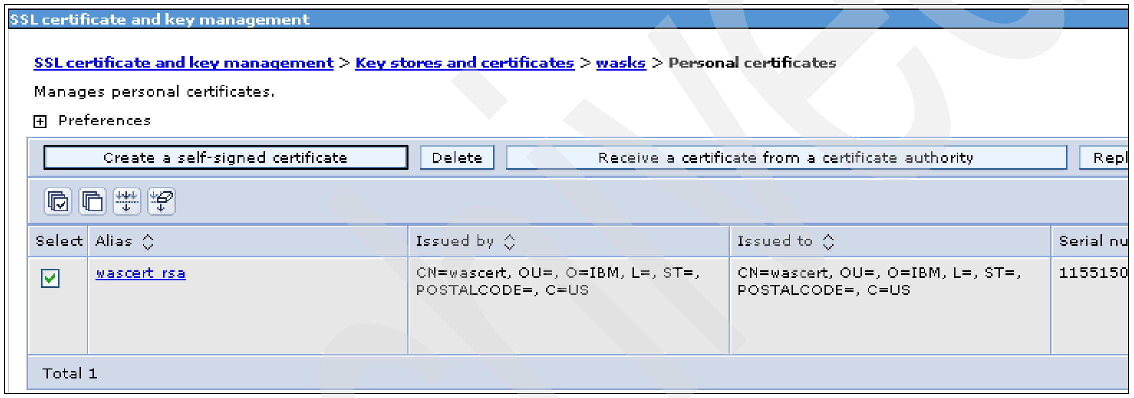
c. Click **Extract**.

d. For Certificate file name, specify `c:\SG247206\keystore\wascert.der`

e. For Data Type select **Binary DER data**.

f. Click **OK**.

Figure 10-17 shows the certificate being extracted using the admin console.


SSL certificate and key management
SSL certificate and key management > Key stores and certificates > wasks > Personal certificates
Manages personal certificates.
Preferences
Create a self-signed certificate
Delete
Receive a certificate from a certificate authority
Repl

Select	Alias	Issued by	Issued to	Serial nu
✓	wascert_rsa	CN=wascert, OU=, O=IBM, L=, ST=, POSTALCODE=, C=US	CN=wascert, OU=, O=IBM, L=, ST=, POSTALCODE=, C=US	1155150

Total 1

Figure 10-17 Extract client X.509 certificate from key store

2. FTP the WebSphere certificate to the host system.

   Before performing the FTP, we first allocated a sequential data set on the host machine SCTS66 to receive the certificate. This data set is allocated with a record format VB, record length 84 and block size 27998.

   We then used FTP as shown in Example 10-16 to send the exported certificate (in binary format) to our host machine.

Example 10-16 FTP client X.509 certificate to host

```
C:\Documents and Settings\Resident>ftp wtsc66.itso.ibm.com
Connected to wtsc66.itso.ibm.com.
220-FTPMVS1 IBM FTP CS V1R7 at wtsc66.itso.ibm.com, 21:04:02 on 2006-08-11.
220 Connection will close if idle for more than 5 minutes.
User (wtsc66.itso.ibm.com:(none)): NIGELW
331 Send password please.
Password:
```

```
230 NIGELW is logged on.  Working directory is "NIGELW.".
ftp> lcd C:\SG247206\keystore
Local directory now C:\SG247206\keystore.
ftp> cd 'CIWS'
250 "CIWS." is the working directory name prefix.
ftp> bin
200 Representation type is Image
ftp> put wascert.der
200 Port request OK.
125 Storing data set CIWS.WASCERT.DER
250 Transfer completed successfully.
ftp: 580 bytes sent in 0.00Seconds 580000.00Kbytes/sec.
ftp> bye
221 Quit command received.
Goodbye.
```

3. Import the client certificate into RACF and connect it to the CICS key ring.

   We used the RACF commands shown in Example 10-17.

Example 10-17 RACF commands to import WebSphere certificate into RACF

```
RACDCERT ID(WEBAS1) ADD('CIWS.WASCERT.DER') WITHLABEL('WASCERT') TRUST
RACDCERT ID(CIWS3D) CONNECT(ID(WEBAS1) LABEL('WASCERT') RING(Ciws.Ciwss3c1))
```

   – The RACDCERT ADD command adds the client certificate with label 'WASCERT' as a trusted certificate.
   – The RACDCERT CONNECT command connects the client certificate to the CICS key ring. ID(WEBAS1) indicates that the certificate being added to the key ring is a user certificate, and WEBAS1 is the user ID that is associated with this certificate.

   The certificate can be listed using this command:

   ```
   RACDCERT ID(WEBAS1) LIST
   ```

   Example 10-18 shows the output of the RACDCERT LIST command.

Example 10-18 Listing of client X.509 certificate (WASCERT)

```
Digital certificate information for user WEBAS1:

  Label: WASCERT
  Certificate ID: 2QbmxcLB4vHmweLDxdnj
  Status: TRUST
  Start Date: 2006/08/09 15:16:07
  End Date:   2007/08/09 15:16:07
  Serial Number:
       >44DA3477<
  Issuer's Name:
```

```
     >CN=wascert.OU=.O=IBM.L=.SP=..C=US<
Subject's Name:
     >CN=wascert.OU=.O=IBM.L=.SP=..C=US<
Private Key Type: None
Ring Associations:
  Ring Owner: CIWS3D
  Ring:
     >Ciws.Ciwss3c1<
```

10.5 Signing a SOAP message

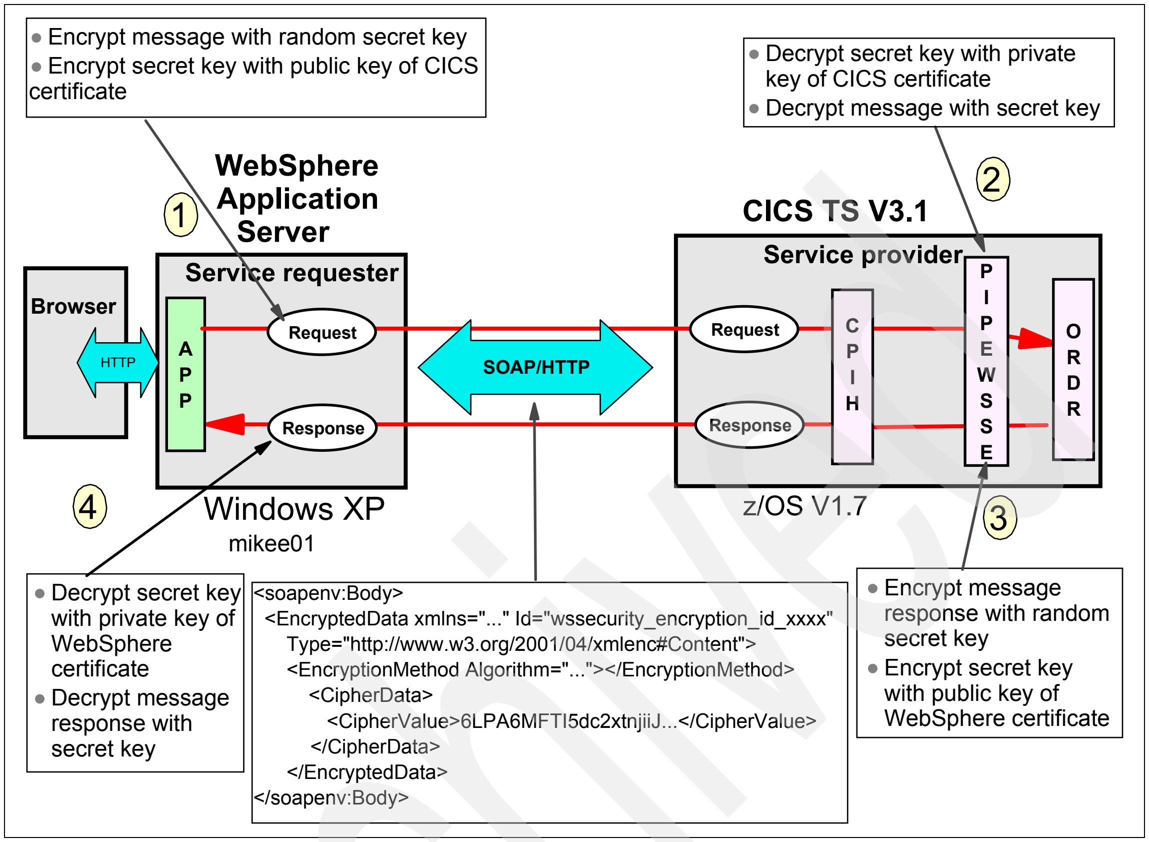
To provide integrity on the client's request message, we can add an XML digital signature to the message. CICS supports digital signatures in both service provider and service requester modes.

In this section we show how a SOAP request message can be signed by WebSphere Application Server and how the signature can be verified by CICS, and how the response message can be signed by CICS and verified by WebSphere.

- The SOAP request message contains an X.509 certificate (the WebSphere certificate) that is used to sign the body of the message.
- The SOAP response message also contains an X.509 certificate (the CICS certificate) that is used to sign the body of the message.

The X.509 certificates are transported within a *BinarySecurityToken* element.

We also show how the WebSphere certificate can be used to authenticate the service requester. DFHWSSE1 verifies the signature and uses the certificate to determine the user ID under which the ORDR transaction will be run. This scenario is shown in Figure 10-18.



• Encrypt message with random secret key
• Encrypt secret key with public key of CICS certificate
• Decrypt secret key with private key of CICS certificate
• Decrypt message with secret key
WebSphere Application Server
1
2
CICS TS V3.1
Browser
HTTP
Service requester
APP
Request
Response
SOAP/HTTP
Service provider
Request
Response
CPIH
PIPEWSSE
ORDR
4
Windows XP
mikee01
z/OS V1.7
3
• Decrypt secret key with private key of WebSphere certificate
• Decrypt message response with secret key
• Encrypt message response with random secret key
• Encrypt secret key with public key of WebSphere certificate

```
<soapenv:Body>
 <EncryptedData xmlns="..." Id="wssecurity_encryption_id_xxxx"
    Type="http://www.w3.org/2001/04/xmlenc#Content">
    <EncryptionMethod Algorithm="..."></EncryptionMethod>
       <CipherData>
          <CipherValue>6LPA6MFTI5dc2xtnjiiJ...</CipherValue>
       </CipherData>
    </EncryptedData>
</soapenv:Body>
```

Figure 10-18 Enabling CICS WS-Security support for signatures

1. The service requester (WebSphere in our example) generates a BinarySecurityToken element from the WebSphere X.509 certificate. It then signs the message with its private key so the service provider (CICS in our example) knows that the message can only be sent by the WebSphere application server.
2. CICS validates the signature with the WebSphere public key. It also uses the owning user ID of the WebSphere X.509 certificate (WEBAS1) to run the ORDR transaction for the placeOrder service.
3. CICS then signs the response message with its private key so that WebSphere knows that the message can only be sent by the CICS region.
4. WebSphere validates the signature with the public key of CICS.

Important: There is a significant performance overhead when using WS-Security and XML digital signatures. See 8.5, "Comparison of transport level and SOAP message security" on page 269 for recommendations on choosing between WS-Security and SSL when implementing an integrity security solution.

We document the following procedures to enable support for XML signature processing:

- Configuring the service requester for signature processing
- Configuring CICS for signature processing
- Testing the signature test scenario

Note: In our signature test scenario we sign both the request and response messages. It is also possible to sign only the request, or to sign only the response.

10.5.1 Configuring the service requester for signature processing

In this section we show how to configure the client J2EE application to send and receive signed SOAP messages. At a high level, the steps to do this are:

- Import the CatalogSec2 WebSphere Application to the Application Server Toolkit (AST).
- Configure the request generator for signing.
- Configure the response consumer for signing.
- Re-deploy the Web Service client application.

In the following sub-sections we provide step-by-step details for each procedure.

Importing the base application

To configure our Web Service client to use WS-Security, we used the IBM WebSphere Application Server Toolkit V6.1 (AST).

See "Importing the base application" on page 333 for instructions on how to import the application. For the signature test scenario, we specified:

- The name of the workspace folder as `C:\AST\CatalogSec2\CatalogSec2Signing`
- The name of the EAR project as `CatalogSec2Sign`.

Configuring the request generator for signing

Perform the following steps to configure the request generator for signing:

1. Expand the **Dynamic Web Project** (**CatalogSec2Web**) in the Project Explorer and open (double-click) the deployment descriptor as shown in Figure 10-19.

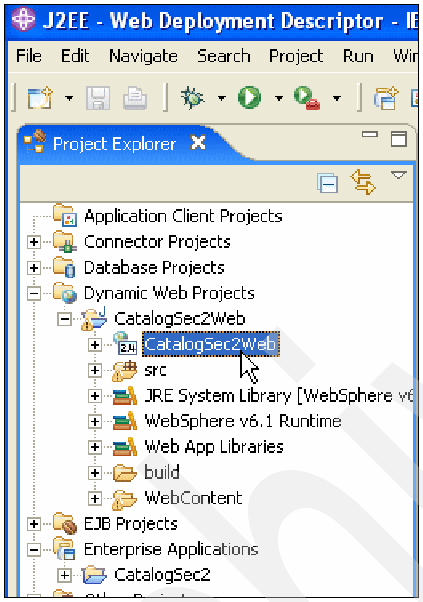


Figure 10-19 Opening the CatalogSec2Web Deployment Descriptor

2. Go to the WS Extension page.

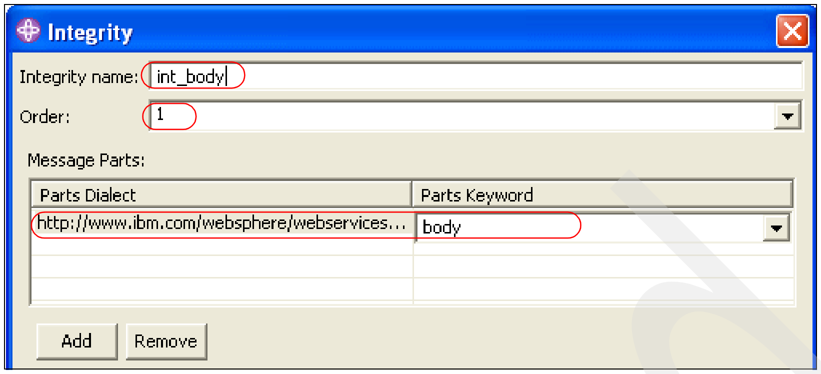
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).

   b. Expand the **Request Generator Configuration**.

   c. Expand **Integrity**, click **Add**, and in the Integrity dialog box (Figure 10-20) enter the following data:

      - Enter a name identifying the part, for example, `int_body`.
      - Select the order in which the signature is generated. Multiple integrity parts can be specified. In our case, we selected 1.
      - Click **Add** for the Message Parts, and one integrity part is created. The default created part is the SOAP body. We accepted the default.

   d. Click **OK** and save the configuration by pressing Ctrl+s.



Integrity name: int_body

Order: 1

Message Parts:

Parts Dialect	Parts Keyword
http://www.ibm.com/websphere/webservices...	body

Figure 10-20 Integrity dialog in the WS Extensions page

3. Open the WS Binding page.
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).
   b. Expand the **Security Request Generator Binding Configuration**.
   c. Expand **Token Generator** and click **Add.** In the Token Generator dialog box (Figure 10-21) enter the following data:
      - Enter a Token generator name, for example, `gen_dsigtgen`.
      - For the Token generator class, select the **X509TokenGenerator**.
      - Do not select a Security Token.
      - Select **Use value type**, and then select **X509 certificate token v3** and the X509CallbackHandler.
      - Select **Use key store** and make the appropriate entries. Ours were:

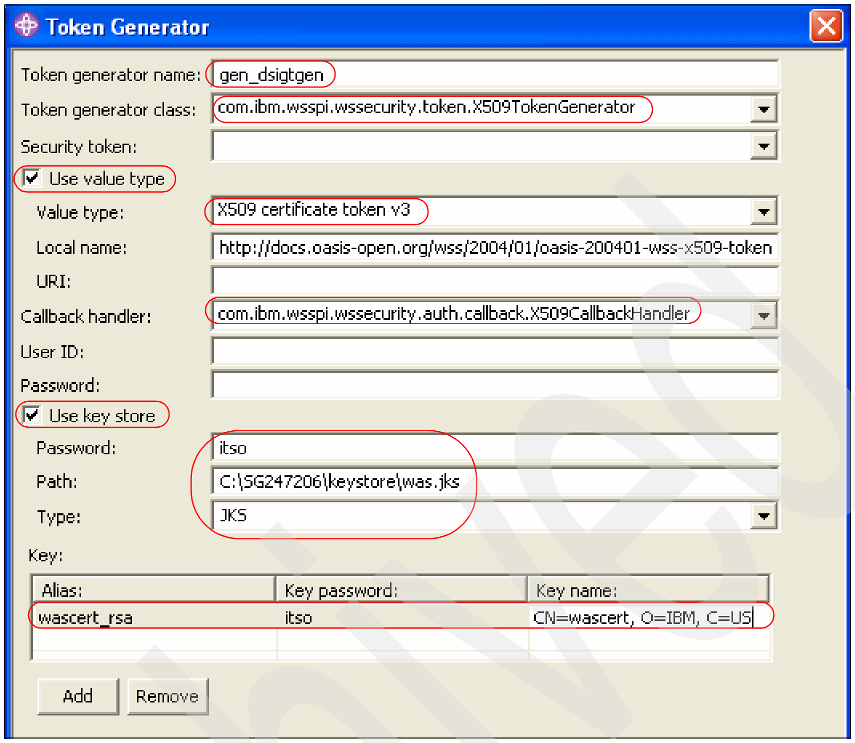
Password:	`itso`
Path:	`C:\SG247206\keystore\was.jks`
Type:	`JKS`

      - Click **Add** under Key and make the appropriate entries. Ours were:

Alias:	`wascert_rsa`
Key password:	`itso`
Key name:	`CN=wascert, O=IBM, C=US`

Note: This is the key name of the WebSphere certificate whose private key is used to sign the request message.

      - Click **OK**.



Token generator name:	gen_dsigtgen
Token generator class:	com.ibm.wsspi.wssecurity.token.X509TokenGenerator
Security token:	
☑ Use value type	
Value type:	X509 certificate token v3
Local name:	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token
URI:	
Callback handler:	com.ibm.wsspi.wssecurity.auth.callback.X509CallbackHandler
User ID:	
Password:	
☑ Use key store	
Password:	itso
Path:	C:\SG247206\keystore\was.jks
Type:	JKS

Key:

Alias:	Key password:	Key name:
wascert_rsa	itso	CN=wascert, O=IBM, C=US

Figure 10-21 Token Generator dialog in the WS Binding page

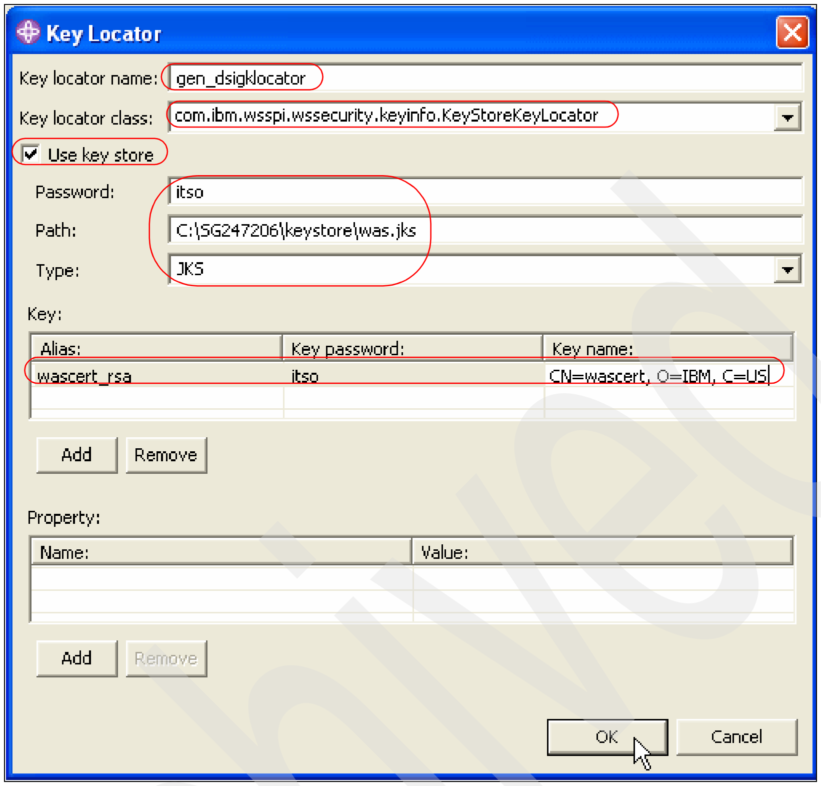
4. Expand **Key Locators**, click **Add**, and in the Key Locator dialog box (Figure 10-22) enter the following data:
   a. Enter a Key locator name, for example, `gen_dsigklocator`.
   b. Select **KeyStoreKeyLocator** as the Key locator class.
   c. Select **Use key store** and make the appropriate entries. Ours were:

Password:	`itso`
Path:	`C:\SG247206\keystore\was.jks`
Type:	`JKS`

   d. Click **Add** under Key and make the appropriate entries. Ours were:

Alias:	`wascert_rsa`
Key password:	`itso`
Key name:	`CN=wascert, O=IBM, C=US`

   e. Click **OK**.



Key locator name: gen_dsigklocator

Key locator class: com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator

[✓] Use key store

Password: itso

Path: C:\SG247206\keystore\was.jks

Type: JKS

Key:

Alias:	Key password:	Key name:
wascert_rsa	itso	CN=wascert, O=IBM, C=US

Figure 10-22 The Key Locator dialog in the WS Binding page

5. Expand **Key Information**, click **Add**, and in the Key Information dialog box (Figure 10-23) enter the following data:
   a. Enter a Key information name, for example, `gen_dsigkeyinfo`.
   b. Select **STRREF** (Direct reference) as the Key information type. The Key information class is filled automatically.

      The possible Key information types are:

• STRREF	Direct reference
• EMB	Embedded reference
• KEYID	Key identifier reference
• KEYNAME	Key name reference
• X509ISSUER	x.509 issuer and serial number reference

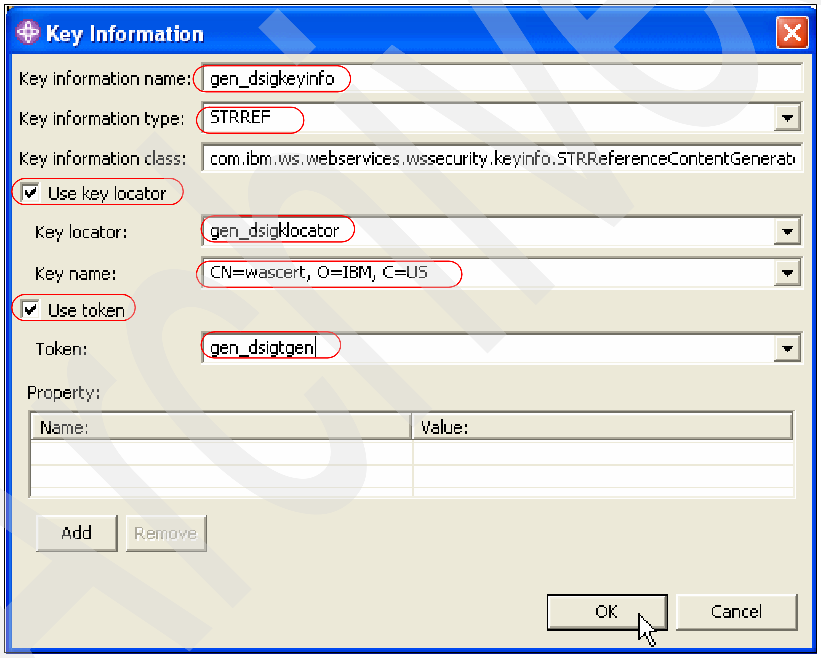
Note: We specify STRREF because we want WebSphere to send the public key as part of the complete certificate, since we will use the certificate for authentication in addition to decrypting the signed message.

It is also possible to configure an indirect reference to the public key, such as the key identifier or the key name. In this case, CICS uses the key reference to identify the RACF certificate that contains the public key.

c. Select **Use key locator** and select **gen_dsigklocator** from the Key locator drop-down list. Select the predefined Key name **CN=wascert, O=IBM, C=US**.

d. Select **Use token**, and select **gen_dsigtgen** from the drop-down list.

e. Click **OK**.



Key information name: gen_dsigkeyinfo
Key information type: STRREF
Key information class: com.ibm.ws.webservices.wssecurity.keyinfo.STRReferenceContentGenerat
☑ Use key locator
Key locator: gen_dsigklocator
Key name: CN=wascert, O=IBM, C=US
☑ Use token
Token: gen_dsigtgen
Property:

Name:	Value:

Figure 10-23 The Key Information dialog in the WS Binding page

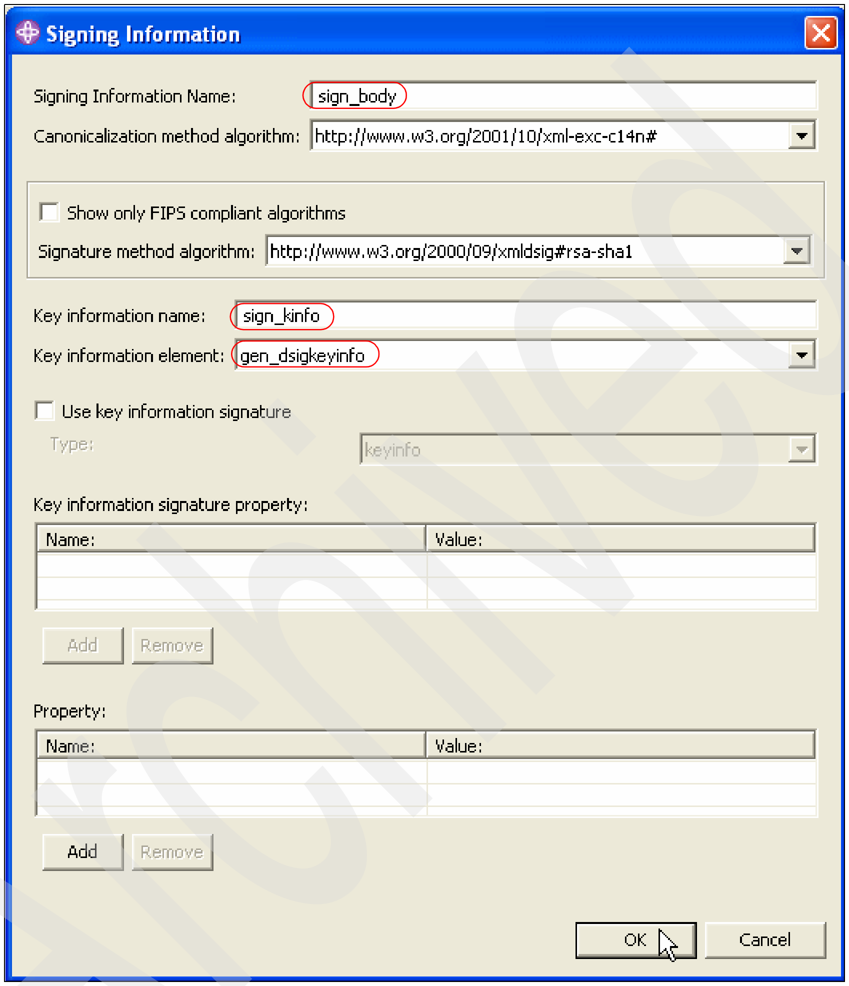
6. Expand **Signing Information**, click **Add**, and in the Signing Information dialog box (Figure 10-24) enter the following data:

a. Enter a Signing Information Name, for example, `sign_body`.

b. Enter a Key information name, for example, `sign_kinfo`.

c. Select the Key information element previously defined. In our case this was **gen_dsigkeyinfo**.

d. Click **OK**.


Signing Information

Signing Information Name: sign_body

Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

☐ Show only FIPS compliant algorithms

Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1

Key information name: sign_kinfo

Key information element: gen_dsigkeyinfo

☐ Use key information signature

Type: keyinfo

Key information signature property:

Name:	Value:

Add Remove

Property:

Name:	Value:

Add Remove

OK Cancel

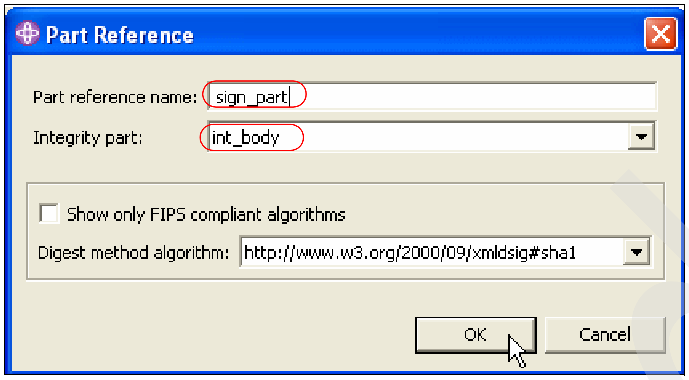
Figure 10-24 The Signing Information dialog in the WS Binding page

7. Expand **Part References**, click **Add**, and in the Part Reference dialog box (Figure 10-25) enter the following data:

   a. Enter a Part reference name, for example, `sign_part`.

   b. Select the Integrity part from the list of parts defined on the WS Extensions page. In our case, we select **int_body**.

   c. Click **OK**.


Part Reference

Part reference name: sign_part

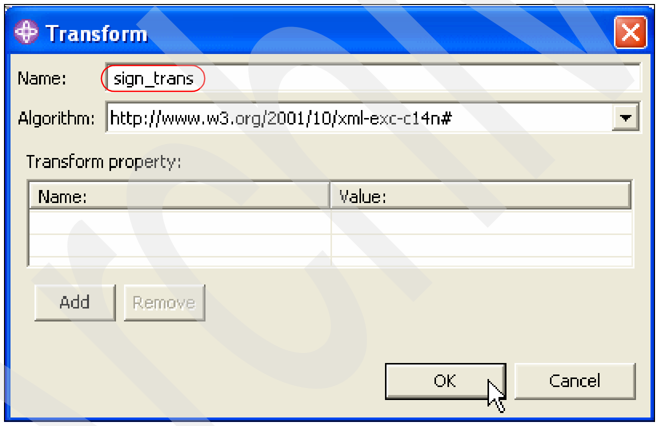
Integrity part: int_body

Show only FIPS compliant algorithms

Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1

Figure 10-25 The Part Reference dialog in the WS Binding page

8. Expand **Transforms**, click **Add**, and in the Transform dialog box (Figure 10-26) enter the following data:
   a. Enter a name, for example, `sign_trans`.
   b. Click **OK**.


Transform

Name: sign_trans

Algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Transform property:

Name:	Value:

Figure 10-26 The Transform dialog in the WS Binding page

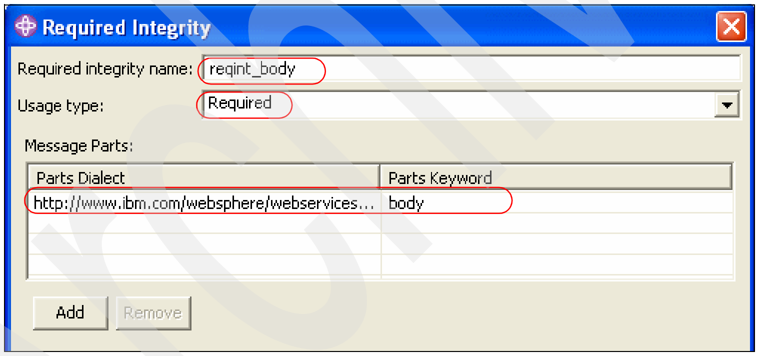
9. Save the configuration by pressing Ctrl+s.

Configure the response consumer for signing

In this section we show how to configure the client J2EE application in order to receive a signed SOAP message.

Use the following steps to do this:

1. Expand the **Dynamic Web Project** (**CatalogSec2Web**) in the Project Explorer and open (double-click) the deployment descriptor.
2. Go to the WS Extension page.
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).
   b. Expand the **Response Consumer Configuration**.
   c. Expand **Required Integrity** and click **Add**. In the Required Integrity dialog box (Figure 10-27) enter the following data:
      - Enter a Required integrity name, for example, `reqint_body`.
      - Select the Usage type, either Required or Optional. If the usage type is Required, an unsigned response message throws a SOAP fault. If the usage type is Optional, an unsigned message is received. We selected **Required**.
      - Click **Add** for Message Parts, and one message part is created. The default created part is the SOAP body. We accepted the default.
   d. Click **OK** and save the configuration by pressing Ctrl+s.



Required integrity name: reqint_body

Usage type: Required

Message Parts:

Parts Dialect	Parts Keyword
http://www.ibm.com/websphere/webservices...	body

Figure 10-27 Required integrity in the WS Extension page for response consumer

3. Open the WS Binding page.
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).
   b. Expand the **Security Response Consumer Binding Configuration**.
   c. Expand **Token Consumer**, click **Add**, and in the Token Consumer dialog box (Figure 10-28) enter the following data:
      - Enter a Token consumer name, for example, `con_dsigtcon`.

- Select **com.ibm.wsspi.wssecurity.token.X509TokenConsumer** as the Token consumer class.
- Select **Use value type**.
- Select **X509 certificate token**. The Local name is selected automatically.

Note: It is important to specify X509 certificate token for the token value type. An error occurred in our test when we specified X509 certificate token v3.

- Select **Use jaas.config** and enter `system.wssecurity.X509BST` as jaas.config name.
- Select **Use certificate path settings**.
- Select **Trust any certificate**.

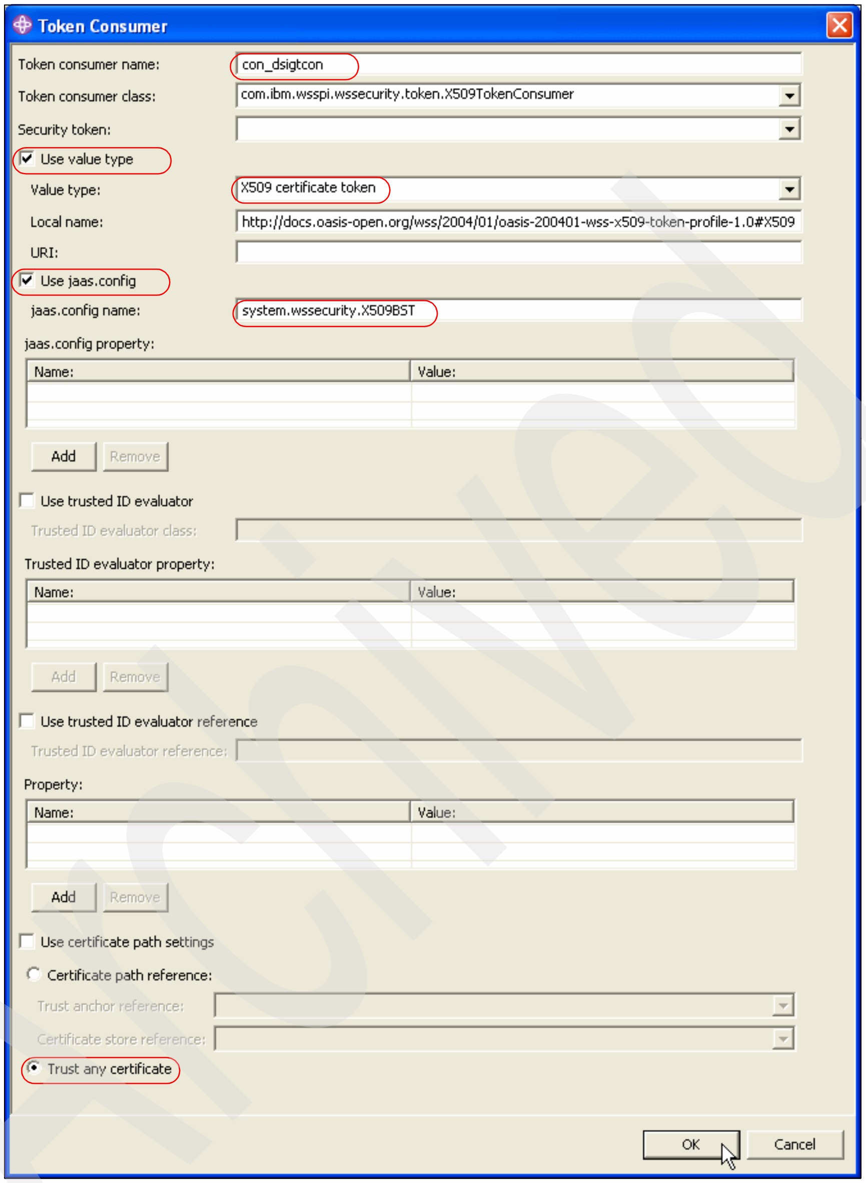
Important: We specified **Trust any certificate** for our test. By selecting this option, the signature is validated by the certificate sent with the message without the certificate itself being validated.

However, in a production environment you are unlikely to use this option because it allows any client with a valid XML digital signature certificate to have access to your WebSphere application server.

WebSphere provides different ways of specifying trust for digital signature certificates.

- If you use a self-signed certificate, you can specify the issuer name or serial number of trusted certificates in a jaas.config property.
- For both self-signed and CA-signed certificates, you can specify a *trust anchor* and a *trusted certificate store* using a Certificate path reference.

d. Click **OK**.


Token Consumer

Token consumer name: con_dsigtcon
Token consumer class: com.ibm.wsspi.wssecurity.token.X509TokenConsumer
Security token:
[✓] Use value type
Value type: X509 certificate token
Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509
URI:
[✓] Use jaas.config
jaas.config name: system.wssecurity.X509BST
jaas.config property:

Name:	Value:

[ ] Use trusted ID evaluator
Trusted ID evaluator class:
Trusted ID evaluator property:

Name:	Value:

[ ] Use trusted ID evaluator reference
Trusted ID evaluator reference:
Property:

Name:	Value:

[ ] Use certificate path settings
○ Certificate path reference:
Trust anchor reference:
Certificate store reference:
● Trust any certificate

Figure 10-28 Token Consumer dialog in the WS Binding page for Response Consumer

4. Expand **Key Locators**, click **Add**, and in the Key Locator dialog box (Figure 10-29) enter the following data:
   a. Enter a Key locator name, for example, `con_dsigklocator`.
   b. Select **KeyStoreKeyLocator** as the Key locator class.

c. Select Use key store and make the appropriate entries. Ours were:

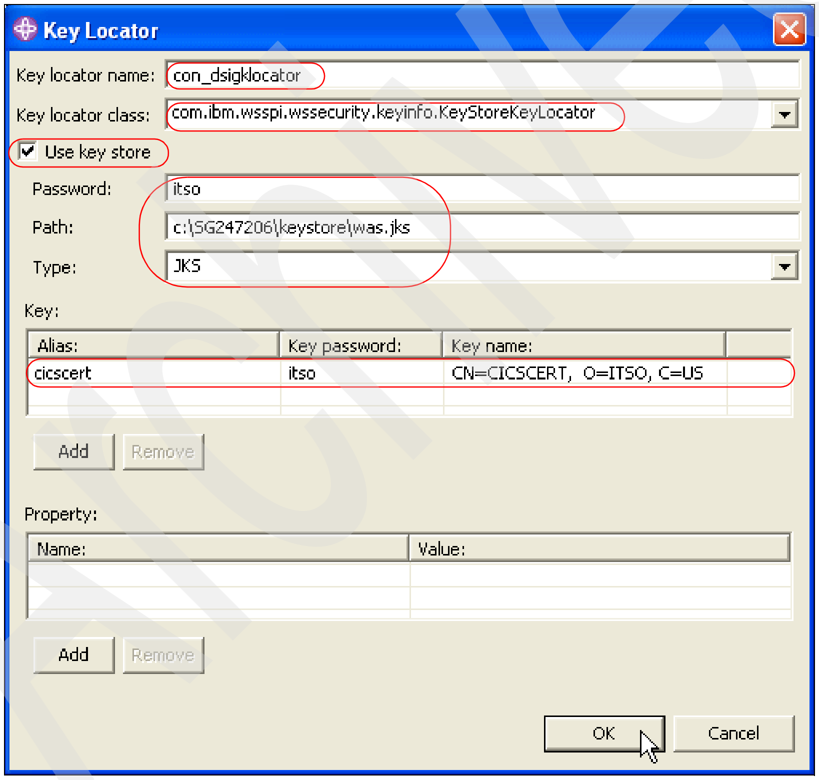
Password:	`itso`
Path:	`C:\SG247206\keystore\was.jks`
Type:	`JKS`

d. Click **Add** under Key and make the appropriate entries. Ours were:

Alias:	`cicscert`
Key password:	`itso`
Key name:	`CN=CICSCERT, O=ITSO, C=US`

Note: This is the key name of the CICS certificate. The private key of this certificate is used to sign the response message.

e. Click **OK**.



Key Locator

Key locator name: con_dsigklocator

Key locator class: com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator

☑ Use key store

Password: itso

Path: c:\SG247206\keystore\was.jks

Type: JKS

Key:

Alias:	Key password:	Key name:
cicscert	itso	CN=CICSCERT, O=ITSO, C=US

Figure 10-29 Key Locator dialog in the WS Binding page for response consumer

5. Expand **Key Information**, click **Add**, and in the Key Information dialog box (Figure 10-30) enter the following data:

   a. Enter a Key information name, for example, `con_dsigkeyinfo`.

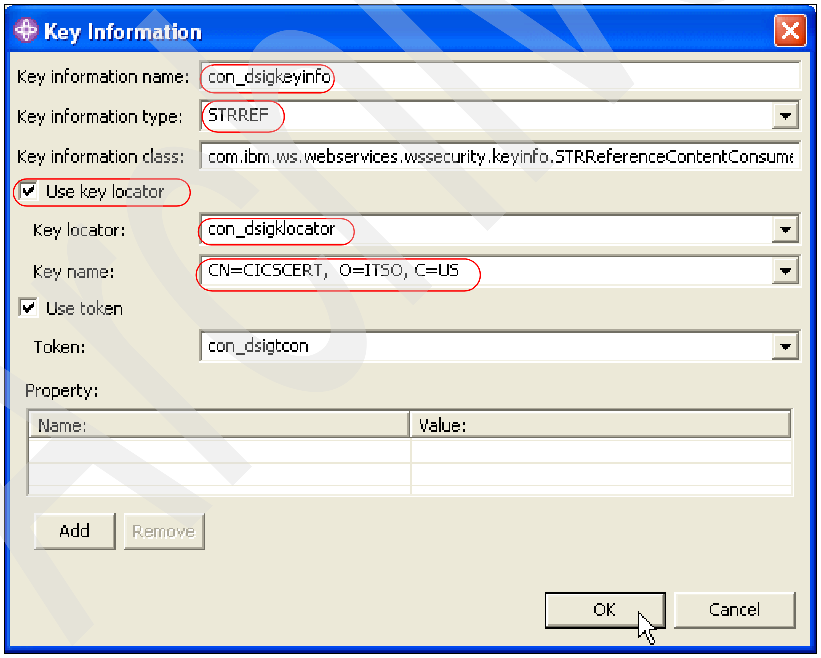
   b. Select **STRREF** as the Key information type. The Key information class is filled automatically.

   > **Note:** We specify STRREF because CICS always sends the complete certificate when signing an outbound message; therefore, the WebSphere response consumer must be configured to expect the CICS certificate in the security header of the response message.

   c. Select **Use key locator** and select **con_dsigklocator** from the Key locator drop-down list. Select the predefined Key name as **CN=CICSCERT, O=ITSO, C=US**.

   d. Select **Use token** and select **con_dsigtcon** from the Token drop-down list.

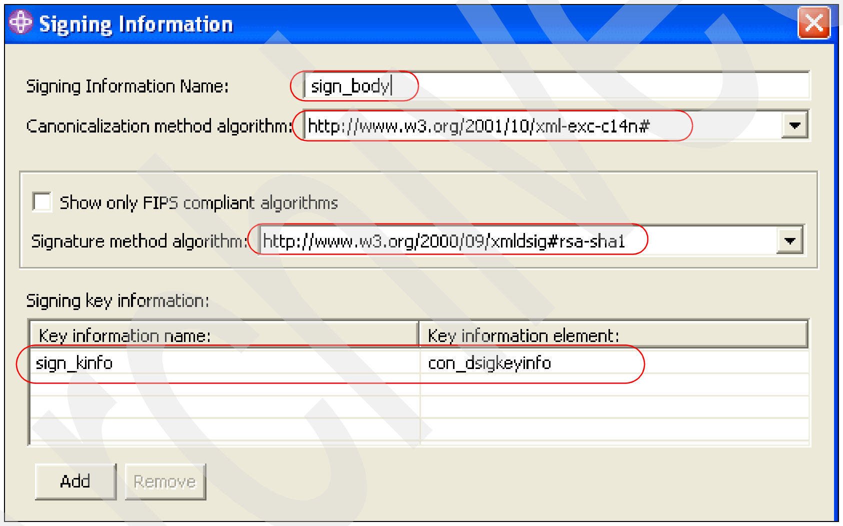
   e. Click **OK**.



Key information name: con_dsigkeyinfo
Key information type: STRREF
Key information class: com.ibm.ws.webservices.wssecurity.keyinfo.STRReferenceContentConsume
[✓] Use key locator
Key locator: con_dsigklocator
Key name: CN=CICSCERT, O=ITSO, C=US
[✓] Use token
Token: con_dsigtcon

Figure 10-30 Key Information dialog in the WS Binding page for response consumer

6. Expand **Signing Information**, click **Add**, and in the Signing Information dialog box (Figure 10-31) enter the following data:
   a. Enter a Signing Information Name, for example, `sign_body`.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as the Canonicalization method algorithm.
   c. Select **http://www.w3.org/2000/09/xmldsig#rsa-sha1** as the Signature method algorithm.
   d. Click **Add** in Signing key information.
      - Enter a Key information name, for example, `sign_kinfo`.
      - Select the **con_dsigkeyinfo** from the Key information element drop-down list.
   e. Click **OK**.



Signing Information Name: sign_body

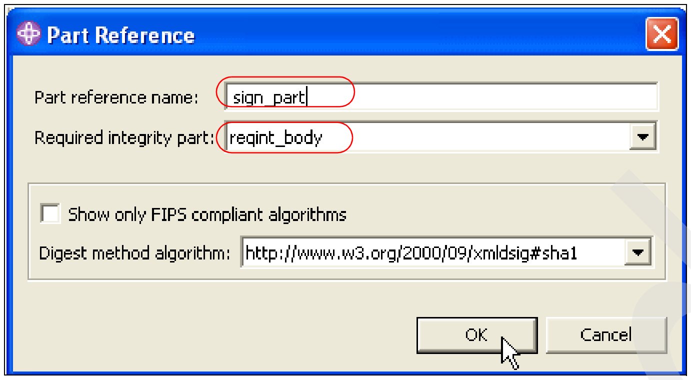
Canonicalization method algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Signature method algorithm: http://www.w3.org/2000/09/xmldsig#rsa-sha1

Key information name:	Key information element:
sign_kinfo	con_dsigkeyinfo

Figure 10-31 Signing Information dialog in the WS Binding page for response consumer

7. Expand **Part References**, click **Add**, and in the Part Reference dialog box (Figure 10-32) enter the following data:
   a. Enter a Part reference Name, for example, `sign_part`.
   b. Select **reqint_body** as the Required integrity.
   c. Click **OK**.


Part Reference

Part reference name: sign_part

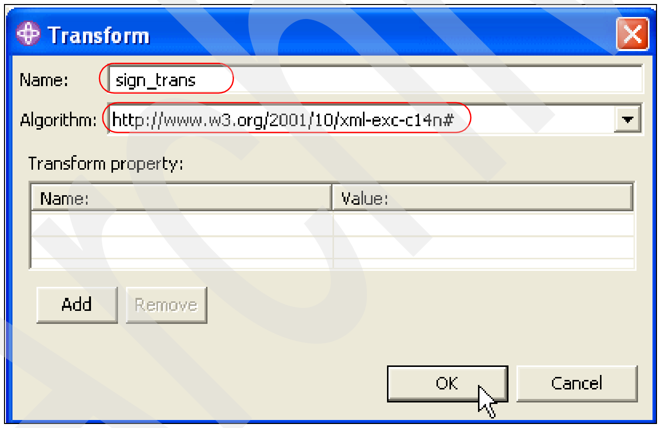
Required integrity part: reqint_body

Show only FIPS compliant algorithms

Digest method algorithm: http://www.w3.org/2000/09/xmldsig#sha1

Figure 10-32 The Part Reference dialog in the WS Binding page for response consumer

8. Expand **Transforms**, click **Add**, and in the Transform dialog box (Figure 10-33) enter the following data:
   a. Enter a Name, for example, `sign_trans`.
   b. Select **http://www.w3.org/2001/10/xml-exc-c14n#** as the algorithm.
   c. Click **OK**.


Transform

Name: sign_trans

Algorithm: http://www.w3.org/2001/10/xml-exc-c14n#

Transform property:

Name:	Value:

Figure 10-33 The Transform dialog in the WS Binding page for response consumer

9. Save the configuration by pressing Ctrl+s.

Redeploying the Web service application

After configuring a signature security constraint for the service requester application, we exported a new EAR file called CatalogSec2_WS-Security_Signature.ear. We followed the same process that is described in “Installing the service requester” on page 87 to deploy the CatalogSec2_WS-Security_Signature.ear.

For further information about configuring WS-Security in WebSphere refer to *Web Services Handbook for WebSphere Application Server 6.1,* SG24-7257.

10.5.2 Configuring CICS for signature processing

In this section we show how to configure the CICS pipeline to receive and send signed SOAP messages, and to run the placeOrder request under the user ID associated with the WebSphere certificate contained in the request message.

Example 10-19 shows the pipeline configuration file that we used for the signature test scenario. It includes the message handler DFHWSSE1, and the configuration information for the handler.

Example 10-19 Pipeline config file, ITSO_7206_wssec_signature_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
    <transport>
      <default_transport_handler_list>
          <handler>
             <program>CIWSMSGH</program>
             <handler_parameter_list/>
          </handler>
      </default_transport_handler_list>
    </transport>
    <service>
      <service_handler_list>
        <handler>
          <program>SNIFFER</program>
          <handler_parameter_list/>
        </handler>
        <wsse_handler>
          <dfhwsse_configuration version="1">
            <authentication mode="signature">
            <algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>
            </authentication>
            <expect_signed_body/>
            <sign_body>
              <algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>
              <certificate_label>CICSCERT</certificate_label>
            </sign_body>
          </dfhwsse_configuration>
        </wsse_handler>
      </service_handler_list>
      </terminal_handler>
        <cics_soap_1.2_handler/>
      </terminal_handler>
    </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

The `<wsse_handler>` element shown in Example 10-19 contains a `<dfhwsse_configuration>` element that specifies configuration information for DFHWSSE1.

Important: If the <wsse-handler> element is included in a provider pipeline configuration, CICS processes signed SOAP messages by default.

- The `<authentication mode="signature">` element specifies that inbound messages must contain an X.509 certificate in a BinarySecurityToken.

  If `<authentication mode="signature">` is specified then the client X.509 certificate must be imported to RACF and attached to the CICS key ring since CICS runs the service request under the user ID associated with the client certificate.

  Note: CICS does not support certificate name filtering for signed SOAP messages.

  If `<authentication mode="signature">` is not specified, and if the complete X.509 certificate is included in the request message, then the client X.509 certificate does not need to be imported to RACF since the public key required to validate the signature is extracted from the X.509 certificate contained in the SOAP message.

- `<algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>` is a child element of the *<authentication>* element that specifies the URI of the signature algorithm.

  Note: We found that it is necessary to specify a valid signature algorithm in service provider mode even if the algorithm used in the decryption of the signature is the one specified in the SOAP message rather than in the pipeline configuration file.

- The `<expect_signed_body/>` element indicates that the *<body>* of the inbound message must be signed. If the body of an inbound message is not correctly signed, CICS rejects the message with a security fault.

  We specify `<expect_signed_body/>` because we want to prevent the placeOrder service being accessed unless the body of the SOAP message is signed.
- The `<sign_body/>` element indicates that the *<body>* of the outbound message must be signed, and provides information about how the message is to be signed.
- `<algorithm>http://www.w3.org/2000/09/xmldsig#rsa-sha1</algorithm>` is a child element of the *<sign-body>* element which specifies the URI of the signature algorithm to be used for signing the response message. This is the only signature algorithm supported by CICS for outbound messages.

  > **Attention:** During our tests we encountered codepage issues when specifying the *<algorithm>* element. The configuration file should be coded in an English encoding such as EBCDIC-CP-US. When using a terminal emulation session with a non-English codepage, we noticed that the symbol ‘#’ was a different hex value in the non-English codepage than in the EBCDIC-CP-US codepage (in which the symbol ‘#’ is x'7B'). This resulted in error DFHPI0723 with the message:
  >
  > `'The value for the algorithm specified for the authentication is not supported'.`
  >
  > We recommend that you use a terminal emulation session with an English codepage when editing the pipeline configuration file.

- `<certificate_label>CICSCERT</certificate_label>` is a child element of the `<sign-body>` element that specifies the label associated with the CICS certificate installed in RACF. This certificate contains the PKDS key label of the private key that is used to sign the message response. The public key associated with the private key is then sent in the SOAP response message, allowing the signature to be validated by WebSphere.

We completed the CICS configuration by changing the pipeline PIPEWSSE (used for the placeOrder service) to specify the pipeline configuration file that we have created for signature processing. We used the following CEDA command:

```
CEDA ALTER PIPELINE(PIPEWSSE) GROUP(S3C1EXWS)
Configfile(/CIWS/S3C1/config/ITSO_7206_wssec_signature_provider.xml)
```

We then re-installed the changed pipeline resource definition.

10.5.3 Testing the signature scenario

An example of the signed request message sent by WebSphere is shown in Example 10-20.

Example 10-20 Signed SOAP request message

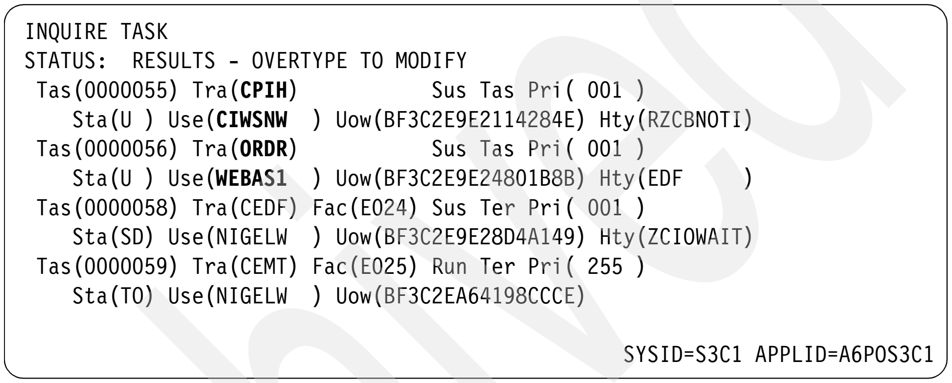
```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssec
      urity-secext-1.0.xsd">
      <wsse:BinarySecurityToken
        EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-s
        oap-message-security-1.0#Base64Binary"
        ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509
        -token-profile-1.0#X509v3" wsu:Id="x509bst_1"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
        curity-utility-1.0.xsd">MIICQDCCAamgAwIBAgIERNoOdzANBgkqhkiG9w0BAQUFADB
        ZMQswCQYDVQQGEwJVUzEJMAcGA1UEERMAMQkwBwYDVQQIEwAxCTAHBgNVBAcTADEMMAoGA1
        UEChMDSUJNMQkwBwYDVQQLEwAxEDAOBgNVBAMTB3dhc2NlcnQwHhcNMDYwODA5MTkxNjA3W
        hcNMDcwODA5MTkxNjA3WjBZMQswCQYDVQQGEwJVUzEJMAcGA1UEERMAMQkwBwYDVQQIEwAx
        CTAHBgNVBAcTADEMMAoGA1UEChMDSUJNMQkwBwYDVQQLEwAxEDAOBgNVBAMTB3dhc2NlcnQ
        wgZ8wDQYJKoZIhvcNAQEBBQADgYOAMIGJAoGBAIEMtFDoacSSCWTqNYVPDd3yMGMIq4GpSN
        UVajzLdW+hlNIgaKzZfhiOo6BJxQcD+Ty+pKRSlbPLyG9B+9LGo+O6dNJmAXDVGNQiSgkNY
        xl12oWRBCJciU5nBIB3a+TUOe2wYEak+rJ3MblB/TjA3ottykjftOyjRohl97wT65j/AgMB
        AAGjFTATMBEGA1UdDgQKBAhJYPbuSs76STANBgkqhkiG9w0BAQUFAAOBgQBP1SEGGNW4ruu
        a8OhItdXERBA356OnzLwO5n+eb2JdZuOilyYHNGfp8+k4b+9F9DjOPzGEJzgspqMTraHKOJ
        P1co7yIG/RUuX+gicGN+dI2A8frLsIS2IUMB66FqQR/uAOYjGJCuYfzKm2CcOPUFTQYqMMk
        nuj+DBTcAoDp+GP4Q==
      </wsse:BinarySecurityToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethodAlgorithm="http://www.w3.org/2001/10/xml-ex
            c-c14n#">
            <ec:InclusiveNamespaces PrefixList="xsi xsd soapenv soapenc wsse
              ds" xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
          </ds:CanonicalizationMethod>
          <ds:SignatureMethod Algorithm=
            "http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
          <ds:Reference URI="#wssecurity_signature_id_0">
            <ds:Transforms>
              <ds:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                <ec:InclusiveNamespaces PrefixList="xsi xsd soapenv soapenc wsu
                  p635" xmlns:ec="http://www.w3.org/2001/10/xml-exc-c14n#" />
```

```
       </ds:Transform>
      </ds:Transforms>
      <ds:DigestMethod Algorithm=
       "http://www.w3.org/2000/09/xmldsig#sha1" />
      <ds:DigestValue>fh93yxjxoa8CwZ9ROqH2mleWdYO=</ds:DigestValue>
     </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue>I53NN8zcbfRmnz3LwVf3vZeFFDpp2WAkgBU9+Sfu/I71G3Wh8tsH
     3fRngwt7qLmRH3qGzjpsMWCbxKssyOP+drSyBwvUIz4FukxyJ5pASVot7qPNRfj9pGO/Y
     OTu26zOW9GQScljFqRpl6+tNZqHwLiMfpmQyJggIF8izQU8nkQ=
    </ds:SignatureValue>
    <ds:KeyInfo>
     <wsse:SecurityTokenReference>
      <wsse:Reference URI="#x509bst_1"
       ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-ws
       s-x509-token-profile-1.0#X509v3" />
     </wsse:SecurityTokenReference>
    </ds:KeyInfo>
   </ds:Signature>
  </wsse:Security>
 </soapenv:Header>
 <soapenv:Body wsu:Id="wssecurity_signature_id_0"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecuri
  ty-utility-1.0.xsd">
  <p635:DFHOXCMN xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
   <p635:ca_request_id>01ORDR</p635:ca_request_id>
   <p635:ca_return_code>0</p635:ca_return_code>
   <p635:ca_response_message>[C@752a752a</p635:ca_response_message>
   <p635:ca_order_request>
    <p635:ca_userid>luis1106</p635:ca_userid>
    <p635:ca_charge_dept>itso</p635:ca_charge_dept>
    <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
    <p635:ca_quantity_req>1</p635:ca_quantity_req>
    <p635:filler1 xsi:nil="true" />
   </p635:ca_order_request>
  </p635:DFHOXCMN>
 </soapenv:Body>
</soapenv:Envelope>
```

- The header includes the `mustUnderstand="1"` attribute, which indicates that either this header must be processed or a SOAP fault thrown.
- The `<wsse:BinarySecurityToken>` contains the base64binary encoding of the WebSphere certificate. This includes the public key that CICS uses to verify the signature.
- The `<ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />` element specifies the algorithm used to sign the message digest.

- The `<ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />` element specifies the algorithm used to produce the message digest.
- The `<ds:DigestValue>` element contains the value of the message digest.
- The `<ds:SignatureValue>` element contains the value of the signed message digest. It is the digest value encrypted with the requester's private key.

Figure 10-34 shows the ORDR transaction running with the user ID WEBAS1 (the user ID associated with the WebSphere certificate), while the CPIH transaction continues to execute with the CIWSNW user ID.



```
INQUIRE TASK
STATUS:  RESULTS - OVERTYPE TO MODIFY
 Tas(0000055) Tra(CPIH)           Sus Tas Pri( 001 )
    Sta(U ) Use(CIWSNW  ) Uow(BF3C2E9E2114284E) Hty(RZCBNOTI)
 Tas(0000056) Tra(ORDR)           Sus Tas Pri( 001 )
    Sta(U ) Use(WEBAS1  ) Uow(BF3C2E9E24801B8B) Hty(EDF     )
 Tas(0000058) Tra(CEDF) Fac(E024) Sus Ter Pri( 001 )
    Sta(SD) Use(NIGELW  ) Uow(BF3C2E9E28D4A149) Hty(ZCIOWAIT)
 Tas(0000059) Tra(CEMT) Fac(E025) Run Ter Pri( 255 )
    Sta(TO) Use(NIGELW  ) Uow(BF3C2EA64198CCCE)

                                                   SYSID=S3C1 APPLID=A6POS3C1
```

Figure 10-34 ORDR executing with user ID associated with WebSphere certificate

The corresponding signed response message sent by CICS is shown in Example 10-21.

Example 10-21 Signed SOAP response message

```
<?xml version="1.0" encoding="UTF8" standalone="no" ?>
  <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
        -200401-wss-wssecurity-secext-1.0.xsd" SOAP-ENV:mustUnderstand="1"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
        wssecurity-utility-1.0.xsd"
        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <wsse:BinarySecurityToken
          EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss
          -soap-message-security-1.0#Base64Binary"
          ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x5
          09-token-profile-1.0#X509"
          wsu:Id="x509cert00">MIIC4DCCAkmgAwIBAgIBADANBgkqhkiG9w0BAQUFADB8MQswC
            QYDVQQGEwJVUzERMA8GA1UECBMITkVXIFlPUksxETAPBgNVBAcTCEVORElDT1RUMQ0w
            CwYDVQQKEwRJVFNPMQ0wCwYDVQQLEwRQU1NDMRYwFAYDVQQMEw1NQURFIFVQIFRJVEx
            FMREwDwYDVQQDEwhDSUNTQ0VSVDAeFw0wNTAxMDEwNDAwMDBaFw0xNTAxMDEwMzU5NT
            laMHwxCzAJBgNVBAYTAlVTMREwDwYDVQQIEwhORVcgWU9SSzERMA8GA1UEBxMIRU5ES
            UNPVFQxDTALBgNVBAoTBElUU08xDTALBgNVBAsTBFBTU0MxFjAUBgNVBAwTDU1BREUg
            VVAgVElUTEUxETAPBgNVBAMTCENJQ1NDRVJUMIGfMA0GCSqGSIb3DQEBAQUAA4GNADC
            BiQKBgQC5EHPavGQtIkVbPO+qNaBFF79tNk3aXDur3Pup4KIycA7JueLtm6sLOyQDNF
            snZgw8llW97EUKIsT55jwYHZcGSR2TjNxswdGrt4rt8EPgwtN3WSl609uWrhVTug4Uu
            MEOXpivdcNDTwRbQgvMtXdOKOWvdSmrbEwBiB4LSdm2pQIDAQABo3IwcDA/BglghkgB
            hvhCAQ0EMhMwR2VuZXJhdGVkIGJ5IHRoZSBTZWN1cml0eSBTZXJ2ZXIgZm9yIHovT1M
            gKFJBQ0YpMA4GA1UdDwEB/wQEAwIEUDAdBgNVHQ4EFgQU6ztx+OkFbSysc93LQBZkSh
            ymaPgwDQYJKoZIhvcNAQEFBQADgYEATtVMjr2pTz47TF92RoTpUneZxq2eMz7LJJSLD
            u37ya+qCLaxbTPB7XfKgqn+egYyYXDYi4mIsfpRq3MHBM/nFIvWtPONpcmCeZ8hfHlq
            Gp3mze+NfuRV4iLpXtHJR9rDPTQsvgwpuIrWfeMX+/IrxYXtY39lh4L78G8s/CQj23U
            =
        </wsse:BinarySecurityToken>
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <ds:SignedInfo
            xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
            xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
            wssecurity-utility-1.0.xsd"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            <ds:CanonicalizationMethod
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
              <c14n:InclusiveNamespaces
                xmlns:c14n="http://www.w3.org/2001/10/xml-exc-c14n#"
                PrefixList="ds wsu xenc SOAP-ENV soapenc soapenv xsd xsi "/>
            </ds:CanonicalizationMethod>
            <ds:SignatureMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
            <ds:Reference URI="#TheBody">
              <ds:Transforms>
                <ds:Transform
                  Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#">
                  <c14n:InclusiveNamespaces
                    xmlns:c14n="http://www.w3.org/2001/10/xml-exc-c14n#"
                    PrefixList="wsu SOAP-ENV soapenc soapenv xsd xsi "/>
                </ds:Transform>
              </ds:Transforms>
              <ds:DigestMethod
                Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
              <ds:DigestValue>tLbsdlSPsgrzGZ5bDOdtyRoHDWO=</ds:DigestValue>
            </ds:Reference>
          </ds:SignedInfo>
```

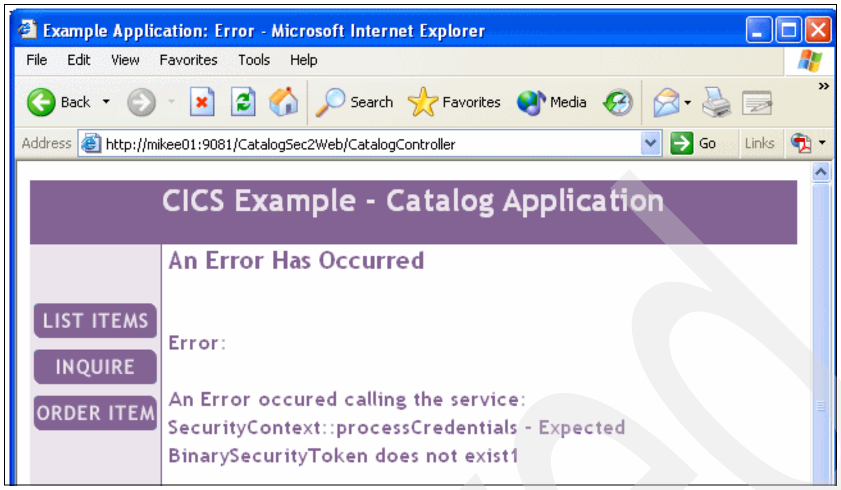
```
        <ds:SignatureValue>Zyxxxq/n7yDaGDYwsIIS21MFbDdMWNruFJ/tT5HuWiODb6N7kS
          DFccM27mQb1uEVFjkNjkKzOnOLNWgIzGqoBU4cV3hu6VOKr4Qg8CnwLO6yDpyQYYC/e
          5rcjfCQUycgJ1JVKdqd+ERN9hwbXX3wZZX3PVZSH5QsOmH/aJOeSME=
        </ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference><wsse:Reference URI="#x509cert00"
            ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
            wss-x509-token-profile-1.0#X509"/>
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
    wssecurity-utility-1.0.xsd" wsu:Id="TheBody">
    <DFHOXCMNResponse
      xmlns="http://www.DFHOXCMN.DFHOXCP5.Response.com">
      <ca_request_id>01ORDR</ca_request_id>
      <ca_return_code>0</ca_return_code>
      <ca_response_message>ORDER SUCESSFULLY PLACED
      </ca_response_message>
      <ca_order_request>
        <ca_userid>luis1106</ca_userid>
        <ca_charge_dept>itso    </ca_charge_dept>
        <ca_item_ref_number>10</ca_item_ref_number>
        <ca_quantity_req>1</ca_quantity_req>
        <filler1>...</filler1>
      </ca_order_request>
    </DFHOXCMNResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The <wsse:BinarySecurityToken> contains the base64binary encoding of the CICS certificate. This includes the public key that WebSphere uses to verify the signature. The rest of the SOAP headers in the response message are similar to those in the request message.

SOAP fault messages

In this section we show some of the SOAP faults that you may see when testing signature processing.

- Figure 10-35 shows the browser response when CICS is expecting a signed message but the service requester does not sign the request.


Example Application: Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mikee01:9081/CatalogSec2Web/CatalogController
Go Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM

An Error Has Occurred

Error:

An Error occured calling the service:
SecurityContext::processCredentials - Expected
BinarySecurityToken does not exist1

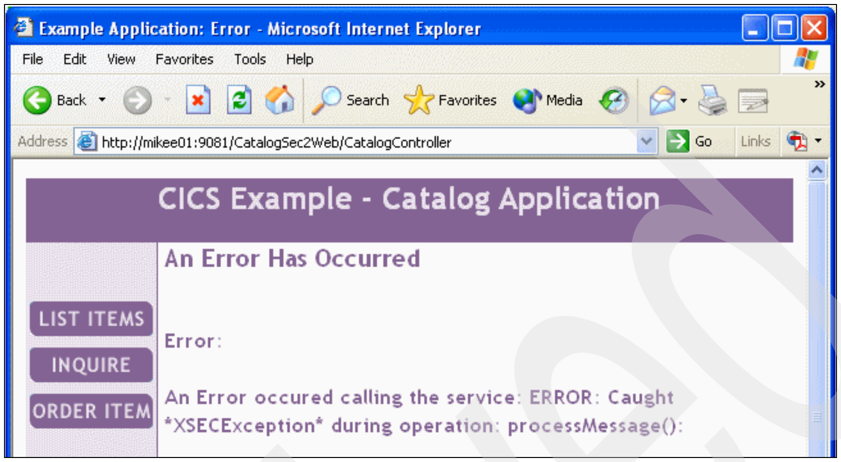
Figure 10-35 DFHWSSE1 SOAP fault - X.509 certificate not known

The SOAP fault for this message, shown in Example 10-22, highlights that CICS is expecting a BinarySecurityToken, which is not sent by WebSphere.

Example 10-22 DFHWSSE1 SOAP fault - X.509 certificate not known

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
    <faultcode>wsse:InvalidSecurity</faultcode>
    <faultstring>ERROR: Caught *XSECException* during operation:
      processMessage()</faultstring>
    <detail>
      <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
        <message>SecurityContext::processCredentials - Expected
          BinarySecurityToken does not exist</message>
        <errorcode>1</errorcode>
        </e:myfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- Figure 10-36 shows the browser response when the client certificate is not associated to a user ID.


Example Application: Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mikee01:9081/CatalogSec2Web/CatalogController
Go Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM

An Error Has Occurred

Error:

An Error occured calling the service: ERROR: Caught *XSECException* during operation: processMessage():

Figure 10-36 DFHWSSE1 SOAP fault - X.509 certificate not known

The SOAP fault for this message, shown in Example 10-23, highlights that there is no RACF user ID defined for the WebSphere certificate.

Example 10-23 DFHWSSE1 SOAP fault - X.509 certificate not known

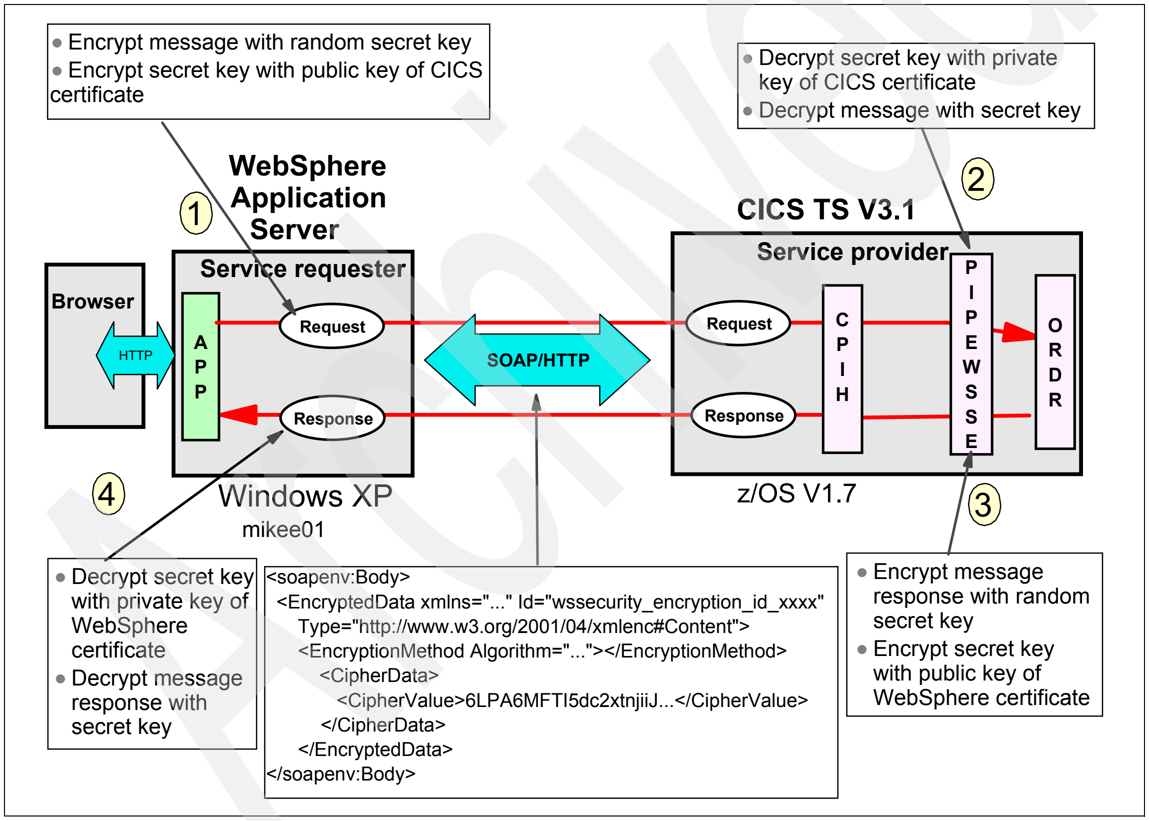
```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>wsse:FailedAuthentication</faultcode>
      <faultstring>ERROR: Caught *XSECException* during operation:
        processMessage()</faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
          <message>XSECKeyInfoResolverZos::extractUserId - Either no RACF user
            ID is defined for this certificate, or the certificate status is
            NOTRUST.</message>
        <errorcode>1</errorcode>
        </e:myfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

10.6 Encrypting a SOAP message

To provide confidentiality we can encrypt the contents of a SOAP message using XML encryption. If the SOAP message is encrypted, only a service consumer that knows the key for confidentiality can decrypt and read the message.

CICS supports XML encryption in both service provider and service requester modes.

In this section we show how a SOAP message can be encrypted by WebSphere Application Server and how the SOAP message can be decrypted by CICS. We also show how the response message can be encrypted by CICS and decrypted by WebSphere. This scenario is shown in Figure 10-37.



• Encrypt message with random secret key
• Encrypt secret key with public key of CICS certificate
• Decrypt secret key with private key of CICS certificate
• Decrypt message with secret key
1
2
3
4
WebSphere Application Server
Service requester
Browser
HTTP
APP
Request
Response
SOAP/HTTP
CICS TS V3.1
Service provider
Request
Response
CPIH
PIPEWSSE
ORDR
Windows XP
mikee01
z/OS V1.7
• Decrypt secret key with private key of WebSphere certificate
• Decrypt message response with secret key
• Encrypt message response with random secret key
• Encrypt secret key with public key of WebSphere certificate

```
<soapenv:Body>
  <EncryptedData xmlns="..." Id="wssecurity_encryption_id_xxxx"
     Type="http://www.w3.org/2001/04/xmlenc#Content">
     <EncryptionMethod Algorithm="..."></EncryptionMethod>
        <CipherData>
           <CipherValue>6LPA6MFTI5dc2xtnjiiJ...</CipherValue>
        </CipherData>
     </EncryptedData>
</soapenv:Body>
```

Figure 10-37 Enabling CICS WS-Security support for encryption

1. The WebSphere application server generates a random secret key and uses it to encrypt the SOAP message body. It also encrypts the secret key with the

CICS public key so only the CICS region can decrypt it. Both the encrypted data and the encrypted secret key are attached to the SOAP message.

2. CICS decrypts the secret key with its private key and uses it to decrypt the SOAP body.
3. CICS then encrypts the SOAP body of the response message with a different randomly generated secret key. It also encrypts the secret key with the public key of WebSphere so only the WebSphere application server can decrypt it.
4. The WebSphere application server decrypts the secret key with its private key and uses it to decrypt the SOAP body of the message response.

Recommendation: In our encryption scenario we use the same key pairs that we used in the signature scenario. In a production environment, however, we recommend that you use different key pairs for signature and encryption processing.

Important: There is a significant performance overhead when using WS-Security and XML encryption. See "Comparison of transport level and SOAP message security" on page 269 for recommendations on choosing between WS-Security and SSL when implementing a confidentiality security solution.

At a high level, the steps to enable support for XML encryption processing are the following:

- Configure the service requester for encryption.
- Configure CICS for encryption.
- Test the encryption test scenario.

Step-by-step details to accomplish these tasks are presented in the following sections.

Note: In our encryption test scenario we encrypt both the request and response messages. It is also possible to encrypt only the request, or to encrypt only the response. Equally, it is possible to both encrypt *and* sign a message request or response.

10.6.1 Configuring the service requester for encryption

In this section we show how to configure the client J2EE application in order to send an encrypted SOAP message. At a high level the steps are:

- Import the Web service client application into AST.
- Configure the request generator for encryption.
- Configure the response consumer for encryption.
- Re-deploy the Web Service client application.

Import the Web service client application into AST

To configure our J2EE application for encryption, we used the IBM WebSphere Application Server Toolkit V6.1 (AST).

See "Importing the base application" on page 333 for instructions on how to import the application. For the signature test scenario, we specified:

- The name of the workspace folder as:
  `C:\AST\CatalogSec2\CatalogSec2Encrypt`
- The name of the EAR project as:
  `CatalogSec2Encrypt`

Configure the request generator for encryption

Perform the following steps to configure the request generator for encryption:

1. Expand the **Dynamic Web Project** (**CatalogSec2Web**) in the Project Explorer and open (double-click) the deployment descriptor as shown in Figure 10-38.

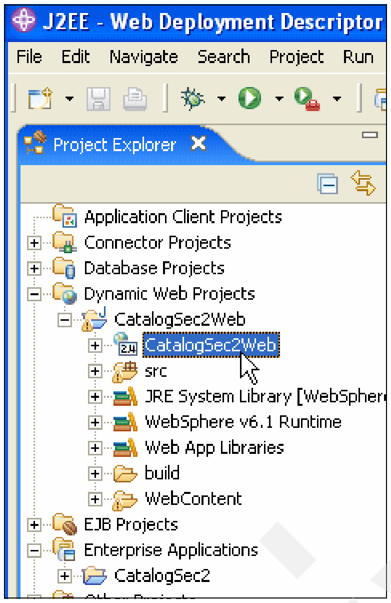


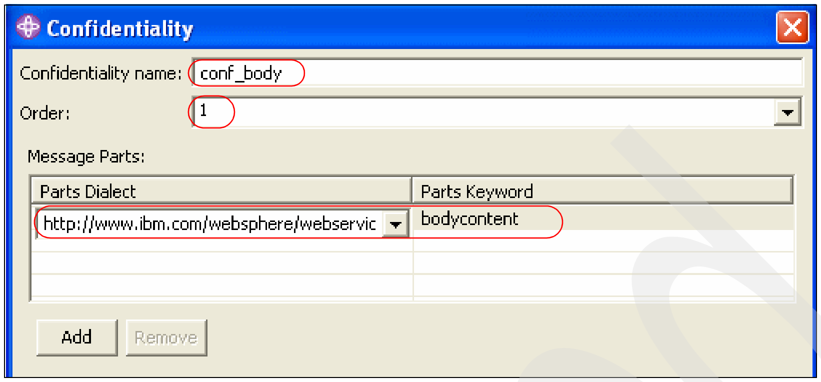
Figure 10-38 Opening the CatalogSec2Web Deployment Descriptor

2. Go to the WS Extension page.
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).
   b. Expand the **Request Generator Configuration**.
   c. Expand **Confidentiality** and click **Add**. In the Confidentiality dialog box (Figure 10-39) enter the following data:
      - Enter a Confidentiality name, for example, `conf_body`.
      - Select the order in which the encryption is generated. Multiple confidentiality parts can be specified, and you have to specify the order of generating the encryption. In our case, we select **1**.

      > **Note:** The WS-Security runtime of WebSphere V6.1 supports multiple signature and encryption parts in one SOAP message. For multiple signature and encryption parts, you need to specify the processing order. For example, if you want to sign *and* encrypt the SOAP body, you should specify 1 in the Integrity dialog and 2 in the Confidentiality dialog.

      - Click **Add** for Message Parts, and one confidentiality part is created. The default created part is the SOAP body. We accepted the default.

d. Click OK and save the configuration by pressing Ctrl+s.



Confidentiality name: conf_body
Order: 1
Message Parts:

Parts Dialect	Parts Keyword
http://www.ibm.com/websphere/webservic	bodycontent

Figure 10-39 Confidentiality dialog in the WS Extensions page

3. Open the *WS Binding* page.
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).
   b. Expand the **Security Request Generator Binding Configuration**.
   c. Expand **Key Locators** and click **Add**. In the Key Locator dialog box (Figure 10-40) enter the following data:
      - Enter a Key locator name, for example, `gen_enclocator`.
      - Select **KeyStoreKeyLocator** as the Key locator class.
   d. Select **Use key store**. For key-related information refer to "Importing the base application" on page 360. Make the appropriate entries. In our case the values were:

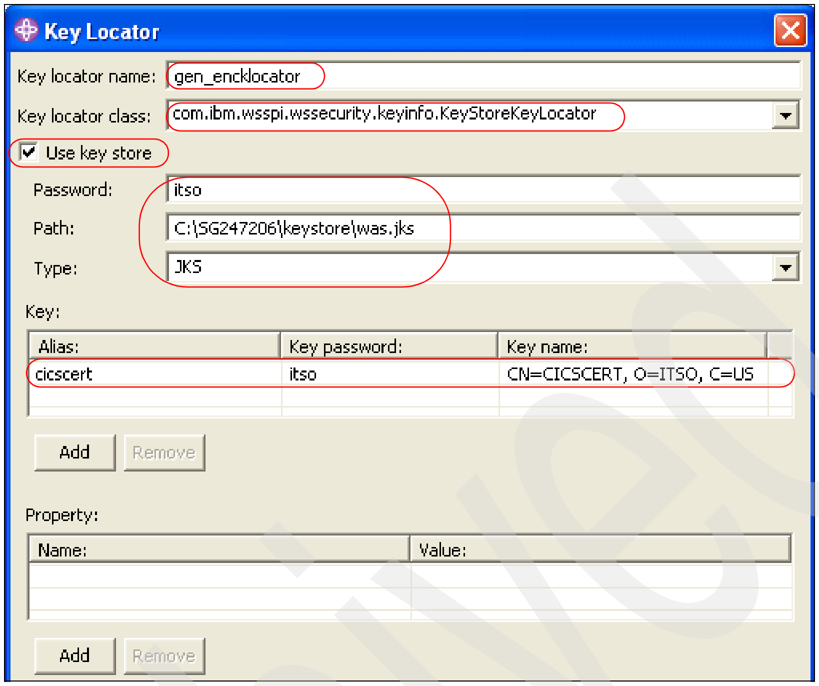
Password:	`itso`
Path:	`C:\SG247206\keystore\was.jks`
Type:	`JKS`

   e. Click **Add** under Key and make the appropriate entries. Ours were:

Alias:	`CICSCERT`
Key password:	`itso`
Key name:	`CN=CICSCERT, O=ITSO, C=US`

Note: This is the key name of the CICS certificate. The public key of this certificate is used by WebSphere to encrypt the request message.

   f. Click **OK**.


Key Locator

Key locator name: gen_encklocator

Key locator class: com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator

[✓] Use key store

Password: itso

Path: C:\SG247206\keystore\was.jks

Type: JKS

Key:

Alias:	Key password:	Key name:
cicscert	itso	CN=CICSCERT, O=ITSO, C=US

Property:

Name:	Value:

Figure 10-40 Key Locator dialog in the WS Binding page

4. Expand **Key Information** and click **Add**. In the Key Information dialog box (Figure 10-41) enter the following data:

   a. Enter a Key information name, for example, `gen_enckeyinfo`.

   b. Select **KEYNAME** as the Key information type. The Key information class is filled automatically. The Key information types are:

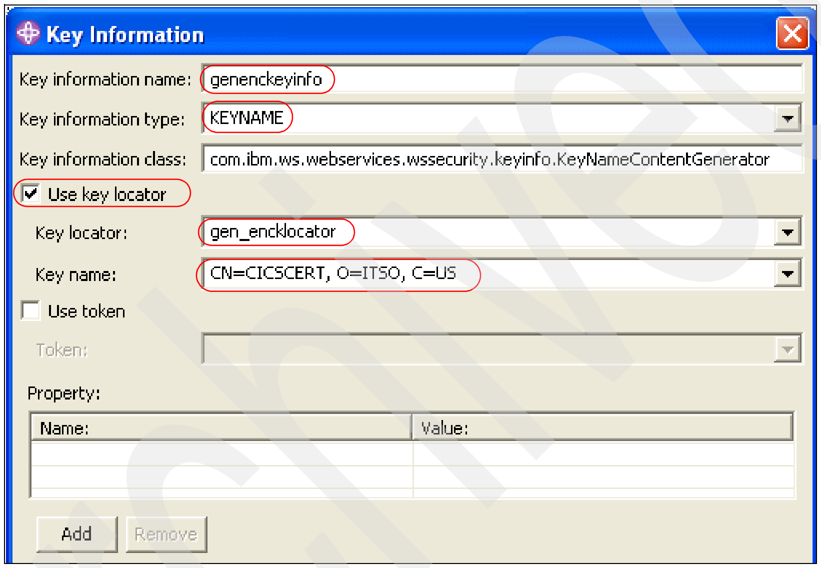
• STRREF	Direct reference
• EMB	Embedded reference
• KEYID	Key identifier reference
• KEYNAME	Key name reference
• X509ISSUER	x.509 issuer and serial number reference

   We specify KEYNAME because we want WebSphere to send the key information using the distinguished name of the CICS certificate.

Recommendation: Use of KEYNAME for the key information type allows CICS to reference the RACF certificate directly because each certificate has a unique distinguished name and RACF provides an API for accessing the certificate using the distinguished name.

c. Select **Use key locator** and select **gen_encklocator** from the Key locator drop-down list. Select the predefined Key name **CN=CICSCERT, O=ITSO, C=US**.

d. Click **OK**.



Key information name: genenckeyinfo
Key information type: KEYNAME
Key information class: com.ibm.ws.webservices.wssecurity.keyinfo.KeyNameContentGenerator
[✓] Use key locator
Key locator: gen_encklocator
Key name: CN=CICSCERT, O=ITSO, C=US
[ ] Use token
Token:
Property:

Name:	Value:

Figure 10-41 Key Information dialog in the WS Binding page

5. Expand **Encryption Information** and click **Add**. In the Encryption Information dialog box (Figure 10-42) enter the following data:

   a. Enter an Encryption Name, for example, `enc_body`.

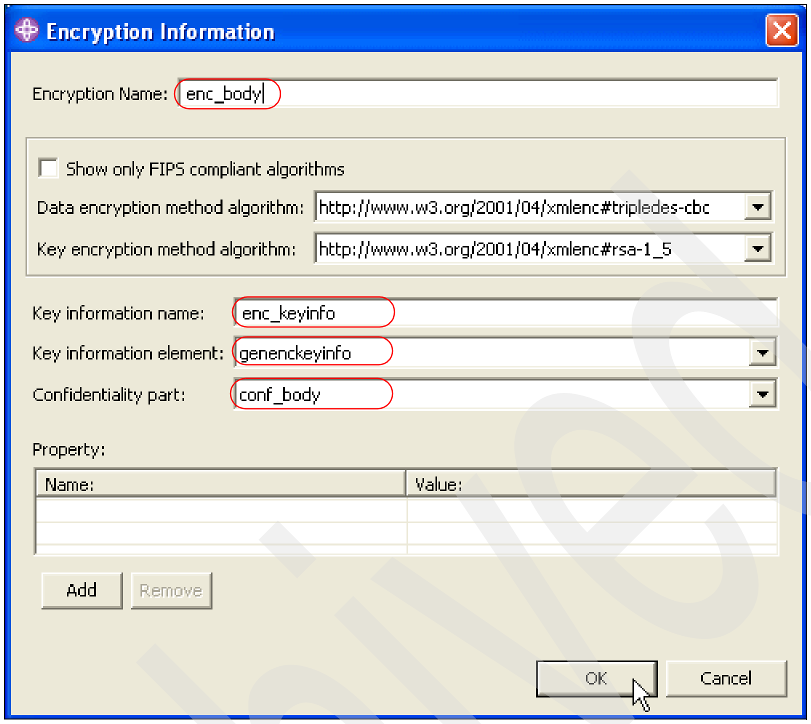
   b. Enter a Key information name, for example, `enc_keyinfo`.

   c. Select **gen_enckey** as a Key information element.

   d. Select **conf_body** as a Confidentiality part.

   e. Allow the encryption algorithms to default.

   f. Click **OK**.



Encryption Name: enc_body

☐ Show only FIPS compliant algorithms

Data encryption method algorithm: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

Key encryption method algorithm: http://www.w3.org/2001/04/xmlenc#rsa-1_5

Key information name: enc_keyinfo

Key information element: genenckeyinfo

Confidentiality part: conf_body

Property:

Name:	Value:

Figure 10-42 The Encryption Information dialog in the WS Binding page

6. Save the configuration by pressing Ctrl+s.

Configure the response consumer for encryption

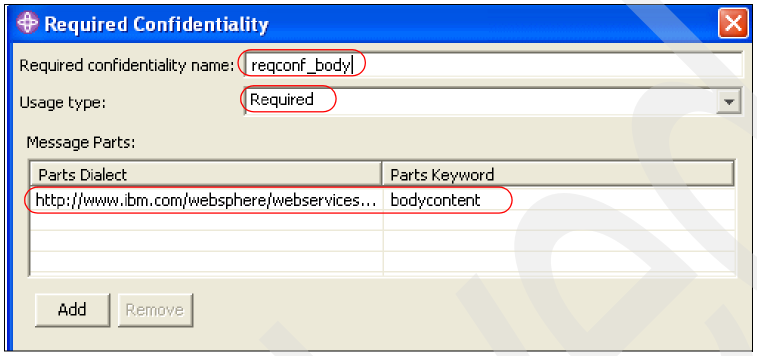
In this section we show how to configure the client J2EE application in order to receive an encrypted SOAP message. To do this, perform the following steps:

1. Expand the **Dynamic Web Project** (**CatalogSec2Web**) in the Project Explorer and open (double-click) the deployment descriptor.
2. Go to the WS Extension page.
   a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).
   b. Expand the **Response Consumer Configuration**.
   c. Expand **Required Confidentiality** and click **Add**. In the Required Confidentiality dialog box (Figure 10-43) enter the following data:
      - Enter a Required confidentiality name, for example, `reqconf_body`.
      - Select the Usage type, either Required or Optional. If the usage type is Required, an unencrypted request message throws a SOAP fault. If the

usage type is Optional, an unencrypted message is received. Select **Required**.

d. Click **Add** for Message Parts, and one message part is created. The default created part is the SOAP body. We accepted the default.

e. Click **OK** and save the configuration by pressing Ctrl+s.



Required confidentiality name: reqconf_body

Usage type: Required

Message Parts:

Parts Dialect	Parts Keyword
http://www.ibm.com/websphere/webservices...	bodycontent

Figure 10-43 Required Confidentiality in the WS Extension page for response consumer

3. Open the WS Binding page.

a. Select the **service/DFH0XCMNService3** reference (for our Order service) and **DFH0XCMNPortPO** (the port binding for the service reference).

b. Expand the **Security Response Consumer Binding Configuration**.

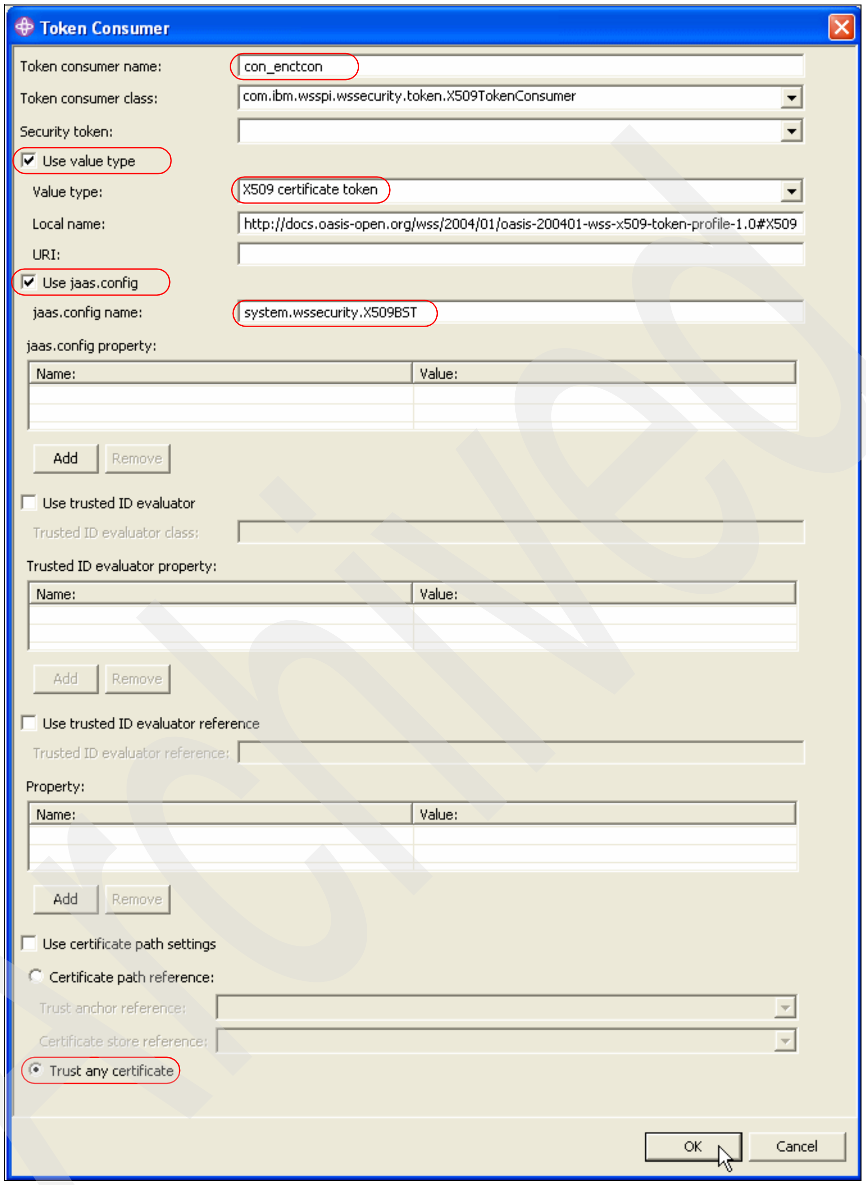
c. Expand **Token Consumer** and click **Add**. In the Token Consumer dialog box (Figure 10-44) enter the following data:

- Enter a Token consumer name, for example, `con_enctcon`.
- Select **com.ibm.wsspi.wssecurity.token.X509TokenConsumer** as the Token consumer class.
- Select **Use value type**.
- Select **X509 certificate token**. The Local name is selected automatically.

Note: It is important to specify X509 certificate token for the token value type. An error occurred in our test when we specified X509 certificate token v3.

- Select **Use jaas.config** and enter `system.wssecurity.X509BST` as the jaas.config name.
- Select **Trust any certificate**.

d. Click **OK**.



Token Consumer

Token consumer name: con_enctcon

Token consumer class: com.ibm.wsspi.wssecurity.token.X509TokenConsumer

Security token:

[✓] Use value type

Value type: X509 certificate token

Local name: http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509

URI:

[✓] Use jaas.config

jaas.config name: system.wssecurity.X509BST

jaas.config property:

Name:	Value:

Add Remove

[ ] Use trusted ID evaluator

Trusted ID evaluator class:

Trusted ID evaluator property:

Name:	Value:

Add Remove

[ ] Use trusted ID evaluator reference

Trusted ID evaluator reference:

Property:

Name:	Value:

Add Remove

[ ] Use certificate path settings

○ Certificate path reference:

Trust anchor reference:

Certificate store reference:

● Trust any certificate

OK Cancel

Figure 10-44 Token Consumer dialog in the WS Binding page for response consumer

4. Expand **Key Locators** and click **Add**. In the Key Locator dialog box (Figure 10-45) enter the following data:

   a. Enter a Key locator name, for example, `con_encklocator`.

b. Select **KeyStoreKeyLocator** as the Key locator class.

c. Select **Use key store** and make the appropriate entries. Ours were:

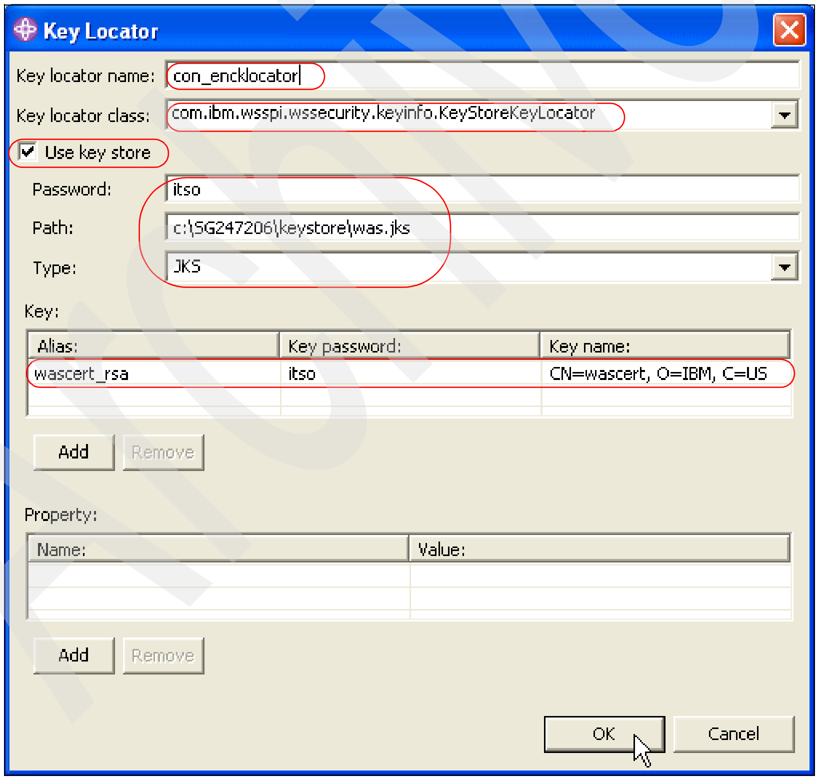
Password:	`itso`
Path:	`C:\SG247206\keystore\was.jks`
Type:	`JKS`

d. Click **Add** under Key and make the appropriate entries. Ours were:

Alias:	`wascert_rsa`
Key password:	`itso`
Key name:	`CN=wascert, O=IBM, C=US`

Note: This is the key name of the WebSphere certificate. The public key of this certificate is used by CICS to encrypt the response message.

e. Click **OK**.



Key locator name: con_encklocator

Key locator class: com.ibm.wsspi.wssecurity.keyinfo.KeyStoreKeyLocator

[✓] Use key store

Password: itso

Path: c:\SG247206\keystore\was.jks

Type: JKS

Key:

Alias:	Key password:	Key name:
wascert_rsa	itso	CN=wascert, O=IBM, C=US

Figure 10-45 Key Locator dialog in the WS Binding page for response consumer

5. Expand **Key Information** and click **Add**. In the Key Information dialog box (Figure 10-46) enter the following data:

   a. Enter a Key information name, for example, `con_enckeyinfo`.

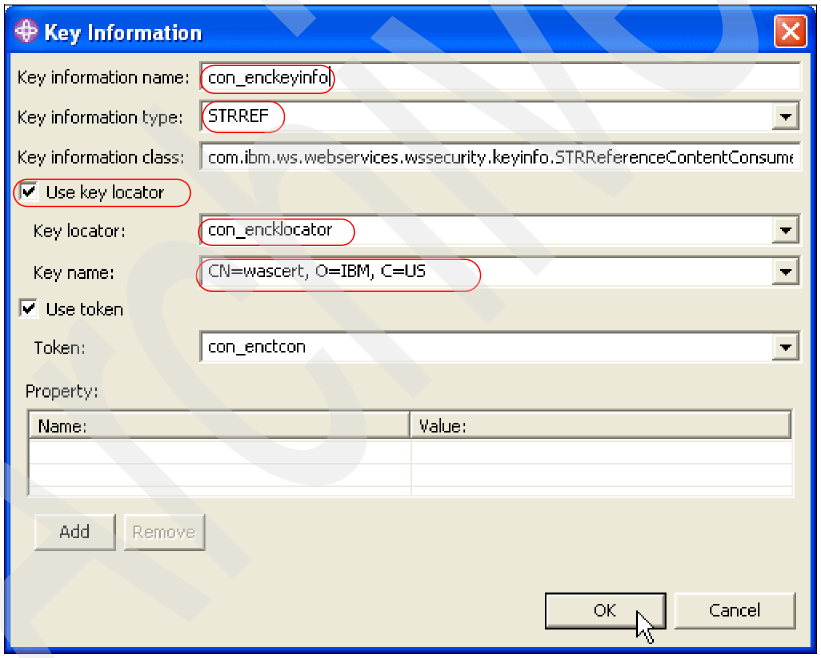
   b. Select **STRREF** as the Key information type. The Key information class is filled automatically.

      We specify STRREF because CICS always sends the complete certificate when encrypting an outbound message. Therefore, the WebSphere response consumer must be configured to expect the WebSphere certificate in the security header of the response message.

   c. Select **Use key locator** and select **con_encklocator** from the Key locator drop-down list. Select the predefined Key name **CN=wascert, O=IBM, C=US**.

   d. Select **Use token** and select **con_enctcon** from the Token drop-down list.

   e. Click **OK**.



Key information name: con_enckeyinfo
Key information type: STRREF
Key information class: com.ibm.ws.webservices.wssecurity.keyinfo.STRReferenceContentConsume
[✓] Use key locator
Key locator: con_encklocator
Key name: CN=wascert, O=IBM, C=US
[✓] Use token
Token: con_enctcon
Property:

Name:	Value:

Figure 10-46 Key Information dialog in the WS Binding page for response consumer

6. Expand **Encryption Information** and click **Add**. In the Encryption Information dialog box (Figure 10-47) enter the following data:

   a. Enter a Encryption Name, for example, `enc_body`.

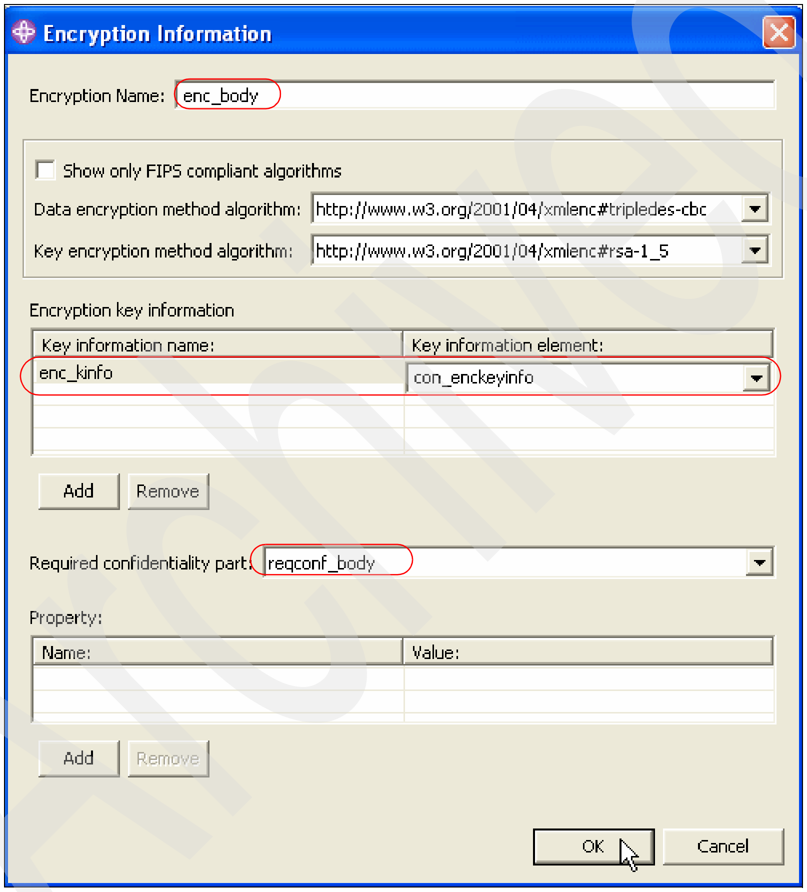
b. Click **Add** in Encryption key information and make the appropriate entries:

   - Enter a Key information name, for example, `enc_kinfo`.
   - Select the **con_enckeyinfo** from the Key information element drop-down list.

c. Select **reqconf_body** as a Required Confidentiality part.

d. Allow the encryption algorithms to default.

e. Click **OK**.



Encryption Information

Encryption Name: enc_body

☐ Show only FIPS compliant algorithms

Data encryption method algorithm: http://www.w3.org/2001/04/xmlenc#tripledes-cbc

Key encryption method algorithm: http://www.w3.org/2001/04/xmlenc#rsa-1_5

Encryption key information

Key information name:	Key information element:
enc_kinfo	con_enckeyinfo

Required confidentiality part: reqconf_body

Property:

Name:	Value:

Figure 10-47 Encryption Information dialog in WS Binding page for Response Consumer

7. Save the configuration by pressing Ctrl+s.

Redeploying the Web service application

After configuring an encryption security constraint for the service requester application, we exported a new EAR file called CatalogSec2_WS-Security_Encryption.ear. We followed the same process that is described in "Installing the service requester" on page 87 to deploy the CatalogSec2_WS-Security_Encryption.ear.

For further information about configuring WS-Security in WebSphere refer to *Web Services Handbook for WebSphere Application Server 6.1,* SG24-7257.

10.6.2 Configuring CICS for encryption

In this section we show how to configure the CICS pipeline to receive and send encrypted SOAP messages.

Example 10-24 shows the pipeline configuration file that we used for the encryption test scenario. It includes the message handler DFHWSSE1, and the configuration information for the handler.

Example 10-24 Pipeline config file, ITSO_7206_wssec_signature_provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
    <transport>
      <default_transport_handler_list>
          <handler>
             <program>CIWSMSGH</program>
             <handler_parameter_list/>
          </handler>
      </default_transport_handler_list>
    </transport>
    <service>
      <service_handler_list>
        <handler>
          <program>SNIFFER</program>
          <handler_parameter_list/>
        </handler>
        <wsse_handler>
          <dfhwsse_configuration version="1">
            <expect_encrypted_body/>
            <encrypt_body>
              <algorithm>http://www.w3.org/2001/04/xmlenc#tripledes-cbc
              </algorithm>
              <certificate_label>WASCERT</certificate_label>
            </encrypt_body>
          </dfhwsse_configuration>
        </wsse_handler>
      </service_handler_list>
      </terminal_handler>
        <cics_soap_1.2_handler/>
      </terminal_handler>
    </service>
  <apphandler>DFHPITP</apphandler>
</provider_pipeline>
```

- The `<wsse_handler>` element contains a `<dfhwsse_configuration>` element that specifies configuration information for DFHWSSE1.
- The `<expect_encrypted_body/>` element indicates that the `<body>` of the inbound message must be encrypted. If the body of an inbound message is not correctly encrypted, CICS rejects the message with a security fault.

  We specify `<expect_encrypted_body/>` because we want to prevent the placeOrder service being accessed unless the body of the SOAP message is encrypted.
- The `<encrypt_body/>` element indicates that the `<body>` of the outbound message will be encrypted, and provides information about how the message is to be encrypted.
- `<algorithm>http://www.w3.org/2001/04/xmlenc#tripledes-cbc</algorithm>` is a child element of the `<encrypt-body>` element that specifies the URI of the encryption algorithm to be used for encrypting the response message. We specified the triple-DES algorithm.

  Important: The encryption algorithm must match the data encryption method algorithm specified in the WebSphere application service response consumer configuration.

- `<certificate_label>WASCERT</certificate_label>` is a child element of the `<encrypt-body>` element that specifies the label associated with the WebSphere certificate installed in RACF. This certificate contains the public key that is used by CICS to encrypt the secret (or symmetric) key. The encrypted secret key is then sent in the SOAP message response.

We completed the CICS configuration by changing the pipeline PIPEWSSE (used for the placeOrder service) to specify the pipeline configuration file that we have created for encryption processing. We used the following CEDA command:

```
CEDA ALTER PIPELINE(PIPEWSSE) GROUP(S3C1EXWS)
Configfile(/CIWS/S3C1/config/ITSO_7206_wssec_encryption_provider.xml)
```

We then re-installed the changed pipeline resource definition.

10.6.3 Testing the encryption scenario

An example of an encrypted SOAP request message for the placeOrder service is shown in Example 10-25.

Example 10-25 Encrypted SOAP request message

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Header>
    <wsse:Security soapenv:mustUnderstand="1"
      xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
      wssecurity-secext-1.0.xsd">
      <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
        <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"
        />
        <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
          <ds:KeyName>CN=CICSCERT, T=Ciwss3c1-cert, OU=PSSC, O=ITSO,
            L=ENDICOTT, ST=NEW YORK, C=US</ds:KeyName>
        </ds:KeyInfo>
        <CipherData>
          <CipherValue>rN8nTy+IlIPN/g4ezibEMaxtnfxEZzQBCGgQQ8HSRMbqy6uy6bhhsty9
            4CZuxDzdKV53vSAxprBgrEYExAIB2Ynn33xo8X4SkYlKKkV/BdiUSjC3x+yEYUKKTDx
            aT4Swi/OINkmxOrT9KU2vSIoqaYA+PTHGn336ldbcDGZjdPo=</CipherValue>
        </CipherData>
        <ReferenceList>
          <DataReference URI="#wssecurity_encryption_id_29" />
        </ReferenceList>
      </EncryptedKey>
    </wsse:Security>
  </soapenv:Header>
  <soapenv:Body>
    <EncryptedData Id="wssecurity_encryption_id_29"
      Type="http://www.w3.org/2001/04/xmlenc#Content"
      xmlns="http://www.w3.org/2001/04/xmlenc#">
      <EncryptionMethod
        Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
      <CipherData>
        <CipherValue>y3FFMZ4ckOZjfpydskgrNHQP9PrFy9kdT9axDeKjqbyj1j6eNeE6nmPpt1
          /UvpGKsEwBchUWvIbIuanXujgc9MGjzcX7lD+xYjAgczoCrykXhO/l53eHgNp445BkPQl
          plV7lea3Q957FLHq1XCa7wIWMwzxZnTkdjzZg8bwKDNJElDNRJCDjtP7wcxK17rJTWKrg
          lRx8bMBWUzEcrHDfFK/Y4FRFmyL9FDKRZFIDnYQ6NSJXqxULZbW9p7vtQmkSvBOteF44X
          iDouMEJCl9Sk6Cf6Nd3xhaxidt2EZYGGK8zrsL8mWrmLPQxfeHxwGuQ6PyMODsvxsPdQN
          m2/s43Hov4VDOYgvWcPd8Q6OONoPcSzPcJoi/kHZnxya7b9vNV1obe+IXX8sej3qFqByW
          uUoEz6nwqDpbHltUfsGUVp36VHUuo91Q7CTcLgLnsZbWLFSqdZLNeRgKIygpVueThLIIp
          ihWNnopZbC9LAag3KmddnCWrw2zw1GV1+alEI4g8rQq4EtlIY5MNqqyDXVyzy2dNnOTiI
          l3fVWOU4r3/Lhl7Yz2HQU8NbDqSEnWptzBJ6xwYD9AcEzQY+kBvuC4Z5AHJMzVHhkVOcu
          glhNjtSxlrcI8K7oVI+HNYL91D6saxF6Aq24zwsoPZJI8FOujWfUCh+G4qzn388u1TnXd
          y2bfpZlo4vpE+r9OYSbZitpqb
        </CipherValue>
      </CipherData>
    </EncryptedData>
  </soapenv:Body>
</soapenv:Envelope>
```

- The header includes the `mustUnderstand="1"` attribute, which indicates that either this header must be processed or a SOAP fault thrown.
- The `<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>` child element of the `<EncryptedKey>` element in the SOAP header specifies the algorithm used to encrypt the secret key.
- The `<ds:KeyName>` element contains the distinguished name that uniquely identifies the CICS certificate that has been used to encrypt the message. This is used by CICS to identify the CICSCERT X.509 certificate, the private key of which CICS uses to decrypt the message.
- The `<CipherValue>` element in the SOAP header contains the encrypted secret key that was used to encrypt the message.
- The `<EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>` child element of the `<EncryptedData>` element in the SOAP body specifies the algorithm used to encrypt the message.
- The `<CipherValue>` element in the SOAP body contains the encrypted message.

The corresponding encrypted response message sent by CICS is shown in Example 10-26.

Example 10-26 Encrypted SOAP response message

```
<?xml version="1.0" encoding="UTF8" standalone="no" ?>
  <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Header>
      <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis
        -200401-wss-wssecurity-secext-1.0.xsd" SOAP-ENV:mustUnderstand="1"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wsse
        curity-utility-1.0.xsd"
        xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
        <wsse:BinarySecurityToken
          EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss
          -soap-message-security-1.0#Base64Binary"
          ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss
          -x509-token-profile-1.0#X509"
          wsu:Id="x509cert00">MIICQDCCAamgAwIBAgIERNoOdzANBgkqhkiG9w0BAQUFADBZM
            QswCQYDVQQGEwJVUzEJMAcGA1UEERMAMQkwBwYDVQQIEwAxCTAHBgNVBAcTADEMMAoG
            A1UEChMDSUJNMQkwBwYDVQQLEwAxEDAOBgNVBAMTB3dhc2NlcnQwHhcNMDYwODA5MTk
            xNjA3WhcNMDcwODA5MTkxNjA3WjBZMQswCQYDVQQGEwJVUzEJMAcGA1UEERMAMQkwBw
            YDVQQIEwAxCTAHBgNVBAcTADEMMAoGA1UEChMDSUJNMQkwBwYDVQQLEwAxEDAOBgNVB
            AMTB3dhc2NlcnQwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAIEMtFDoacSSCWTq
            NYVPDd3yMGMIq4GpSNUVajzLdW+hlNIgaKzZfhiOo6BJxQcD+Ty+pKRSlbPLyG9B+9L
            Go+O6dNJmAXDVGNQiSgkNYxl12oWRBCJciU5nBIB3a+TUOe2wYEak+rJ3MblB/TjA3o
            ttykjftOyjRohl97wT65j/AgMBAAGjFTATMBEGA1UdDgQKBAhJYPbuSs76STANBgkqh
            kiG9w0BAQUFAAOBgQBP1SEGGNW4ruua80hItdXERBA356OnzLwO5n+eb2JdZuOilyYH
            NGfp8+k4b+9F9DjOPzGEJzgspqMTraHKOJP1co7yIG/RUuX+gicGN+dI2A8frLsIS2I
            UMB66FqQR/uAOYjGJCuYfzKm2CcOPUFTQYqMMknuj+DBTcAoDp+GP4Q==
        </wsse:BinarySecurityToken>
        <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
          <xenc:EncryptionMethod
            Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
          <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
            <wsse:SecurityTokenReference>
              <wsse:Reference URI="#x509cert00"
                ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
                wss-x509-token-profile-1.0#X509"/>
            </wsse:SecurityTokenReference>
          </ds:KeyInfo>
          <xenc:CipherData>
            <xenc:CipherValue>SLqhQ7hIdj2CugP94jRO63DA6uEJO82NBVsrF8SKWLyYCsaRJ
              mocHahVoUAqTeyYCr7ihPSJifGmC6bbqkxOvLA5nSuKC4IPSzPdK4k7BqBO25wyrJ
              2RCWmKm3MOz6IDRwXvwn9U/OgfsUuXEU3+9CwmddLrM9KisCC8BndvCHE=
            </xenc:CipherValue>
          </xenc:CipherData>
          <xenc:ReferenceList>
            <xenc:DataReference URI="#Enc1"/>
          </xenc:ReferenceList>
        </xenc:EncryptedKey>
      </wsse:Security>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
      <xenc:EncryptedData xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
        Id="Enc1" Type="http://www.w3.org/2001/04/xmlenc#Content">
        <xenc:EncryptionMethod
          Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
        <xenc:CipherData>
```

```
            <xenc:CipherValue>zZJhly8COD1lRF7aSd53yJzi+6XYv4kCw/+y+s9rQm7/yXzK41W
              J6frbz4hv/EN37yYRRzhMJaMz5aTI/wmj5ztTLeUev5J9DNxxGovtBVRiM4iJts2Kby
              QUw//Ig7e7hotYAoHas2lrpROV9tnJUuhkgbLrzOr7J6SsyPwxl6vJLC8txDrJiZSOE
              lAKt9wcDesozuCBJ23kj6A2qwAp2BJzXmVrOUhXVBwmkjqRQn+K5gq/fpnhZwnXSWmM
              n6bOtYhc1a9QkzztJUTdO2PmXjxZEGu6GCt/amh9RxqkmwPwfX22biWUbbHjLrQD4VU
              e3MJnLeimdmP9fENoJOf4Tp/5Cf8hXhdULrdnlKiHFHiSLG3HaSHkBPabWmUMNkXdjk
              p/b2VxfmqRuLuEXIh1qlLDN+oGnQKPQUI9nBt7PNZ4acoFtdjRYGpZavHyTIwvbditY
              SnH+RTpmp16RKsDwmT7/225mhFFfrddc35eO/FeZGnKR1+F+GcLDOWdmSEp/CYcOvqg
              r8Yxq+Huy6r4nbcCxXNI54O4RlxJ8cXiOWIF5ocWXjSDqdFVBGoH573OfTPrMXj6T1F
              5FoX48BboqCUSGggo3TsWKYWyUGZzMQRz7qBgdLcWw/r8gTBVHEEgp7juXlaxqMykcT
              jsvrCO5AxteBhq6zmpqBAS5SuomgUAZ8CtBwVuO26WRF4NHdKbWi+4Sj5MVmsL8ALkZ
              fyKdGOTy5X7nDCGCyY1viqyrGwx/eHCJrkEieqG/lksgrUKUcMPUQlRpyTI7pgcFSIL
              gFUOKqdzrXXCGMJarOmodcLFMSfjPWgnYlIcrZl+IuPRa3LCePGFGiuwYBob+h7Va8w
              5+Wp+ukdsljEpvTXhsAeOnTmQgHrUdV9bblhryGc7b65HIK4ONf3H9nayNlqABW1cqn
              eGVzGO2xGdfjSRSf/G5w9m15sQt/89irNUM5odxaQMiwIYY7woi4DhzKyqM8HftmlnX
              3T8izIzhaT2OjnOKKxrIE+CzWqyIpYgF7GRuyVRJRMPh/xQNFxE6bLspcWazNtsWFOl
              dHZZe6uyOQ7TH6DXvyIa9feDawZ3aSIERTN/garMKFimoYAVH5ShoBH3UvJQ23ixU2J
              z8N7HLVBpwOaHDGpaLBqoM/Lqii4mLG8B9jNIzBV7iXVaB6vUBgcJhYd/FZ3bEn2kp/
              kc1j2YKRSQhRCLKosFBY1KXmNCt7IClfu1XzyTiofnb7kEI3dM3Us7tufOOefmyG/jM
              UzED5f3y6en1fhmvEGNV7+R7luc1gvTT2Rs9V8cBrUzSriaJQr2NZgJY7jTnnULDzUU
              FI7TLygbpVjCRNu/hkjDHarFRgnzixocUdAHhMuiWil+Hwyms3//osjQ8+VTFfQRVee
              Y3RwkOkEnceunyFOH/ICQdJCJdgxyAQ/ObtOPOAnUR3SCpcWV8jcjM5LlGB7a+TSPQB
              poVb/P9hzQNTiekYpTWkaTITM96nktUvtmXq6ymAhkcCbXvqbstUdOEJn1d73hMuCkK
              OX9oBSYOmbLsTsTV6yfS5Q3nSlPv12sSf3ogxzrzX8TOWp/yhhFdW4bnb1Cyckkn2Ya
              NTLcmrlCYsMRpVMZWIkvNesK8pDZijyprDjMmaezBv6DcnNqn3WntgUrkFsBvI7EySC
              OgW1MqvFV6/E3hD+mIJItSWhvU4i4gBZGKNggtAuItJ4oWV+DCSPOPgiJ/FV3MhNCXO
              xyyQlcg+uFXXXQpkDjOs5pHruWnMF5FULQCq6cSVF+bXqN1EzGrxK5F64rh74INJjBe
              nm6AwEVGWyEiTT9ZRysfD9wvbZMd2fzlTOFOA6yJXz7bnqAB2oAng==
            </xenc:CipherValue>
          </xenc:CipherData>
        </xenc:EncryptedData>
      </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

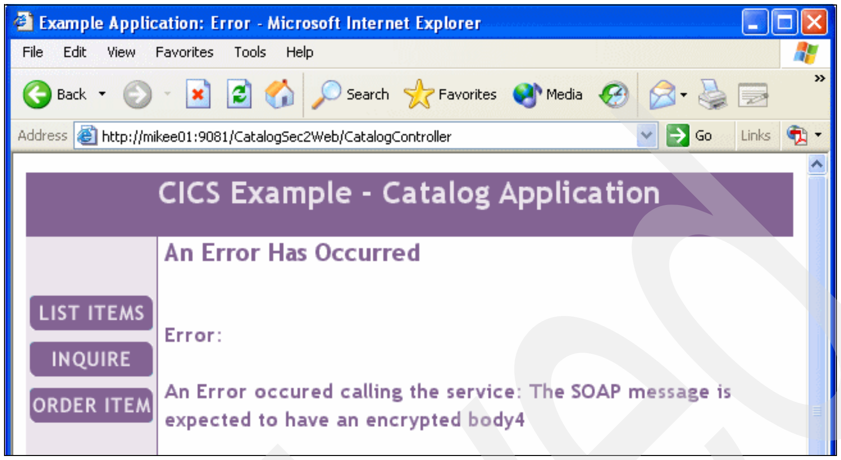
- The <wsse:BinarySecurityToken> contains the base64binary encoding of the WebSphere certificate. This includes the public key that WebSphere uses to decrypt the secret key.
- <wsse:SecurityTokenReference> is a child element of the <ds:KeyInfo> element which contains a reference to the BinarySecurityToken.

The rest of the SOAP headers in the response message are similar to those in the request message.

SOAP fault messages

In this section we show some of the SOAP faults that you may see when testing encryption processing.

- Figure 10-48 shows the browser response when CICS is expecting an encrypted message but the service requester does not encrypt the request.


Example Application: Error - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://mikee01:9081/CatalogSec2Web/CatalogController

CICS Example - Catalog Application

LIST ITEMS
INQUIRE
ORDER ITEM

An Error Has Occurred

Error:

An Error occured calling the service: The SOAP message is expected to have an encrypted body4

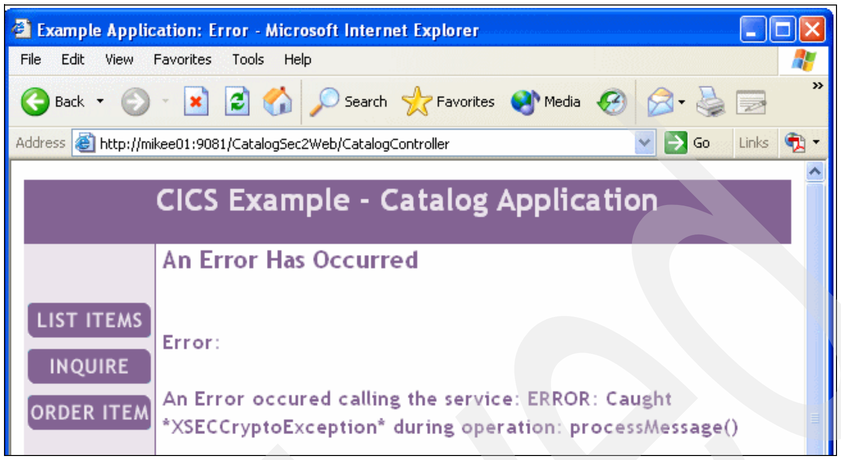
Figure 10-48 DFHWSSE1 SOAP fault: X.509 certificate not known

The SOAP fault for this message, shown in Example 10-27, highlights that CICS is expecting an encrypted message.

Example 10-27 DFHWSSE1 SOAP fault - X.509 certificate not known

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>wsse:FailedCheck</faultcode>
      <faultstring>INVALID_SOAP_REQUEST</faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
          <message>The SOAP message is expected to have an encrypted body
          </message>
          <errorcode>4</errorcode>
        </e:myfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- Figure 10-49 shows the browser response when CICS is attempting to decrypt an encrypted message but ICSF is not available.


Example Application: Error - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://mikee01:9081/CatalogSec2Web/CatalogController
Go Links
CICS Example - Catalog Application
LIST ITEMS
INQUIRE
ORDER ITEM

An Error Has Occurred

Error:

An Error occured calling the service: ERROR: Caught *XSECCryptoException* during operation: processMessage()

Figure 10-49 DFHWSSE1 SOAP fault: ICSF not available

The SOAP fault for this message, shown in Example 10-28, shows that there is an error creating RSA key token.

Example 10-28 DFHWSSE1 SOAP fault - ICSF not available

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>wsse:FailedCheck</faultcode>
      <faultstring>ERROR: Caught *XSECCryptoException* during operation:
        processMessage()</faultstring>
      <detail>
        <e:myfaultdetails xmlns:e="http://www.ibm.com/software/htp/cics/wssec">
          <message>ZosCryptoKeyRSA::privateDecrypt - Error encoding under RSA
            key</message>
          <errorcode>1</errorcode>
        </e:myfaultdetails>
      </detail>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Part 4

Transaction management

In this part, we begin by providing an introduction to Web services atomic transactions. We then outline the steps for enabling WS-Atomic Transaction (WS-AT) support in CICS. Finally, we show several scenarios that demonstrate how you can synchronize WebSphere and CICS updates using the WS-AT standard.

11 Introduction to Web services: Atomic transactions

We begin this chapter by describing an example of a classic transaction. We expect that most of our readers will be familiar with examples similar to ours. Then we map our classical transaction to a Web services *atomic transaction* as a means of introducing the new terminology that we use in this and succeeding chapters: *coordinator*, *transactional context*, *activation service*, *registration service*, and *completion protocol*. We continue by providing an overview of the flows and message contents prescribed by the three specifications upon which CICS TS V3.1 support for atomic transactions is built:

- Web Services - Addressing (WS-Addressing or WS-A)
- Web Services - Coordination (WS-Coordination or WS-C)
- Web Services - Atomic Transaction (WS-Atomic Transaction or WS-AT)

11.1 Beginner's guide to atomic transactions

We begin by describing an example of a classic transaction. For this discussion we borrow freely from the paper *Tour Web Services Atomic Transaction operations: Beginner's guide to classic transactions, data recovery, and mapping to WS-Atomic Transactions* which Thomas Freund and Daniel House published on September 2, 2004, on the IBM developerWorks Web site at:

```
http://www-128.ibm.com/developerworks/webservices/library/ws-introwsat
```

Not losing money is quite important. Just ask Waldo. Waldo's situation typifies the need for a transaction. Waldo uses a Web browser or an Automatic Teller Machine (ATM) to move some money from one account to another account. These accounts may be in different branches of the same financial institution, or they may be in different institutions.

It is never acceptable to Waldo for his money to disappear. Should Waldo ever doubt the safety of his money, he would probably switch financial institutions.

Waldo's money is represented by data in two databases that cooperate to ensure that the data they contain is always in a known and consistent state. That is, these two databases allow actions or tasks between them to be within a common activity or work scope as shown in Figure 11-1. Put yet another way, a single transaction can manipulate data in both databases and *something* will guarantee that only one of two possible outcomes occurs: *all* the changes are successfully made or *none* of them is made at all.

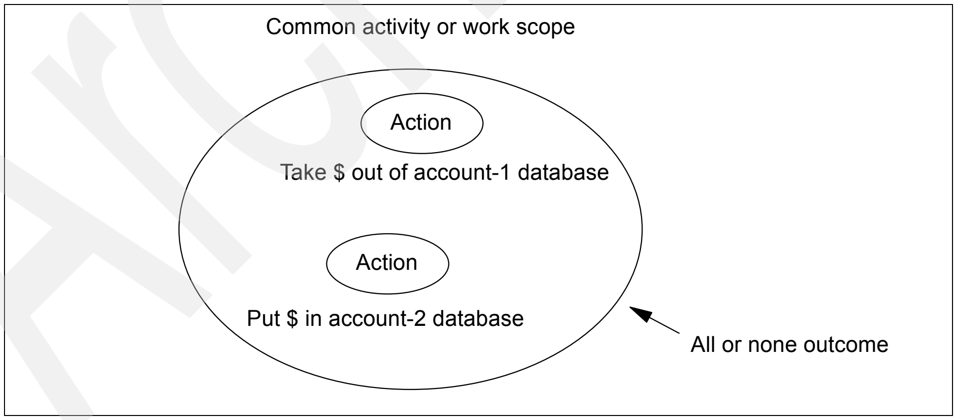

Common activity or work scope
Action
Take $ out of account-1 database
Action
Put $ in account-2 database
All or none outcome

Figure 11-1 Common activity encompasses various recoverable actions

The *something* that guarantees the common outcome of all the actions is a protocol supported by both databases, and some supporting middleware. The protocol the databases use to keep data (such as Waldo's balances) coordinated is called *two phase commit*, or simply 2PC. Our example uses a common variation of 2PC called *presumed abort*, where the default behavior in the absence of a successful outcome is to rollback or undo all the actions in the activity.

From a programming perspective, there are different ways to specify that multiple actions should be within the scope of a single transaction. One particularly clear way to specify transactional behavior is shown in Example 11-1. The code is the small piece of logic running somewhere behind the ATM Waldo is using–perhaps in the data center of one of the financial institutions involved.

Example 11-1 Pseudo-code for Waldo's transaction

```
TransferCash(fromAcct, toAcct, amount)
    BeginTransaction
    fromAcct = fromAcct - amount
    toAcct   = toAcct + amount
    CommitTransaction
Return
```

11.1.1 What is a classic transaction

For our simple purposes, a *recoverable* action is anything that modifies protected data. For example, taking money out of one of Waldo's accounts (fromAcct = fromAcct - amount) is a recoverable action that can be reversed up to the end of the transaction. A *classic* transaction, then, is just a grouping of recoverable actions, the guaranteed outcome of which is that either all the actions are taken, or none of them is taken (see Figure 11-1).

In Waldo's case, his transaction is composed of two actions: taking money out of one account and putting money into another account. It's okay for both of these actions to occur, and it's even okay if neither of these actions occurs. It's never okay for one action to occur without the other also occurring, which would result in corrupt data and either Waldo's net worth or the bank's assets disappearing or appearing from nowhere. Hence, both actions need to be within a single transaction with a single outcome: either both actions occur (a *commit* outcome), or neither action occurs (a *rollback* outcome).

Assuming no errors happen, the code in Example 11-1 shows that a commit outcome is desired. The code could just as easily have specified rollback instead of commit (for when Waldo presses the Cancel key on the ATM), which means reverse all actions in the transactional work scope (between beginning and end). The transaction monitor, which is the underlying middleware helping the code in Example 11-1 support transaction processing, would automatically specify rollback if the program suffered an unhandled exception. Such an automatic rollback on the part of the transaction monitor is a protection mechanism to make sure that data is not corrupted. For example, even if the ATM application fails unexpectedly, the middleware will “clean up” and guarantee the outcome.

Now let’s see how one common variant of 2PC, *presumed abort*, can be used to effect Waldo’s transaction and move money from one account to another in a recoverable way. A key part of this illustration is to see that no matter what kind of failure occurs, data integrity is preserved and Waldo remains a loyal customer.

Figure 11-2 shows Waldo’s transaction on a time line with all of the interacting components needed to execute the logic shown in Example 11-1.

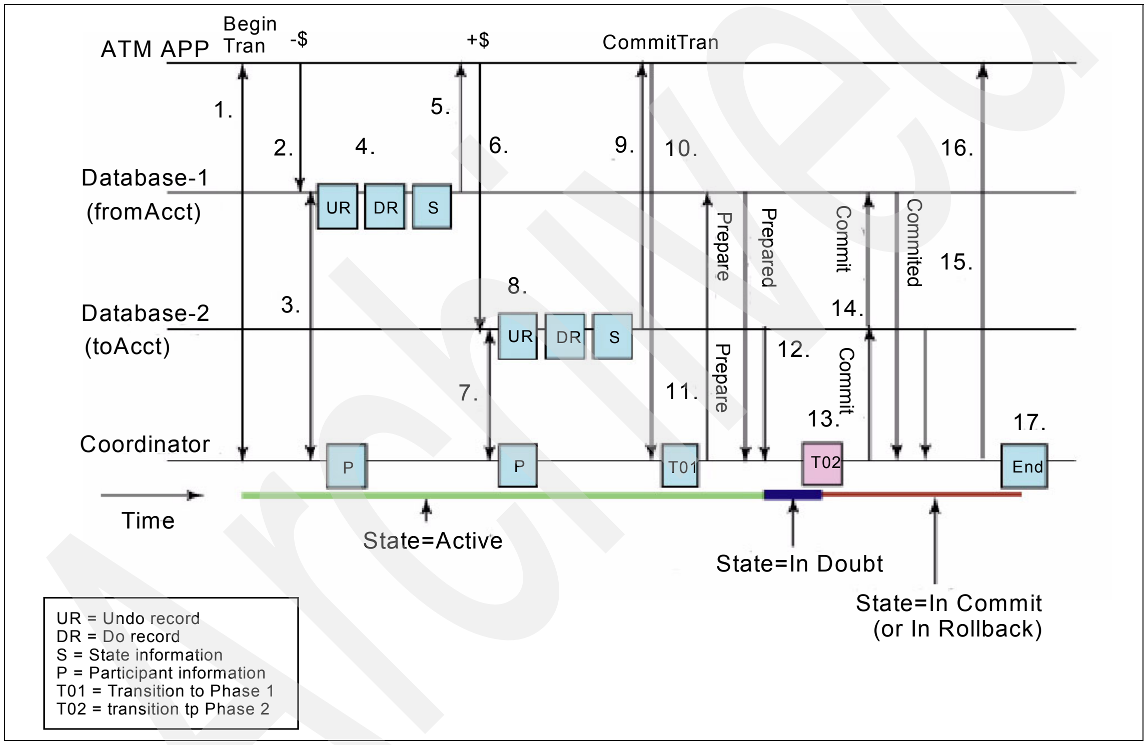


Figure 11-2 Behind the scenes of Waldo’s ATM transaction

The top line represents the ATM application itself. The next two lines represent the account databases that the application manipulates. The databases will be transactional participants. The next line is a transactional coordinator, or middleware, which will orchestrate the 2PC protocol. The line at the very bottom indicates the state of Waldo’s transaction at different points in time. The state of the transaction dictates recovery processing in the event of a failure.

The lines for Database-1, Database-2, and Coordinator represent both time (flowing left to right) and also some key records recorded onto a recovery log. These records include images of the data before it is modified (Undo records), images of the data after it has been modified (Do records), and state information. The recovery log is used to insure data integrity during recovery processing.

Now let's walk through Waldo's transaction. In the following discussion, when we talk about *the ATM application*, take it to mean either the application itself, or some middleware supporting the application. For example, when we say the application begins a transactional scope, it could be that middleware begins the transactional scope on behalf of the application.

Here is narration to explain the numbered steps shown in Figure 11-2:

1. The ATM application indicates the beginning of a transactional scope. The Coordinator creates a context for this transaction; the context includes a unique identifier and some other information about the transaction. Importantly, this transaction context flows back to the application. The context flows with other interactions between the application and resource managers; it is the context that helps glue together a whole set of actions into one transactional activity.
2. The application takes money out of Database-1. The context (from step 1) is inserted into this flow.
3. Database-1 sees the request for action, but also sees the transactional context. Database-1 uses this context to contact the transactional Coordinator and register interest in this transaction or activity (so that the Coordinator will help Database-1 through 2PC processing later to guarantee a commit or rollback outcome of all actions). The Coordinator remembers that Database-1 is a participant in the transaction.
4. Database-1 looks at the request to modify recoverable data. It writes records to a recovery log, plus transaction state information. One record describes the database change to be made if the decision later is to commit (the Do record). The other record describes the database change to be made if the decision is to rollback (the Undo record).
   - In this case, the Undo record says *make Waldo's balance = x* and the Do record says *make Waldo's balance = x - $*. (X is the amount of the balance before this transaction ever started and $ is the amount to transfer). Notice that we are only looking at the recovery log – not database files.
   - The Do records are not strictly required if Database-1 makes database file updates when the application requests it to, instead of waiting. However, waiting to write the data can have advantages for performance and concurrency. In addition, the Do records may be used for audit or other advanced reasons. Since they are so useful, our example databases use them.

5. Return to the application.
6. Similarly to step 2, the application makes a request to manipulate the other database, Database-2. The application wants to add in the amount taken out of Database-1.
7. Database-2 registers interest in the transaction with the Coordinator the same way Database-1 did. The Coordinator remembers that Database-2 is a participant in the transaction.
8. Database-2 writes Undo and Do records and state information to its recovery log, again just as Database-1 did.
9. Return to the application.
10. The application chooses to commit the transaction. The Coordinator now takes over. When `Commit` is received, the Coordinator writes a log record indicating that Phase 1 of 2PC has begun.
11. In Phase 1, the Coordinator goes down the list of all participants (Database-1 and Database-2 in this example) who expressed interest in this transaction, asking each one to `Prepare`. Prepare means get ready to receive the order to either commit or rollback.
12. Database-1 and Database-2 both respond with `Prepared`, meaning that they are ready to be told the final outcome (commit or rollback all the changes made) and support it.
    - They must have committed something (at least on their logs) by this point, because responding `Prepared` means they guarantee being able to commit or rollback when told - actions up to this point were just tentative.
    - If either Database had some kind of failure preparing, it would respond `Aborted` instead of `Prepared`, and the Coordinator would broadcast `Rollback` to all participants.
13. The Coordinator forces a log record indicating a Transition to Phase 2 (T02).
    - Once this record is hardened on a log, we know and have recorded that:
        - All participants are prepared to go either way (commit or rollback).
        - The ultimate outcome of the transaction is known (commit in our example).
        - The outcome is guaranteed by recovery processing.
    - If this record fails to make it to the log for any reason, the ultimate outcome will be to rollback (we are using *presumed abort* in this example). The recovery processing will enforce the outcome.
14. The Coordinator informs each participant that the decision is to commit the changes. The participants can then do whatever they need to do, such as perhaps writing the results to the real database data.

15. The participants return to the Coordinator with **`Committed`**. Once the Coordinator knows that all the participants acknowledged the **`Commit`** order with **`Committed`**, it can forget about this transaction because the transaction was acknowledged by all to be done.
16. Return to the application.
17. At some point, since it knows the participants have succeeded in the 2PC flow by acknowledging the common outcome, the Coordinator writes an *End* indicator on its log.

11.1.2 Mapping from classic transactions to WS-Atomic Transaction

In Figure 11-2 on page 412 we did not mention how Database-1 contacted the Coordinator, nor did we specify how the application called the databases. In fact, we didn't specify the mechanisms for anything to contact anything else. In the past, these were mostly non-universal mechanisms that sometimes only worked between certain combinations of entities (applications, resource managers, and coordinators or transaction monitors).

The combination of Web services, WS-Coordination (WS-C) and WS-Atomic Transaction (WS-AT) maps all of the flows shown in Figure 11-2 on page 412 and specifies precise communications mechanisms for achieving the same results. However, instead of only working between certain combinations, the Web services based flows can work with just about anything.

Figure 11-3 illustrates how the classic flows are converted to Web services. Significantly changed steps are described following the figure. As before, when we say *application*, take it to mean the application or some helper middleware. Likewise, when we say *database*, it might mean the actual database, or some helper middleware.

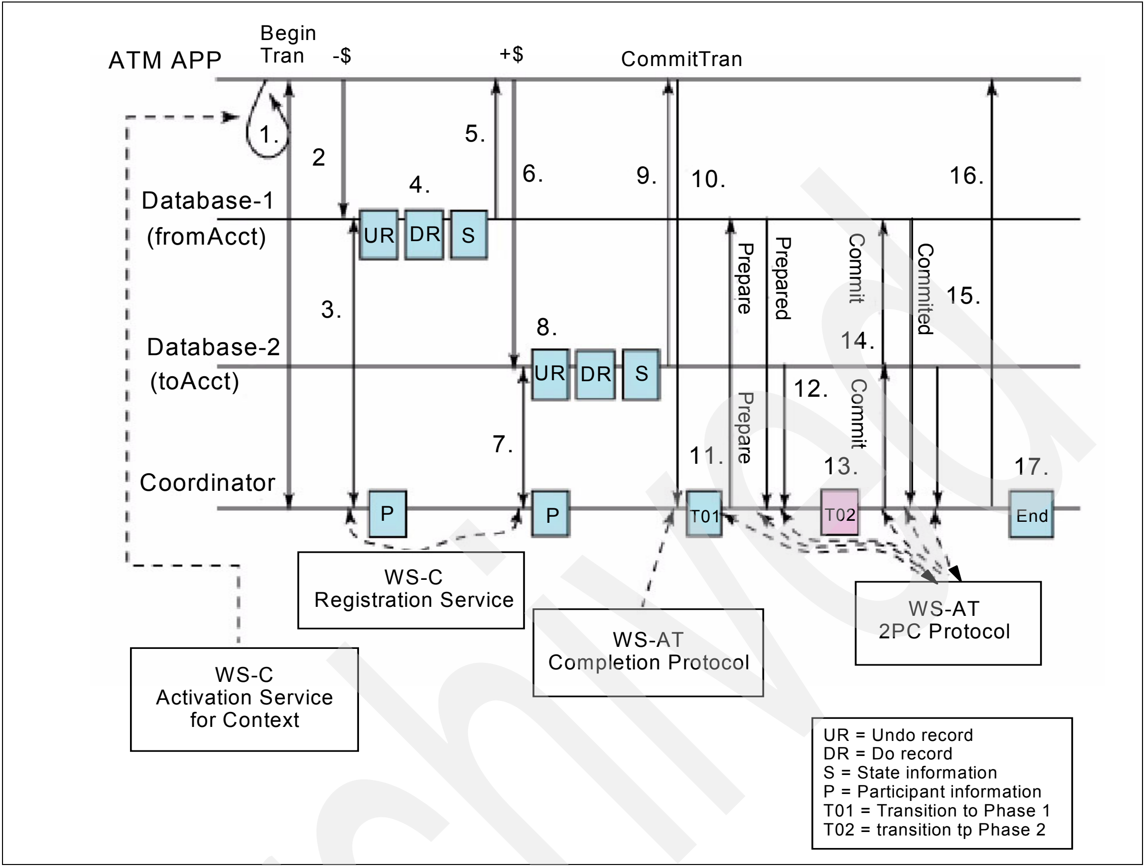


Figure 11-3 Waldo's transaction revisited

1. The application uses the Activation Service defined in WS-C to obtain a transactional context.
2. The application invokes a Web service exposed by Database-1 (alternatively, exposed by an application server that then talks to Database-1) to subtract money from Waldo's balance. The context flows along with the Web service invocation, although the application is not aware of that.
3. Database-1 uses information in the context to invoke the Registration Service defined in WS-C to register interest in this transaction.
4. No change.
5. No change.
6. The application invokes a Web service exposed by Database-2 to add money to Waldo's balance. Just like in step 2, the context flows along with the Web service invocation.

7. Just like step 3, Database-2 uses information in the context to invoke the Registration Service and register interest in this transaction.
8. No change.
9. No change.
10. The application uses the Completion Protocol defined in WS-AT to indicate that it wishes to commit the transaction.
11. to 15. Databases and the Coordinator participate in 2PC flows as defined in the WS-AT 2PC Protocol.

From Figure 11-3 it is clear that atomic transactions using Web services (WS-C and WS-AT) are substantially the same as without Web services (Figure 11-2 on page 412, for example). The primary differences are almost cosmetic from the outside and involve *how entities communicate with each other, not the substance of what they communicate*. However, these differences in how the entities communicate have a big impact on flexibility and interoperability.

You can achieve universal interoperability with Web services because instead of changing resource manager X to interoperate with transaction monitor Y, you can change both X and Y to use Web services and then interoperate with many other resource managers and transaction monitors. So instead of two-at-a-time interoperability, or interoperability only within a specific kind of domain, n-way universal interoperability is possible.

Recovery processing using Web services between the interested parties is the same as before Web services. Resource managers are the only ones who know their resources and how to commit them or roll them back.

As an example, suppose that Database-1 fails between steps 5 and 6 in Figure 11-3. Database-1 comes back up and, just like before Web services, it reads its log, notices that it has an incomplete transaction, and realizes that it needs to contact the Coordinator. Information about how to contact the Coordinator is in the state saved on its recovery log; with Web services it will be an endpoint reference (as defined in WS-Addressing; see “Endpoint references” on page 419). Database-1 contacts the Coordinator at that endpoint reference with a message defined in WS-AT called **Replay**. Replay causes the Coordinator to resend the last protocol message to Database-1, which lets Database-1 deduce the transaction state and then apply the appropriate recovery rule. In our example the Coordinator tells Database-1 that it has no knowledge of this transaction. Database-1 therefore applies its Undo record, making the data consistent again.

Important: WS-AT is a two-phase commit transaction protocol that is suitable for short duration transactions only. WS-AT is well suited for distributed transactions within a single enterprise, but is it is generally not recommended that WS-AT transactions be distributed across enterprise domains. Inter-enterprise transactions typically require a looser semantic than two-phase commit.

11.2 WS-Addressing

Figure 11-3 on page 416 shows several messages flowing:

- The application sends a message to the Activation Service asking for a transactional context.
- The Activation Service sends a response containing a transactional context to the application.
- Database-1 and Database-2 each send a `Register` message to the Registration Service and receives a reply.
- The application sends a `Commit` message to the Coordinator.
- The Coordinator sends `Prepare` and `Commit` messages to Database-1 and Database-2.
- Database-1 and Database-2 each send `Prepared` and `Committed` messages to the Coordinator.

The application, Database-1, Database-2, the Activation Service, the Registration Service, and the Coordinator are endpoints for these messages. As with messages in the everyday business world, we need a way to identify the recipient of each message, the sender of the message, what previous message (if any) the message relates to, what action we want the recipient to take, and where the recipient should send the reply (if any) to the message. Furthermore, we want to be able to do this in a way that does not depend on the transport mechanism (such as HTTP or WebSphere MQ) that we use to send the message.

To this end IBM, Microsoft, Sun™ Microsystems, BEA, and SAP® formally submitted the *Web Services- Addressing (WS - Addressing)* specification to the World Wide Web Consortium (W3C) on 10 August 2004. You can find this specification at:

`http://www.w3.org/Submission/ws-addressing`

Note: As the specification has moved through the W3C standards process, it has been divided into three parts:

- Web Services Addressing 1.0 - Core

  This is currently a W3C Candidate Recommendation dated 17 August 2005.

- Web Services Addressing 1.0 - SOAP Binding

  This is also currently a W3C Candidate Recommendation dated 17 August 2005.

- Web Services Addressing 1.0 - WSDL Binding

  This is currently a W3C Working Draft dated 13 April 2005.

However, the WS-Coordination and WS-Atomic Transaction specifications that we discuss later in this chapter are based on the 10 August 2004 specification, and, therefore, so is CICS TS V3.1 support for Web services.

All information items defined by the 10 August 2004 specification are identified by the XML namespace URI:

```
http://schemas.xmlsoap.org/ws/2004/08/addressing
```

We associate the namespace prefix *wsa* with this namespace by using the attribute

```
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
```

The specification defines two constructs:

- Endpoint references
- Message information headers

11.2.1 Endpoint references

A Web service *endpoint* is a (referenceable) entity, processor, or resource to which Web services messages can be addressed.

An *endpoint reference* conveys the information needed to address a Web service endpoint. As we shall see later:

- An endpoint reference for the Registration Service forms part of the coordination context.
- An endpoint reference for the participant's Protocol Service forms part of the Register request.

- An endpoint reference for the coordinator's Protocol Service forms part of the response to a Register request.

Example 11-2 shows the pseudo schema for an `EndpointReference` element.

Example 11-2 Pseudo schema for EndpointReference element

```
<wsa:EndpointReference>
   <wsa:Address>.......................</wsa:Address>
   <wsa:ReferenceProperties>...........</wsa:ReferenceProperties>
   <wsa:ReferenceParameters>...........</wsa:ReferenceParameters>
   <wsa:PortType>......................</wsa:PortType>
   <wsa:ServiceName PortName="...">.....</wsa:ServiceName>
   <wsp:Policy>........................</wsp:Policy>
</wsa:EndpointReference>
```

For each child element of EndpointReference, Table 11-1 describes what the element contains, the minimum number of times the element can be used, and the maximum number of times the element can be used.

Table 11-1 Children of the Endpoint Reference element

Element	Description	Min	Max
Address	Contains an address URI that identifies the endpoint. This may be a network address or a logical address.	1	1
ReferenceProperties	Contains child elements each of which represents an individual reference property. The number of child elements is not limited.	0	1
ReferenceParameters	Contains child elements each of which represents an individual reference parameter. The number of child elements is not limited.	0	1
PortType	Contains the name of the primary portType of the endpoint being conveyed.	0	1
ServiceName	Contains the name of the WSDL `<service>` element that contains a WSDL description of the endpoint being referenced. The service name provides a link to a full description of the service endpoint. The ServiceName element may optionally have a PortName attribute which specifies the name of the specific WSDL <port> definition in that service which corresponds to the endpoint being referenced.	0	1
Policy	Contains an XML policy element as described in WS-Policy that describes the behavior, requirements, and capabilities of the endpoint.	0	No limit

Here is the difference between a reference property and a reference parameter:

- A *reference property* is required to identify the endpoint. It is required to properly dispatch a message to an endpoint at the endpoint side of the interaction.
- A *reference parameter* is required to facilitate a particular interaction with the endpoint. It is required to properly interact with the endpoint.

Tip: In "Transaction scenarios" on page 463 we look at the messages that flow between CICS TS V3.1 and WebSphere Application Server V6.0. In these messages we see only the following children of the `EndpointReference` element:

- Address
- ReferenceProperties

Example 11-3 shows an endpoint reference for the Registration Service running in a CICS TS V3.1 region that is monitoring port 15301 on a z/OS system whose IP address is MVSG3.mop.ibm.com. The endpoint reference has two reference properties: `UOWID` and `PublicID`. We replaced the ending characters of the `PublicID` property with ... for brevity.

Example 11-3 Sample EndpointReference element for Registration Service in CICS

```
<wsa:EndpointReference
      xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
      xmlns:cicswsat="http://www.ibm.com/xmlns/prod/CICS/pipeline">
   <wsa:Address>
       http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
   </wsa:Address>
   <wsa:ReferenceProperties>
      <cicswsat:UOWID>BDFC52CD7C57D466</cicswsat:UOWID>
      <cicswsat:PublicId>310FD7E2E2...</cicswsat:PublicId>
   </wsa:ReferenceProperties>
</wsa:EndpointReference>
```

Example 11-4 shows an endpoint reference for the Registration Coordinator Port running in a WebSphere Application Server V6.0 region that is monitoring port 9080 on a Windows system whose IP address is 9.100.199.156. The endpoint reference has two reference properties: `txID` and `instanceID`. We replaced the ending characters of these properties with ... for brevity.

Example 11-4 Sample EndpointReference element for Registration Service in WAS

```
<wsa:EndpointReference
   xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
   xmlns:websphere-wsat="http://wstx.Transaction.ws.ibm.com/extension">
   <wsa:Address>
      http://9.100.199.156:9080/_IBMSYSAPP/wscoor/services/Registration
      CoordinatorPort
   </wsa:Address>
   <wsa:ReferenceProperties>
      <websphere-wsat:txID>
           com.ibm.ws.wstx:00000107d70ad26c0000000700000005cbb17a7d5f7e...
      </websphere-wsat:txID>
      <websphere-wsat:instanceID>
           com.ibm.ws.wstx:00000107d70ad26c0000000700000005cbb17a7d5f7e...
      </websphere-wsat:instanceID>
   </wsa:ReferenceProperties>
</wsa:EndpointReference>
```

11.2.2 Message information headers

Example 11-5 shows pseudo schema for the message information headers defined in the WS-Addressing specification.

Example 11-5 Pseudo schema for message information header elements

```
<wsa:MessageID>.........................</wsa:MessageID>
<wsa:RelatesTo RelationshipType="...">...</wsa:RelatesTo>
<wsa:To>................................</wsa:To>
<wsa:Action>............................</wsa:Action>
<wsa:From>.............................</wsa:From>
<wsa:ReplyTo>...........................</wsa:ReplyTo>
<wsa:FaultTo>...........................</wsa:FaultTo>
```

Table 11-2 describes what each message information header element contains, the minimum number of times the element may be used, and the maximum number of times the element may be used.

Table 11-2 Message information header elements

Element	**Description**	**Min**	**Max**
MessageID	Contains a URI that uniquely identifies this message in space and time.	0 (but see Note 1)	1
RelatesTo	Contains a URI that corresponds to a related message's MessageID property. The RelatesTo element has an optional RelationshipType attribute that indicate the type of relationship this message has to the related message. The specification defines one relationship type, namely wsa:Reply. When absent, the implied value of this attribute is wsa:Reply.	0	No limit

Element	Description	Min	Max
To	Contains a URI that specifies the address of the intended receiver of this message.	1	1
Action	Contains a URI that uniquely identifies the semantics implied by this message.	1	1
From	Contains an endpoint reference that identifies the endpoint from which the message originated.	0	1
ReplyTo	Contains an endpoint reference that identifies the intended receiver for replies to this message.	0 (but see Note 2)	1
FaultTo	Contains an endpoint reference that identifies the intended receiver for faults related to this message.	0 (see also Note 3)	1

Table notes:

1. If ReplyTo or FaultTo is present, MessageID must be present.
2. If a reply is expected, ReplyTo *must* be present.
3. When formulating a fault message, the sender *must* use the contents of the FaultTo element of the message to which the Fault reply is being sent. If the FaultTo element is absent, the sender *may* use the contents of the ReplyTo element to formulate the fault message. If both the FaultTo element and the ReplyTo element are absent, the sender *may* use the contents of the From element to formulate the fault message. The FaultTo element may be absent if the sender cannot receive fault messages.

Example 11-6 shows an example of a set of message information headers.

Example 11-6 Sample message information header elements

```
<wsa:To>
   http://9.100.199.156:9080/_IBMSYSAPP/wscoor/services/Registration
   CoordinatorPort
</wsa:To>
<wsa:Action>
   http://schemas.xmlsoap.org/ws/2004/10/wscoor/Register
</wsa:Action>
<wsa:MessageID>PIAT-MSG-A6POT3C1-003342266297785C</wsa:MessageID>
<wsa:ReplyTo>
   <wsa:Address>
      http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
   </wsa:Address>
   <wsa:ReferenceProperties>
      <cicswsat:UOWID>BDFC52CD7C57D466</cicswsat:UOWID>
      <cicswsat:PublicId>
```

```
         310FD7E2E2...
      </cicswsat:PublicId>
   </wsa:ReferenceProperties>
</wsa:ReplyTo>
```

The To header shows that the message is being sent to the Registration Coordinator Port running in a WebSphere Application Server V6.0 region that is monitoring port 9080 on a Windows system whose IP address is 9.100.199.156.

The Action header indicates that the sender wishes to register with the Registration Coordinator Port.

The ID of the message, PIAT-MSG-A6POT3C1-003342266297785C, uniquely identifies the message in space and time:

- A6POT3C1 is the VTAM® APPLID of the CICS TS V3.1 region that sent the message.
- 003342266297785C is the abstime value returned by an EXEC CICS INQUIRE TIME issued in that CICS TS V3.1 region.

The ReplyTo header shows that the reply to this message should be sent to the Registration Service running in a CICS TS V3.1 region that is monitoring port 15301 on a z/OS system whose IP address is MVSG3.mop.ibm.com.

11.2.3 SOAP binding for endpoint references

When a SOAP message must be addressed to an endpoint, the information contained in the endpoint reference is mapped to the SOAP message by the following two rules:

- The contents of the Address element in the endpoint reference is copied to the To message information header of the SOAP message.
- Each reference property or reference parameter is added as a header block in the new message.

Example 11-7 shows how we use these rules to address a message to the CICS Registration Service endpoint whose endpoint reference is given in Example 11-3 on page 421.

Example 11-7 SOAP message addressed to CICS RegistrationService endpoint

```
<S:Envelope xmlns:S=”...” xmlns:wsa=”...” xmlns:cicswsat=”...”>
   <S:Header>
       ...
       <wsa:To>
           http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
```

```
        </wsa:To>
        <cicswsat:UOWID>BDFC52CD7C57D466</cicswsat:UOWID>
        <cicswsat:PublicId>310FD7E2E2...</cicswsat:PublicId>
        ...
    </S:Header>
    <S:Body>
      ...
    </S:Body>
</S:Envelope>
```

11.3 WS-Coordination

In Section 11.1, “Beginner’s guide to atomic transactions” on page 410 and Section 11.2, “WS-Addressing” on page 418 we see that the application, Database-1, Database-2, the Activation service, the Registration service, and the Coordinator are endpoints for messages. We also see that:

- The application sends a message to the Activation service asking for a transactional context.
- The Activation service sends a response containing a transactional context to the application.
- Database-1 and Database-2 each sends a **Register** message to the Registration service and receives a reply.

We still need to define what these messages should contain.

For this purpose IBM, Microsoft, and BEA published the *Web Services-Coordination (WS - Coordination)* specification in September of 2003; they updated it in November of 2004. Arjuna Technologies Ltd., Hitachi Ltd., and IONA Technologies joined IBM, Microsoft, and BEA in publishing *Web Services-Coordination (WS-Coordination) Version 1.0* in August of 2005. You can find these at:

```
http://www-128.ibm.com/developerworks/library/specification/ws-tx
```

Although CICS TS V3.1 was developed and tested when the November, 2004 version was the current version, the nature of the differences between the November, 2004 version and the August, 2005 version is such that it may accurately be said that CICS TS V3.1 also supports the August, 2005 version.

All information items defined by the November, 2004 and August, 2005 versions are identified by the XML namespace URI:

```
http://schemas.xmlsoap.org/ws/2004/10/wscoor
```

We associate the namespace prefix `wscoor` with this namespace by using the attribute:

```
xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/wscoor"
```

The specification defines:

- A coordination service
- The following messages:
  - CreateCoordinationContext
  - CreateCoordinationContextResponse
  - Register
  - RegisterResponse

11.3.1 Coordination service

As shown in Figure 11-4, a *Coordination service* (or *Coordinator*) is an aggregation of the following services:

- Activation service

  When the application sends a `CreateCoordinationContext` element, the Activation service creates a new activity and returns its coordination context in a `CreateCoordinationContextResponse` element.

  The Coordination service may, but does not have to, support the Activation service.

  Note: Some products provide this as an external service, for others to call. CICS does not do this, and only supports the creation of coordination contexts internally, for use by the workloads that it manages.

- Registration service

  The Registration service defines a **`Register`** operation that allows a Web service to register to participate in a coordination protocol.

  The Coordination service must support the Registration service.

- A set of coordination Protocol services for each supported coordination type.

  These are defined in the specification that defines the coordination type (for example, in the WS-Atomic Transaction specification).

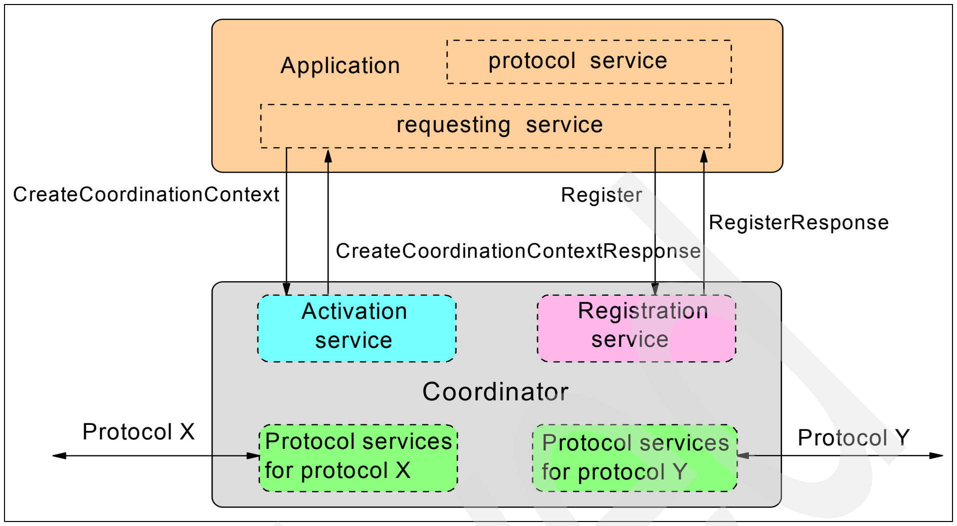


Figure 11-4 A Coordination service (or Coordinator)

11.3.2 CreateCoordinationContext

Example 11-8 shows the pseudo schema for the `CreateCoordinationContext` element.

Example 11-8 Pseudo schema for CreateCoordinationContext element

```
<wscoor:CreateCoordinationContext...>
   <wscoor:CoordinationType>...........</wscoor:CoordinationType>
   <wscoor:Expires>....................</wscoor:Expires>
   <wscoor:CurrentContext>.............</wscoor:CurrentContext>
</wscoor:CreateCoordinationContext>
```

For each child element of `CreateCoordinationContext`, Table 11-3 describes what the element contains, the minimum number of times the element can be used, and the maximum number of times the element can be used.

Table 11-3 Children of the CreateCoordinationContext element

Element	Description	Min	Max
CoordinationType	The unique identifier for the desired coordination type for the activity.	1	1
Expires	The expiration for the returned CoordinationContext expressed as an unsigned integer in milliseconds. Specifies the earliest point in time at which a transaction may be terminated solely due to its length of operation.	0	1
CurrentContext	The current Coordination Context.	0	1

Currently there are two specifications that define coordination types:

- WS-Atomic Transaction

  This specification defines the coordination type for atomic transactions where the results of operations are not made visible until the completion of the unit of work:

  `http://schemas.xmlsoap.org/ws/2004/10/wsat`

- WS-Business Activity

  This specification defines two coordination types for business activities:

  `http://schemas.xmlsoap.org/ws/2004/10/wsba/AtomicOutcome`

  `http://schemas.xmlsoap.org/ws/2004/10/wsba/MixedOutcome`

  Business activities have the following characteristics:

  – A business activity may consume many resources over a long duration.

  – There may be a significant number of atomic transactions involved.

  – Individual tasks within a business activity can be seen prior to the completion of the business activity; their results may have an impact outside of the computer system.

  – Responding to a request may take a very long time. Human approval, assembly, manufacturing, or delivery may have to take place before a response can be sent.

  – In the case where a business exception requires an activity to be logically undone, abort is typically not sufficient. Exception handling mechanisms may require business logic, for example in the form of a compensation task, to reverse the effects of a previously completed task.

  For example, selling a vacation is a business activity that may involve the travel agent in actions such as recording customer details, booking seats on an aircraft, booking a hotel, booking a rental car, invoicing the customer, checking for receipt of payment, processing the payment, and arranging foreign currency.

Table 11-4 compares an atomic transaction with a business activity.

Table 11-4 Comparison of features of atomic transaction and business activity

Atomic transaction	Business activity
Short duration	Longer duration
Locks	Avoid locks
Suited for a more controlled environment	Suited for a loosely coupled environment
Classical resource manager mapping - think database (not business processes crossing business boundaries)	Business process mapping
Easier to think about and program ▶ Rollback or Commit ▶ Automatic rollback in case of abnormal termination	More complex ▶ Compensate
All resource managers move in one direction (everybody commits or rolls back in unison)	More flexible resource manager participation. They don't have to trust applications so much.

Note: CICS TS V3.1 does not support the WS-Business Activity specification. At this time there are no plans for future releases of CICS to support it either.

Example 11-9 shows an example of a `CreateCoordinationContext` element in which the coordination type is WS-Atomic Transaction.

Example 11-9 Sample CreateCoordinationContext element

```
<wscoor:CreateCoordinationContext>
   <wscoor:CoordinationType>
      http://schemas.xmlsoap.org/ws/2004/10/wsat
   </wscoor:CoordinationType>
   <wscoor:Expires>5000</wscoor:Expires>
</wscoor:CreateCoordinationContext>
```

11.3.3 CreateCoordinationContextResponse

The `CreateCoordinationContextResponse` element contains the `CoordinationContext` element.

The `CoordinationContext element` contains four elements:

- A URI which identifies the `CoordinationContext`.
- The `CoordinationType`.
- The expiration period for the `CoordinationContext`.
- An endpoint reference for the Registration Service which is part of this Coordination service. Recall from “Endpoint references” on page 419 that this means that the `CoordinationContext` must contain the address of the Registration Service and may contain reference properties for the Registration Service.

Example 11-10 shows the pseudo schema for the `CreateCoordinationContextResponse` element.

Example 11-10 Pseudo schema for CreateCoordinationContext Response element

```
<wscoor:CreateCoordinationContextResponse>
   <wscoor:CoordinationContext>
      <wscoor:Identifier>.......................</wscoor:Identifier>
      <wscoor:CoordinationType>.................</wscoor:CoordinationType>
      <wscoor:Expires>..........................</wscoor:Expires>
      <wscoor:RegistrationService>..............</wscoor:RegistrationService>
   </wscoor:CoordinationContext>
</wscoor:CreateCoordinationContextResponse>
```

The application will place the `CoordinationContext` element within an application message to pass the coordination information to other parties. Conveying a `CoordinationContext` on an application message is commonly referred to as *flowing* the context. When a context is flowed as a SOAP header, the header must have the `mustUnderstand` attribute and the value of the `mustUnderstand` attribute must be `true`.

Note: When an application flows the context, it passes the address of its Registration Service and the coordination type.

Example 11-11 shows a `CoordinationContext` created by a CICS TS V3.1 region.

Example 11-11 Sample CoordinationContext created by CICS

```
<wscoor:CoordinationContext>
   <wscoor:Identifier>
      PIAT-CCON-A6POT3C1-003322404576825C
   </wscoor:Identifier>
   <wscoor:CoordinationType
      http://schemas.xmlsoap.org/ws/2004/10/wsat
   </wscoor:CoordinationType>
   <wscoor:RegistrationService>
      <wsa:Address>
         http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
      </wsa:Address>
      <wsa:ReferenceProperties>
         <cicswsat:NetName>A6POT3C1</cicswsat:NetName>
         <cicswsat:Token>F0F0F0F0</cicswsat:Token>
         <cicswsat:UOWID>BCDB8F2E852B924C</cicswsat:UOWID>
      </wsa:ReferenceProperties>
   </wscoor:RegistrationService>
</wscoor:CoordinationContext>
```

Example 11-12 shows a `CoordinationContext` created by WebSphere Application Server V6.0.

Example 11-12 Sample CoordinationContext created by WebSphere

```
<wscoor:CoordinationContext>
   <wscoor:Expires>Never</wscoor:Expires>
   <wscoor:Identifier>
      com.ibm.ws.wstx:00000107d70ad2...
   </wscoor:Identifier>
   <wscoor:CoordinationType
      http://schemas.xmlsoap.org/ws/2004/10/wsat
   </wscoor:CoordinationType>
   <wscoor:RegistrationService>
      <wsa:Address>
         http://9.100.199.156:9080/_IBMSYSAPP/wscoor/services/RegistrationCoord
         inatorPort
      </wsa:Address>
      <wsa:ReferenceProperties>
         <websphere-wsat:txID xmlns:websphere-wsat=”......”>
            com.ibm.ws.wstx:00000107d70ad2...
         </websphere-wsat:txID>
         <websphere-wsat:instanceID xmlns:websphere-wsat=”......”
            com.ibm.ws.wstx:00000107d70ad2...
         </websphere-wsat:instanceID>
      </wsa:ReferenceProperties>
   </wscoor:RegistrationService>
</wscoor:CoordinationContext>
```

Note: We noted from our tests that the content of the `Expires` element (that is, `Never`) is not an unsigned integer as required by the specification. This is planned to be changed in a future release of WebSphere Application Server.

11.3.4 Register

Before we provide the details of the Register request, we consider some concepts in this section so that you don't lose sight of the "big picture."

The Coordinator provides the application with the Endpoint reference of its Registration service in the `CreateCoordinationContextResponse`. The application then knows where and how to send a `Register` request.

Figure 11-5 shows how Endpoint references are used during and after registration.

1. The Register message targets the Endpoint reference of the Coordinator's Registration Service and includes the Endpoint reference of the application's Protocol service as a parameter.
2. The Register Response includes the Endpoint reference of the Coordinator's Protocol service.
3. At this point, both sides have the Endpoint Reference of the other's Protocol service, so the protocol messages can target the other side.

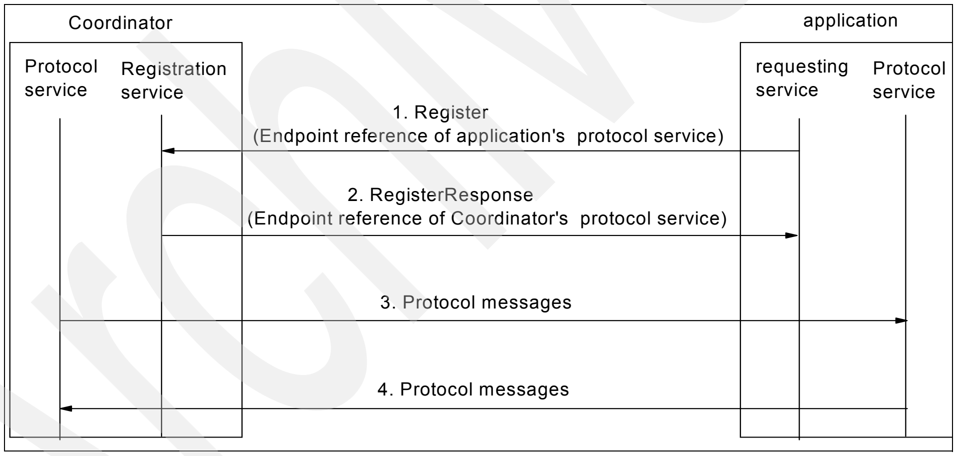


Figure 11-5 Using Endpoint references during and after registration

Now that you understand the big picture, we provide the details. Example 11-13 shows the pseudo schema for the `Register` element.

Example 11-13 Pseudo schema for the Register element

```
<wscoor:Register>
   <wscoor:ProtocolIdentifier>.............</wscoor:ProtocolIdentifier>
   <wscoor:ParticipantProtocolService>.....</wscoor:ParticipantProtocolService>
</wscoor:Register>
```

The `ProtocolIdentifier` element contains a URI that provides the identifier of the coordination protocol selected for registration. The contents of the `CoordinationType` element of the `CoordinationContext` element determine the possible choices for the `ProtocolIdentifier` element as follows:

- If the `CoordinationType` is atomic transaction, then the `ProtocolIdentifier` must be one of the following:
  - `http://schemas.xmlsoap.org/ws/2004/10/wsat/Completion`
  - `http://schemas.xmlsoap.org/ws/2004/10/wsat/Volatile2PC`
  - `http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC`
- If the `CoordinationType` is business activity (either AtomicOutcome or MixedOutcome), then the `ProtocolIdentifier` must be one of the following:
  - `http://schemas.xmlsoap.org/ws/2004/10/wsba/BusinessAgreementWithParticipantCompletion`
  - `http://schemas.xmlsoap.org/ws/2004/10/wsba/BusinessAgreementWithCoordinatorCompletion`

The `ParticipantProtocolService` element contains the `EndpointReference` that the registering participant wants the Coordinator to use for the Protocol service.

Note: As we noted earlier, CICS TS V3.1 does not support the WS-Business Activity specification. Therefore, it does not support either the `BusinessAgreementWithParticipantCompletion` or the `BusinessAgreementWithCoordinatorCompletion` protocol identifiers.

Example 11-14 shows a `Register` element created by a CICS TS V3.1 region.

Example 11-14 Sample Register element

```
<wscoor:Register>
   <wscoor:ProtocolIdentifier>
      http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
   </wscoor:ProtocolIdentifier>
   <wscoor:ParticipantProtocolService>
      <wsa:Address>
         http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
      </wsa:Address>
      <wsa:ReferenceProperties>
         <cicswsat:UOWID>BDFC52CD7C57D466</cicswsat:UOWID>
         <cicswsat:PublicId>310FD7E2E2...</cicswsat:PublicId>
      </wsa:ReferenceProperties>
   </wscoor:ParticipantProtocolService>
</wscoor:Register>
```

Note: CICS TS V3.1 uses a single endpoint address for its Registration service, Protocol service, and fault messages. Other products, such as WebSphere Application Server V6.0, use separate addresses for each of these.

11.3.5 Register response

Example 11-15 shows the pseudo schema for the `RegisterResponse` element.

Example 11-15 Pseudo schema for the RegisterResponse element

```
<wscoor:RegisterResponse>
   <wscoor:CoordinatorProtocolService>.....</wscoor:CoordinatorProtocolService>
</wscoor:RegisterResponse>
```

The `CoordinatorProtocolService` element contains the `EndpointReference` that the Coordination service wants the registered participant to use for the Protocol service.

We note, once again, that when the application has received the `RegisterResponse`, each side has the `EndpointReference` of the other's Protocol Service.

11.3.6 Two applications with their own coordinators

In this section we see how two application services (App1 and App2) with their own coordinators (Coordinator A and Coordinator B) interact as an activity propagates between them. Coordinator A provides Activation service ASa, Registration service RSa, and Protocol service Pa. Coordinator B provides Activation service ASb, Registration service RSb, and Protocol service Pb. Figure 11-6 shows the two applications with their own coordinators.

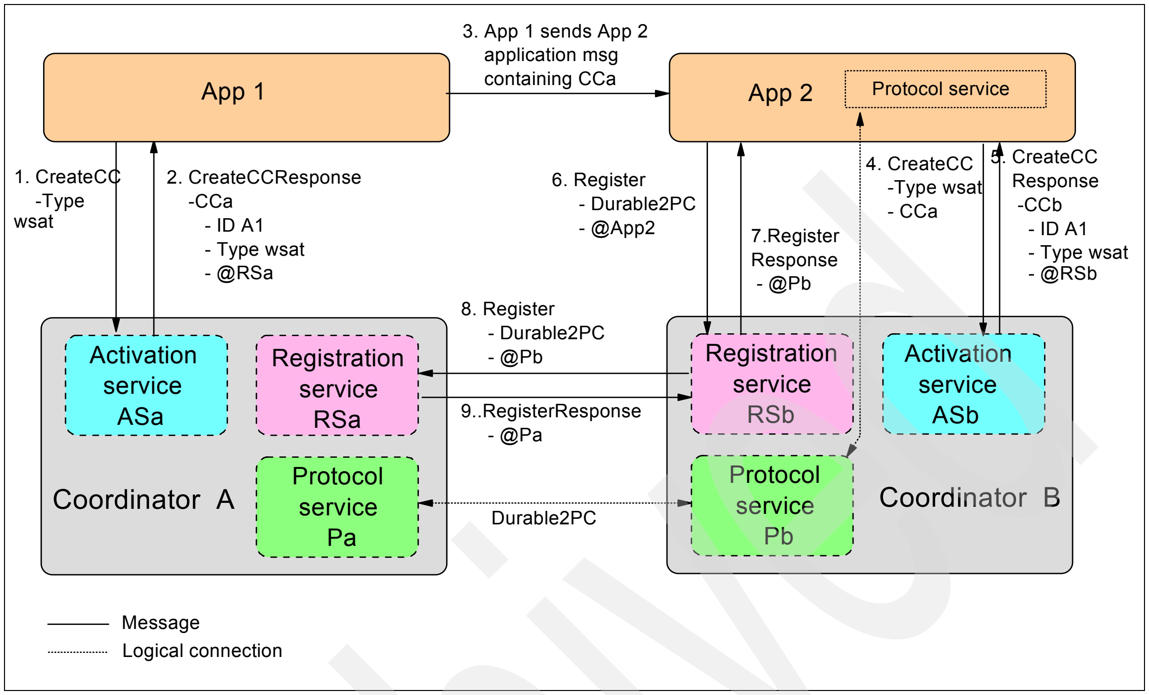

3. App 1 sends App 2 application msg containing CCa
App 1
App 2
Protocol service
1. CreateCC -Type wsat
2. CreateCCResponse -CCa - ID A1 - Type wsat - @RSa
6. Register - Durable2PC - @App2
7.Register Response - @Pb
4. CreateCC -Type wsat - CCa
5. CreateCC Response -CCb - ID A1 - Type wsat - @RSb
8. Register - Durable2PC - @Pb
9..RegisterResponse - @Pa
Activation service ASa
Registration service RSa
Protocol service Pa
Coordinator A
Registration service RSb
Activation service ASb
Protocol service Pb
Coordinator B
Durable2PC
Message
Logical connection

Figure 11-6 Two applications with their own coordinators

1. App1 sends to ASa a `CreateCoordinationContext` element that specifies a `CoordinationType` of wsat.
2. ASa sends to App1 a `CreateCoordinationContextResponse` element that includes `CoordinationContext` CCa. CCa contains an `Identifier` element whose content is A1, a `CoordinationType` element whose content is wsat, and a `RegistrationService` element that contains an `EndpointReference` to Coordinator A's Registration service RSa.
3. App1 then sends to App2 a SOAP message that has a SOAP header that contains CCa and that has a `mustUnderstand` attribute with a value of `True`.
4. App2 prefers Coordinator B, so it sends to ASb a `CreateCoordinationContext` element that specifies a `CurrentContext` of CCa.
5. Coordinator B creates its own `CoordinationContext` element CCb that contains the same `Identifier` and `CoordinationType` as CCa but with an `EndpointReference` to its own RegistrationService RSb.

The *WS-Atomic Transaction* specification which we discuss in detail in "WS-Atomic Transaction" on page 436 states that:

   – If the `CreateCoordinationContext` request includes the `CurrentContext` element, then the target coordinator is interposed as a subordinate to the coordinator stipulated inside the `CurrentContext` element.

   – If the `CreateCoordinationContext` request does not include a `CurrentContext` element, then the target coordinator creates a new transaction and acts as the root coordinator.

6. App2 determines the coordination protocols supported by the wsat coordination type and then sends a `Register` element to RSb. This `Register` element contains a `ProtocolIdentifier` of `Durable2PC` and an `EndpointReference` to App2's Protocol service.
7. RSb sends back to App2 a `RegisterResponse` element that contains an `EndpointReference` to Protocol service Pb. This forms a logical connection (which the Durable2PC protocol can use) between the Endpoint reference for App2's Protocol service and the Endpoint reference for Coordinator B's Protocol service Pb.
8. This registration causes Coordinator B to forward the registration on to Coordinator A's Registration service RSa.
9. RSa sends back to Coordinator B a `RegisterResponse` element that contains an `EndpointReference` to Protocol Service Pa. This forms a logical connection between the `EndpointReference`s for Pa and Pb that the Durable2PC protocol can use.

11.3.7 Addressing requirements for WS-Coordination message types

`CreateCoordinationContext` and `Register` messages:

- *Must* include a `wsa:MessageID` header
- *Must* include a `wsa:ReplyTo` header

`CreateCoordinationContextResponse` and `RegisterResponse` messages:

- *Must* include a `wsa:RelatesTo` header that specifies the `MessageID` from the corresponding request message

11.4 WS-Atomic Transaction

As we discussed in Section 11.3, "WS-Coordination" on page 425, the WS-Coordination specification defines an extensible framework for defining coordination types. The WS-Atomic Transaction specification builds on

WS-Coordination by providing the definition of the atomic transaction coordination type.

Atomic transactions have an *all-or-nothing* property. The actions taken prior to commit are only tentative (that is, not persistent and not visible to other activities). When an application finishes, it requests the Coordinator to determine the outcome for the transaction. The Coordinator determines if there were any processing failures by asking the participants to vote. If the participants all vote that they were able to execute successfully, the Coordinator commits all actions taken. If a participant votes that it needs to abort or a participant does not respond at all, the Coordinator aborts all actions taken. Commit makes the tentative actions visible to other transactions. Abort makes the tentative actions appear as though the actions never happened.

IBM, Microsoft, and BEA published the *Web Services- Atomic Transaction (WS - Atomic Transaction)* specification in September, 2003; they updated it in November, 2004. Arjuna Technologies Ltd., Hitachi Ltd., and IONA Technologies joined IBM, Microsoft, and BEA in publishing *Web Services - Atomic Transaction (WS - Atomic Transaction) Version 1.0* in August, 2005. You can find these at:

```
http://www-128.ibm.com/developerworks/library/specification/ws-tx
```

Although CICS TS V3.1 was developed and tested when the November, 2004, version was the current version, the nature of the differences between the November, 2004 version and the August, 2005 version is such that it may accurately be said that CICS TS V3.1 also supports the August, 2005 version.

All information items defined by the November, 2004 and August, 2005 versions are identified by the XML namespace URI:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat
```

We associate the namespace prefix `wsat` with this namespace by using the attribute:

```
xmlns:wsat="http://schemas.xmlsoap.org/ws/2004/10/wsat"
```

The WS-AT specification defines the following protocols for atomic transactions:

- Completion
- Volatile Two-Phase Commit
- Durable Two-Phase Commit

11.4.1 Completion protocol

The `Completion` protocol is used by an application to tell the Coordinator to try to either commit or abort an atomic transaction. The `Completion` protocol initiates commitment processing. Based on each protocol's registered participants, the Coordinator begins with Volatile 2PC and then proceeds through Durable 2PC. After the transaction has completed, a status (Committed or Aborted) is returned to the application.

An initiator registers for this protocol by specifying the following URI for the contents of the `ProtocolIdentifier` element in the `Register` element:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Completion
```

Figure 11-7 illustrates the protocol abstractly.

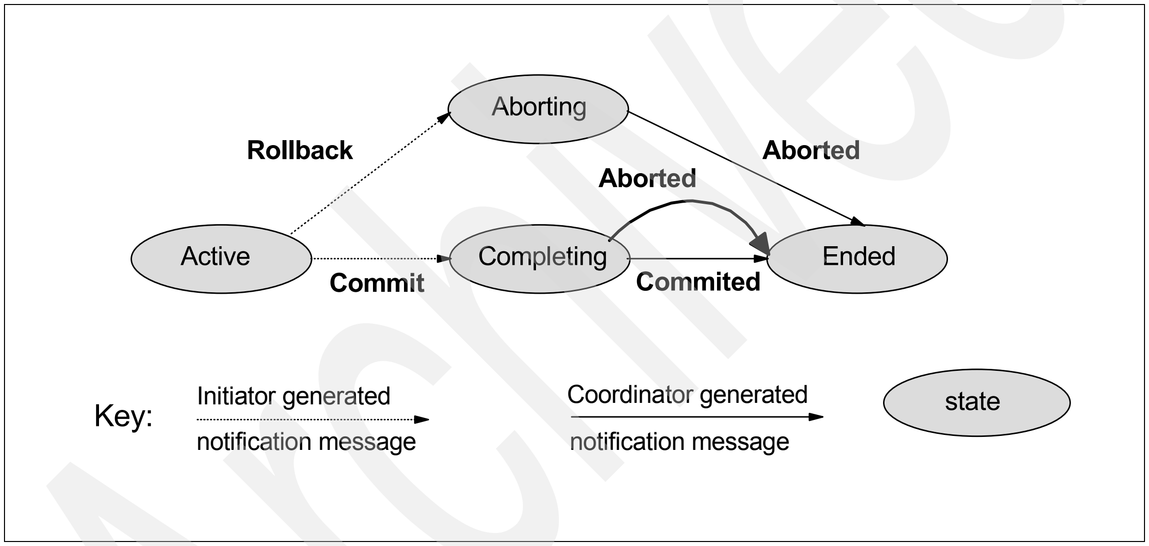

Aborting
Rollback
Aborted
Aborted
Active
Completing
Ended
Commit
Commited
Key:
Initiator generated notification message
Coordinator generated notification message
state

Figure 11-7 Completion protocol

The initiator generates:

- Commit

  Upon receipt of this notification, the Coordinator knows that the initiator has *completed* application processing and that it should attempt to commit the transaction.

- Rollback

  Upon receipt of this notification, the Coordinator knows that the initiator has *terminated* application processing and that it should abort the transaction.

The Coordinator generates:

- Committed

  Upon receipt of this notification, the initiator knows that the Coordinator reached a decision to commit.

- Aborted

  Upon receipt of this notification, the initiator knows that the Coordinator reached a decision to abort.

11.4.2 Two-Phase Commit protocol

The Two-Phase Commit (2PC) protocol defines how multiple registered participants reach agreement on the outcome of an atomic transaction. The 2PC protocol has two variants: Volatile 2PC and Durable 2PC.

Participants managing volatile resources such as a cache should register for this protocol by using the following protocol identifier:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Volatile2PC
```

Participants managing durable resources such as a database should register for this protocol by using the following protocol identifier:

```
http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
```

Note: When CICS TS V3.1 is a participant in an atomic transaction, it always requests the Durable2PC protocol when it sends a Register request. When CICS TS V3.1 is the coordinator of an atomic transaction, it will tolerate a Register request for Volatile2PC but it will treat it as a Durable2PC request.

After receiving a **`Commit`** notification in the Completion protocol, the root Coordinator begins the Prepare phase of all participants registered for the Volatile 2PC protocol. All participants registered for this protocol must respond before a Prepare is issued to a participant registered for the Durable 2PC protocol. We illustrate this in Figure 11-8, where participants P1 and P3 registered for the Volatile 2PC protocol and participant P2 registered for the Durable 2PC protocol. Both P1 and P3 must respond to the **`Prepare`** notification before the Coordinator can send **`Prepare`** to P2.

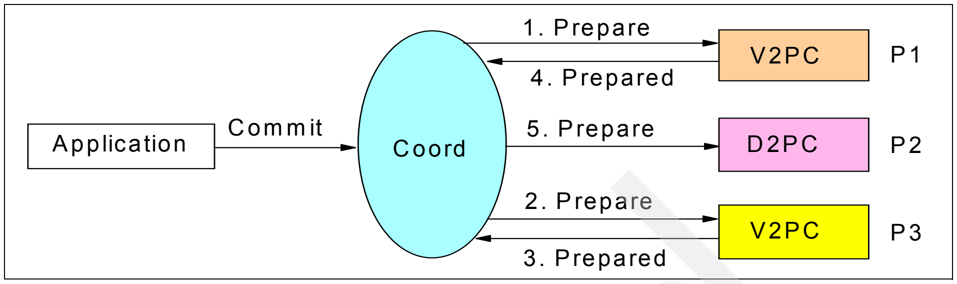


Figure 11-8 Mixture of participants registered for Durable 2PC and Volatile 2PC

Upon successfully completing the prepare phase for Volatile 2PC participants, the root Coordinator begins the Prepare phase for Durable 2PC participants. All participants registered for this protocol must respond `Prepared` or `ReadOnly` before a Commit notification is issued to a participant registered for either protocol. A volatile participant is not guaranteed to receive a notification of the transaction's outcome.

Figure 11-9 illustrates the 2PC protocol abstractly.

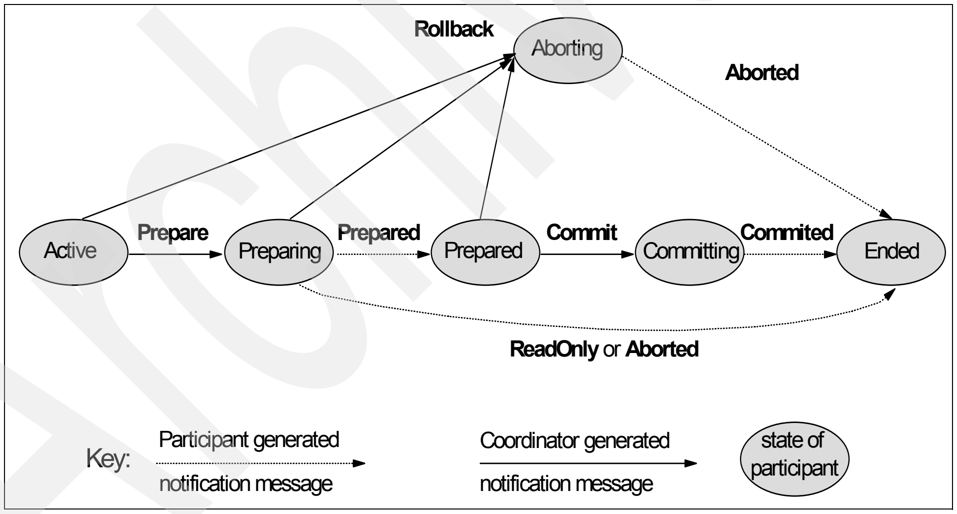


Figure 11-9 Two-Phase Commit protocol

The Coordinator generates:

- **`Prepare`**

  Upon receipt of this notification, the participant should enter phase 1 and vote on the outcome of the transaction.

  – If the participant has already voted, it should resend the same vote.

  - If the participant does not know of the transaction, it must vote to abort.

- **`Rollback`**

  Upon receipt of this notification, the participant should abort, and forget, the transaction. This notification can be sent in either phase 1 or phase 2. Once sent, the Coordinator may forget all knowledge of this transaction.

- **`Commit`**

  Upon receipt of this notification, the participant should commit the transaction.This notification can only be sent after phase 1, and if the participant voted to commit. If the participant does not know of the transaction, it must send a **`Committed`** notification to the Coordinator.

The participant generates:

- **`Prepared`**

  The participant is prepared and votes to commit the transaction.

- **`ReadOnly`**

  The participant votes to commit the transaction and has forgotten the transaction. The participant does not wish to participate in phase two.

  Suppose, for example, that the participant received an account number that it could not match to an entry in a database. It might return an error to the requesting application, but, having registered as a participant in the atomic transaction, it would then go on to be coordinated during 2PC processing. When the Coordinator sends **`Prepare`**, the participant replies **`ReadOnly`** and then terminates without waiting for the **`Commit`**. The Coordinator, on receipt of the **`ReadOnly`**, would then delete its own record of the interaction with the participant and would not attempt to send a **`Commit`** to it.

- **`Aborted`**

  The participant has aborted, and forgotten, the transaction.

- **`Committed`**

  The participant has committed the transaction. The Coordinator may safely forget that participant.

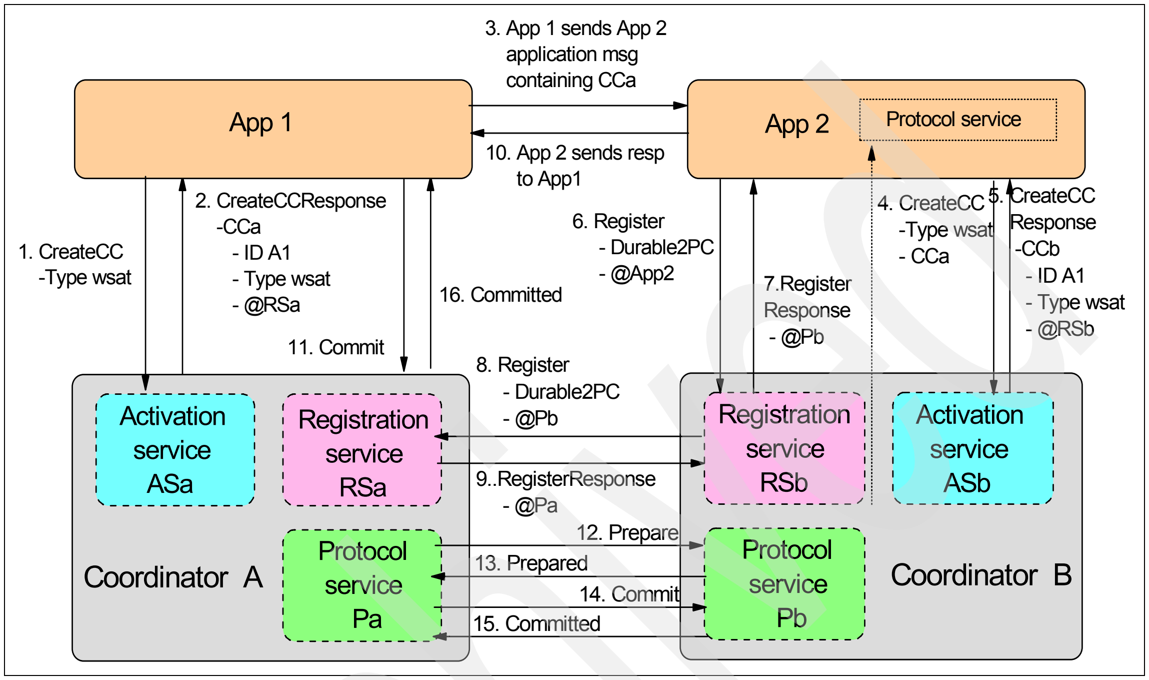
- **`Replay`**

  The participant has suffered a recoverable failure. The Coordinator should resend the last appropriate protocol notification.

11.4.3 Two applications with their own coordinators (continued)

In Section 11.3.6, “Two applications with their own coordinators” on page 434 we saw how two application services (App1 and App2) with their own coordinators

(Coordinator A and Coordinator B) used the WS-Coordination specification to interact as an activity propagated between them. Figure 11-10 shows how they use the WS-AT specification to complete their global unit of work.



3. App 1 sends App 2 application msg containing CCa
App 1
App 2
Protocol service
10. App 2 sends resp to App1
1. CreateCC -Type wsat
2. CreateCCResponse -CCa - ID A1 - Type wsat - @RSa
11. Commit
16. Committed
6. Register - Durable2PC - @App2
7.Register Response - @Pb
4. CreateCC -Type wsat - CCa
5. CreateCC Response -CCb - ID A1 - Type wsat - @RSb
8. Register - Durable2PC - @Pb
9..RegisterResponse - @Pa
12. Prepare
13. Prepared
14. Commit
15. Committed
Activation service ASa
Registration service RSa
Protocol service Pa
Coordinator A
Registration service RSb
Activation service ASb
Protocol service Pb
Coordinator B

Figure 11-10 Two applications with their own coordinators (continued)

After completing its work, App2 sends its response back to App1 (step 10 in Figure 11-10). When App1 completes its work, its sends a **Commit** message to Coordinator A (step 11). This causes Coordinator A's Durable 2PC Protocol service Pa to send a **Prepare** notification to Coordinator B's Durable 2PC Protocol service Pb (step 12). We assume that Pb responds with a **Prepared** notification. If Pa encounters no other errors, it makes the decision to commit the atomic transaction and sends a **Commit** notification to Pb (step 14). Pb returns a **Committed** notification and then completes its updates before terminating. When Coordinator A receives this notification, it commits its own updates and notifies App1 of the outcome (step 16).

11.4.4 Addressing requirements for WS-AT message types

The messages defined in the WS-AT specification are *notification* messages; that is, they are *one-way* messages. There are two types of notification messages:

- A notification message is a *terminal* message when it indicates the end of a coordinator/participant relationship. **`Committed`**, **`Aborted`**, and **`ReadOnly`** are terminal messages.
- A notification message is a *non-terminal* message when it does not indicate the end of a coordinator/participant relationship. **`Commit`**, **`Rollback`**, **`Prepare`**, **`Prepared`**, and **`Replay`** are non-terminal messages.

Non-terminal notification messages *must* include a `wsa:ReplyTo` header.

11.4.5 CICS TS V3.1 and resynchronization processing

We have completed our discussion of the WS-AT specification. Unfortunately, the current version of the specification does not completely cover all of the issues surrounding the use of the 2PC protocol. In particular, it does not completely describe the resynchronization processing that should take place following a failure in one of the systems involved in the 2PC protocol or in the network connections that link the systems together. The only thing that the specification mentions relating to resynchronization is the **`Replay`** notification. Therefore, in this section we describe some aspects of how CICS TS V3.1 handles resynchronization processing for transactions that use the 2PC protocol.

Note: For the sake of brevity we do not describe all of the possible issues. For example, we do not describe:

- What happens when resync processing is driven from both sides and a *race* condition results
- What happens when a resync request fails

Network failures can result in messages not being delivered in a timely manner. System failures prevent processing altogether until a restart takes place.

Within the 2PC processing sequence there is a period of time, known as the *in-doubt* window, during which one system is unable to complete processing because it does not know what the other system has done. The distributed UOW is said to be in-doubt when:

- A participant Protocol service has replied **`Prepared`** in response to a **`Prepare`** notification, *and*
- Has written a log record of its response to signify that it has entered the in-doubt state, *and*
- Does not yet know the decision of its coordinator (to **`Commit`** or to **`Rollback`**).

Barring system or network failures, the UOW remains in-doubt until the coordinator issues either the **`Commit`** or **`Rollback`** request as a result of responses received from all UOW participants. If a failure occurs that causes loss of connectivity between a participant and its coordinator, the UOW remains in-doubt until either:

- Recovery from the failure has taken place and synchronization can resume, *or*
- The in-doubt waiting period is terminated by some built-in control mechanism, and an arbitrary (heuristic) decision is then taken (to commit or back out).

Note that while the UOW remains in-doubt, the recoverable resources that it owns remain locked.

If a system or network failure occurs during the in-doubt window, additional steps must be taken to ensure that the updates are completed in a consistent manner by both systems. This is known as *resynchronization processing*.

Previous releases of CICS provided a Recovery Manager that dealt with resynchronization processing for distributed workloads that made use of *VTAM* networks or that used *MRO* connections. These releases dealt with failures during the in-doubt window in one of three ways:

- Automatic heuristic decision

  You could cause CICS to make an automatic heuristic decision by specifying the WAIT, WAITTIME, and ACTION attributes on a TRANSACTION definition as follows:

  - If you set the WAIT attribute to NO, then CICS took whatever action was specified on the ACTION attribute (either COMMIT or BACKOUT) *immediately*.
  - If you set the WAIT attribute to YES and the WAITTIME attribute to a non-zero value, then CICS took whatever action was specified on the ACTION attribute after waiting for the amount of time specified in WAITTIME (assuming normal recovery and resynchronization had not already taken place).

- Manual heuristic decision

  You could force an in-doubt UOW to complete by issuing a CEMT SET UOW(*uowid*) [COMMIT | BACKOUT] command or its EXEC CICS equivalent.

- Automatic resynchronization

  If you set the WAIT attribute to YES and the WAITTIME attribute to 00.00.00, the transaction would wait until it could communicate with its partner system, after which it could either explicitly request that the message it was waiting for be sent again, or it could resend the last message that it generated.

CICS TS V3.1 extends the Recovery Manager for use by WS-AT workloads. CICS applications that form part of a WS-AT workload can be controlled by any of these mechanisms. However, automatic resynchronization is somewhat different for WS-AT workloads.

The principle difference arises from the fact that WS-AT processing takes place over a TCP/IP network.

- Other forms of distributed transactions make use of communication mechanisms such as VTAM, and resynchronization across a VTAM network can be triggered when the connection between a pair of systems is re-established.
- CICS does not currently support TCP/IP connections in the same way that it does its VTAM connections, and so CICS can only drive resynchronization of WS-AT requests when a region starts.

Important: CICS only drives resynchronization of WS-AT requests during a CICS region restart.

During any type of startup except an initial start, CICS reads the system log to discover any units of work that were in-doubt when the region previously shut down or failed. While reading through the log, CICS may find that it has outstanding units of work that indicate they were involved in an atomic transaction. These log records also indicate whether the UOW was acting as a coordinator or a participant.

- Coordinator

  If it is a coordinator and the log record indicates that the UOW was waiting for a `Committed` or `Aborted` response from a participant when CICS shut down, then the UOW is reactivated (unshunted) and sends out its decision message (`Commit` or `Rollback`) to the participant identified in the log record.

  – If a response is received, then the UOW completes its processing and terminates.

  – If a response is not received before the coordination UOW times out, then CICS shunts the UOW (moves it aside for processing later on). The UOW then persists until another resynchronization attempt takes place or until someone manually forces it to complete. (The coordination UOW times out after 30 seconds, a value set internally by CICS).

- Participant

  If it is a participant and the log record indicates that the UOW had voted in response to a **`Prepare`** message and was waiting for a `Commit` or `Rollback` decision from its coordinator when communication was lost, then the UOW is reactivated, sends a `Replay` message to its coordinator, and once again waits for the decision message to arrive.

– If the decision message is then received, the participant acts on it and sends a `Committed` or `Aborted` message back to the coordinator before terminating.

– If the decision message does not arrive before the participant UOW times out, then CICS shunts the participant UOW. The UOW then persists until another resynchronization attempt takes place or until someone manually forces it to complete.

12

Enabling atomic transactions

We begin this chapter by showing you how to enable atomic transactions in CICS. We start with the simple case where we have a service requester application running in CICS AOR1 invoking a service provider application running in CICS AOR2. Then we move to the more elaborate case where two CICSplexes are working together, one acting as service requester and the other as service provider. We conclude the chapter by showing you how to enable atomic transactions in WebSphere Application Server.

12.1 Enabling atomic transactions in CICS

We recognize that it is not likely that many customers will choose to use WS-AT for workloads distributed entirely within CICS. More typically, CICS would participate in an atomic transaction as a Web service provider with the service requester running under the control of another product such as WebSphere Application Server V6 or later. Customers might also use a CICS transaction acting as a Web service requester to participate in an atomic transaction with a service provider running under the control of another product.

Nevertheless, we start with the special case of one CICS region acting as a service requester participating in an atomic transaction with another CICS region acting as a service provider. This will illustrate the capacity of CICS to undertake both the role of a coordinator of an atomic transaction and the role of a participant in an atomic transaction.

12.1.1 CICS to CICS configuration

Figure 12-1 shows two CICS regions: AOR1 and AOR2. A service requester application running in AOR1 invokes a service provider application running in AOR2.

In AOR1 the request passes through a pipeline that contains a CICS-provided message handler module (either DFHPISN1, if SOAP 1.1 is being used; or DFHPISN2, if SOAP 1.2 is being used). DFHPISNx invokes the CICS-provided header processing program DFHWSATH. DFHWSATH adds a SOAP header containing a `CoordinationContext` to each message that it sends out.

In AOR2 the request passes through a pipeline that supports the Web service that AOR1's application is calling and also invokes DFHPISNx. DFHPISNx invokes the header processing program DFHWSATH when it detects a SOAP header that contains a `CoordinationContext`.

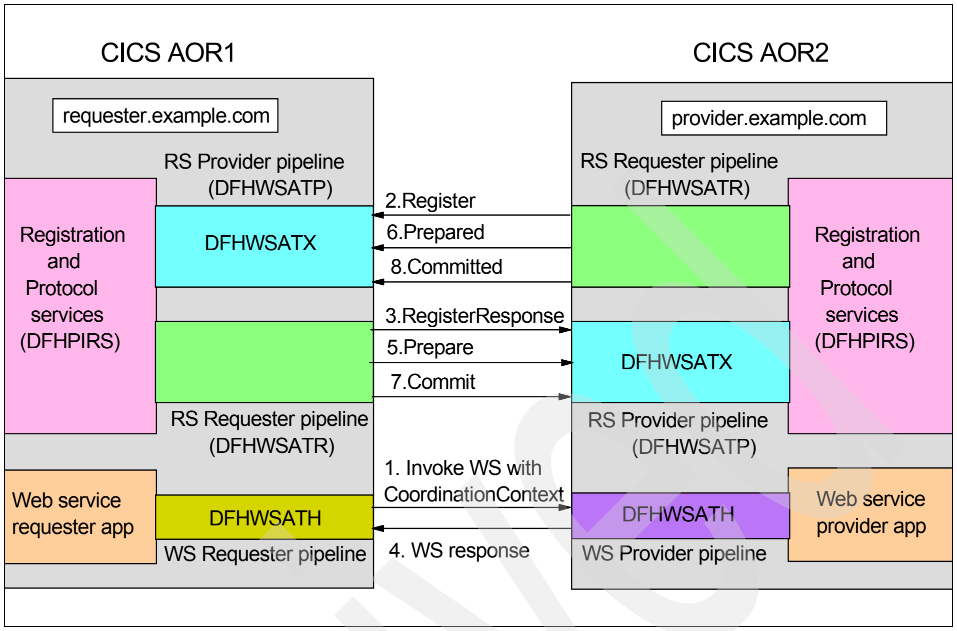

CICS AOR1
CICS AOR2
requester.example.com
provider.example.com
RS Provider pipeline (DFHWSATP)
RS Requester pipeline (DFHWSATR)
Registration and Protocol services (DFHPIRS)
DFHWSATX
2.Register
6.Prepared
8.Committed
3.RegisterResponse
5.Prepare
7.Commit
RS Requester pipeline (DFHWSATR)
RS Provider pipeline (DFHWSATP)
1. Invoke WS with CoordinationContext
Web service requester app
Web service provider app
DFHWSATH
WS Requester pipeline
4. WS response
WS Provider pipeline

Figure 12-1 The special case: CICS to CICS

Both regions have a requester pipeline named DFHWSATR and a provider pipeline named DFHWSATP for registration and protocol processing. The DFHWSATP pipeline invokes the CICS-supplied message handler DFHWSATX as the last message handler in the pipeline.

For the workload we have shown, AOR1's DFHWSATP pipeline receives registration requests and protocol notifications, while its DFHWSATR pipeline sends registration responses and protocol instructions.

AOR2's DFHWSATP pipeline receives registration responses and protocol instructions, while its DFHWSATR pipeline sends registration requests and protocol notifications.

Note: The DFHWSATP pipeline acts as the registration endpoint for CICS.

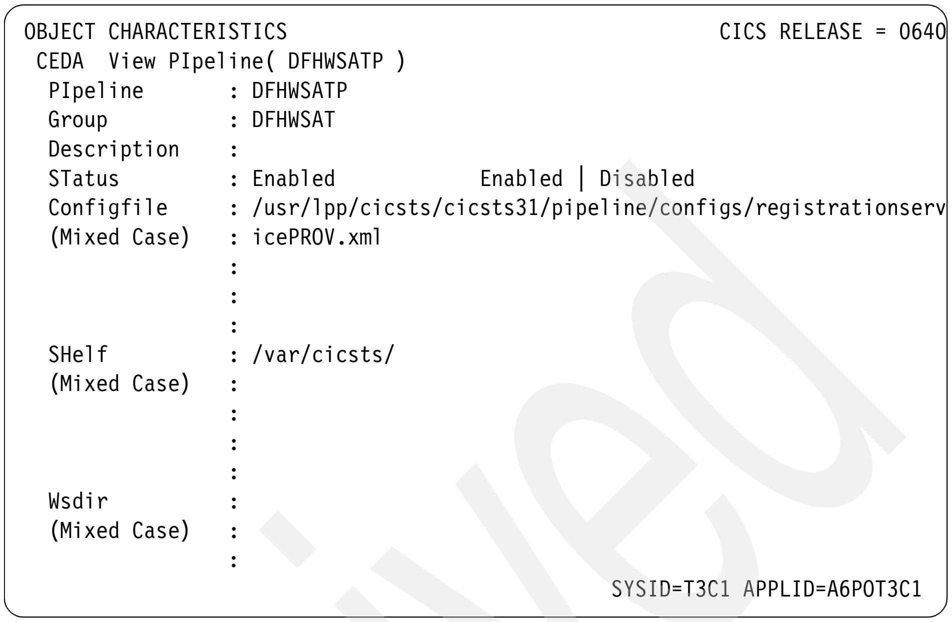
CICS TS V3.1 provides a new resource group DFHWSAT to assist customers with setting up WS-AT support in CICS. The DFHWSAT group contains the resources shown in Table 12-1.

Table 12-1 CICS supplied resource definitions for WS-AT

Resource	Resource name	Description
Pipeline	DFHWSATP	Registration services provider PIPELINE
Pipeline	DFHWSATR	Registration services requester PIPELINE
Urimap	DFHRSURI	URIMAP used by the Registration services provider
Program	DFHPIRS	Registration and protocol services program
Program	DFHWSATH	SOAP header processing program
Program	DFHWSATR	Registration and coordination services handler program
Program	DFHWSATX	CICS message handler program

Since DFHLIST does not include the DFHWSAT group and you cannot add the DFHWSAT group to DFHLIST, specifying DFHLIST in the system initialization table GRPLIST parameter will not cause CICS to install DFHWSAT automatically during an initial start.

Figure 12-2 shows the definition of the DFHWSATP PIPELINE resource.



```
OBJECT CHARACTERISTICS                                          CICS RELEASE = 0640
 CEDA  View PIpeline( DFHWSATP )
  PIpeline       : DFHWSATP
  Group          : DFHWSAT
  Description    :
  STatus         : Enabled            Enabled | Disabled
  Configfile     : /usr/lpp/cicsts/cicsts31/pipeline/configs/registrationserv
  (Mixed Case)   : icePROV.xml
                 :
                 :
                 :
  SHelf          : /var/cicsts/
  (Mixed Case)   :
                 :
                 :
                 :
  Wsdir          :
  (Mixed Case)   :
                 :
                                                     SYSID=T3C1 APPLID=A6POT3C1
```

Figure 12-2 Definition of the DFHWSATP PIPELINE resource

Note that the definition contains the fully-qualified name of the pipeline configuration file. If you install the CICS-supplied registrationservicePROV.xml configuration file in a different directory when you install CICS (as we did), then you must make a copy of the entire DFHWSAT group, change the definition of the DFHWSATP pipeline, and install the modified group.

Tip: If you attempt to modify the definition of the DFHWSATP pipeline in group DFHWSAT, you will get the message "`Unable to perform operation: DFHWSAT is IBM protected.`"

If you add the DFHWSAT group to your startup list and then attempt to override the definition of DFHWSATP that is provided in the DFHWSAT group by adding a modified DFHWSATP definition that appears later in the startup list, then you will get message DFHAM4892 W indicating that the install of the second group completed with errors.

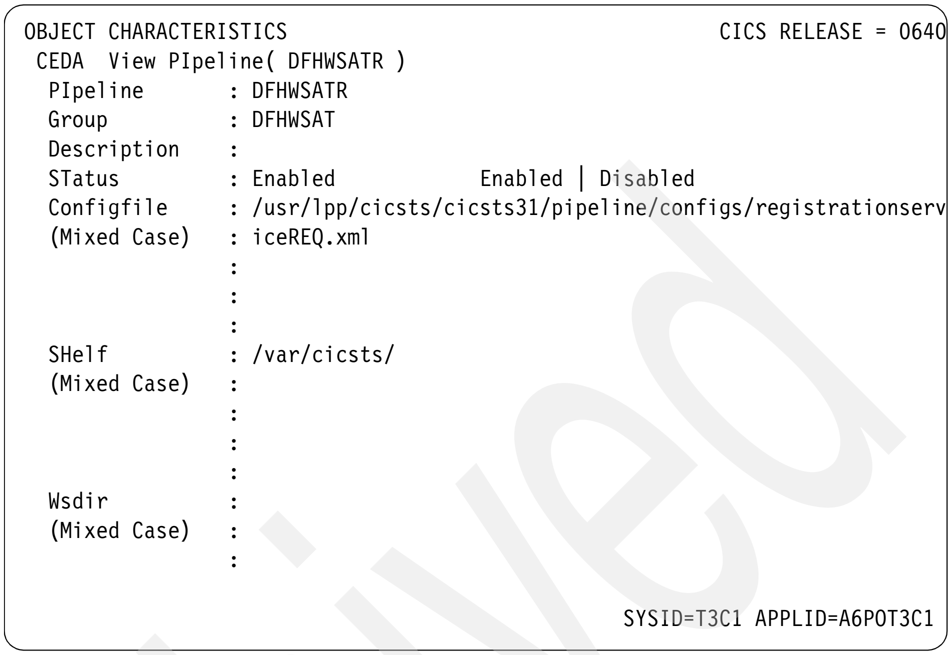
If you copy only the DFHWSATP, DFHWSATR, and DFHRSURI definitions to a new group and try to let program autoinstall automatically install the definitions of the programs DFHWSATH, DFHWSATR, DFHWSATX, and DFHPIRS, you may also have problems. These four programs need access to containers which use CICS-key storage, and therefore they must run with EXECKEY(CICS) unless storage protection is turned off. You would have a problem if the model for the program you use for program autoinstall does not specify EXECKEY(CICS).

Example 12-1 shows the contents of the registrationservicePROV.xml file. This configuration file defines a pipeline that contains only one message handler program, DFHWSATX.

Example 12-1 The registrationservicePROV.xml pipeline configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline
  xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
  <service>
    <terminal_handler>
      <handler>
        <program>DFHWSATX</program>
        <handler_parameter_list/>
      </handler>
    </terminal_handler>
  </service>
  <service_parameter_list/>
</provider_pipeline>
```

Figure 12-3 shows the definition of the DFHWSATR PIPELINE resource.



```
OBJECT CHARACTERISTICS                                    CICS RELEASE = 0640
 CEDA  View PIpeline( DFHWSATR )
  PIpeline       : DFHWSATR
  Group          : DFHWSAT
  Description    :
  STatus         : Enabled            Enabled | Disabled
  Configfile     : /usr/lpp/cicsts/cicsts31/pipeline/configs/registrationserv
  (Mixed Case)   : iceREQ.xml
                 :
                 :
                 :
  SHelf          : /var/cicsts/
  (Mixed Case)   :
                 :
                 :
                 :
  Wsdir          :
  (Mixed Case)   :
                 :

                                                  SYSID=T3C1 APPLID=A6POT3C1
```

Figure 12-3 Definition of the DFHWSATR PIPELINE resource

Example 12-2 shows the contents of the registrationserviceREQ.xml file. This configuration file defines a pipeline that does not contain any message handlers.

Example 12-2 The registrationserviceREQ.xml pipeline configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<requester_pipeline
    xmlns="http://www.ibm.com/software/htp/cics/pipeline"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
    requester.xsd">
</requester_pipeline>
```

Important: We recommend that you do not modify the registrationserviceREQ.xml pipeline configuration file or the registrationservicePROV.xml pipeline configuration file. If you modify one of these files, you might inadvertently alter the flow of the registration or protocol service messages and thereby affect the integrity of your data.

Figure 12-4 shows the definition of the DFHRSURI URIMAP resource.

```
OBJECT CHARACTERISTICS                                          CICS RELEASE = 0640
 CEDA  View Urimap( DFHRSURI )
  Urimap         : DFHRSURI
  Group          : DFHWSAT
  Description    :
  STatus         : Enabled            Enabled | Disabled
  USAge          : Pipeline           Server | Client | Pipeline
 UNIVERSAL RESOURCE IDENTIFIER
  SCheme         : HTTP               HTTP | HTTPS
  HOST           : *
  (Lower Case)   :
  PAth           : /cicswsat/RegistrationService
  (Mixed Case)   :
                 :
                 :
                 :
 ASSOCIATED CICS RESOURCES
  TCpipservice   :
  Analyzer       : No                 No | Yes
  COnverter      :
  TRansaction    : CPIH
  PRogram        :
  PIpeline       : DFHWSATP
  Webservice     :                                         (Mixed Case)
 SECURITY ATTRIBUTES
  USErid         :
  CIphers        :
```

Figure 12-4   Definition of the DFHRSURI URIMAP resource

When the CICS-supplied CWXN transaction finds that the URI in an HTTP request matches the PATH attribute of the DFHRSURI URIMAP definition, it uses the PIPELINE attribute of that definition to get the name of the PIPELINE definition that it will use to process the incoming request. As Figure 12-4 shows, this is the DFHWSATP pipeline. As we have already seen, the DFHWSATP PIPELINE definition specifies that CICS should use registrationservicePROV.xml as the pipeline configuration file. As we have also already seen, this configuration file defines a pipeline that contains only one message handler program, DFHWSATX. Thus when the URI in an HTTP request contains /cicswsat/RegistrationService, CICS invokes the DFHWSATX message handler.

This raises the question: What causes CICS to receive an HTTP request whose URI field contains /cicswsat/RegistrationService?

The answer is that the pipeline configuration file for the service requester application (running in CICS AOR1) must specify:

- One of the CICS-provided SOAP message handlers (cics_soap_1.1_handler or cics_soap_1.2_handler)
- The mandatory invocation of the DFHWSATH header processing program to add a `CoordinationContext` header to the SOAP request
- A `<registration_service_endpoint>` element within a `<service_parameter_list>`

This is shown in Example 12-3.

Example 12-3 Service requester pipeline configuration file which supports WS-AT

```
<?xml version="1.0" encoding="UTF-8"?>
<requester_pipeline
      xmlns="http://www.ibm.com/software/htp/cics/pipeline"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
      requester.xsd">
  <service>
    <service_handler_list>
      <cics_soap_1.1_handler>
        <headerprogram>
          <program_name>DFHWSATH</program_name>
          <namespace>
             http://schemas.xmlsoap.org/ws/2004/10/wscoor
          </namespace>
          <localname>CoordinationContext</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.1_handler>
    </service_handler_list>
  </service>
  <service_parameter_list>
    <registration_service_endpoint>
       http://requester.example.com:3207/cicswsat/RegistrationService
    </registration_service_endpoint>
  </service_parameter_list>
</requester_pipeline>
```

The `<registration_service_endpoint>` element contains the address of the Registration service endpoint that runs in the requesting CICS region (AOR1). The path component of this address matches the PATH attribute defined in the DFHRSURI URIMAP resource definition of AOR1. Participant Web services should send `Register` requests and `Prepared` and `Committed` (or `Aborted`) notifications to this address.

In the service requester pipeline in Example 12-3:

- Since the `<mandatory>` element contains `True`, the pipeline will flow a `CoordinationContext` with the message.
- If you change the `<mandatory>` element to `False` or remove DFHWSATH from the pipeline, the pipeline will not flow a `CoordinationContext` with the message.

The pipeline configuration file (Example 12-4) for the service *provider* application must specify:

- One of the CICS-provided SOAP message handlers (`cics_soap_1.1_handler` or `cics_soap_1.2_handler`)
- Invocation of the DFHWSATH header processing program whenever the SOAP message contains a `CoordinationContext` header
- A `<registration_service_endpoint>` element within a `<service_parameter_list>`

Example 12-4 Service provider pipeline configuration file that supports WS-AT

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline
   xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
   provider.xsd ">
 <service>
  <terminal_handler>
   <cics_soap_1.1_handler>
    <headerprogram>
     <program_name>DFHWSATH</program_name>
     <namespace>
      http://schemas.xmlsoap.org/ws/2004/10/wscoor
     </namespace>
     <localname>CoordinationContext</localname>
     <mandatory>false</mandatory>
    </headerprogram>
   </cics_soap_1.1_handler>
  </terminal_handler>
 </service>
 <apphandler>DFHPITP</apphandler>
 <service_parameter_list>
  <registration_service_endpoint>
  http://provider.example.com:3207/cicswsat/RegistrationService
  </registration_service_endpoint>
 </service_parameter_list>
</provider_pipeline>
```

This time the `<registration_service_endpoint>` element contains the address of the Registration service endpoint that runs in the provider CICS region. The Coordinator should send `RegisterResponse` messages and `Prepare` and `Commit` (or `Abort`) notifications to this address.

In the service provider pipeline in Example 12-4:

- The pipeline will accept flows with a `CoordinationContext`, and such flows will be treated as part of a WS-AT transaction.
- Since the `<mandatory>` element contains `False`, the pipeline will also accept messages *without* a `CoordinationContext` but they will *not* be part of any WS-AT transaction.
- If you change the `<mandatory>` element to `True`, the pipeline will *require* that a `CoordinationContext` flow with the message. A Fault will be raised if a requester attempts to use the service without a `CoordinationContext`.
- If you remove DFHWSATH from the pipeline, the pipeline will raise a `mustUnderstand` fault when a `CoordinationContext` arrives with `mustUnderstand` set to `True`.

We conclude this section with a few remarks on the function provided by two programs in the CICS-supplied DFHWSAT group.

DFHPIRS

DFHPIRS is the central component of CICS support for WS-AT. It provides the bulk of the function that is needed for registration and protocol processing. The actions that it can be called to service include the following: Register, RegisterResponse, Prepare, Prepared, Aborted, ReadOnly, Commit, Rollback, Committed, and Replay.

If DFHPIRS encounters an unrecoverable error, it abends with an abend code of APIO.

DFHWSATH

When included in the configuration file for a Web service *requester* pipeline, DFHWSATH controls the processing that causes a `CoordinationContext` to be created and added to the SOAP message before it is sent. To do this DFHWSATH calls DFHPIAT, which finds the local netname, the unit-of-work ID (UOWID), and the value of the DTIMOUT attribute on the TRANSACTION definition for the transaction ID under which the requesting application is running. DFHPIAT also finds the endpoint address of the Registration service for the requesting region. From all of this information DFHPIAT creates a `CoordinationContext`.

When included in the configuration file for a Web service *provider* pipeline, this program is invoked if the CICS-supplied SOAP message handler detects a `CoordinationContext` header in a message. When called, DFHWSATH controls the processing that extracts data from the `CoordinationContext` header.

If DFHWSATH encounters an unrecoverable error, it abends with an APIP abend, which causes the pipeline manager to terminate the processing of the current request.

12.1.2 More elaborate CICS to CICS configuration

In the simple CICS to CICS configuration shown in 12.1.1, "CICS to CICS configuration" on page 448, all the requester registration endpoints reside in the same region as the service requester application, and all the provider registration endpoints reside in the same region as the service provider application. This does not have to be the case, and a number of different configurations are possible, including the one shown in Figure 12-5.

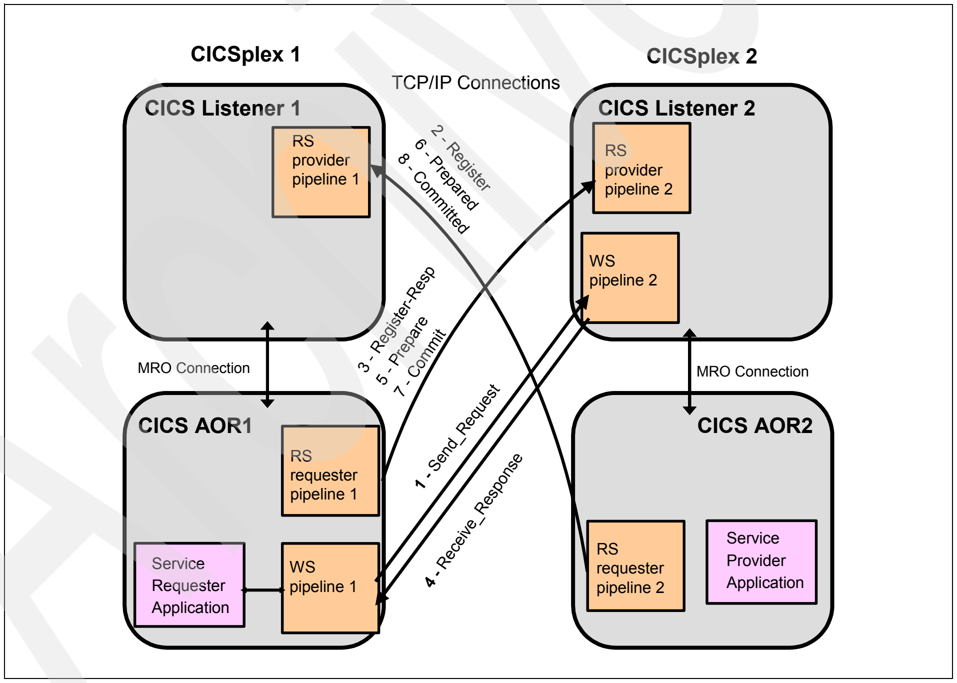


Figure 12-5 More elaborate configuration

The service requester pipeline (WS pipeline 1) must run in the same region as the service requester application that it supports. However, the service provider pipeline (WS pipeline 2) does not have to run in the same region as the service provider application that it manages. Instead, these applications are eligible for dynamic routing as described in 3.4, “Configuring for high availability” on page 101. Similarly, Registration services provider pipelines, such as RS provider pipeline 1 and RS provider pipeline 2 in Figure 12-5, do not have to be located in the same region as the applications that they service.

In the case of RS provider pipeline 1, the requests arriving there are for registration with the service requester application running in AOR1, and protocol response messages that refer to this application’s role in coordinating the atomic transaction. The RS provider pipeline 2 receives registration response messages and protocol notification messages intended for the service provider application running in AOR2.

Since eventually these messages must be processed in the same region as the service provider application, the WS-AT implementation in CICS TS makes use of *RequestStreams* to pass these messages from one region to another. RequestStreams are dependent on MRO connections when they exchange information between CICS regions.

Note: If a Registration Service provider pipeline is configured in a different region than the application that it services, then an MRO connection is required between the CICS regions.

The use of RequestStreams means that individual Registration services provider pipelines can be used by different applications, and those applications do not have to reside in the same region. A set of cloned AORs can share a common Registration services provider pipeline that is located in a specific WS-AT CICS region. Work can then be distributed across the AORs and each of the WS-AT messages, arriving at the Registration services provider pipeline, contains sufficient information to allow it to be forwarded to the correct AOR.

Tip: A set of cloned AORs can share a common RS provider pipeline located in a specific WS-AT region, thus reducing the number of pipeline configurations required. In this configuration, it is recommended that you do not run applications in the specific WS-AT region and that you create clones for improved failover and availability.

Figure 12-5 shows the Registration services requester pipelines are in the same region as the applications that they service. RS requester pipeline 1 services the service requester application in AOR1, and RS requester pipeline 2 services the service provider application in AOR2. This configuration cannot be changed because CICS requires that each WS-AT message must be dispatched from the same region as the application that it relates to. This means that each AOR in a cloned set of AORs used to support a common Web services workload, must have it’s own Registration services requester pipeline configured.

Note: A Registration services requester pipeline must be configured in each AOR that hosts applications which participate in atomic transactions.

12.2 Enabling atomic transactions in WebSphere

WebSphere Application Server Version 6 implements the WS-AT specification. A J2EE application programmer demarcates a global transaction in a program by using the Java Transaction API (JTA) `UserTransaction` interface as shown in Example 12-5.

Example 12-5 Demarcating a global transaction using the JTA

```
UserTransaction userTransaction = null;
try {
      InitialContext context = new InitialContext();
      userTransaction = (UserTransaction)
                             context.lookup(“java:comp/UserTransaction”);
      userTransaction.begin();

      // insert record into database
        .
        .
        .
      // commit
      userTransaction.commit();
    } catch (java.rmi.RemoteException re) {
        try {
           userTransaction.rollback();
        }
        .....
     }
```

If a Web service request is made by an application component running under a global transaction, WebSphere Application Server implicitly propagates a `CoordinationContext` to the target Web service if the appropriate application deployment descriptors have been specified.

If WebSphere Application Server is the system hosting the target endpoint for a Web service request that contains a WS-AT `CoordinationContext`, WebSphere automatically establishes a subordinate JTA transaction in the target run-time environment that becomes the transactional context under which the target Web service application executes.

Application developers do not have to explicitly register WS-AT participants. The WebSphere Application Server run time takes responsibility for the registration of WS-AT participants. At transaction completion time, all WS-AT participants are atomically coordinated by the WebSphere Application Server transaction service.

There are no specific development tasks required for Web service applications to take advantage of WS-AT; however, there are some application deployment descriptors that have to be set appropriately:

- In a Web module that invokes a Web service, specify `Send Web Services Atomic Transactions on requests` to propagate the transaction to the target Web service.

  See "Change the deployment descriptor" on page 478 for information about how we enabled our Web application to send WS-AT SOAP headers in requests to a CICS service provider application.

- In a Web module that implements a Web service, specify `Execute using Web Services Atomic Transaction on incoming requests` to run under a received client transaction context.
- In an EJB module that invokes a Web service, specify `Use Web Services Atomic Transaction` to propagate the EJB transaction to the target Web service.
- In an EJB module, bean methods must be specified with transaction type `Required`, which is the default, to participate in a global atomic transaction.

Transaction scenarios

In this chapter we show different scenarios that demonstrate how you can synchronize resource updates using the WS-Atomic Transaction support in CICS and WebSphere Application Server.

We start with an explanation of the scenarios that we set out to test, and how we prepared the system and the settings that we used for the configuration of our system.

We then cover in detail the changes that we made to the CICS pipeline configuration and WebSphere Application Server to enable Web services transactional integration.

13.1 Introduction to our scenarios

Figure 13-1 shows three different atomic transaction scenarios that we considered testing.

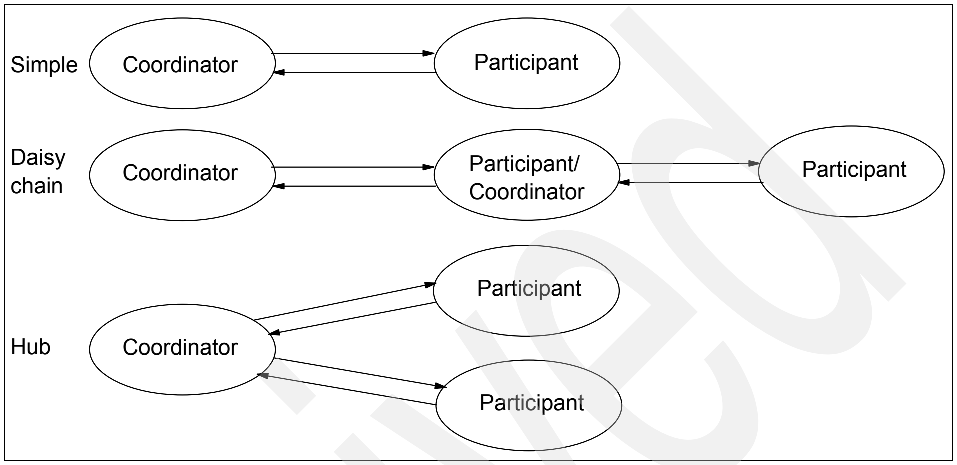


Figure 13-1 Three possible atomic transaction scenarios

The scenario at the top is the simple case where a coordinator controls a single participant. Both parties may make recoverable updates. Another possibility is that only the participant makes recoverable updates. A third possibility is that the participant does nothing recoverable; in this case it will send a `ReadOnly` notification when it is coordinated.

The second scenario is often referred to as a *daisy chain*. Here there is a primary (or root) coordinator that invokes a Web service, and the invoked Web service then invokes a second Web service. The entire atomic transaction is controlled by the primary coordinator. The middle system takes on the role of a coordinator *and* the role of a participant at different times in its life cycle. When the primary coordinator instructs it during transaction termination, this system acts as a participant. However, before it responds to the primary coordinator, it then takes on the role of coordinator of its own participant.

The third scenario is a *hub* configuration. A single coordinator invokes one Web service and then another. It then coordinates them together.

You can probably think of many other scenarios. In this chapter we more closely look at two of these scenarios:

- The simple scenario in which WebSphere Application Server V6.0 is the coordinator and CICS TS V3.1 is the participant. See “The simple atomic transaction scenario” on page 467.
- The daisy chain scenario in which CICS TS V3.1 is both a coordinator and a participant. See 13.3, “The daisy chain atomic transaction scenario” on page 510.

13.1.1 Software checklist

Table 13-1 shows the software we used in the scenarios described in this chapter.

Table 13-1 Software used in the atomic transaction scenarios

Windows	**z/OS**
Windows 2000 SP4	z/OS V1.6
IBM WebSphere Application Server - ND V6.0.0.2	CICS Transaction Server V3.1
Internet Explorer V6.0	
DB2 V8.1.7.445	
User-supplied programs: ► CatalogAtomic.ear A modified service requester application used for the WS-AT scenarios ► DispatchAtomic.ear A modified service provider application used for the WS-AT scenarios	User-supplied programs: ► SNIFFER (message handler program written in COBOL) ► WSATHND (header processing program written in C)

Important: You should install the fix for APAR PK16509 if you are using a later version of WebSphere than IBM WebSphere Application Server - Network Deployment V6.0.0.2.

13.1.2 Definition checklist

Table 13-2 shows the definitions we used in the scenarios described in this chapter.

Table 13-2 Definitions used in the atomic transaction scenarios

Value	CICS TS	WebSphere Application Server
IP name	mvsg3.mop.ibm.com	cam21-pc11.mop.ibm.com
IP address	9.100.193.167	9.100.199.238
Job name	CIWST3C1	
APPLID	A6POT3C1	
SIT parameter	SEC=NO (see APAR PK10849)	
TCPIPSERVICE definition	T3C1	
PORT attribute on T3C1 TCPIPSERVICE definition	15301	
FILE definition for sample catalog VSAM file	EXMPCAT	
RECOVERY attribute on EXMPCAT FILE definition	BACKOUTONLY	
Web service provider PIPELINE definition	PIPE1	
CONFIGFILE attribute on PIPE1 PIPELINE definition	/CIWS/T3C1/config /ITSO_7206_wsat_soap11 provider.xml	
Web service requester PIPELINE definition	PIPE2	
CONFIGFILE attribute on PIPE2 PIPELINE definition	/CIWS/T3C1/config /ITSO_7206_wsat_soap11 requester.xml	
RDO group containing copy of DFHWSAT group	CTS310C	
Registration Service provider PIPELINE	DFHWSATP	
CONFIGFILE attribute on DFHWSATP PIPELINE definition	/CIWS/T3C1/config /ITSO_7206_wsat_registr ationservicePROV.xml	
Registration Service requester PIPELINE	DFHWSATR	

Value	CICS TS	WebSphere Application Server
CONFIGFILE attribute on DFHWSATR PIPELINE definition	/CIWS/T3C1/config /ITSO_7206_wsat_registrationserviceREQ.xml	

Important: As we write this book, APAR PK10849 is open. It reports that if you attempt to use WS-AT in a CICS region that is running with the SIT parameter SEC=YES, then during the RegisterResponse step of coordination RACF will issue messages ICH408I and IRR012I and CICS will issue message DFHPI0002. Therefore, we ran our region with SEC=NO.

13.2 The simple atomic transaction scenario

The sample Catalog application provided with CICS TS V3.1 provides three Web services:

- inquireSingle
- inquireCatalog
- placeOrder

Only the placeOrder Web service *updates* the VSAM file that contains the information about the company's products. Therefore, we naturally decided that the service requester running in WebSphere Application Server should invoke the placeOrder Web service in our WS-AT scenario.

We modified the service requester application (see "Installing the service requester" on page 87) to create a global transaction and to update a DB2 table. We call the new service requester application *AtomicClient*.

Note: Rather than using JDBC™ to update a DB2 table, we could have chosen to use another J2EE connector such as the JCA or JMS. Updates to resources accessed by these connectors can be synchronized with Web service requests in the same way.

To be more specific, we create the table ITSO.ORDER in DB2. Before calling the placeOrder Web service, the AtomicClient inserts a row in this table so that we have a log of all of the orders placed through our application. We can now have a global transaction that updates two resources: a DB2 table in the Windows environment and a VSAM file in the z/OS environment. Figure 13-2 shows the sequence of events in AtomicClient as it begins a global transaction, updates a DB2 database, calls the placeOrder Web service, and then either commits or rolls back the updates.

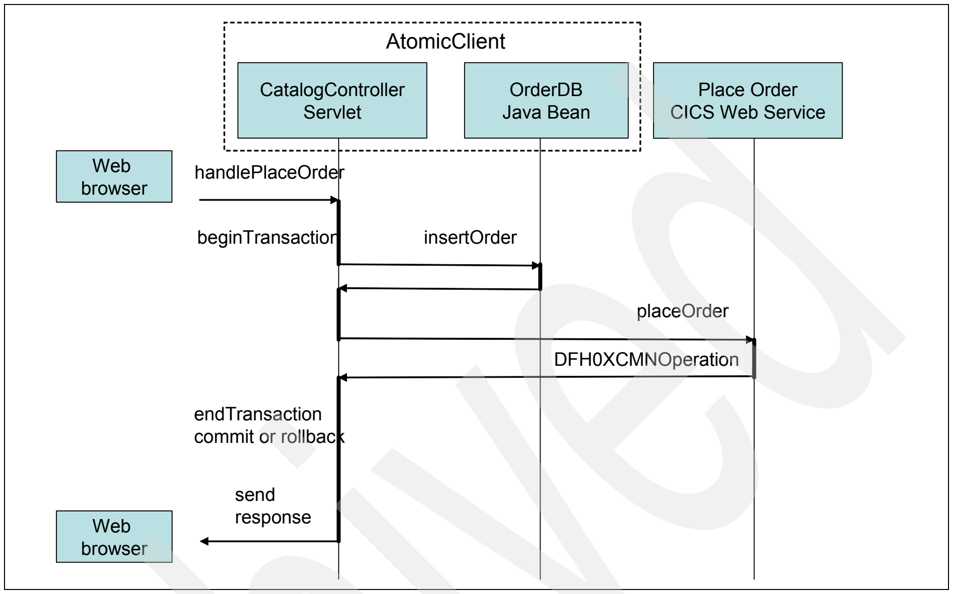


Figure 13-2 Simple atomic transaction sequence of events

Figure 13-3 shows a more global view of the sequence of events:

- The user uses his Web browser to invoke the AtomicClient that runs in WebSphere Application Server.
- The AtomicClient updates the ITSO.ORDER table in DB2.
- WebSphere Application Server sends the insertOrder SOAP message containing the order to CICS.
- CICS uses Web service provider PIPELINE definition PIPE1 to process the SOAP message. PIPE1 contains our SNIFFER program and the CICS-supplied SOAP 1.1 message handler DFHPISN1.
- DFHPISN1 links to the CICS-supplied header processing program DFHWSATH.
- CICS converts the SOAP message to a COMMAREA for the sample catalog manager program, DFH0XCMN.
- DFH0XCMN passes the data in the COMMAREA to the sample catalog program DFH0XVDS, which updates the recoverable VSAM file.

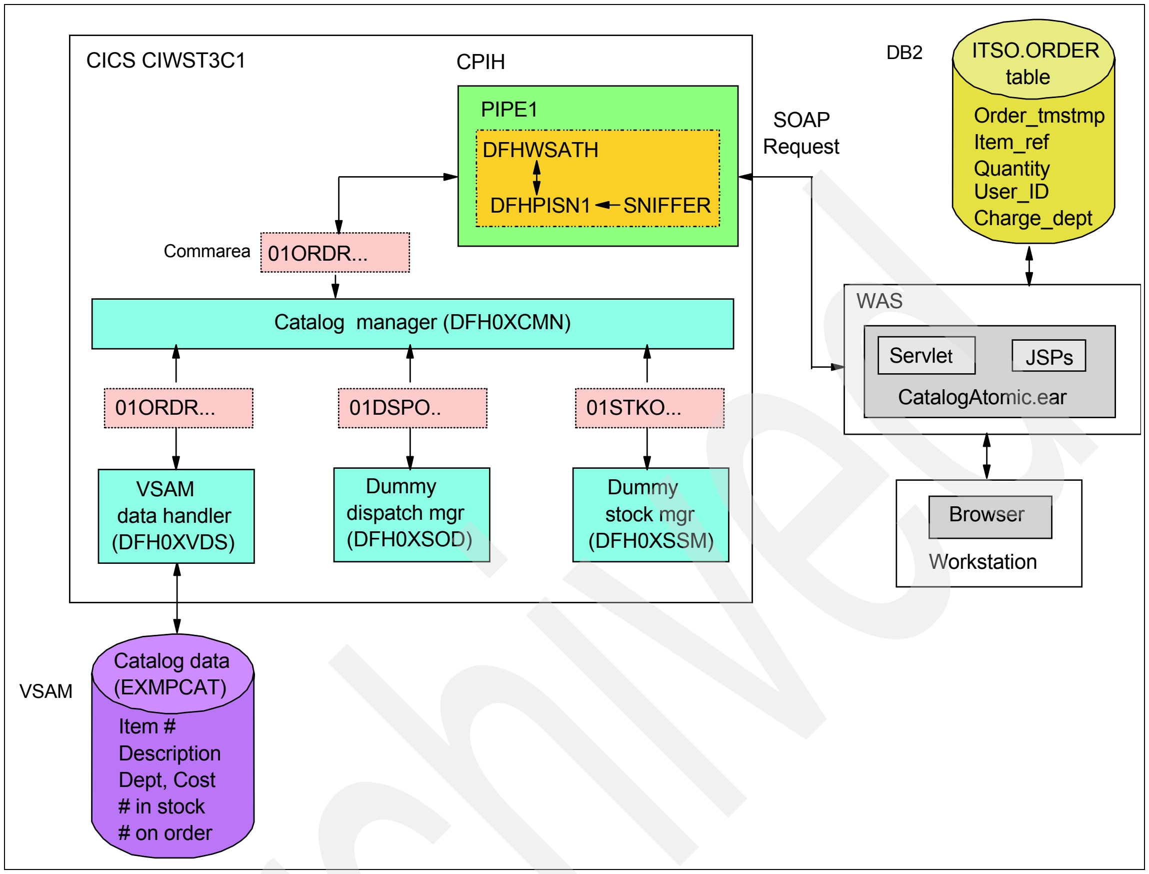


Figure 13-3 WebSphere as service requester and CICS as service provider

In the following sections we describe how we set up CICS for this scenario, how we created the AtomicClient and the ITSO.ORDER table, and the results of testing this scenario.

13.2.1 Setting up CICS for the simple scenario

To set up CICS for this scenario we performed the following steps:

- Set the value of the SEC system initialization table parameter to NO; see 13.1.2, “Definition checklist” on page 465.
- Changed the value of the RECOVERY attribute of the EXMPCAT FILE definition to BACKOUTONLY so that the file is recoverable. (The EXMPCAT FILE definition defines the VSAM file that contains the catalog data for the sample application).

- Edited PIPE1’s configuration file

  /CIWS/T3C1/config/ITSO_7206_wsat_soap11provider.xml

  so that it contains the XML shown in Example 13-1. This XML contains:

  – The same XML as shown in Example 12-4 on page 456, except that we changed the address in the `registration_service_endpoint` element to be specific to the CICS region CIWST3C1
  – XML to add the SNIFFER message handler

Example 13-1 PIPE1: /CIWS/T3C1/config/ITSO_7206_wsat_soap11provider.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline
      xmlns="http://www.ibm.com/software/htp/cics/pipeline"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
      provider.xsd ">
  <service>
    <service_handler_list>
      <handler>
        <program>SNIFFER</program>
        <handler_parameter_list/>
      </handler>
    </service_handler_list>
    <terminal_handler>
      <cics_soap_1.1_handler>
        <headerprogram>
          <program_name>DFHWSATH</program_name>
          <namespace>
             http://schemas.xmlsoap.org/ws/2004/10/wscoor
          </namespace>
          <localname>CoordinationContext</localname>
          <mandatory>false</mandatory>
        </headerprogram>
      </cics_soap_1.1_handler>
    </terminal_handler>
  </service>
  <apphandler>DFHPITP</apphandler>
  <service_parameter_list>
    <registration_service_endpoint>
    http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
    </registration_service_endpoint>
  </service_parameter_list>
</provider_pipeline>
```

Tip: For readability Example 13-1 shows the `registration_service_endpoint` element split across three lines. However, during our testing we found that we had to place its start tag, contents, and end tag on the *same* line.

- Copied the group DFHWSAT to the group CTS310C and added the group CTS310C to our startup list:

  ```
  CEDA COPY GROUP(DFHWSAT) TO(CTS310C)
  CEDA ADD GROUP(CTS310C) LIST(LISTT3C1)
  ```

- Edited DFHWSATP's configuration file

  /CIWS/T3C1/config/ITSO_7206_wsat_registrationservicePROV.xml

  so that it contained the XML shown in Example 13-2. This XML contains:

  – The same XML as shown in Example 12-1 on page 452

  – XML to add the WSATHND header processing program (see "How we monitored the exchange of WS-AT messages" on page 472)

Example 13-2 DFHWSATP: /CIWS/.../ITSO_7206_wsat_registrationservicePROV.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<provider_pipeline
  xmlns="http://www.ibm.com/software/htp/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
  provider.xsd ">
  <service>
    <service_handler_list>
      <cics_soap_1.2_handler>
        <headerprogram>
          <program_name>WSATHND</program_name>
          <namespace>*</namespace>
          <localname>wsatHeader</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.2_handler>
    </service_handler_list>
    <terminal_handler>
      <handler>
        <program>DFHWSATX</program>
        <handler_parameter_list/>
      </handler>
    </terminal_handler>
  </service>
  <service_parameter_list/>
</provider_pipeline>
```

–

- Edited DFHWSATR's configuration file

  /CIWS/T3C1/config/ITSO_7206_wsat_registrationserviceREQ.xml

  so that it contained the XML shown in Example 13-3. This XML contains

  – The same XML shown in Example 12-2 on page 453

  – XML to add the WSATHND header processing program

Example 13-3 DFHWSATR: /CIWS/.../ITSO_7206_wsat_registrationserviceREQ.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<requester_pipeline
    xmlns="http://www.ibm.com/software/htp/cics/pipeline"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
    requester.xsd">
  <service>
    <service_handler_list>
      <cics_soap_1.2_handler>
        <headerprogram>
          <program_name>WSATHND</program_name>
          <namespace>*</namespace>
          <localname>wsatHeader</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.2_handler>
    </service_handler_list>
  </service>
</requester_pipeline>
```

How we monitored the exchange of WS-AT messages

The user-written WSATHND header processing program writes messages to the CESO transient data queue, which normally is an extrapartition queue with DDname CEEOUT. We used WSATHND to monitor the exchange of registration and protocol service messages between WebSphere Application Server and CICS.

The message output of WSATHND depends on the values the programmer has assigned to the two variables MESSAGES_ON and FULL_MESSAGES_ON:

1. If MESSAGES_ON is set to 1, the program writes the following message:

   WSAT: REACHED HANDLER - function

   where function is the contents of the DFHFUNCTION container (RECEIVE-REQUEST, SEND-RESPONSE, SEND-REQUEST, RECEIVE-RESPONSE, PROCESS-REQUEST, NO-RESPONSE, or HANDLER-ERROR).

When the function is NO-RESPONSE, then the handler is being invoked after processing a request, when there is no response to be processed.

2. If the function is not NO-RESPONSE, then:
   - If FULL_MESSAGES_ON is set to 1, WSATHND writes the following message:

     WSAT: contents of the DFHREQUEST container

   - If MESSAGES_ON is set to 1, it will write the following message:

     WSAT: ACTION: action

     where action has one of the following values: Register, RegisterResponse, Prepare, Prepared, Commit, Committed, ReadOnly, Abort, Aborted, Rollback.

Thus, WSATHND provides a way to see the registration and protocol service messages that are being sent between the service requester and the service provider as they happen. We found it easier to use this program than to take a TCP/IP trace or to search for these messages in a CICS auxiliary trace. The full WSATHND program is shown A.7, “Sample header processing program - WSATHND” on page 567.

13.2.2 Creating the AtomicClient and ITSO.ORDER table

To create the AtomicClient and the ITSO.ORDER table, we did the following:

- Created the OrderBean JavaBean that represents the order.
- Created the OrderDB JavaBean that inserts the order into the DB2 database.
- Changed the handlePlaceOrder method of the CatalogController servlet so that it creates an OrderBean and then calls OrderDB to insert it into the DB2 database as part of a global transaction.
- Changed the deployment descriptor of the Web module CatalogAtomicWeb. This is the module that invokes the Web Service.
- Created the ITSO.ORDER table.
- Created a data source.

We explain each step in detail in the following sections.

Create the OrderBean JavaBean

The OrderBean JavaBean represents the order; it has all the fields from the PlaceOrder JavaServer™ Page (JSP™) plus a timestamp that records the time when the Web browser user placed the order. Example 13-4 shows the code for the OrderBean JavaBean.

Example 13-4 The OrderBean JavaBean

```
package itso.mop.objects;

import java.sql.Timestamp;

public class OrderBean {

   private Timestamp orderTmstmp;
   private short itemRef;
   private short quantity;
   private String userId;
   private String chargeDept;

// getters and setters
...
}
```

Create the OrderDB JavaBean

Example 13-5 shows the code for the OrderDB JavaBean, which inserts the order into the DB2 table.

Example 13-5 The OrderDB JavaBean

```
package itso.mop.db;

import itso.mop.objects.OrderBean;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class OrderDB {

   private static final String _JNDI_NAME = "jdbc/ORDERDS";

   public int insertOrder(OrderBean order){
      Connection con = getConnection();
      PreparedStatement pm = null;
      int rs = 0;
      try {
         pm = con.prepareStatement("INSERT INTO ITSO.ORDER (ORDER_TMSTMP, 
ITEM_REF, QUANTITY, USER_ID, CHARGE_DEPT) VALUES(?,?,?,?,?)");
         pm.setTimestamp(1, order.getOrderTmstmp());
         pm.setInt(2, order.getItemRef());
         pm.setInt(3, order.getQuantity());
```

```
          pm.setString(4, order.getUserId());
          pm.setString(5, order.getChargeDept());

          rs = pm.executeUpdate();

          System.out.println("OrderDB.insertOrder() - inserted the order in the 
database!!!!");
       } catch (SQLException e) {
          System.out.println("OrderDB.insertOrder() - Exception inserting the 
order in the database!!");
          e.printStackTrace(System.err);
          rs = 0;
       } finally {
          try {
             if (pm != null)
                pm.close();
             if (con != null)
                con.close();
          } catch (Exception e) {
             e.printStackTrace(System.err);
          }
       }
       return rs;
    }

    private Connection getConnection() {
       Connection con = null;
       try {
          InitialContext ic = new InitialContext();
          DataSource ds = (DataSource) ic.lookup(_JNDI_NAME);
          con = ds.getConnection();
       } catch (NamingException e) {
          e.printStackTrace(System.err);
       } catch (SQLException e) {
          e.printStackTrace(System.err);
       }
       return con;
    }
}
```

Change the handlePlaceOrder method

In the Catalog application the CatalogController servlet handles all the requests from the Web browser. In particular, the `handlePlaceOrder` method of this servlet handles requests from the Web browser to place an order. We made the following changes to this method:

1. Created the OrderBean and populated all of its fields

2. Created the transaction (UserTransaction) and then began the transaction
3. Inserted the order into the DB2 database using the OrderDB JavaBean
4. Called the CICS Web service
5. Coded the method to throw a RemoteException when the “ROLLBACK” user ID enters the order (for testing transaction rollback)
6. Committed the transaction
7. Rolled back the transaction in the case of an Exception

Note: The rollback logic was added to the application for testing purposes only; normal applications would not be expected to behave in this way.

Example 13-6 shows the code for the `handlePlaceOrder` method.

Example 13-6 The handlePlaceOrder method of the CatalogController servlet

```
private void handlePlaceOrder(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException
{
   OrderDetails orderDetails = new OrderDetails();
   try
   {
      // Retrieve the data from the submitted form
      orderDetails.setItemRefNumber(
short.parseShort(request.getParameter("itemRef")));
      orderDetails.setQuantityRequired(Short.parseShort(
         request.getParameter("quantity")));
      orderDetails.setUserId(request.getParameter("userName"));
      orderDetails.setChargeDepartment(request.getParameter("deptName"));
   }
   catch(Exception e)
   {
      handleError(request, response, "Input data format incorrect, please try
again");
      return;
   }

   UserTransaction userTransaction = null;
   try
   {
      //1. Create the order bean and populate all of its fields
      System.out.println("CatalogController.handlePlaceOrder() - creating the
order");
      OrderBean order = new OrderBean();
      order.setOrderTmstmp(new Timestamp(Calendar.getInstance().
          getTime().getTime()));
      order.setItemRef(orderDetails.getItemRefNumber());
      order.setQuantity(orderDetails.getQuantityRequired());
      order.setUserId(orderDetails.getUserId());
      order.setChargeDept(orderDetails.getChargeDepartment());

      // 2. Create and begin the Transaction
      InitialContext context = new InitialContext();
      userTransaction = (UserTransaction)
                      context.lookup("java:comp/UserTransaction");
      System.out.println("CatalogController.handlePlaceOrder() - beginning the 
transaction");
      userTransaction.begin();

      // 3. Insert the order into the database
      System.out.println("CatalogController.handlePlaceOrder() - inserting the 
order in the database");
      OrderDB orderDB = new OrderDB();
      orderDB.insertOrder(order);

      // 4. Make the Web service call to CICS
      System.out.println("CatalogController.handlePlaceOrder() - calling the 
CICS web service");
      orderDetails.populateResponse(
getOrderProxy(request).DFHOXCMNOperation(orderDetails.getRequestProgramInterfac
e()) );
      System.out.println("CatalogController.handlePlaceOrder() - response back 
from the CICS web service");
      // 5. Throw exception if user ID ROLLBACK
      if(order.getUserId().equalsIgnoreCase("ROLLBACK"))
      {
         System.out.println("CatalogController.handlePlaceOrder() - simulating 
the RemoteException");
         throw new RemoteException("Throwing the RemoteException");
      }
      // 6. Commit the transaction
      System.out.println("CatalogController.handlePlaceOrder() - commit the 
transaction");
      userTransaction.commit();
      System.out.println("CatalogController.handlePlaceOrder() - after 
commit");
   }
      // 7. Rollback in case of exception
   catch (Exception e)
   {
      try {
         System.out.println("CatalogController.handlePlaceOrder() - 
rollingback the transaction");
         userTransaction.rollback();
```

```
      } catch (Exception e1) {
         System.out.println("CatalogController.handlePlaceOrder() - Exception
when rollingback the transaction");
         e1.printStackTrace();
      }

      if (e.getMessage().startsWith("java.net.ConnectException"))
      {
         String errorMessage = "Unable to connect to service endpoint: "+
                            getOrderProxy(request).getEndpoint()+
                            "<BR> Please ensure service is running.";
         handleError(request, response, errorMessage);
         return;
      }
      else
      {
         e.printStackTrace();
         handleError(request, response, "An Error occured calling the service:
"+e.getMessage());
         return;
      }
   }
   // Call the response page setting the OrderDetails bean on the request
   RequestDispatcher dispatcher =
request.getRequestDispatcher(orderResponsePage);
   request.setAttribute("orderDetails",orderDetails);
   dispatcher.forward(request,response);
}
```

Note: Since the service requester application is a Web application (as opposed to an enterprise bean) we used a bean-managed transaction. For an enterprise bean, it is generally recommended to use container-managed transactions.

Change the deployment descriptor

The deployment descriptor of a J2EE application is used to specify whether the application component, if it makes any Web service requests, expects any transaction context to be propagated with the Web service requests (in accordance with the WebSphere WS-AT support).

To activate the WS-AT support:

- We imported the client application archive CatalogAtomic.ear into RAD. We then expanded the **Dynamic Web Project** (CatalogAtomicWeb) in the Project Explorer and opened (double-clicked) the **Deployment Descriptor** (Figure 13-4).

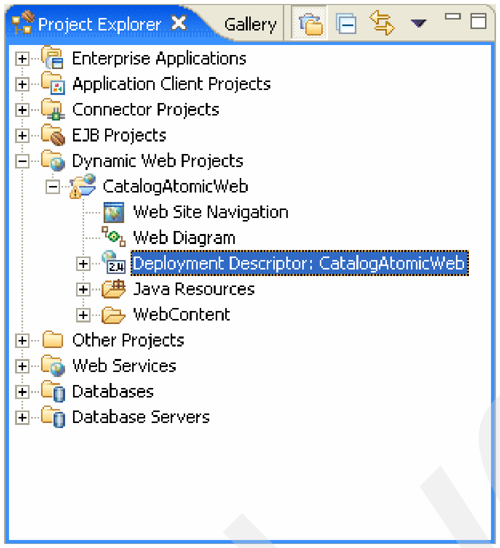
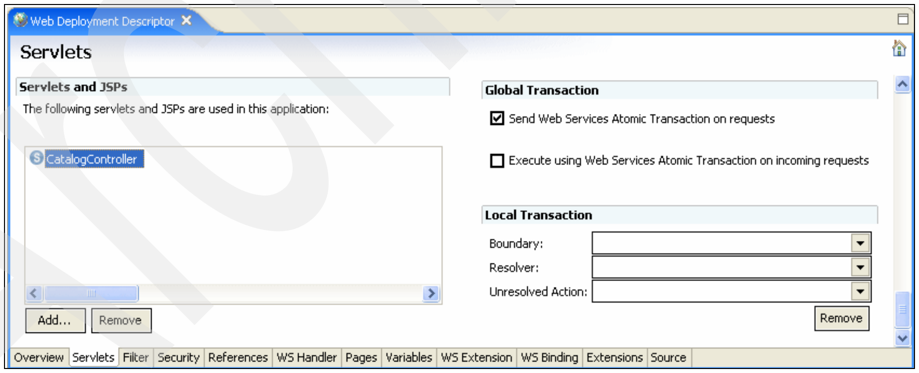


Figure 13-4 Project Explorer for CatalogAtomicWeb application

► In the Deployment Descriptor we clicked the **Servlets** tab and selected the **CatalogController** servlet. Then we scrolled down to Global Transaction and selected **Send Web Services Atomic Transactions on requests** (Figure 13-5).



Servlets and JSPs

The following servlets and JSPs are used in this application:

CatalogController

Global Transaction

[✓] Send Web Services Atomic Transaction on requests

[ ] Execute using Web Services Atomic Transaction on incoming requests

Local Transaction

Boundary:

Resolver:

Unresolved Action:

Figure 13-5 Activating atomic transaction in the web deployment descriptor

► We saved and closed the file.

Note: A similar deployment descriptor attribute **Use Web Services Atomic Transaction** can be used for enterprise beans.

Create the ITSO.ORDER table

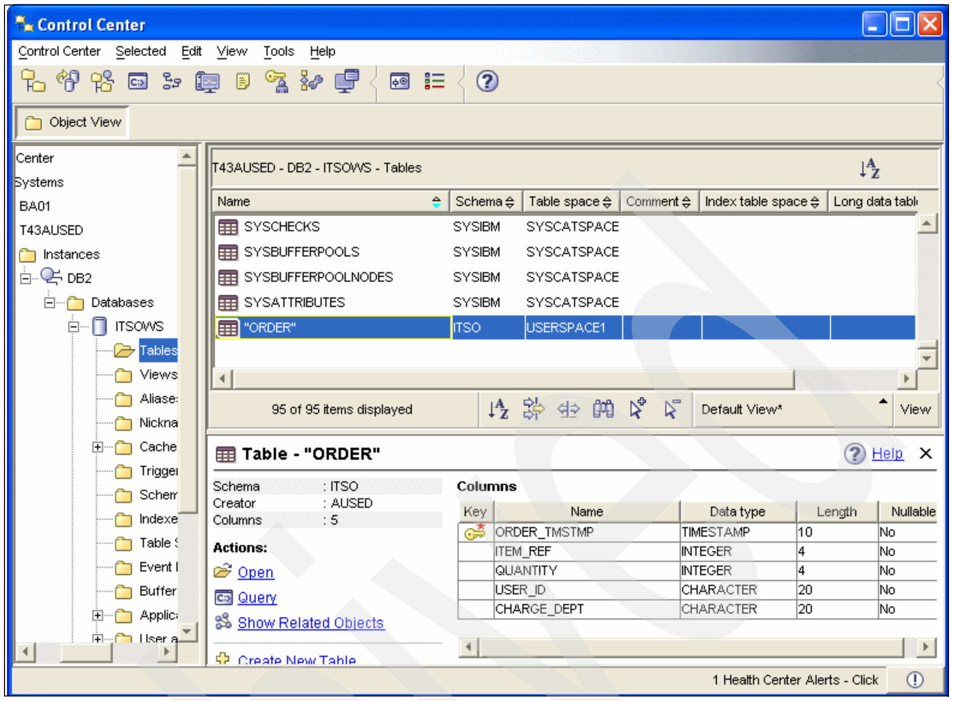
We created the ITSO.ORDER table to record all the orders that the Web browser user sends to the Catalog application. Example 13-7 shows the script we used to create the database and the table.

Example 13-7 Creation of ITSOWS database and ITSO.ORDER table

```
-- IBM ITSO
CONNECT RESET;
-- Create database ITSOWS and schema ITSO
CREATE DATABASE ITSOWS;
CONNECT TO ITSOWS;
CREATE SCHEMA ITSO;

-- Table definitions for Order
CREATE TABLE ITSO.ORDER
  (ORDER_TMSTMP TIMESTAMP  NOT NULL ,
   ITEM_REF INTEGER  NOT NULL ,
   QUANTITY INTEGER  NOT NULL ,
   USER_ID CHARACTER (20)  NOT NULL ,
   CHARGE_DEPT CHARACTER (20)  NOT NULL  ,
   CONSTRAINT ORDERKEY PRIMARY KEY ( ORDER_TMSTMP)  ) ;
```

The ITSO.ORDER table has a column for every field in the Catalog Place Order JSP. Also, there is a TimeStamp column that is used as a unique key for the table. Figure 13-6 shows the ITSOWS database in the Control Center.



Control Center

T43AUSED - DB2 - ITSOWS - Tables

Name	Schema	Table space	Comment	Index table space	Long data tabl
SYSCHECKS	SYSIBM	SYSCATSPACE			
SYSBUFFERPOOLS	SYSIBM	SYSCATSPACE			
SYSBUFFERPOOLNODES	SYSIBM	SYSCATSPACE			
SYSATTRIBUTES	SYSIBM	SYSCATSPACE			
"ORDER"	ITSO	USERSPACE1			

95 of 95 items displayed

Table - "ORDER"

Schema : ITSO
Creator : AUSED
Columns : 5

Columns

Key	Name	Data type	Length	Nullable
🔑	ORDER_TMSTMP	TIMESTAMP	10	No
	ITEM_REF	INTEGER	4	No
	QUANTITY	INTEGER	4	No
	USER_ID	CHARACTER	20	No
	CHARGE_DEPT	CHARACTER	20	No

Actions:
Open
Query
Show Related Objects
Create New Table

1 Health Center Alerts - Click

Figure 13-6 The ITSOWS database in the Control Center

Create a WebSphere data source

Next we configured the data source in WebSphere through the Administrative Console.

- We opened the Admin console by pointing a Web browser at:

  `http://cam21-pc11:9060/admin`

- In the main window, we opened **Resources** and then clicked **JDBC Providers** (Figure 13-7).

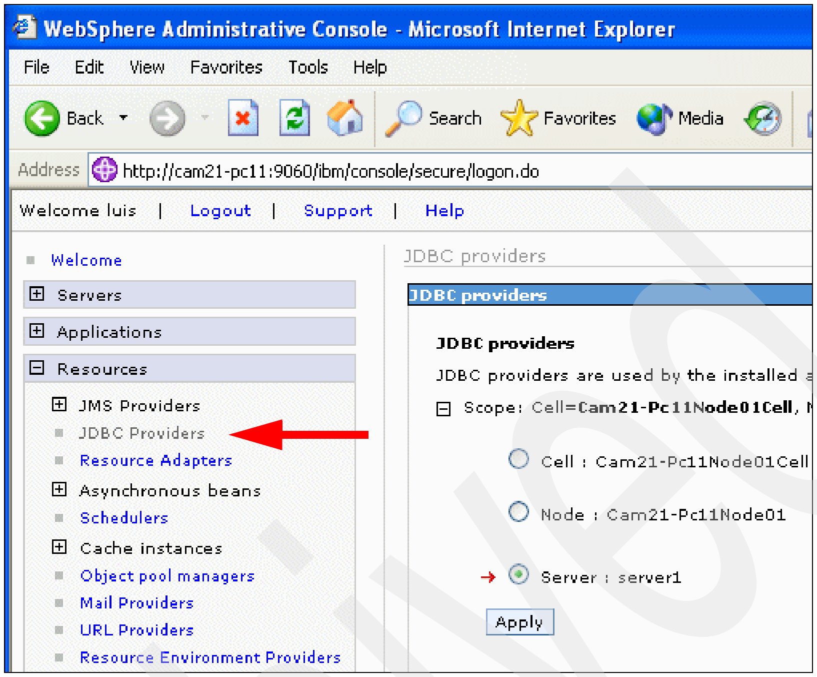
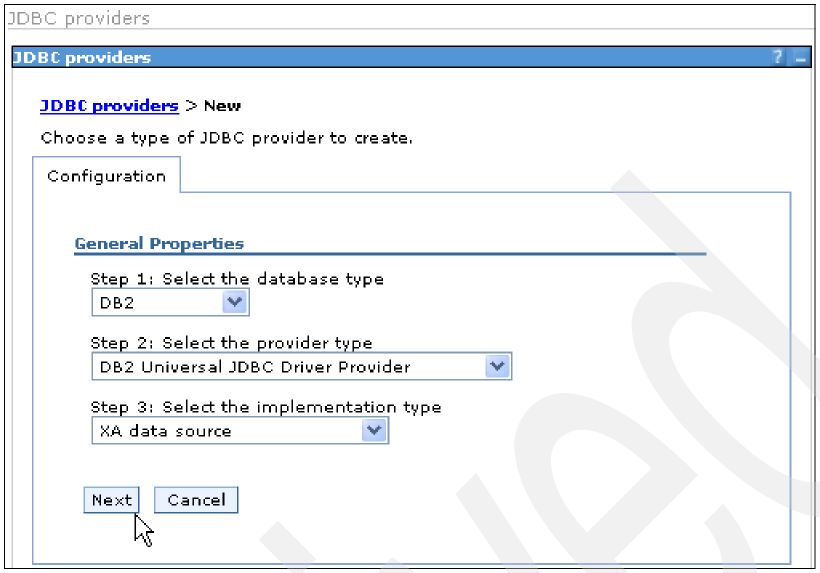

WebSphere Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://cam21-pc11:9060/ibm/console/secure/logon.do
Welcome luis | Logout | Support | Help
Welcome
Servers
Applications
Resources
JMS Providers
JDBC Providers
Resource Adapters
Asynchronous beans
Schedulers
Cache instances
Object pool managers
Mail Providers
URL Providers
Resource Environment Providers
JDBC providers
JDBC providers
JDBC providers
JDBC providers are used by the installed a
Scope: Cell=Cam21-Pc11Node01Cell, N
Cell : Cam21-Pc11Node01Cell
Node : Cam21-Pc11Node01
Server : server1
Apply

Figure 13-7 Admin console - JDBC providers

- Because we are running a single server, we chose the Server scope.
- To create a new JDBC provider we clicked **New**.


JDBC providers

JDBC providers

JDBC providers > New

Choose a type of JDBC provider to create.

Configuration

General Properties

Step 1: Select the database type
DB2

Step 2: Select the provider type
DB2 Universal JDBC Driver Provider

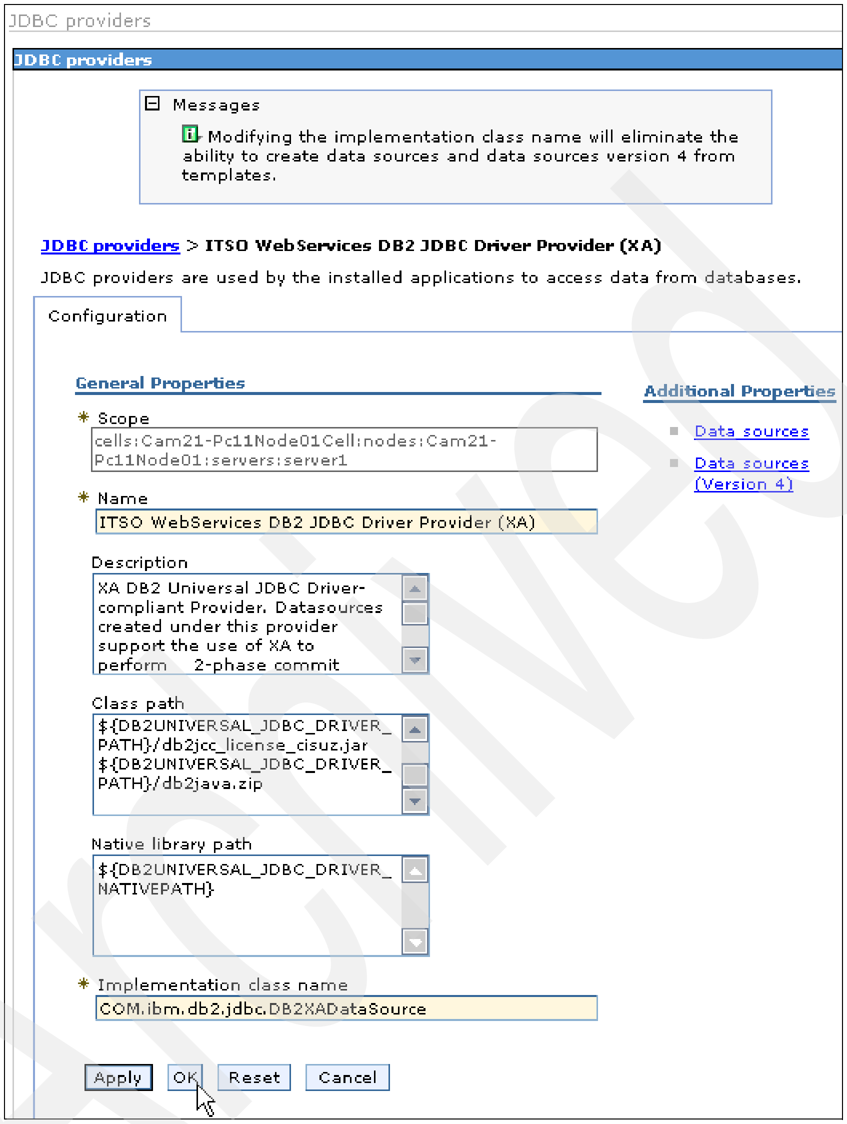
Step 3: Select the implementation type
XA data source

Next Cancel

Figure 13-8 Admin console - New JDBC provider

- In the new JDBC provider window (Figure 13-8), we selected:
  - DB2 as database type
  - DB2 Universal JDBC Driver Provider as provider type
  - XA data source as the implementation type

  We clicked **Next**, and we were then presented with the window shown in Figure 13-9.


JDBC providers

JDBC providers

Messages

Modifying the implementation class name will eliminate the ability to create data sources and data sources version 4 from templates.

JDBC providers > ITSO WebServices DB2 JDBC Driver Provider (XA)

JDBC providers are used by the installed applications to access data from databases.

Configuration

General Properties

* Scope: cells:Cam21-Pc11Node01Cell:nodes:Cam21-Pc11Node01:servers:server1

* Name: ITSO WebServices DB2 JDBC Driver Provider (XA)

Description: XA DB2 Universal JDBC Driver-compliant Provider. Datasources created under this provider support the use of XA to perform 2-phase commit

Class path:
```
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2jcc_license_cisuz.jar
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2java.zip
```

Native library path: ${DB2UNIVERSAL_JDBC_DRIVER_NATIVEPATH}

* Implementation class name: COM.ibm.db2.jdbc.DB2XADataSource

Additional Properties

- Data sources
- Data sources (Version 4)

Figure 13-9 Admin console - New JDBC provider 2

- We provided a name for the the JDBC provider, in this instance:

  `ITSO WebServices DB2 JDBC Driver Provider (XA)`

- We noted that WebSphere offers the Implementation class name COM.ibm.db2.jdbc.DB2XADataSource. This is the class of the DB2 driver that has two-phase commit capability. Since this class is not in the default class path provided by WebSphere, we added to the end of the class path the following:

```
${DB2UNIVERSAL_JDBC_DRIVER_PATH}/db2java.zip
```

- We clicked **OK** and saved the configuration changes.

Now we have the new JDBC provider defined and we can see it in the list of JDBC providers.

Define the DB2UNIVERSAL_JDBC_DRIVER_PATH variable

We defined the DB2UNIVERSAL_JDBC_DRIVER_PATH WebSphere variable as follows:

- In the Admin console main window we clicked **Environment** and then clicked **WebSphere Variables** as shown in Figure 13-10.

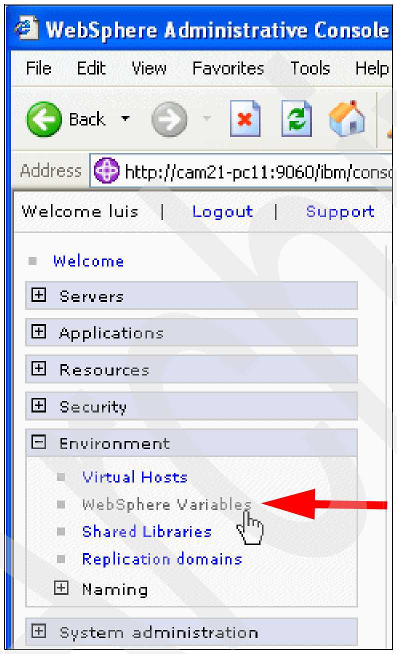
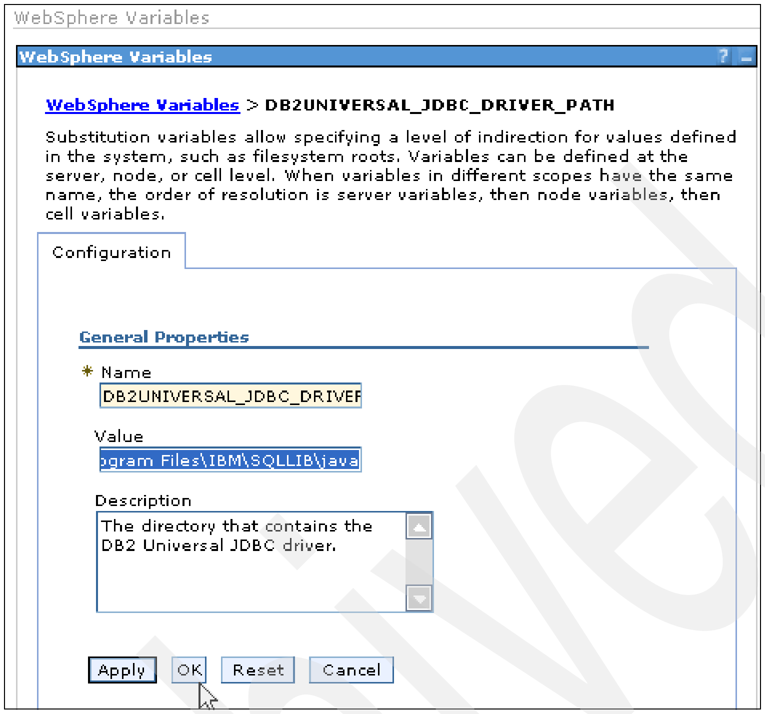


Figure 13-10 Admin console - WebSphere variables window

- We set the value of the DB2UNIVERSAL_JDBC_DRIVER_PATH variable to the default JDBC driver location C:\Program Files\IBM\SQLLIB\java (Figure 13-11).


WebSphere Variables

WebSphere Variables

WebSphere Variables > DB2UNIVERSAL_JDBC_DRIVER_PATH

Substitution variables allow specifying a level of indirection for values defined in the system, such as filesystem roots. Variables can be defined at the server, node, or cell level. When variables in different scopes have the same name, the order of resolution is server variables, then node variables, then cell variables.

Configuration

General Properties

* Name
DB2UNIVERSAL_JDBC_DRIVEF

Value
ogram Files\IBM\SQLLIB\java

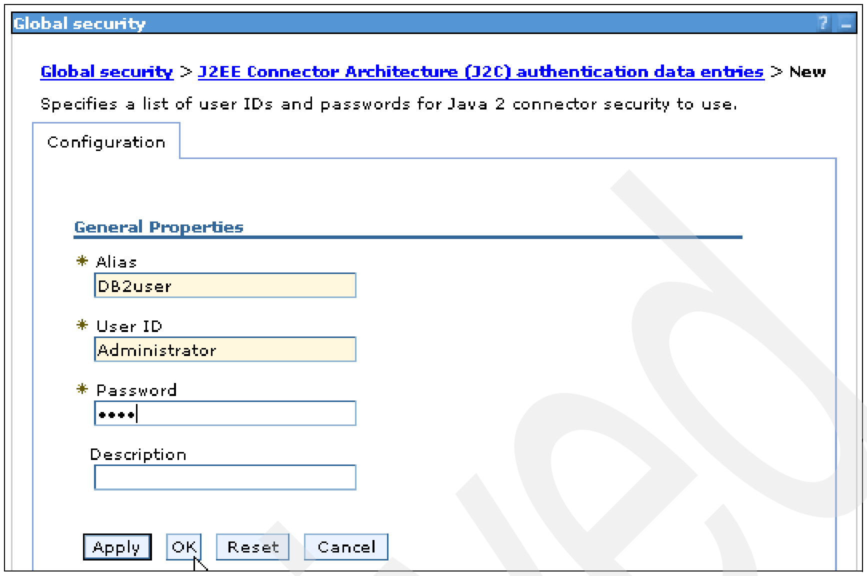
Description
The directory that contains the DB2 Universal JDBC driver.

Figure 13-11 Admin console - Defining the DB2 driver path variable

Create the J2C Authentication data

Before creating the data source, we needed to create the J2EE Connector Architecture (J2C) Authentication data in JAAS Configuration.

- In the Admin console main window we selected **Security** and then **Global Security**.
- In the Global security page we clicked **JAAS Configuration under Authentication**, and selected **J2C Authentication data**.
- In the **J2EE Connector Architecture (J2C) authentication data entries** page we clicked **New**. The window shown in Figure 13-12 was displayed.


Global security

Global security > J2EE Connector Architecture (J2C) authentication data entries > New

Specifies a list of user IDs and passwords for Java 2 connector security to use.

Configuration

General Properties

* Alias: DB2user

* User ID: Administrator

* Password: ••••

Description:

Figure 13-12 Admin console - Define J2C authentication alias

- In the General Properties we entered the following values:
  - DB2user as Alias
  - Administrator as User ID
  - User password as Password
- We clicked **OK** and saved the configuration changes.

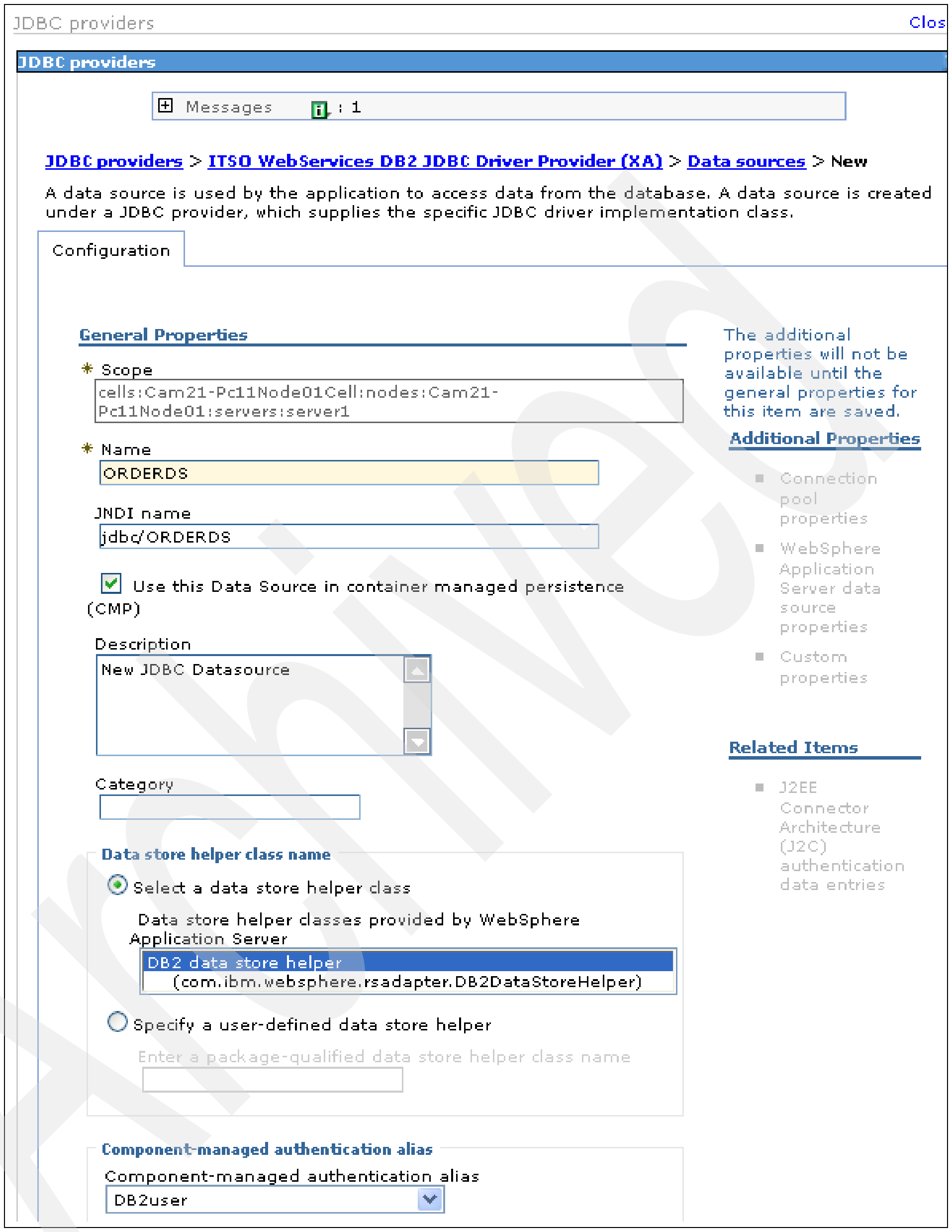
Configure the data source for the JDBC provider

Next we configured the data source for the JDBC provider.

- In the Admin console main window we selected **Resources** and then **JDBC Providers**.
- In the JDBC providers window, we clicked the new provider that we defined in "Create a WebSphere data source" on page 481:

```
ITSO WebServices DB2 JDBC Driver Provider (XA)
```

- In **Additional Properties** we clicked **Data sources**, then clicked **New**. We were presented with the window shown in Figure 13-13.


JDBC providers Clos

JDBC providers

⊞ Messages : 1

JDBC providers > ITSO WebServices DB2 JDBC Driver Provider (XA) > Data sources > New

A data source is used by the application to access data from the database. A data source is created under a JDBC provider, which supplies the specific JDBC driver implementation class.

Configuration

General Properties

* Scope
cells:Cam21-Pc11Node01Cell:nodes:Cam21-Pc11Node01:servers:server1

* Name
ORDERDS

JNDI name
jdbc/ORDERDS

✓ Use this Data Source in container managed persistence (CMP)

Description
New JDBC Datasource

Category

Data store helper class name

● Select a data store helper class

Data store helper classes provided by WebSphere Application Server
DB2 data store helper (com.ibm.websphere.rsadapter.DB2DataStoreHelper)

○ Specify a user-defined data store helper

Enter a package-qualified data store helper class name

Component-managed authentication alias

Component-managed authentication alias
DB2user

The additional properties will not be available until the general properties for this item are saved.

Additional Properties

- Connection pool properties
- WebSphere Application Server data source properties
- Custom properties

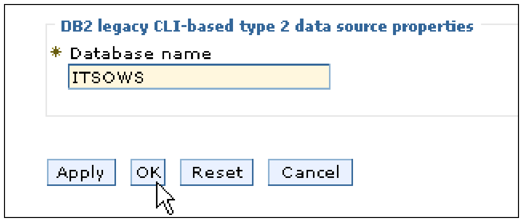
Related Items

- J2EE Connector Architecture (J2C) authentication data entries

Figure 13-13 Admin console - New Data source (1 of 2)

- In the General Properties we entered the following values:
  – ORDERDS as the Data source name.

  - jdbc/ORDERDS as the JNDI name.
  - DB2user as the Component-managed authentication alias.
- We then scrolled down to the bottom of the window and entered ITSOWS as the Database name (Figure 13-14).



* Database name: ITSOWS

Figure 13-14 Admin console - New data source (2 of 2)

- We clicked **OK** and then saved the changes.

13.2.3 Testing the simple scenario

We performed two tests of the simple scenario:

- Normal transaction termination
- Abnormal transaction termination (see “Simple scenario: abnormal transaction termination” on page 507)

Simple scenario: normal transaction termination

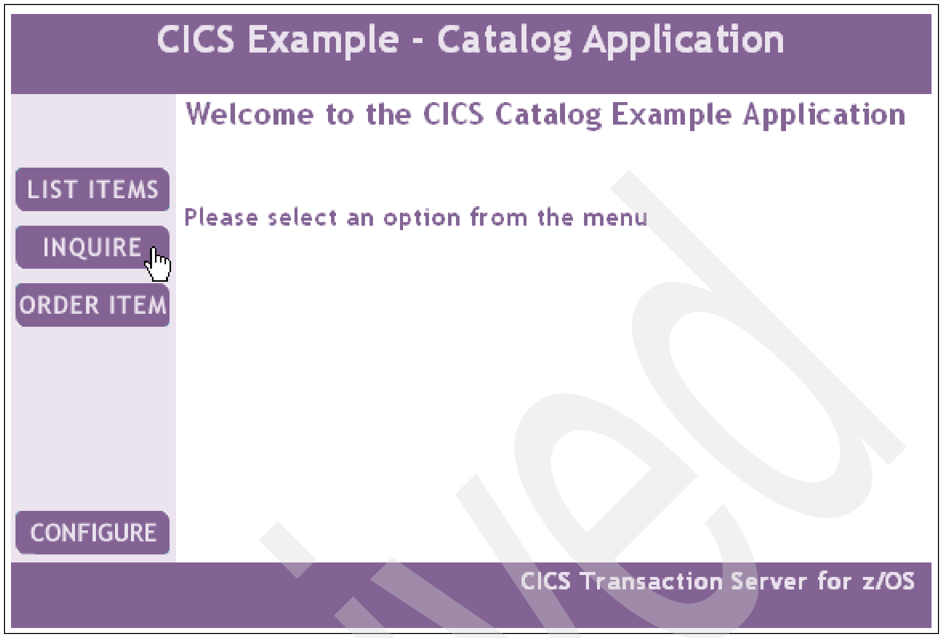
In this section, we explain how we ran the AtomicClient application and then show the registration and protocol service messages that are exchanged between WebSphere Application Server and CICS during the normal termination of the atomic transaction.

To run the scenario we followed these steps:

- We opened the welcome window of the Catalog application using the URL:

  `http://cam21-pc11:9080/CatalogAtomicWeb/Welcome.jsp`

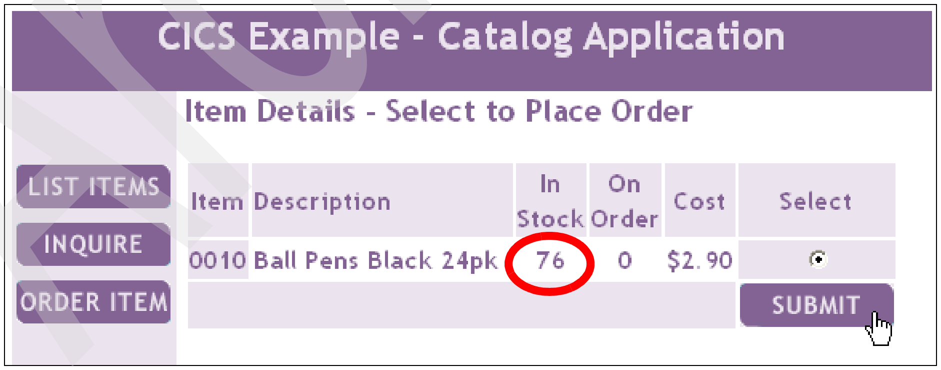
  We were presented with the window shown in Figure 13-15.


CICS Example - Catalog Application
Welcome to the CICS Catalog Example Application
LIST ITEMS
INQUIRE
ORDER ITEM
CONFIGURE
CICS Transaction Server for z/OS

Please select an option from the menu

Figure 13-15 Catalog Application - Welcome window

- We clicked **INQUIRE**.
- In the Inquire Single window we used the Item Reference Number default value of 0010 and clicked **SUBMIT**. The Web service request was sent to CICS and we were presented with the results of the inquiry as shown in Figure 13-16.

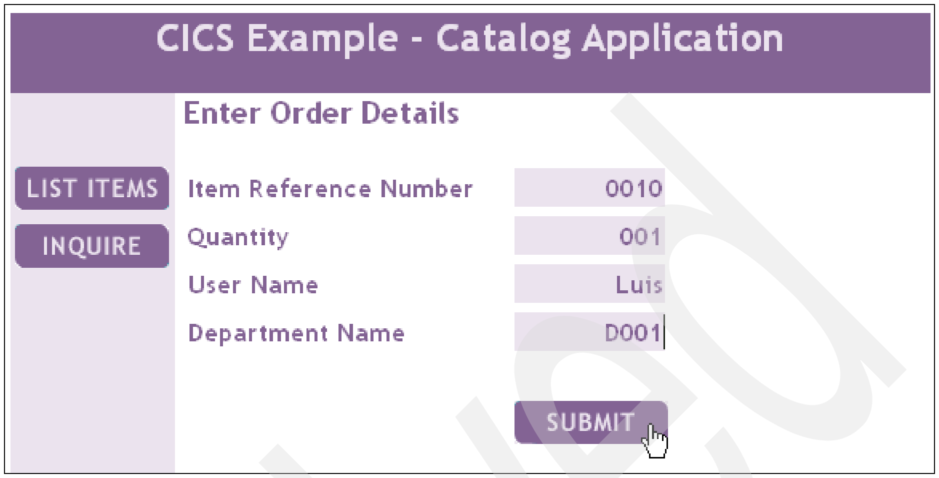

CICS Example - Catalog Application
Item Details - Select to Place Order
LIST ITEMS
INQUIRE
ORDER ITEM
SUBMIT

Item	Description	In Stock	On Order	Cost	Select
0010	Ball Pens Black 24pk	76	0	$2. 90	●

Figure 13-16 Catalog Application - Inquire single

- We noted that the number of items in stock was 76. This value is taken from the CICS VSAM file.

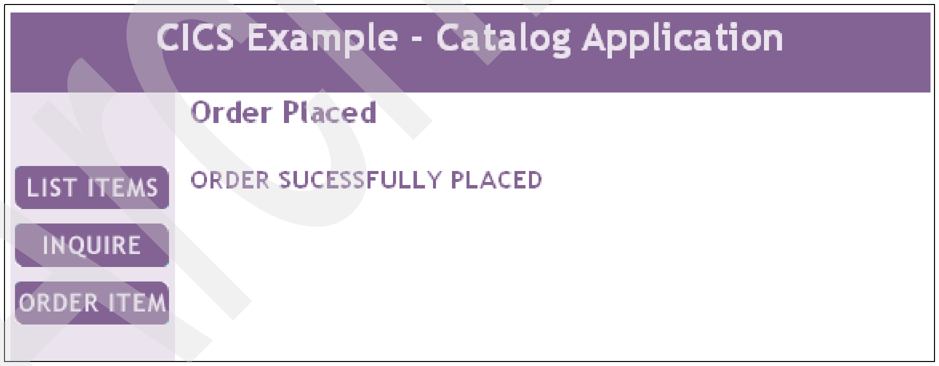
- We clicked **SUBMIT** to go to the Enter Order Details window shown in Figure 13-17.



Item Reference Number	0010
Quantity	001
User Name	Luis
Department Name	D001

Figure 13-17 Catalog Application - Enter order details

- In the Enter Order Details window we provided a User Name and a Department Name and clicked **SUBMIT**.
- After the CICS Web service processed the order, we got a response telling us that the order was successfully placed (Figure 13-18).



ORDER SUCESSFULLY PLACED

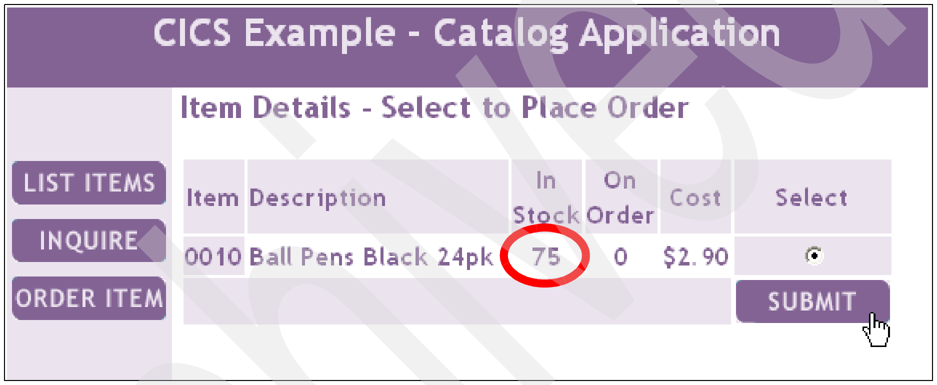
Figure 13-18 Catalog Application - Order placed response

In the WebSphere server log we see trace entries, generated by the Catalog Controller servlet, which show that the transaction was successfully committed (Example 13-8).

Example 13-8 WebSphere server log with successful Place Order

```
CatalogContoller:doPost() - Action = Place Order
CatalogController.handlePlaceOrder() - creating the order
CatalogController.handlePlaceOrder() - beginning the transaction
CatalogController.handlePlaceOrder() - inserting the order in the database
OrderDB.insertOrder() - inserted the order in the database!!!!
CatalogController.handlePlaceOrder() - calling the CICS web service
CatalogController.handlePlaceOrder() - response back from the CICS web service
CatalogController.handlePlaceOrder() - commit the transaction
CatalogController.handlePlaceOrder() - after commit
```

- Next we checked the same item number through the Inquire Single service and verified that the stock level decreased by one item (Figure 13-19).


CICS Example - Catalog Application
Item Details - Select to Place Order
LIST ITEMS
INQUIRE
ORDER ITEM
SUBMIT

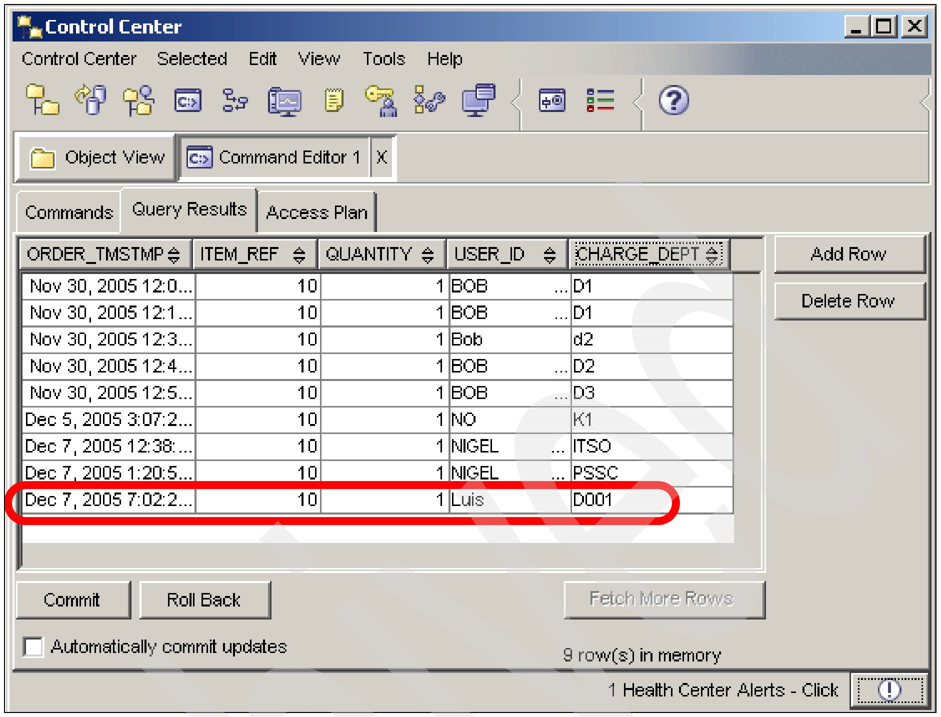
Item	Description	In Stock	On Order	Cost	Select
0010	Ball Pens Black 24pk	75	0	$2.90	●

Figure 13-19 Catalog Application - Inquire single

- We opened a DB2 Control Center and issued the SQL command:

  `SELECT * FROM ITSO.ORDER`

  This lists all of the records in our ITSO.ORDER table. Figure 13-20 shows that our new record is in the table.



ORDER_TMSTMP	ITEM_REF	QUANTITY	USER_ID	CHARGE_DEPT
Nov 30, 2005 12:0...	10	1	BOB ...	D1
Nov 30, 2005 12:1...	10	1	BOB ...	D1
Nov 30, 2005 12:3...	10	1	Bob	d2
Nov 30, 2005 12:4...	10	1	BOB ...	D2
Nov 30, 2005 12:5...	10	1	BOB ...	D3
Dec 5, 2005 3:07:2...	10	1	NO	K1
Dec 7, 2005 12:38:...	10	1	NIGEL ...	ITSO
Dec 7, 2005 1:20:5...	10	1	NIGEL ...	PSSC
Dec 7, 2005 7:02:2...	10	1	Luis	D001

☐ Automatically commit updates

9 row(s) in memory

1 Health Center Alerts - Click

Figure 13-20 The new record in the ITSO .ORDER table

Figure 13-21 shows the registration and protocol service messages that were exchanged between CICS and WebSphere Application Server during our test. Note that WebSphere Application Server uses separate endpoint addresses for its Registration service, Protocol service, and fault messages.

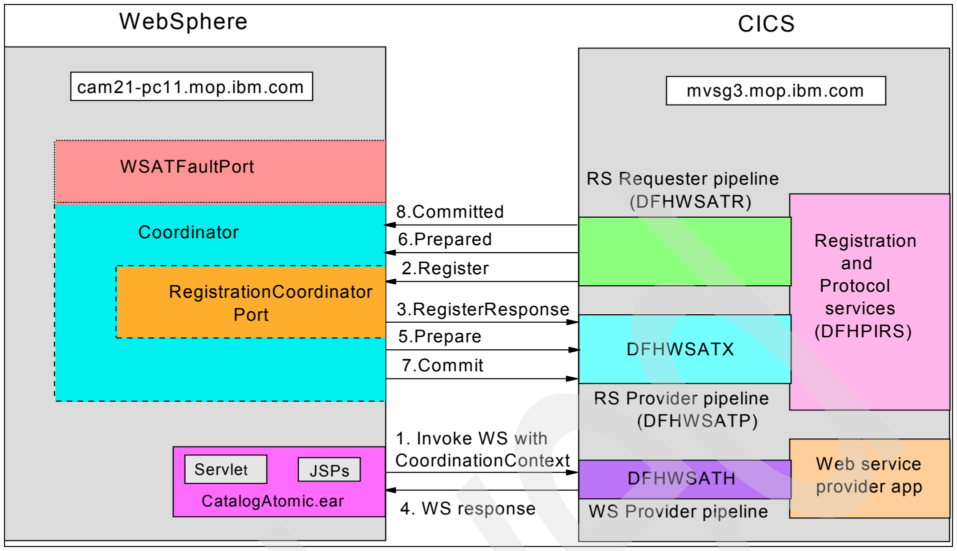


Figure 13-21 Messages exchanged during test of normal atomic transaction termination

We conclude our discussion of this test by showing the complete messages summarized by Figure 13-21. To make each message easier to understand we:

- Format it
- Replace the URL in each xmlns attribute with "..." for each namespace in Example 13-9. For instance, we replaced xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" with xmlns:soap="..." .

Example 13-9 Namespaces used in messages between WebSphere and CICS

```
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:SOAP-ENV=”http://schemas.xmlsoap.org/soap/envelope/”
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wscoor="http://schemas.xmlsoap.org/ws/2004/10/wscoor"
xmlns:wsat="http://schemas.xmlsoap.org/ws/2004/10/wsat"
xmlns:websphere-wsat="http://wstx.Transaction.ws.ibm.com/extension"
xmlns:cicswsat="http://www.ibm.com/xmlns/prod/CICS/pipeline"
```

Invoke Web service with coordination context (message 1)

Example 13-10 shows the SOAP 1.1 message that WebSphere sends to CICS to invoke the placeOrder Web service. The message consists of a SOAP envelope that contains a SOAP header and a SOAP body.

Example 13-10 WebSphere sends to CICS a Web service request with a CoordinationContext header

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:soapenc="..." xmlns:xsd="..."
xmlns:xsi="..."
                  xmlns:wscoor="..."  xmlns:wsa="...">
   <soapenv:Header>
      <wscoor:CoordinationContext soapenv:mustUnderstand="1">
         <wscoor:Expires>Never</wscoor:Expires>
         <wscoor:Identifier>
            com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
            ecc889b8eab92e29d91254fd4fb0ff47b9
         </wscoor:Identifier>
         <wscoor:CoordinationType>
            http://schemas.xmlsoap.org/ws/2004/10/wsat
         </wscoor:CoordinationType>
         <wscoor:RegistrationService xmlns:wscoor="...">
            <wsa:Address xmlns:wsa="...">

http://9.100.199.238:9080/_IBMSYSAPP/wscoor/services/RegistrationCoordinatorPort
            </wsa:Address>
            <wsa:ReferenceProperties xmlns:wsa="...">
               <websphere-wsat:txID xmlns:websphere-wsat="...">
                  com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
                  ecc889b8eab92e29d91254fd4fb0ff47b9
               </websphere-wsat:txID>
               <websphere-wsat:instanceID xmlns:websphere-wsat="...">
                  com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
                  ecc889b8eab92e29d91254fd4fb0ff47b9
               </websphere-wsat:instanceID>
            </wsa:ReferenceProperties>
         </wscoor:RegistrationService>
      </wscoor:CoordinationContext>
   </soapenv:Header>
   <soapenv:Body>
      <p635:DFHOXCMN xmlns:p635="http://www.DFHOXCMN.DFHOXCP5.Request.com">
         <p635:ca_request_id>01ORDR</p635:ca_request_id>
         <p635:ca_return_code>0</p635:ca_return_code>
         <p635:ca_response_message></p635:ca_response_message>
         <p635:ca_order_request>
             <p635:ca_userid>Luis</p635:ca_userid>
             <p635:ca_charge_dept>D001</p635:ca_charge_dept>
             <p635:ca_item_ref_number>10</p635:ca_item_ref_number>
             <p635:ca_quantity_req>1</p635:ca_quantity_req>
             <p635:filler1 xsi:nil="true"/>
         </p635:ca_order_request>
      </p635:DFHOXCMN>
   </soapenv:Body>
</soapenv:Envelope>
```

- SOAP header

  The SOAP header contains only one header, a `CoordinationContext` header. The `CoordinationContext` header has a `mustUnderstand` attribute whose value is *1*. This means that CICS must process the `CoordinationContext` header. If CICS cannot process the `CoordinationContext` header (for example, because the provider pipeline configuration file does not specify the DFHWSATH message handler) then CICS must stop all further processing of the message and generate a SOAP fault whose fault code is `MustUnderstand`.

  The `CoordinationContext` element contains four elements:

  – `Expires`

    CICS treats the content of the `Expires` element as a character string and determines whether the string represents a number.

    - If it does, it is treated as a millisecond value, which is then converted to an integer representing the number of seconds for which the Web service transaction waits for a response to a **`Register`** request or for the Coordinator to send various 2PC messages. If this integer is 0, or greater than 4080, then CICS uses 4080 seconds (the maximum value which CICS allows for the DTIMOUT attribute of the TRANSACTION resource definition).
    - If it does not, then CICS does not set a suspend time and the Web service transaction waits forever.

    Note: We noted that the Expires element was not an unsigned integer but the text `Never`. The default CICS behavior is not to terminate the transaction but to wait indefinitely for WebSphere to commit or roll back the atomic transaction.

  – `Identifier`

    WebSphere creates this identifier as a unique global identifier for each WS-AT transaction as required by the WS-Coordination specification.

– `CoordinationType`

  The `CoordinationType` element specifies the WS-AtomicTransaction coordination type.

– `RegistrationService`

  The `RegistrationService` element contains an endpointReference for WebSphere's Registration service. The endpointReference has two elements:

  - `Address`

    The `Address` element tells CICS where to send its **`Register`** request. CICS copies the contents of the `Address` element to the `To` message information header when it builds its **`Register`** request.

  - `ReferenceProperties`

    WebSphere provides two reference properties: `txID` and `instanceID`. The content of the `txID` element is the same as the content of the `Identifier` element. The content of the `instanceID` element is initially the same as that of the `txID` element, but that will change later.

    CICS adds these reference properties to the SOAP `Header` when it builds its **`Register`** request. This will allow WebSphere to map the **`Register`** request to this service request. The detail which WebSphere puts into the `txID` and `instanceID` properties is implementation specific and is only ever parsed and understood by WebSphere.

► SOAP body

  The content of the `<p635:ca_request_id>` element tells CICS which Web service to invoke; 01ORDR indicates the placeOrder Web service. The contents of the `<p635:ca_userid>`, `<p635:ca_charge_dept>`, `<p635:ca_item_ref_number>`, and `<p635:ca_quantity_req>` elements provide the placeOrder Web service with the details of the order (see Figure 13-17 on page 491).

Register (message 2)

Example 13-11 shows the SOAP message which contains the **`Register`** request that CICS sends to WebSphere. The message consists of a SOAP envelope which contains a SOAP header and a SOAP body.

Example 13-11 CICS sends a Register request to WebSphere

```
<soap:Envelope xmlns:wscoor="..." xmlns:wsa="..." xmlns:cicswsat="..."
xmlns:soap="...">
   <soap:Header>
      <wsa:Action>
         http://schemas.xmlsoap.org/ws/2004/10/wscoor/Register
      </wsa:Action>
      <wsa:MessageID>PIAT-MSG-A6POT3C1-003343146052627C</wsa:MessageID>
      <wsa:ReplyTo>
         <wsa:Address>
            http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
         </wsa:Address>
         <wsa:ReferenceProperties>
            <cicswsat:UOWID>BE092025168B596B</cicswsat:UOWID>
            <cicswsat:PublicId>
               310FD7E2E2C3C7F34BC1F6D7D6E3F3C3F10FD7E2E2C3C7F34BC1F6D7D6E3F3C3
               F1C3C9E6E2F3C44040BE092025167B0000092025167B00004040404040404040
            </cicswsat:PublicId>
         </wsa:ReferenceProperties>
      </wsa:ReplyTo>
      <wsa:To>

http://9.100.199.238:9080/_IBMSYSAPP/wscoor/services/RegistrationCoordinatorPort
      </wsa:To>
      <websphere-wsat:txID xmlns:websphere-wsat="...">
         com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
         ecc889b8eab92e29d91254fd4fb0ff47b9
      </websphere-wsat:txID>
      <websphere-wsat:instanceID xmlns:websphere-wsat="...">
         com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
         ecc889b8eab92e29d91254fd4fb0ff47b9
      </websphere-wsat:instanceID>
   </soap:Header>
   <soap:Body>
      <wscoor:Register>
         <wscoor:ProtocolIdentifier>
            http://schemas.xmlsoap.org/ws/2004/10/wsat/Durable2PC
         </wscoor:ProtocolIdentifier>
         <wscoor:ParticipantProtocolService>
            <wsa:Address>
               http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
            </wsa:Address>
            <wsa:ReferenceProperties>
               <cicswsat:UOWID>BE092025168B596B</cicswsat:UOWID>
               <cicswsat:PublicId>
                  310FD7E2E2C3C7F34BC1F6D7D6E3F3C3F10FD7E2E2C3C7F34BC1F6D7D6E3F3C3
             F1C3C9E6E2F3C44040BE092025167B0000092025167B00004040404040404040
               </cicswsat:PublicId>
            </wsa:ReferenceProperties>
         </wscoor:ParticipantProtocolService>
      </wscoor:Register>
```

```
    </soap:Body>
</soap:Envelope>
```

- SOAP header

  The SOAP `Header` contains several message information headers and the two reference properties (`txID` and `instanceID`) which WebSphere sent in its Web service request. These reference properties allow WebSphere to match this **`Register`** request to its initial request for the placeOrder service. The message information headers are as follows:

  – `To`

    The `To` header shows that the message is being sent to the Registration Coordinator Port running in a WebSphere Application Server V6.0 region which is monitoring port 9080 on a system whose IP address is 9.100.199.238. CICS copied this from the `Address` element of the endpointReference for the RegistrationService in WebSphere's service request.

  – `Action`

    The `Action` header indicates that CICS wishes to register with the Registration Coordinator Port.

  – `MessageID`

    The ID of the message uniquely identifies the message in space and time:

    - PIAT is part of the name of module DFHPIAT; DFHPIAT generates `CoordinationContext` elements for CICS and interprets `CoordinationContext` elements received from other systems.
    - A6POT3C1 is the VTAM APPLID of the CICS region which is sending the message.
    - `003343146052627C` is the abstime value returned by an EXEC CICS INQUIRE TIME issued in our CICS region.

  – `ReplyTo`

    The `Address` element of the `ReplyTo` header shows that the reply to this message should be sent to the Registration service running in a CICS region which is monitoring port 15301 on a z/OS system whose IP address is MVSG3.mop.ibm.com. This Registration service has two reference properties:

    - `UOWID`
    - `PublicID`

    When WebSphere sends the **`RegisterResponse`** to CICS, it will add each of these reference properties to the response as a SOAP header. CICS will use the PublicID to find the region and the UOW in that region where the Web service provider is waiting for the response; then CICS will route it there for processing. The detail which CICS puts into the UOWID and PublicID properties is there for CICS to understand; other products do not use it.

- SOAP body

  The SOAP Body contains the **Register** request, which contains two elements:

  – ProtocolIdentifier

    CICS uses this element to register for the durable two-phase commit protocol.

  – ParticipantProtocolService

    Note that the address of CICS's Protocol service is the same as the address of its Registration service.

RegisterResponse (message 3)

Example 13-12 shows the **RegisterResponse** that WebSphere sends to CICS.

Example 13-12 WebSphere sends a RegisterResponse to CICS

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:soapenc="..." xmlns:xsd="..."
xmlns:xsi="..."
                  xmlns:wsa="...">
   <soapenv:Header>
      <wsa:MessageID>uuid:10A19D35-0108-4000-E000-094409D4835B</wsa:MessageID>
      <wsa:To>http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService</wsa:To>
      <wsa:Action>
         http://schemas.xmlsoap.org/ws/2004/10/wscoor/RegisterResponse
      </wsa:Action>
      <wsa:FaultTo xmlns:wsa="...">
         <wsa:Address>
            http://9.100.199.238:9080/_IBMSYSAPP/wsatfault/services/WSATFaultPort
         </wsa:Address>
         <wsa:ReferenceProperties xmlns:wsa="...">
            <websphere-wsat:txID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b020000000010000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:txID>
            <websphere-wsat:instanceID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b020000000010000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:instanceID>
         </wsa:ReferenceProperties>
      </wsa:FaultTo>
      <wsa:From xmlns:wsa="...">
         <wsa:Address>

http://9.100.199.238:9080/_IBMSYSAPP/wscoor/services/RegistrationCoordinatorPort
         </wsa:Address>
         <wsa:ReferenceProperties xmlns:wsa="...">
            <websphere-wsat:txID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:txID>
            <websphere-wsat:instanceID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:instanceID>
         </wsa:ReferenceProperties>
      </wsa:From>
      <wsa:RelatesTo>PIAT-MSG-A6POT3C1-003343146052627C</wsa:RelatesTo>
      <cicswsat:UOWID xmlns:cicswsat="...">BE092025168B596B</cicswsat:UOWID>
      <cicswsat:PublicId xmlns:cicswsat="...">
         310FD7E2E2C3C7F34BC1F6D7D6E3F3C3F10FD7E2E2C3C7F34BC1F6D7D6E3F3C3
         F1C3C9E6E2F3C44040BE092025167B0000092025167B00004040404040404040
      </cicswsat:PublicId>
   </soapenv:Header>
   <soapenv:Body>
      <RegisterResponse xmlns="http://schemas.xmlsoap.org/ws/2004/10/wscoor">
         <wscoor:CoordinatorProtocolService xmlns:wscoor="...">
            <wsa:Address xmlns:wsa="...">
               http://9.100.199.238:9080/_IBMSYSAPP/wsat/services/Coordinator
            </wsa:Address>
            <wsa:Reference Properties xmlns:wsa="...">
               <websphere-wsat:txID xmlns:websphere-wsat="...">
                  com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
                  ecc889b8eab92e29d91254fd4fb0ff47b9
               </websphere-wsat:txID>
               <websphere-wsat:instanceID xmlns:websphere-wsat="...">
                  http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
               </websphere-wsat:instanceID>
            </wsa:ReferenceProperties>
         </wscoor:CoordinatorProtocolService>
      </RegisterResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

- SOAP header

  The SOAP `Header` contains several message information headers and the two reference properties (`UOWID` and `PublicID`), which CICS sent in its **`Register`** request. If we were running a CICSPlex, CICS would use these reference properties to route the response to the region and the UOW in that region where the Web service provider is waiting for it. The message information headers are as follows:

  – `MessageID`

  – `To`

    WebSphere is sending this message to CICS's RegistrationService. It obtained this address from the `ReplyTo` header in the **`Register`** request.

  – `Action`

    This message is a **`RegisterResponse`**.

  – `FaultTo`

    WebSphere wants CICS to send any SOAP faults that it has to generate to WebSphere's WSATFaultPort.

  – `From`

    This message is coming from WebSphere's RegistrationCoordinatorPort.

  – `RelatesTo`

    This message relates to the **`Register`** request which CICS sent.

- SOAP body

  The SOAP `Body` contains the `RegisterResponse` element, which in turn contains only a `CoordinatorProtocolService` element. The address of WebSphere's Protocol Service is
  http://9.100.199.238:9080/_IBMSYSAPP/wsat/services/Coordinator
  Note that its two reference properties, `txID` and `instanceID`, no longer have the same contents; `instanceID` now contains the address of CICS's Registration service.

Web service response (message 4)

Since CICS has registered its interest in the atomic transaction and WebSphere has responded to that, CICS can now run the placeOrder service. When the placeOrder service has completed its work, CICS sends a response to WebSphere as shown in Example 13-13. The essence of the response is the message `ORDER SUCCESSFULLY PLACED`.

Example 13-13 CICS sends the Web service response to WebSphere

```
<SOAP-ENV:Envelope xmlns:soapenv="..." xmlns:soapenc="..." xmlns:xsd="..."
```

```
                xmlns:xsi="..."     xmlns:wscoor="..."  xmlns:wsa="..."
                xmlns:SOAP-ENV="...">
   <SOAP-ENV:Body>
      <DFHOXCMNResponse xmlns="http://www.DFHOXCMN.DFHOXCP5.Response.com">
         <ca_request_id>01ORDR</ca_request_id>
         <ca_return_code>0</ca_return_code>
         <ca_response_message>ORDER SUCESSFULLY PLACED</ca_response_message>
         <ca_order_request>
            <ca_userid>Luis    </ca_userid>
            <ca_charge_dept>D001    </ca_charge_dept>
            <ca_item_ref_number>10</ca_item_ref_number>
            <ca_quantity_req>1</ca_quantity_req>
            <filler1>
            ...
            </filler1>
         </ca_order_request>
      </DFHOXCMNResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Prepare (message 5)

When the `handlePlaceOrder` method of the CatalogController servlet issues the **`userTransaction.commit`** command, WebSphere sends a **`Prepare`** command to CICS as shown in Example 13-14 in the `Action` message information header and in the `<p320:Prepare.../>` element of the SOAP Body.

Example 13-14 WebSphere sends a Prepare notification to CICS

```
<soapenv:Envelope   xmlns:soapenv="..."   xmlns:soapenc="..."   xmlns:xsd="..."
                    xmlns:xsi="..."       xmlns:wsa="...">
   <soapenv:Header>
      <wsa:MessageID>uuid:10A19F38-0108-4000-E000-094409D4835B</wsa:MessageID>
      <wsa:To>http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService</wsa:To>
      <wsa:Action>http://schemas.xmlsoap.org/ws/2004/10/wsat/Prepare</wsa:Action>
      <wsa:FaultTo xmlns:wsa="...">
         <wsa:Address>
            http://9.100.199.238:9080/_IBMSYSAPP/wsatfault/services/WSATFaultPort
         </wsa:Address>
         <wsa:ReferenceProperties xmlns:wsa="...">
            <websphere-wsat:txID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:txID>
            <websphere-wsat:instanceID xmlns:websphere-wsat="...">
               http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
            </websphere-wsat:instanceID>
         </wsa:ReferenceProperties>
      </wsa:FaultTo>
      <wsa:ReplyTo xmlns:wsa="...">
         <wsa:Address>
            http://9.100.199.238:9080/_IBMSYSAPP/wsat/services/Coordinator
         </wsa:Address>
         <wsa:ReferenceProperties xmlns:wsa="...">
            <websphere-wsat:txID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:txID>
            <websphere-wsat:instanceID xmlns:websphere-wsat="...">
               http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
            </websphere-wsat:instanceID>
         </wsa:ReferenceProperties>
      </wsa:ReplyTo>
      <cicswsat:UOWID xmlns:cicswsat="...">BE092025168B596B</cicswsat:UOWID>
      <cicswsat:PublicId xmlns:cicswsat="...">
         310FD7E2E2C3C7F34BC1F6D7D6E3F3C3F10FD7E2E2C3C7F34BC1F6D7D6E3F3C3
         F1C3C9E6E2F3C44040BE092025167B0000092025167B00004040404040404040
      </cicswsat:PublicId>
   </soapenv:Header>
   <soapenv:Body>
      <p320:Prepare xsi:nil="true"
xmlns:p320="http://schemas.xmlsoap.org/ws/2004/10/wsat"/>
   </soapenv:Body>
</soapenv:Envelope>
```

Prepared (message 6)

CICS responds with **Prepared** as shown in Example 13-15 in the `Action` message information header and in the `<wsat:Prepared.../>` element of the SOAP `Body`. Since **Prepared** is a terminating message, it does not contain a `ReplyTo` message information header.

Example 13-15 CICS sends Prepared notification to WebSphere

```
<soap:Envelope xmlns:wscoor="..."  xmlns:wsa="..."   xmlns:cicswsat="..."
xmlns:soap="...">
   <soap:Header>
      <wsa:Action>http://schemas.xmlsoap.org/ws/2004/10/wsat/Prepared</wsa:Action>
      <wsa:MessageID>PIAT-MSG-A6POT3C1-003343146053278C</wsa:MessageID>
      <wsa:To>http://9.100.199.238:9080/_IBMSYSAPP/wsat/services/Coordinator</wsa:To>
      <websphere-wsat:txID xmlns:websphere-wsat="...">
         com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
```

```
            ecc889b8eab92e29d91254fd4fb0ff47b9
         </websphere-wsat:txID>
         <websphere-wsat:instanceID xmlns:websphere-wsat="...">
              http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
         </websphere-wsat:instanceID>
      </soap:Header>
      <soap:Body>
         <wsat:Prepared xmlns:wsat="..."/>
      </soap:Body>
</soap:Envelope>
```

Commit (message 7)

Since CICS has voted **yes** in response to the `Prepare` command, and since no other system has registered an interest in this transaction, WebSphere sends a `Commit` command to CICS. See the `Action` header and the `<p320:Commit.../>` element of the SOAP `Body` in Example 13-16.

Example 13-16 WebSphere sends Commit notification to CICS

```
<soapenv:Envelope xmlns:soapenv="..."  xmlns:soapenc="..." xmlns:xsd="..."
xmlns:xsi="..."
                  xmlns:wsa="...">
   <soapenv:Header>
      <wsa:MessageID>uuid:10A19F67-0108-4000-E000-094409D4835B</wsa:MessageID>
      <wsa:To>http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService</wsa:To>
      <wsa:Action>http://schemas.xmlsoap.org/ws/2004/10/wsat/Commit</wsa:Action>

      <wsa:FaultTo xmlns:wsa="...">
         <wsa:Address>
            http://9.100.199.238:9080/_IBMSYSAPP/wsatfault/services/WSATFaultPort
         </wsa:Address>
         <wsa:ReferenceProperties xmlns:wsa="...">
            <websphere-wsat:txID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:txID>
            <websphere-wsat:instanceID xmlns:websphere-wsat="...">
               http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
            </websphere-wsat:instanceID>
         </wsa:ReferenceProperties>
      </wsa:FaultTo>
      <wsa:ReplyTo xmlns:wsa="...">
         <wsa:Address>
            http://9.100.199.238:9080/_IBMSYSAPP/wsat/services/Coordinator
         </wsa:Address>
         <wsa:ReferenceProperties xmlns:wsa="...">
            <websphere-wsat:txID xmlns:websphere-wsat="...">
               com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
               ecc889b8eab92e29d91254fd4fb0ff47b9
            </websphere-wsat:txID>
            <websphere-wsat:instanceID xmlns:websphere-wsat="...">
               http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
            </websphere-wsat:instanceID>
         </wsa:ReferenceProperties>
      </wsa:ReplyTo>
      <cicswsat:UOWID xmlns:cicswsat="...">BE092025168B596B</cicswsat:UOWID>
      <cicswsat:PublicId xmlns:cicswsat="...">
         310FD7E2E2C3C7F34BC1F6D7D6E3F3C3F10FD7E2E2C3C7F34BC1F6D7D6E3F3C3
         F1C3C9E6E2F3C44040BE092025167B0000092025167B000040404040404040
      </cicswsat:PublicId>
   </soapenv:Header>
   <soapenv:Body>
      <p320:Commit xsi:nil="true"
xmlns:p320="http://schemas.xmlsoap.org/ws/2004/10/wsat"/>
   </soapenv:Body>
</soapenv:Envelope>
```

Committed (message 8)

After committing the update to the EXMPCAT VSAM file, CICS sends a **Committed** notification to WebSphere. See the Action message information header and the <wsat:Committed.../> element of the SOAP Body in Example 13-17. Since **Committed** is a terminating message, it does not contain a ReplyTo message information header.

Example 13-17 CICS sends Committed notification to WebSphere

```
<soap:Envelope   xmlns:wscoor="..."   xmlns:wsa="..."   xmlns:cicswsat="..."
                 xmlns:soap="...">
   <soap:Header>
      <wsa:Action>http://schemas.xmlsoap.org/ws/2004/10/wsat/Committed</wsa:Action>
      <wsa:MessageID>PIAT-MSG-A6POT3C1-003343146053324C</wsa:MessageID>
      <wsa:To>http://9.100.199.238:9080/_IBMSYSAPP/wsat/services/Coordinator</wsa:To>
      <websphere-wsat:txID xmlns:websphere-wsat="...">
         com.ibm.ws.wstx:0000010810a19b0200000001000000683e8b0
         ecc889b8eab92e29d91254fd4fb0ff47b9
      </websphere-wsat:txID>
      <websphere-wsat:instanceID xmlns:websphere-wsat="...">
         http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
      </websphere-wsat:instanceID>
   </soap:Header>
   <soap:Body>
      <wsat:Committed xmlns:wsat="..."/>
   </soap:Body>
</soap:Envelope>
```

Let's think back to Waldo and his money transfer transaction (see "Beginner's guide to atomic transactions" on page 410). What Waldo requires is that both databases cooperate to ensure that they are always in a known and consistent state.

The set of eight Web service, registration, and protocol service messages we have described here are the flows that guarantee database consistency in our scenario, a scenario in which the databases are accessed by middleware (WebSphere Application Server on a Windows platform and CICS on z/OS); and communication between the middleware products is based on open Web services standards. The same basic flows could be used between any middleware products which support the Web services, Web Services-Coordination (WS-C) and Web Services-Atomic Transaction (WS-AT) specifications.

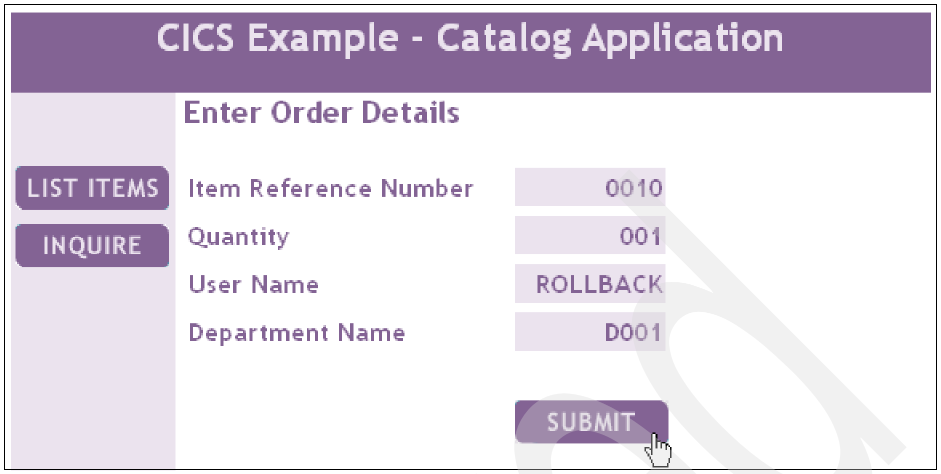
Simple scenario: abnormal transaction termination

We return to the Place Order window to place a new order. This time we used `ROLLBACK` as the User Name. This tells the servlet to throw an exception after inserting the order in the DB2 table and placing the order with CICS.

Example 13-18 CatalogController servlet - Throwing the RemoteException

```
if(order.getUserId().equalsIgnoreCase("ROLLBACK"))
{
   System.out.println("CatalogController.handlePlaceOrder() - simulating the 
RemoteException");
   throw new RemoteException("Throwing the RemoteException");
}
```

On the Order Entry page we entered the User Name `ROLLBACK` and clicked **SUBMIT** to place a new order (Figure 13-22). Note that, at this time, we still had 75 items in stock (see Figure 13-19 on page 492).

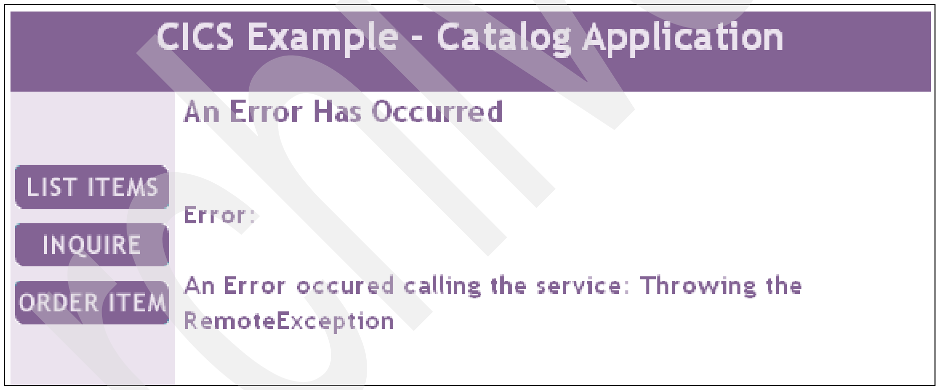

CICS Example - Catalog Application

Enter Order Details

Item Reference Number	0010
Quantity	001
User Name	ROLLBACK
Department Name	D001

Figure 13-22 Catalog Application - Place order for ROLLBACK

Figure 13-23 shows the expected error response.


CICS Example - Catalog Application

An Error Has Occurred

Error:

An Error occured calling the service: Throwing the RemoteException

Figure 13-23 Catalog Application - Error window

When the Exception is thrown, the transaction is rolled back; the order is deleted from the table and the CICS transaction is rolled back too.

We again check in the WebSphere server log to see what happened (Example 13-19).

Example 13-19 WebSphere server log with ROLLBACK Place Order

```
CatalogContoller:doPost() - Action = Place Order
CatalogController.handlePlaceOrder() - creating the order
CatalogController.handlePlaceOrder() - beginning the transaction
CatalogController.handlePlaceOrder() - inserting the order in the database
```

```
OrderDB.insertOrder() - inserted the order in the database!!!!
CatalogController.handlePlaceOrder() - calling the CICS web service
CatalogController.handlePlaceOrder() - response back from the CICS web service
CatalogController.handlePlaceOrder() - simulating the RemoteException
CatalogController.handlePlaceOrder() - rollingback the transaction
```

We can see in the log that the order was inserted in the database and a good response was returned from calling the CICS Web service. Then we simulated the RemoteException and the rollback of the whole transaction took place.

When we inquired on the stock level, we noted that there were still 75 items in stock.

Figure 13-24 shows the registration and protocol service messages that CICS and WebSphere Application Server exchanged during our test.

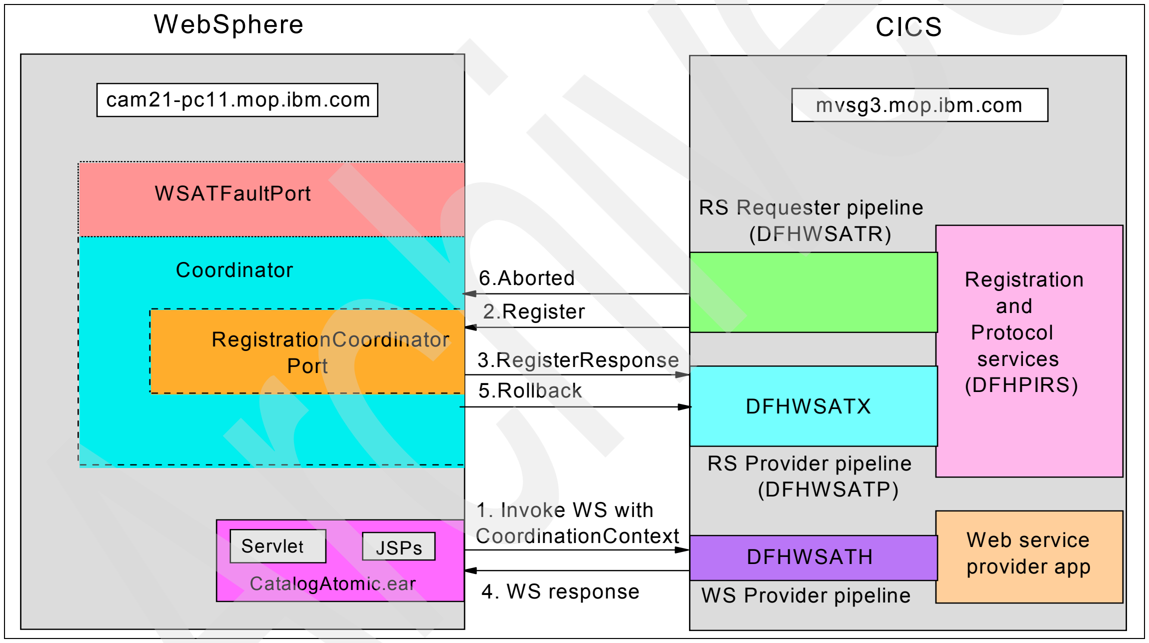


Figure 13-24 Messages exchanged when both sides back out

13.3 The daisy chain atomic transaction scenario

We now extend the simple atomic transaction scenario to a *daisy chain* scenario by making the following modifications:

- Changing the configuration of the CICS sample Catalog application such that it makes an outbound Web service call
- Changing the ExampleAppDispatchOrder.ear file provided with CICS TS V3.1

  ExampleAppDispatchOrder.ear is an enterprise archive provided with the Catalog application that can be deployed in WebSphere Application Server and used as an order dispatch endpoint. See "Catalog manager example application" on page 67 for more information about the sample application.

  We modified the ExampleAppDispatchOrder.ear file to insert records into a DB2 table called ITSO.DISPATCH. The ITSO.DISPATCH table logs all of the orders dispatched through the dispatchOrder Web service. We called the modified program DispatchOrderAtomic.ear.

By making these changes, we have a global transaction that updates three resources:

- A DB2 table in the Windows environment
- A VSAM file in the z/OS environment
- Another DB2 table in a Windows environment

We have a daisy chain from WebSphere to CICS to WebSphere. CICS acts as both a coordinator and a participant. Figure 13-25 shows the sequence of events for the whole transaction.

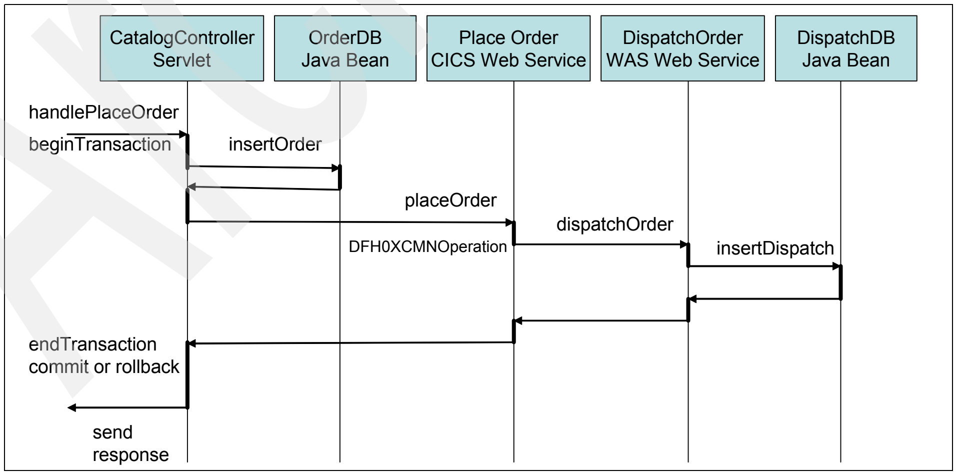


Figure 13-25 Daisy chain scenario sequence of events

Figure 13-26 shows a more global view of the sequence of events:

- The user uses his Web browser to invoke the AtomicClient CatalogController servlet, which runs in WebSphere Application Server.
- The AtomicClient updates the ITSO.ORDER table in DB2.
- WebSphere Application Server sends the placeOrder SOAP message containing the order to CICS.
- CICS uses the Web services provider PIPELINE definition PIPE1 to process the SOAP message. PIPE1 contains our SNIFFER program and the CICS-supplied SOAP 1.1 message handler DFHPISN1.
- DFHPISN1 links to the CICS-supplied header processing program DFHWSATH.
- CICS converts the SOAP message to a COMMAREA for the sample catalog manager program, DFH0XCMN.
- DFH0XCMN passes the data in the COMMAREA to the sample catalog program DFH0XVDS, which updates the recoverable VSAM file.
- DFH0XCMN passes the data in the COMMAREA to DFH0XWOD.
- DFH0XWOD invokes the dispatchOrder Web service, which points to the PIPELINE PIPE2.
- PIPE2 contains DFHPISN1, which:
  - Converts the COMMAREA to a SOAP message body
  - Calls DFHWSATH to create a CoordinationContext SOAP header
  - Sends the dispatchOrder SOAP message to WebSphere
- DispatchOrderAtomic updates the ITSO.DISPATCH table in DB2.

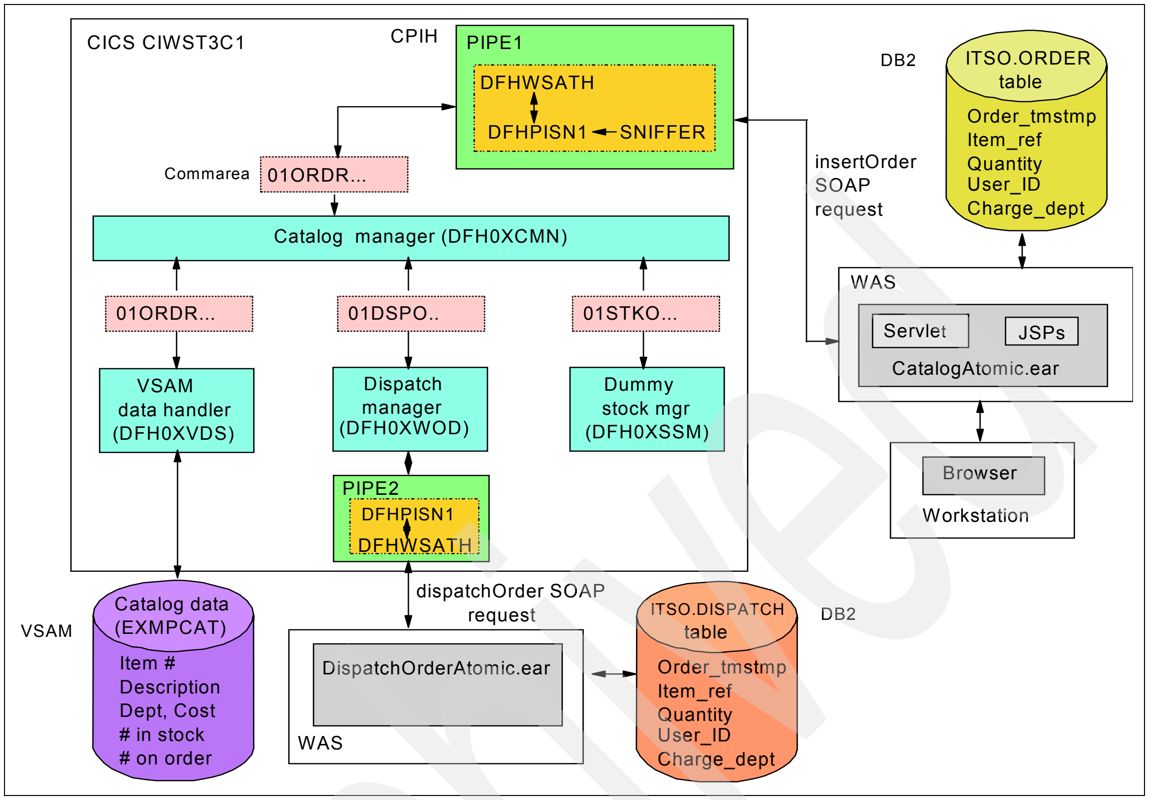


Figure 13-26 Daisy chain: CICS as a service provider and as a service requester

In the following sections we describe how we set up CICS for the daisy chain scenario, and how we created the DispatchOrderAtomic application and the ITSO.DISPATCH table. We then show the results of testing this scenario.

13.3.1 Setting up CICS for the daisy chain scenario

To set up CICS for this scenario we performed the following steps in addition to those shown in 13.2.1, “Setting up CICS for the simple scenario” on page 469:

- We edited PIPE2’s configuration file /CIWS/T3C1/config/ITSO_7206_wsat_soap11request.xml

  It now contains the XML shown in Example 13-20.

Example 13-20 PIPE2:/CIWS/T3C1/config/ITSO_7206_wsat_soap11request.xml

```
<?xml version="1.0" encoding="EBCDIC-CP-US"?>
<requester_pipeline
     xmlns="http://www.ibm.com/software/htp/cics/pipeline"
```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ibm.com/software/htp/cics/pipeline
      requester.xsd “>
  <service>
    <service_handler_list>
      <cics_soap_1.1_handler>
        <headerprogram>
          <program_name>DFHWSATH</program_name>
          <namespace>
             http://schemas.xmlsoap.org/ws/2004/10/wscoor
          </namespace>
          <localname>CoordinationContext</localname>
          <mandatory>true</mandatory>
        </headerprogram>
      </cics_soap_1.1_handler>
    </service_handler_list>
  </service>
  <default_transport_handler_list>
    <handler>
      <program>SNIFFER</program>
      <handler_parameter_list/>
    </handler>
  </default_transport_handler_list>
  <service_parameter_list>
    <registration_service_endpoint>
       http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
    </registration_service_endpoint>
  </service_parameter_list>
</requester_pipeline>
```

- We modified the configuration file for the sample Catalog application using the ECFG transaction at a 3270 terminal connected to our CICS region (see Figure 3-17 on page 98). In the 3270 screen titled Configure CICS Example Catalog Application:
  - We set the value of the Outbound WebService? field on this screen to **Yes**; this causes the Catalog Manager program (DFH0XCMN) to invoke the Dispatch Manager program (DFH0XWOD) rather than the Dummy Dispatch Manager program (DFH0XSOD) that we used in the simple scenario. While DFH0XSOD simply sets the return code in the COMMAREA to 0 and returns to its caller, DFH0XWOD issues an **`EXEC CICS INVOKE WEBSERVICE('dispatchOrder') URI(outboundWebServiceURI)`** command to make an outbound Web service call to an order dispatcher.
  - We set the value of the Outbound WebService URI field to the location of the Web service that implements the order dispatcher function. We ran the dispatchOrder service on WebSphere Application Server for Windows.

13.3.2 Creating DispatchOrderAtomic and the ITSO.DISPATCH table

To create the DispatchOrderAtomic application and the ITSO.DISPATCH table, we did the following:

- Created the DispatchDB JavaBean that inserts the dispatch order into the DB2 table.
- Changed the `dispatchOrder` method in the DispatchOrderSoapBindingImpl class in order to call the DispatchDB JavaBean. DispatchOrderSoapBindingImpl is the bean that serves the Web service and `dispatchOrder` is the method that is called when the Web service is requested.
- Changed the deployment descriptor of the DispatchOrderAtomic application so that it is executed as part of a Web Services Atomic Transaction.
- Created the ITSO.DISPATCH table.

We explain each step in detail in the following sections.

Create the DispatchDB JavaBean

In the same way that we previously created the OrderDB JavaBean, we created the DispatchDB JavaBean. This bean has a method that inserts the dispatch order into the ITSO.DISPATCH table. The input parameter is an OrderBean that has all the data needed for the insert. We do not show the DispatchDB JavaBean code here because it is very similar to the OrderDB code shown in Example 13-4 on page 474.

Change the dispatchOrder method

When the dispatchOrder Web service is called, the `dispatchOrder` method in the DispatchOrderSoapBindingImpl Java class is called. It is responsible to perform the service and send a response. We changed this method to create an OrderBean and to perform the insert by calling the `insertDispatch` method in the DispatchDB JavaBean.

The `dispatchOrder` method is shown in Example 13-21.

Example 13-21 The dispatchOrder method

```
public com.Response.dispatchOrder.exampleApp.www.DispatchOrderResponse
   dispatchOrder(com.Request.dispatchOrder.exampleApp.www.DispatchOrderRequest
requestPart) throws java.rmi.RemoteException {
      System.out.println("DispatchOrderSoapBindingImpl.dispatchOrder():"+
            " ItemRef="+requestPart.getItemReferenceNumber()+
            " Quantity="+requestPart.getQuantityRequired()+
            " CustomerName="+requestPart.getCustomerId()+
            " Dept="+requestPart.getChargeDepartment());
      // Creating the order
      System.out.println("DispatchOrderSoapBindingImpl.dispatchOrder() -
creating the order");
      OrderBean order = new OrderBean();
      order.setOrderTmstmp(new
Timestamp(Calendar.getInstance().getTime().getTime()));
      order.setItemRef(requestPart.getItemReferenceNumber());
      order.setQuantity(requestPart.getQuantityRequired());
      order.setUserId(requestPart.getCustomerId());
      order.setChargeDept(requestPart.getChargeDepartment());

      // inserting the order in the database
      System.out.println("DispatchOrderSoapBindingImpl.dispatchOrder() -
inserting the order in the database");
      DispatchDB dispatchDB = new DispatchDB();
      dispatchDB.insertDispatch(order);
      System.out.println("DispatchOrderSoapBindingImpl.dispatchOrder() - after
the insert!!!");

      com.Response.dispatchOrder.exampleApp.www.DispatchOrderResponse
      response = new
com.Response.dispatchOrder.exampleApp.www.DispatchOrderResponse();
      response.setConfirmation("Order in Dispatch");
        return response;
    }
```

Change the deployment descriptor

In order to specify that the dispatchOrder service should participate in the global transaction, we needed to modify the Web Application deployment descriptor.

To activate WS-AT support:

- We imported the application archive DispatchOrderAtomic.ear into RAD. We then expanded the **Dynamic Web Project** (DispatchOrderAtomicWeb) in the Project Explorer and opened (double-clicked) the **Deployment Descriptor** (Figure 13-27).

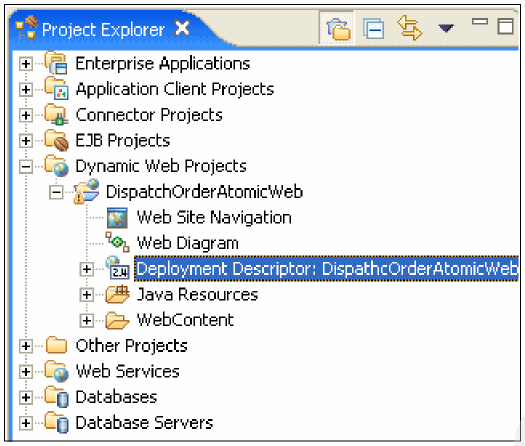
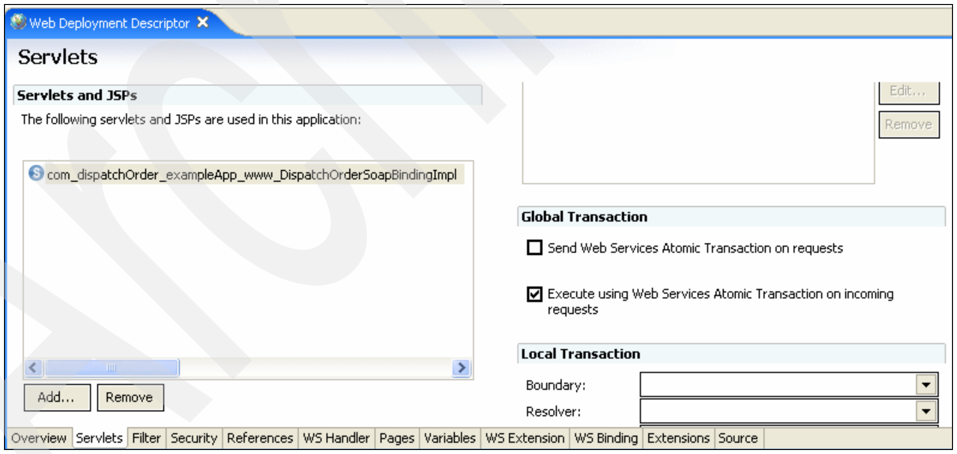


Figure 13-27 Opening the DispatchOrderAtomicWeb Deployment Descriptor

- In the Deployment Descriptor we clicked the **Servlets** tab and then we selected the **com_dispatchOrder_exampleApp_www_DispatchOrderSoapBindingImpl** servlet. Then we scrolled down to Global Transaction and selected **Execute using Web Services Atomic Transaction on incoming request** (Figure 13-28).


Web Deployment Descriptor

Servlets

Servlets and JSPs

The following servlets and JSPs are used in this application:

com_dispatchOrder_exampleApp_www_DispatchOrderSoapBindingImpl

Global Transaction

☐ Send Web Services Atomic Transaction on requests

☑ Execute using Web Services Atomic Transaction on incoming requests

Local Transaction

Boundary:

Resolver:

Edit...
Remove
Add...
Remove
Overview | Servlets | Filter | Security | References | WS Handler | Pages | Variables | WS Extension | WS Binding | Extensions | Source

Figure 13-28 The DispatchOrderAtomicWeb Deployment Descriptor

- We saved and closed the file.

Now when an incoming request contains a coordination context, the dispatchOrder Web service will join the global transaction.

Create the ITSO.DISPATCH table

We created the ITSO.DISPATCH table to record all the dispatch orders that come to the dispatchOrder Web service. Example 13-22 shows the script we used to create the table.

Example 13-22 Creation of ITSO.DISPATCH table

```
-- IBM ITSO

CONNECT RESET;

CONNECT TO ITSOWS;

-- Table definitions for Dispatch
CREATE TABLE ITSO.DISPATCH
  (DISPATCH_TMSTMP TIMESTAMP  NOT NULL ,
   ITEM_REF INTEGER  NOT NULL ,
   QUANTITY INTEGER  NOT NULL ,
   USER_ID CHARACTER (20)  NOT NULL ,
   CHARGE_DEPT CHARACTER (20)  NOT NULL  ,
   CONSTRAINT DISPATCHKEY PRIMARY KEY (DISPATCH_TMSTMP)  ) ;
```

The ITSO.DISPATCH table has a column for every field in the Catalog Place Order JSP. Also, there is a TimeStamp column that is used as a unique key for the table.

13.3.3 Testing the daisy chain scenario

We performed two tests of the daisy chain scenario:

- Normal transaction termination
- Abnormal transaction termination (see “Daisy chain scenario: abnormal transaction termination” on page 523)

Daisy chain scenario: normal transaction termination

In this section, we explain how we ran the AtomicClient application and then show the registration and protocol service messages that are exchanged between WebSphere Application Server and CICS during the normal termination of the atomic transaction.

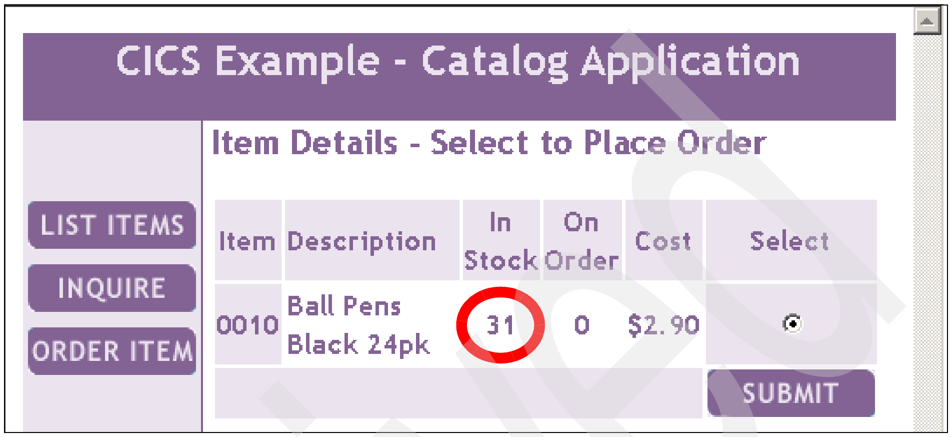
To run the scenario we did the following:

- Opened the welcome page of the Catalog application.

  `http://cam21-pc11:9080/CatalogAtomicWeb/Welcome.jsp`

- Clicked **INQUIRE** to perform an inquireSingle operation.

- In the **Inquire Single** window we used the Item Reference Number default value of 0010 and clicked **SUBMIT**. The Web service request was sent to CICS and we were presented with the results of the inquiry as shown in Figure 13-29.

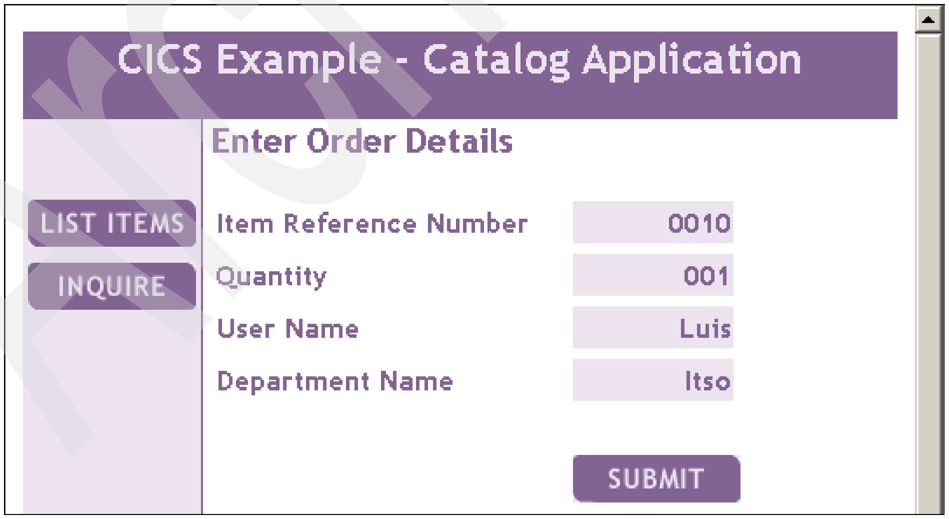

CICS Example - Catalog Application

Item Details - Select to Place Order

Item	Description	In Stock	On Order	Cost	Select
0010	Ball Pens Black 24pk	31	0	$2.90	●

Figure 13-29 Catalog Application - Inquire single before

- We noted that the number of items in stock was 31. This value is taken from the CICS VSAM file.
- We clicked **SUBMIT** to go to the Enter Order Details window shown in Figure 13-30.

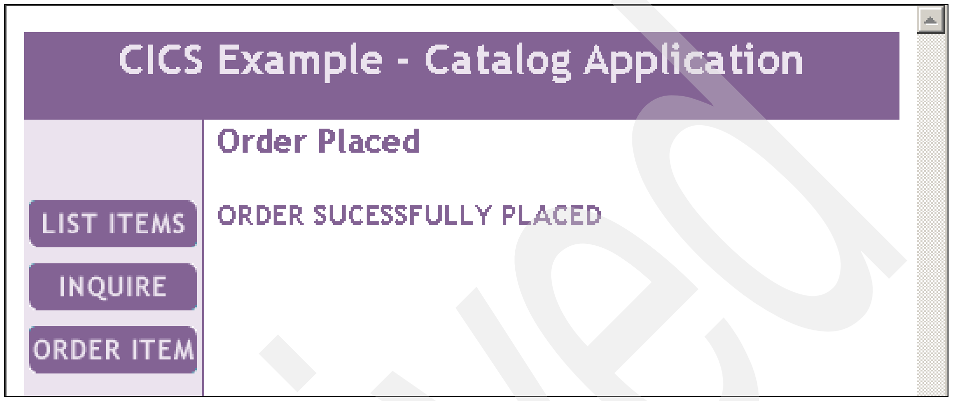

CICS Example - Catalog Application

Enter Order Details

Item Reference Number	0010
Quantity	001
User Name	Luis
Department Name	Itso

Figure 13-30 Catalog Application - Enter order details

- In the Enter Order Details window we provided a User Name and a Department Name and clicked **SUBMIT**.
- After the CICS Web service processed the order and called the dispatchOrder WebSphere Web service, we got the response telling us that the order was successfully placed (Figure 13-31).



ORDER SUCESSFULLY PLACED

Figure 13-31 Catalog Application - Order successfully placed

- In the WebSphere server log we see trace entries generated by the CatalogController servlet, which show that the transaction was successfully committed (Example 13-23).

Example 13-23 WebSphere server log with successful Place Order for CatalogController

```
CatalogContoller:doPost() - Action = Place Order
CatalogController.handlePlaceOrder() - creating the order
CatalogController.handlePlaceOrder() - beginning the transaction
CatalogController.handlePlaceOrder() - inserting the order in the database
OrderDB.insertOrder() - inserted the order in the database!!!!
CatalogController.handlePlaceOrder() - calling the CICS web service
CatalogController.handlePlaceOrder() - response back from the CICS web service
CatalogController.handlePlaceOrder() - commit the transaction
CatalogController.handlePlaceOrder() - after commit
```

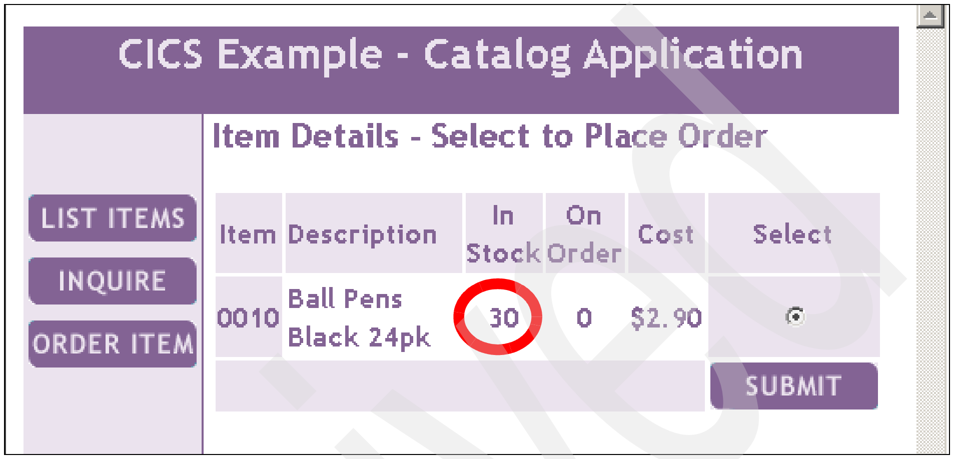
- We also can see the log for the WebSphere Application Server on which the DispatchOrderAtomic application is installed (Example 13-24).

Example 13-24 WebSphere server log with successful Place Order for DispatchOrder

```
DispatchOrderSoapBindingImpl.dispatchOrder(): ItemRef=10 Quantity=1
CustomerName=Luis     Dept=Itso
DispatchOrderSoapBindingImpl.dispatchOrder() - creating the order
DispatchOrderSoapBindingImpl.dispatchOrder() - inserting the order in the
database
```

```
DispatchDB.insertDispatch() - inserted the order in the database!!!!
DispatchOrderSoapBindingImpl.dispatchOrder() - after the insert!!!
```

- Next we checked the same item number through the inquireService service and verified that the stock level decreased by one item (Figure 13-32).



CICS Example - Catalog Application

Item Details - Select to Place Order

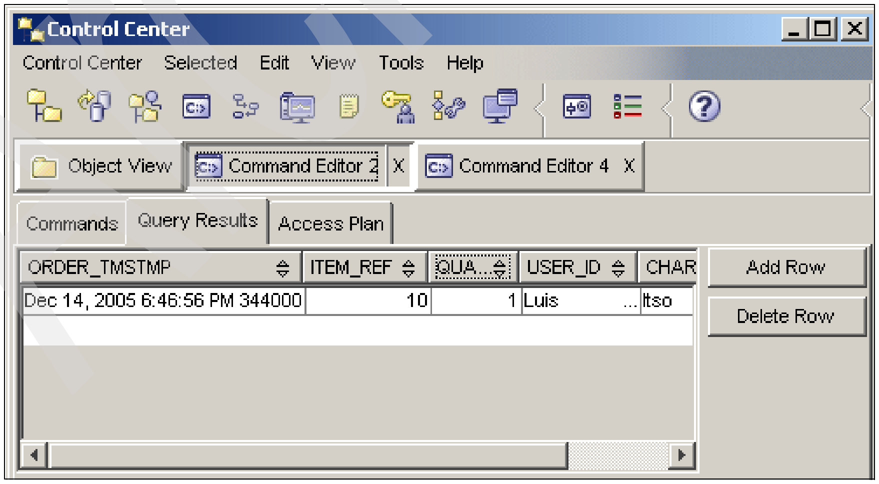
Item	Description	In Stock	On Order	Cost	Select
0010	Ball Pens Black 24pk	30	0	$2.90	●

Figure 13-32 Catalog Application - Inquire single after

- We opened the DB2 Control Center and used the following SQL command on the Windows machine which hosted the ITSO.ORDER table:

  ```
  SELECT * FROM ITSO.ORDER
  ```

  Figure 13-33 shows the new record in the table.



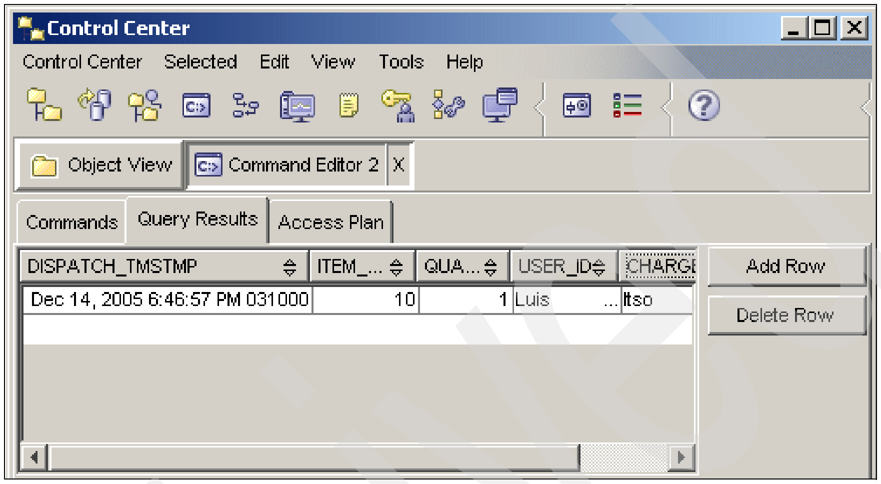
ORDER_TMSTMP	ITEM_REF	QUA...	USER_ID	CHAR
Dec 14, 2005 6:46:56 PM 344000	10	1	Luis ...	Itso

Figure 13-33 The new record in the ITSO.ORDER table

- We also used the following SQL command on the Windows machine which hosted the ITSO.DISPATCH table:

  ```
  SELECT * FROM ITSO.DISPATCH
  ```

  Figure 13-34 shows the new record in the table.



DISPATCH_TMSTMP	ITEM_...	QUA...	USER_ID	CHARGE
Dec 14, 2005 6:46:57 PM 031000	10	1	Luis ...	Itso

Figure 13-34 The new record in the ITSO.DISPATCH table

Figure 13-35 shows the registration and protocol service messages that are exchanged between CICS and the two WebSphere Application Servers during our test.

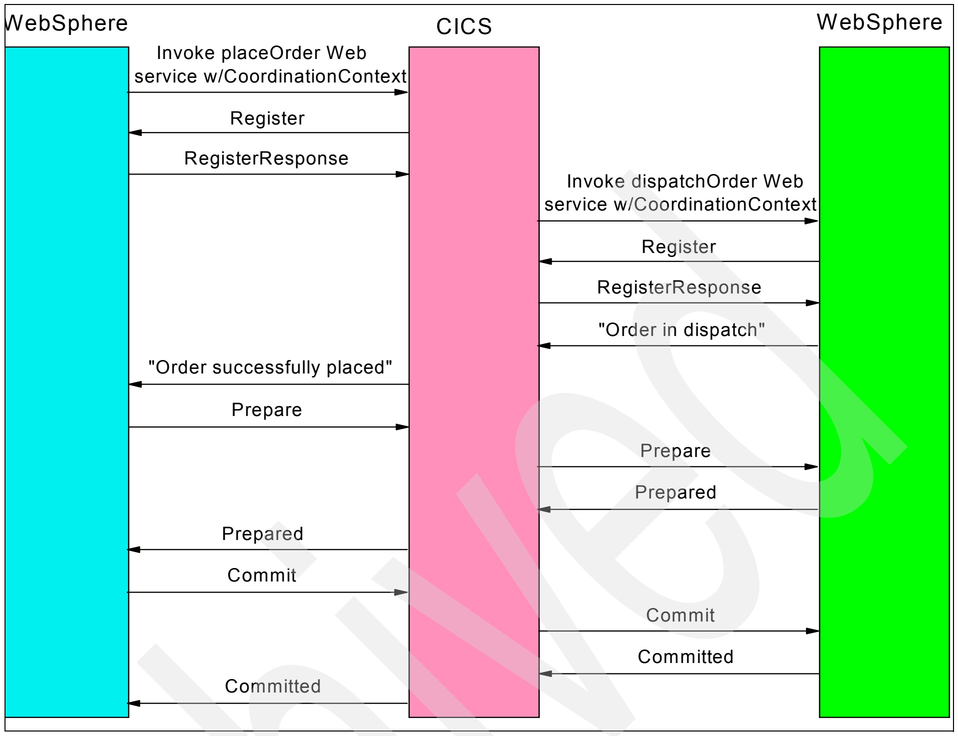


Figure 13-35 Successful daisy chain - Message flow

Example 13-25 shows the dispatchOrder request that CICS sends to WebSphere. Recall that when CICS is a *participant* in a WS-AT transaction, it uses two reference properties: `UOWID` and `PublicID`. We see in Example 13-25 that when CICS is the *coordinator*, it uses three reference properties: `UOWID`, `Token`, and `Netname`.

Example 13-25 CICS sends dispatchOrder request to WebSphere

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="..." xmlns:wscoor="..." xmlns:wsa="..."
xmlns:cicswsat="..."                xmlns:soap="...">
   <SOAP-ENV:Header>
      <wscoor:CoordinationContext>
          <wscoor:Identifier>
             PIAT-CCON-A6POT3C1-003343578075119C
          </wscoor:Identifier>
          <wscoor:CoordinationType>
             http://schemas.xmlsoap.org/ws/2004/10/wsat
          </wscoor:CoordinationType>
          <wscoor:RegistrationService>
```

```
            <wsa:Address>
               http://MVSG3.mop.ibm.com:15301/cicswsat/RegistrationService
            </wsa:Address>
            <wsa:ReferenceProperties>
               <cicswsat:Netname>A6POT3C1</cicswsat:Netname>
               <cicswsat:Token>F0F0F0F0</cicswsat:Token>
               <cicswsat:UOWID>BE0F698D97751D67</cicswsat:UOWID>
            </wsa:ReferenceProperties>
         </wscoor:RegistrationService>
      </wscoor:CoordinationContext>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <dispatchOrderRequest xmlns="http://www.exampleApp.dispatchOrder.Request.com">
         <itemReferenceNumber>10</itemReferenceNumber>
         <quantityRequired>1</quantityRequired>
         <customerId>Luis    </customerId>
         <chargeDepartment>Itso    </chargeDepartment>
      </dispatchOrderRequest>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Daisy chain scenario: abnormal transaction termination

Finally, we tested the daisy chain scenario, but this time with a RemoteException thrown at the end of the execution. We returned to the Place Order window to place a new order. This time we used `ROLLBACK` as the User Name (see Figure 13-22 on page 508). In the daisy chain scenario, this exception takes place after the insertions in the two DB2 tables and the update of the CICS VSAM file.

When the exception is thrown, the transaction is rolled back and the updates are backed out from:

- The ITSO.ORDER table
- The CICS VSAM file
- The ITSO.DISPATCH table

We can see in the WebSphere logs that the AtomicClient application successfully inserted the order record in the database prior to rolling back the transaction (Example 13-26).

Example 13-26 Catalog Application log

```
CatalogContoller:doPost() - Action = Place Order
CatalogController.handlePlaceOrder() - creating the order
CatalogController.handlePlaceOrder() - beginning the transaction
CatalogController.handlePlaceOrder() - inserting the order in the database
OrderDB.insertOrder() - inserted the order in the database!!!!
CatalogController.handlePlaceOrder() - calling the CICS web service
CatalogController.handlePlaceOrder() - response back from the CICS web service
CatalogController.handlePlaceOrder() - simulating the RemoteException
CatalogController.handlePlaceOrder() - rollingback the transaction
```

We also can see that the DispatchOrderAtomic application successfully inserted the order record (Example 13-27).

Example 13-27 DispatchOrder Application log

```
DispatchOrderSoapBindingImpl.dispatchOrder(): ItemRef=10 Quantity=1
CustomerName=ROLLBACK Dept=Itso
DispatchOrderSoapBindingImpl.dispatchOrder() - creating the order
DispatchOrderSoapBindingImpl.dispatchOrder() - inserting the order in the
database
DispatchDB.insertDispatch() - inserted the order in the database!!!!
DispatchOrderSoapBindingImpl.dispatchOrder() - after the insert!!!
```

Following the RemoteException, the atomic transaction is rolled back. When we inquired on the stock level, we noted that there were still 30 items in stock.

13.4 Transaction scenario summary

These test scenarios demonstrate how you can synchronize WebSphere and CICS updates using WS-AT. They show how the classical 2PC distributed transaction can be implemented using Web services, and that the distributed global transaction can be committed or rolled back based on a set of Web service flows that are managed entirely by the WebSphere and CICS middleware.

Important: Before implementing a solution based on WS-AT, you should be aware of the general issues that can arise from any implementation of a distributed transaction (for example, locked records preventing access to important data) and you should also compare the solution with alternatives such as the J2EE Connector Architecture.

Appendixes

A

Sample handler programs

In this appendix we show the sample message handler and header processing programs used in our test scenarios:

- Sample message handler program - CIWSMSGH
- Sample header processing program - CIWSSECH
- Sample handler program - SNIFFER
- Sample XML parser program - MYPARSER
- Sample header processing program - CIWSSECR
- Sample header processing program - CIWSSECS
- Sample header processing program - WSATHND

A.1 Sample message handler program - CIWSMSGH

The CIWSMSGH program in Example A-1 was used to change the transaction ID for the Web service requests received in the pipeline.

The program obtains the value in the DFHFUNCTION container and if the program was invoked for a RECEIVE-REQUEST function, then it continues processing. It then obtains the data in the DFHWS-WEBSERVICE container and makes a decision based on the contents of the requested Web service name as to which transaction ID should be used. This new value for the transaction ID is put into the DFHWS-TRANID container. The program puts the contents of the DFHWS-TRANID and the DFHWS-WEBSERVICE containers to the CESE CICS transient data queue.

Example: A-1 Sample message handler program - CIWSMSGH

```
000100 IDENTIFICATION DIVISION.
000200 PROGRAM-ID.  CIWSMSGH
000300*****************************************************************
000400*                                                               *
000500* This program:-                                                *
000600* 1. CHANGES THE DEFAUL TRANID : CPIH/CPIL                      *
000700*    BASED ON THE INCOMING SOAP REQUESTS                        *
000800*****************************************************************
000900
001000 AUTHOR.      CHIEREGATTI.
001100 DATE-COMPILED.
001200 ENVIRONMENT DIVISION.
001300 CONFIGURATION SECTION.
001400 SPECIAL-NAMES.
001500 DATA DIVISION.
001600 WORKING-STORAGE SECTION.
001700 01 WS-START.
001800* Nesting offset for DISPLAYS
001900   03 NN                    PIC X(11) VALUE 'CIWSMSGH: '.
002000
002100   03 RESP                     PIC S9(8) COMP-5 SYNC.
002200   03 RESP2                    PIC S9(8) COMP-5 SYNC.
002300   03 BOD-PTR                  USAGE IS POINTER.
002400   03 BOD-LEN                  PIC S9(8) COMP-4.
002500   03 CONTAINER-LEN            PIC S9(8) BINARY.
002600   03 GETMAIN-PTR              USAGE IS POINTER.
002700   03 GETMAIN-LEN              PIC S9(8) COMP-4.
002800   03 WS-FAULT-STRING          PIC X(21) value spaces.
002900   03 WS-DFHFUNCTION           PIC X(16) value spaces.
003000   03 WS-WEBSERVICES           PIC X(30) value spaces.
003100   03 WS-WEBSERVICES-LEN       PIC S9(8) BINARY.
003200*
```

```
003300* Not Found SOAP Fault Detail section
003400*
003500 01  WS-Fault-NotFnd.
003600   03 WS-Fault-Namespace   pic x(53).
003700   03 WS-Fault-RC-Lit      pic X(17).
003800   03 WS-Fault-RC          pic X(2).
003900   03 WS-Fault-RC-ELit     pic X(18).
004000   03 WS-Fault-Item-Lit    pic X(11).
004100   03 WS-Fault-Item        pic X(4).
004200   03 WS-Fault-Item-ELit   pic X(12).
004300   03 WS-Fault-ENamespace  pic x(19).
004400
004500 01 EXPARSER-COMLEN            PIC S9(4) COMP-4.
004600 01 CA-PARSER-RSP.
004700    03 CA-PARSER-REQUEST.
004800      05 FILLER               PIC X(2).
004900      05 CA-TRANID            PIC X(4).
005000    03 CA-PARSER-REF-REQ      pic x(4).
005100    03 CA-PARSER-RET-CODE     pic x(2).
005200    03 FILLER                 PIC X(200).
005300
005400*****************************************************************
005500* Externally referenced data areas
005600*****************************************************************
005700 LINKAGE SECTION.
005800 01 BOD-AREA.
005900    02 FILLER    PIC X OCCURS 64000 DEPENDING ON BOD-LEN.
006000
006100 01 GETMAIN-AREA.
006200    02 FILLER    PIC X OCCURS 64000 DEPENDING ON GETMAIN-LEN.
006300 01 CONTAINER-DATA.
006400    05 FILLER            PIC X OCCURS 32768
006500                            DEPENDING ON CONTAINER-LEN.
006600*---------------------------------------------------------------*
006700    EJECT
006800*---------------------------------------------------------------*
006900 PROCEDURE DIVISION.
007000*---------------------------------------------------------------*
007100     EXEC CICS GET CONTAINER('DFHFUNCTION')
007200                   INTO(WS-DFHFUNCTION)
007300                   FLENGTH(length of WS-DFHFUNCTION)
007400                   NOHANDLE
007500     END-EXEC.
007600* If not RECEIVE-REQUEST then exit
007700     IF WS-DFHFUNCTION equal 'RECEIVE-REQUEST'
007800         PERFORM VALIDATE-REQUEST THRU END-VAL-REQUEST
007900         PERFORM CHANGE-TRANID    THRU END-CHANGE-TRANID
008000         EXEC CICS
008100            DELETE CONTAINER('DFHRESPONSE')
```

```
008200         END-EXEC
008300     END-IF
008400     EXEC CICS RETURN END-EXEC.
008500     GOBACK.
008600*-----------------------------------------------------------------
008700     EJECT
008800*-----------------------------------------------------------------
008900*------------- REQUEST VALIDATION -------------------------------
009000*-----------------------------------------------------------------
009100 VALIDATE-REQUEST.
009200     PERFORM GET-SOAP-WEBSERVICE THRU END-GET-SOAP-WEBSERVICE.
009300     IF WS-WEBSERVICES = 'ERROR'
009400         PERFORM FAULT-INVREQ
009500     END-IF.
009600 END-VAL-REQUEST. EXIT.
009700*-----------------------------------------------------------------
009800     EJECT
009900*-----------------------------------------------------------------
010000* Retrieve the DFHWS-WEBSERVICE that contains type of requests
010100*-----------------------------------------------------------------
010200 GET-SOAP-WEBSERVICE.
010300     EXEC CICS
010400     GET CONTAINER('DFHWS-WEBSERVICE')
010500     SET(ADDRESS OF CONTAINER-DATA)
010600     FLENGTH(CONTAINER-LEN)
010700     END-EXEC.
010800* Copy the input container to our storage
010900     MOVE CONTAINER-DATA(1:30) TO WS-WEBSERVICES.
011000     MOVE CONTAINER-LEN TO WS-WEBSERVICES-LEN.
011100 END-GET-SOAP-WEBSERVICE. EXIT.
011200*-----------------------------------------------------------------
011300     EJECT
011400*---------------- CHANGE DEFAULT TRANID CPIH/CPIL ---------------
011500 CHANGE-TRANID.
011600       EXEC CICS GET CONTAINER('DFHWS-TRANID')
011700            SET(ADDRESS OF CONTAINER-DATA)
011800            FLENGTH(CONTAINER-LEN)
011900       END-EXEC.
012000     IF WS-WEBSERVICES = 'inquireSingle'
012100        MOVE 'INQS' TO CA-TRANID
012200        PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
012300     END-IF
012400     IF WS-WEBSERVICES = 'inquireCatalog'
012500        MOVE 'INQC' TO CA-TRANID
012600        PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
012700     END-IF
012800     IF WS-WEBSERVICES = 'placeOrder'
012900        MOVE 'ORDR' TO CA-TRANID
012901     END-IF.
```

```
012910     IF WS-WEBSERVICES = 'dispatchOrderEndpoint'
012920        MOVE 'DISP' TO CA-TRANID
013000        PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
013100     END-IF.
013200 END-CHANGE-TRANID. EXIT.
013300*-----------------------------------------------------------------
013400     EJECT
013500*-----------------------------------------------------------------
013600 CHANGE-CONTAINER.
013700     MOVE CA-TRANID TO CONTAINER-DATA(1:4)
013800     EXEC CICS PUT CONTAINER('DFHWS-TRANID')
013900          FROM(CONTAINER-DATA)
014000          FLENGTH(CONTAINER-LEN)
014100     END-EXEC.
014200     DISPLAY NN '>================================<'
014300     DISPLAY NN 'Container Name: : DFHWS-WEBSERVICE '.
014400*    DISPLAY NN 'Content length: '    WS-WEBSERVICES-LEN.
014500     DISPLAY NN 'Container content: ' WS-WEBSERVICES.
014600     DISPLAY NN '----------------------------------'
014700     DISPLAY NN 'Container Name: : DFHWS-TRANID '.
014800*    DISPLAY NN 'Content length: '    CONTAINER-LEN
014900     DISPLAY NN 'Container content: ' CONTAINER-DATA.
015000 END-CHANGE-CONTAINER. EXIT.
015100     EJECT
015200****************************************************************
015300* We detected that the ca_request_id field specifies an invalid
015400* request. This is a CLIENT error.
015500****************************************************************
015600 FAULT-INVREQ SECTION.
015700*--------------------------------------------------------------*
015800* Generate a SOAP Fault
015900*--------------------------------------------------------------*
016000     MOVE 'Request code invalid' to WS-FAULT-STRING
016100     EXEC CICS SOAPFAULT CREATE
016200               FAULTCODE(dfhvalue(CLIENT))
016300               FAULTSTRING(WS-FAULT-STRING)
016400               FAULTSTRLEN(length of WS-FAULT-STRING)
016500     END-EXEC.
016600 FAULT-INVREQ-END. EXIT.
016700****************************************************************
016800* The supplied ca_item_req_ref reference is not in our database
016900* We decide to send a SOAP Fault.
017000* This is a SERVER error.
017100* We supply detailed information in the DETAIL Fault element
017200****************************************************************
017300 FAULT-NOTFND SECTION.
017400*--------------------------------------------------------------*
017500* Build the Detail section.
017600*  we do it this way for pedagogical reasons
```

```
*----------------------------------------------------------------*
     MOVE ':o(   NOT FOUND   )o:' to WS-FAULT-STRING
     MOVE
     '<hdlh:FaultDetail xmlns:hdlh="http://HDRHDLRX.fault">'
                               to WS-Fault-Namespace.
     MOVE '<hdlh:ReturnCode>'  to WS-Fault-RC-Lit.
     MOVE CA-PARSER-RET-CODE   to WS-Fault-RC.
     MOVE '</hdlh:ReturnCode>' to WS-Fault-RC-ELit.
     MOVE '<hdlh:Item>'        to WS-Fault-Item-Lit.
     MOVE ca-PARSER-REF-REQ    to WS-Fault-Item.
     MOVE '</hdlh:Item>'       to WS-Fault-Item-ELit.
     MOVE '</hdlh:FaultDetail>'
            to WS-Fault-ENamespace.
     EXEC CICS SOAPFAULT CREATE
               DETAIL(WS-Fault-NotFnd)
               DETAILLENGTH(length of WS-Fault-NotFnd)
               FAULTCODE(dfhvalue(SERVER))
               FAULTSTRING(WS-FAULT-STRING)
               FAULTSTRLEN(length of WS-FAULT-STRING)
     END-EXEC.
 FAULT-INVREQ-END. EXIT.
```

Sample output from CIWSMSGH

Example A-2 shows the output sent to the CESE transient data queue by the CIWSMSGH message handler program, whenever it changes the transaction ID in the DFHWS-TRANID container.

Example: A-2   CIWSMSGH - sample output

```
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: inquireCatalog
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: INQC
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: inquireSingle
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: INQS
CIWSMSGH:  >================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: placeOrder
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: ORDR
```

A.2 Sample header processing program - CIWSSECH

The CIWSSECH program (Example A-3) extracts security credentials from the WS-Security header, validates them and changes the user ID under which the business logic of the invoked Web service executes.

- It first determines that it is invoked during the RECEIVE-REQUEST phase of processing.
- It then ensures that this is only executed for a “place an order” request by examining the data in the DFHWS-URI container. In the event that this is not the case, it returns control to the calling program.
- When it is processing an order request, the WS-Security header is obtained from the DFHHEADER container.
- The header is then passed in a COMMAREA to a COBOL XML parsing program that extracts the user ID and password from the XML.
- CIWSSECH then uses the EXEC CICS VERIFY PASSWORD command to determine if the user ID and password are correct. If this is true, then the user ID is written to the DFHWS-USERID container, which will cause a context switch to occur when invoking the business logic. If the verification returns an error, then a SOAP fault is created and returned to the requester.

Example: A-3 Sample header processing program - CIWSSECH

```
000100  PROCESS CICS
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID.  CIWSSECH
000400*****************************************************************
000500*                                                               *
000600* This program:-                                                *
000700* 1. Checks for RECEIVE-REQUEST function, else returns          *
000800* 2. Obtains data from DFHHEADER container                      *
000900* 2. Uses the MYPARSER program to extract the CA-USER-ID        *
001000*                                     and the CA-PASSWORD       *
001100*    fields from the XML message contained within the Body      *
001200* 2. Verifies the user ID/password combination                  *
001300*    If it is succesful, then it places the CA-USER-ID into     *
001400*        the DFHWS-USERID container.                            *
001500*    If it fails, it returns a SOAP fault message in DFHRESPONSE*
001600*****************************************************************
001700
001800 AUTHOR.      CI11M1.
001900 DATE-COMPILED.
002000 ENVIRONMENT DIVISION.
002100 CONFIGURATION SECTION.
002200 SPECIAL-NAMES.
002300 DATA DIVISION.
002400 WORKING-STORAGE SECTION.
002500 01 WS-START.
```

```
   03 FILLER                   PIC X(44)
       VALUE '*** CIWSSECH WORKING STORAGE STARTS HERE ***'.
   03 WS-RESP                  PIC S9(8) COMP-5 SYNC.
   03 WS-RESP2                 PIC S9(8) COMP-5 SYNC.
   03 WS-HEAD-PTR              USAGE IS POINTER.
   03 WS-HEAD-LEN              PIC S9(8) COMP-4.
   03 WS-URI-PTR               USAGE IS POINTER.
   03 WS-URI-LEN               PIC S9(8) COMP-4.
   03 WS-FUNC-LEN              PIC S9(8) COMP-4 VALUE 16.
   03 WS-GETMAIN-PTR           USAGE IS POINTER.
   03 WS-GETMAIN-LEN           PIC S9(8) COMP-4.
   03 WS-FAULT-STRING          PIC X(40) value spaces.
   03 WS-FAULT-CODE            PIC S9(8) COMP-4.
   03 WS-SOAP-LEVEL            PIC S9(8) COMP-4.
     88 WS-SOAP-11  VALUE 1.
     88 WS-SOAP-12  VALUE 2.
     88 WS-NOT-SOAP VALUE 10.
   03 WS-FUNC-AREA             PIC X(16).
   03 WS-NOT-AUTH              PIC X(40)
               VALUE 'Not authorized to place order.'.
   03 WS-AUTH-FAILED           PIC X(40)
               VALUE 'Authorization failed for order request.'.

 01 MYPARSER-COMLEN            PIC S9(4) COMP-4.
 01 CA-PARSER-RSP.
    03 CA-USER-ID             PIC X(8).
    03 CA-PASSWORD            PIC X(8).

 LINKAGE SECTION.
 01 WS-HEAD-AREA.
    02 FILLER    PIC X OCCURS 1024 DEPENDING ON WS-HEAD-LEN.

 01 WS-URI-AREA.
    02 FILLER    PIC X OCCURS 256 DEPENDING ON WS-URI-LEN.

 01 WS-GETMAIN-AREA.
    02 FILLER    PIC X OCCURS 1024 DEPENDING ON WS-GETMAIN-LEN.

******************************************************************
* Main line code begins                                          *
******************************************************************
 PROCEDURE DIVISION.
 MAIN-PROCESSING SECTION.

***********************************
*  Receive the SOAP Body namespace
***********************************
     PERFORM GET-DFHFUNCTION.

```

```
***********************************
*  Receive the SOAP Body namespace
***********************************
     PERFORM GET-DFHWS-URI.
     IF WS-URI-AREA(1:WS-URI-LEN)
              NOT = '/exampleApp/placeOrder'
        EXEC CICS RETURN END-EXEC
     END-IF.

***********************************
*  Receive the SOAP Body namespace
***********************************
     PERFORM GET-SOAP-HEADER.

************************************************************
*  The SOAP body XML data can now be parsed by MYPARSER
*    LINK to the XML parser, it will return the CA-USER-ID
*                                           and CA-PASSWORD
*                                           or  'EXCEPTION'
*                                           or  'NOT FOUND'
*                                           or  'BAD COMMA'
************************************************************
     EXEC CICS LINK PROGRAM('MYPARSER')
                    COMMAREA(WS-HEAD-AREA)
                    LENGTH(MYPARSER-COMLEN)
     END-EXEC.
     IF WS-HEAD-AREA(1:9) = 'EXCEPTION' or
                         'NOT FOUND' or
                         'BAD COMMA'
******************************************************
*  Error found during the XML PARSE program execution
******************************************************
         MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
         PERFORM FAULT-MESSAGE
     ELSE
         MOVE WS-HEAD-AREA to CA-PARSER-RSP
************************************
*  OK, data found, so do the VERIFY
************************************
         EXEC CICS VERIFY
            PASSWORD(CA-PASSWORD)
            USERID(CA-USER-ID)
            RESP(WS-RESP)
         END-EXEC
***************
*  Succesful ?
***************
         IF WS-RESP = DFHRESP(NORMAL)
            PERFORM SET-USER-ID
```

```
012400         ELSE
012500            MOVE WS-NOT-AUTH TO WS-FAULT-STRING
012600            PERFORM FAULT-MESSAGE
012700         END-IF
012800     END-IF.
012900****************************
013000*  and  is the end, bye bye
013100****************************
013200     EXEC CICS RETURN END-EXEC.
013300
013400 MAIN-PROCESSING-END. EXIT.
013500
013600****************************************************
013700*     ****************************************     *
013800*              SUBROUTINES FOLLOW                  *
013900*     ****************************************     *
014000****************************************************
014100
014200*****************************************************************
014300* Retrieve the function type ftom DFHFUNCTION container
014400*****************************************************************
014500 GET-DFHFUNCTION.
014600     EXEC CICS
014700        GET CONTAINER('DFHFUNCTION')
014800        INTO(WS-FUNC-AREA)
014900        FLENGTH(WS-FUNC-LEN)
015000        RESP(WS-RESP)
015100     END-EXEC.
015200
015300*************************************************
015400* Check for correct length. Create SOAP fault if
015500* not correct. If for incoming request then we
015600* do the work, else we exit immediately.
015700*************************************************
015800     IF WS-FUNC-LEN NOT = 16
015900        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
016000        PERFORM FAULT-MESSAGE
016100     ELSE
016200        IF WS-FUNC-AREA NOT = 'RECEIVE-REQUEST '
016300           EXEC CICS RETURN END-EXEC
016400        END-IF
016500     END-IF.
016600
016700*****************************************************************
016800* Retrieve the URI from the DFHWS-URI container
016900*****************************************************************
017000 GET-DFHWS-URI.
017100     EXEC CICS
017200        GET CONTAINER('DFHWS-URI')
```

```
017300         SET(WS-URI-PTR)
017400         FLENGTH(WS-URI-LEN)
017500         RESP(WS-RESP)
017600      END-EXEC.
017700
017800*******************************************
017900* Copy the input container to our storage
018000*******************************************
018100      IF WS-URI-LEN > 0
018200         SET ADDRESS OF WS-URI-AREA TO WS-URI-PTR
018300         MOVE WS-URI-LEN TO WS-GETMAIN-LEN
018400
018500         EXEC CICS GETMAIN
018600             SET(WS-GETMAIN-PTR)
018700             FLENGTH(WS-GETMAIN-LEN)
018800         END-EXEC
018900
019000         SET ADDRESS OF WS-GETMAIN-AREA TO WS-GETMAIN-PTR
019100         MOVE WS-URI-AREA TO WS-GETMAIN-AREA
019200         SET WS-URI-PTR TO WS-GETMAIN-PTR
019300         SET ADDRESS OF WS-URI-AREA TO WS-URI-PTR
019400      ELSE
019500         MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
019600         PERFORM FAULT-MESSAGE
019700         EXEC CICS RETURN END-EXEC
019800      END-IF.
019900
020000***************************************
020100* Retrieve the <wsse:Security> header
020200***************************************
020300 GET-SOAP-HEADER.
020400      EXEC CICS
020500         GET CONTAINER('DFHHEADER')
020600         SET(WS-HEAD-PTR)
020700         FLENGTH(WS-HEAD-LEN)
020800         RESP(WS-RESP)
020900      END-EXEC.
021000
021100*******************************************
021200* Copy the input container to our storage
021300*******************************************
021400      IF WS-HEAD-LEN > 0
021500         SET ADDRESS OF WS-HEAD-AREA TO WS-HEAD-PTR
021600         MOVE WS-HEAD-LEN TO WS-GETMAIN-LEN MYPARSER-COMLEN
021700
021800         EXEC CICS GETMAIN
021900             SET(WS-GETMAIN-PTR)
022000             FLENGTH(WS-GETMAIN-LEN)
022100         END-EXEC
```

```

        SET ADDRESS OF WS-GETMAIN-AREA TO WS-GETMAIN-PTR
        MOVE WS-HEAD-AREA TO WS-GETMAIN-AREA
        SET WS-HEAD-PTR TO WS-GETMAIN-PTR
        SET ADDRESS OF WS-HEAD-AREA TO WS-HEAD-PTR
     ELSE
        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
        PERFORM FAULT-MESSAGE
        EXEC CICS RETURN END-EXEC
     END-IF.

******************************************************************
* The user ID has been verified. We will now set the contents of
* DFHWS-USERID container with this value.This will cause the
* business logic (back-end program/s) to be executed with this
* user ID, using it's access rights.
******************************************************************
 SET-USER-ID.
     EXEC CICS
         PUT CONTAINER('DFHWS-USERID')
         FROM(CA-USER-ID)
         FLENGTH(length of CA-USER-ID)
         DATATYPE(DFHVALUE(CHAR))
         RESP(WS-RESP)
     END-EXEC.

******************************************************************
* We detected that the ca_request_id field specifies an invalid
* request. This is a CLIENT error.
******************************************************************
 FAULT-MESSAGE SECTION.
************************
* Generate a SOAP Fault
************************
     EXEC CICS
         GET CONTAINER('DFHWS-SOAPLEVEL')
         INTO(WS-SOAP-LEVEL)
         FLENGTH(WS-HEAD-LEN)
         RESP(WS-RESP)
     END-EXEC.

************************************
* MOVE CORRECT VERSION OF FAULTCODE
************************************
     IF WS-SOAP-11 MOVE DFHVALUE(CLIENT) TO WS-FAULT-CODE
     ELSE
        MOVE DFHVALUE(SENDER) TO WS-FAULT-CODE
     END-IF

     EXEC CICS SOAPFAULT CREATE
               FAULTSTRING(WS-FAULT-STRING)
               FAULTSTRLEN(LENGTH OF WS-FAULT-STRING)
               FAULTCODE(WS-FAULT-CODE)
     END-EXEC.
 FAULT-MESSAGE-END. EXIT.
```

A.3 Sample handler program - SNIFFER

The SNIFFER handler program can be used as a message handler program or a header processing program.

It is a very simple program that browses through the containers available in the pipeline. It does this by issuing a STARTBROWSE CONTAINER command followed by GETNEXT CONTAINER until all have been browsed, then it issues an ENDBROWSE CONTAINER command. For each container that it issues a GETNEXT CONTAINER for, it displays the name and contents to the CICS transient data queue CESE.

Example: A-4 Sample handler program - SNIFFER

```
0ST TRUNC(OPT)
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SNIFFER.
*****************************************************************
* HEADER START                                                  *
*                                                               *
*                                                               *
* PROGRAM NAME: See above                                       *
*                                                               *
* TITLE: Pipeline Channel container sniffer                     *
*                                                               *
* DATE: 03/06/05                                                *
*                                                               *
* AUTHOR: Ian Noble                                             *
*                                                               *
* CHANGE HISTORY:                                               *
*                                                               *
*   DATE MODIFIED  CHANGED BY       REASON FOR CHANGE           *
*                                                               *
* PROGRAM DESCRIPTION:                                          *
*                                                               *
*   This program writes a report to the CESE transient data     *
*   queue which is usually an extra-partition queue having a    *
*   DDname of CEEMSG. The report contains the name, length, and *
*   contents of every container which is visible to user code   *
```

```
002600*   within a pipeline as shown in the following example:         *
002700*                                                                *
002800*   SNIFFER :  *** Start ***                                     *
002900*   SNIFFER :  >================================<                *
003000*   SNIFFER :  Container Name   : DFHFUNCTION                    *
003100*   SNIFFER :  Content length   : 00000016                       *
003200*   SNIFFER :  Container content:                                *
003300*   RECEIVE-REQUEST                                              *
003400*   SNIFFER :  Containers on channel: List starts.               *
003500*   SNIFFER :  >================================<                *
003600*   SNIFFER :  Container Name   : DFH-HANDLERPLIST               *
003700*   SNIFFER :  Content length   : 00000000                       *
003800*   SNIFFER :  Container EMPTY                                   *
003900*   SNIFFER :  >================================<                *
004000*   SNIFFER :  Container Name   : DFHRESPONSE                    *
004100*   SNIFFER :  Content length   : 00000000                       *
004200*   SNIFFER :  Container EMPTY                                   *
004300*   SNIFFER :  >================================<                *
004400*   SNIFFER :  Container Name   : DFHFUNCTION                    *
004500*   SNIFFER :  Content length   : 00000016                       *
004600*   SNIFFER :  Container content:                                *
004700*   RECEIVE-REQUEST                                              *
004800*   SNIFFER :  >================================<                *
004900*   SNIFFER :  Container Name   : DFHWS-SOAPACTION               *
005000*   SNIFFER :  Content length   : 00000002                       *
005100*   SNIFFER :  Container content:                                *
005200*   ""                                                           *
005300*   SNIFFER :  >================================<                *
005400*   SNIFFER :  Container Name   : DFHWS-URI                      *
005500*   SNIFFER :  Content length   : 00000022                       *
005600*   SNIFFER :  Container content:                                *
005700*   /exampleApp/placeOrder                                       *
005800*   SNIFFER :  >================================<                *
005900*   SNIFFER :  Container Name   : DFHREQUEST                     *
006000*   SNIFFER :  Content length   : 00002094                       *
006100*   SNIFFER :  Container content:                                *
006200*   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoa...    *
006300*   " xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:x...    *
006400*   lsoap.org/ws/2004/10/wscoor" xmlns:wsa="http://schemas...    *
006500*   etc. etc. etc. etc. etc. etc. etc. etc. etc. etc. etc.       *
006600*   SNIFFER :  >================================<                *
006700*   SNIFFER :  Container Name   : DFH-SERVICEPLIST               *
006800*   SNIFFER :  Content length   : 00000134                       *
006900*   SNIFFER :  Container content:                                *
007000*   <registration_service_endpoint>       http://MVSG3.mop etc. *
007100*   SNIFFER :  >================================<                *
007200*   SNIFFER :  Container Name   : DFHWS-PIPELINE                 *
007300*   SNIFFER :  Content length   : 00000008                       *
007400*   SNIFFER :  Container content:                                *
```

```
007500*   PIPE1                                                     *
007600*   SNIFFER :  >=================================<            *
007700*   SNIFFER :  Container Name   : DFHWS-USERID                *
007800*   SNIFFER :  Content length   : 00000008                    *
007900*   SNIFFER :  Container content:                             *
008000*   CIWS3D                                                    *
008100*   SNIFFER :  >=================================<            *
008200*   SNIFFER :  Container Name   : DFHWS-TRANID                *
008300*   SNIFFER :  Content length   : 00000004                    *
008400*   SNIFFER :  Container content:                             *
008500*   CPIH                                                      *
008600*   SNIFFER :  >=================================<            *
008700*   SNIFFER :  Container Name   : DFHWS-WEBSERVICE            *
008800*   SNIFFER :  Content length   : 00000032                    *
008900*   SNIFFER :  Container content:                             *
009000*   placeOrder                                                *
009100*   SNIFFER :  >=================================<            *
009200*   SNIFFER :  Container Name   : DFHWS-APPHANDLER            *
009300*   SNIFFER :  Content length   : 00000008                    *
009400*   SNIFFER :  Container content:                             *
009500*   DFHPITP                                                   *
009600*   SNIFFER :  Containers on channel: List ends               *
009700*   SNIFFER :  DFHRESPONSE      container deleted             *
009800*   SNIFFER :  **** End ****                                  *
009900*                                                             *
010000***************************************************************
010100
010200 ENVIRONMENT DIVISION.
010300 DATA DIVISION.
010400 WORKING-STORAGE SECTION.
010500
010600 77  WS-EYE-CATCHER         PIC X(44)     VALUE
010700                  '*** SNIFFER WORKING STORAGE STARTS HERE ***'.
010800
010900* Nesting offset for DISPLAYS
011000 77  NN                     PIC X(11) VALUE 'SNIFFER : '.
011100
011200* Miscellaneous variables
011300
011400 77 CONTAINER-FOUND         PIC X(01).
011500
011600 77 CHANNEL-NAME               PIC X(16).
011700 77 CONTAINER-NAME             PIC X(16).
011800 77 CONTAINER-LEN              PIC S9(8) COMP-4 SYNCHRONIZED
011900                                             VALUE 0.
012000
012100 77 DFHFUNCTION-IN-PTR         USAGE IS POINTER.
012200 77 DFHFUNCTION-IN-LEN         PIC S9(8) COMP-4 SYNCHRONIZED
012300                                             VALUE 0.
```

```

 77 GETMAIN-PTR                USAGE IS POINTER.
 77 GETMAIN-LEN                PIC S9(8) COMP-4 SYNCHRONIZED
                                                 VALUE 0.

* CTS SIBUS declarations
 COPY DFHPIUCO.

 77 FUNCTION-DRIVEN         PIC X(16).

 77 CALL-LABEL                PIC X(08).
 77 CALL-DETAILS              PIC X(80).

 77 RESP                      PIC S9(8)  COMP-5 SYNC.
 77 RESP2                     PIC S9(8)  COMP-5 SYNC.

* Channel browse stuff
 01 CHANNEL-BROWSING.
    03 BROWSE-TOKEN           PIC S9(8) COMP-4 SYNC.
    03 BROWSED-NAME           PIC X(16).

***************************************************************
* Constants
***************************************************************

 77 UNEXPECTED-RESP-ABCODE    PIC X(04) VALUE 'SNIF'.

*----------------------------------------------------------------
 LINKAGE SECTION.

 01 DFHFUNCTION-IN.
    03 FILLER                  PIC X OCCURS 1 TO 64
                               DEPENDING ON DFHFUNCTION-IN-LEN.

 01 GETMAIN-AREA.
    03 FILLER                  PIC X OCCURS 1 TO 65536
                                     DEPENDING ON GETMAIN-LEN.


*----------------------------------------------------------------
 PROCEDURE DIVISION.
 MAIN-PROG SECTION.

     Display NN '*** Start ***'.

     PERFORM GET-DFHFUNCTION-CONTAINER.

     PERFORM BROWSE-ALL-CONTAINERS.

     IF FUNCTION-DRIVEN = PI-SEND-REQUEST
     OR FUNCTION-DRIVEN = PI-RECEIVE-REQUEST
     OR FUNCTION-DRIVEN = PI-PROCESS-REQUEST
     THEN
        PERFORM DELETE-DFHRESPONSE
     END-IF.

     Display NN '**** End ****'.

     EXEC CICS RETURN
     END-EXEC.


*****************************************************************
*
* Retrieve the contents of the DFHFUNCTION container
*
*****************************************************************
 GET-DFHFUNCTION-CONTAINER.

     MOVE PI-DFHFUNCTION TO CONTAINER-NAME.
     PERFORM GET-NAMED-CONTAINER.

     IF CONTAINER-FOUND = 'Y'
     THEN
        MOVE CONTAINER-LEN TO DFHFUNCTION-IN-LEN
        SET DFHFUNCTION-IN-PTR TO GETMAIN-PTR
        SET ADDRESS OF DFHFUNCTION-IN TO DFHFUNCTION-IN-PTR
        MOVE DFHFUNCTION-IN TO FUNCTION-DRIVEN
     END-IF.



*****************************************************************
*
* Retrieve a named container
*
*****************************************************************
 GET-NAMED-CONTAINER.

     MOVE 'N' TO CONTAINER-FOUND
* Get the length of the name container from the input pipe.
     MOVE 'SNIFF001' TO CALL-LABEL.
     EXEC CICS GET
               CONTAINER(CONTAINER-NAME)
               NODATA
               FLENGTH(CONTAINER-LEN)
               NOHANDLE
               RESP(RESP)
               RESP2(RESP2)
     END-EXEC.

* If the container wasn't found, do nothing.  Else GETMAIN
* suitable storage and retrieve it for real.
     EVALUATE RESP
        WHEN DFHRESP(CONTAINERERR)
           DISPLAY NN 'Container NOT FOUND'
        WHEN DFHRESP(NORMAL)
           MOVE 'Y' TO CONTAINER-FOUND

           DISPLAY NN '>================================<'
           DISPLAY NN 'Container Name   : 'CONTAINER-NAME
           DISPLAY NN 'Content length   : ' CONTAINER-LEN

           IF CONTAINER-LEN > 0
           THEN
              MOVE 'SNIFF002' TO CALL-LABEL
              MOVE CONTAINER-LEN TO GETMAIN-LEN
              EXEC CICS GETMAIN SET(GETMAIN-PTR)
                                FLENGTH(GETMAIN-LEN)
              END-EXEC
              SET ADDRESS OF GETMAIN-AREA TO GETMAIN-PTR
              EXEC CICS GET CONTAINER(CONTAINER-NAME)
                            INTO(GETMAIN-AREA)
                            FLENGTH(CONTAINER-LEN)
              END-EXEC
              DISPLAY NN 'Container content: '
                      GETMAIN-AREA

           ELSE
              DISPLAY NN 'Container EMPTY'
           END-IF
        WHEN OTHER
           DISPLAY NN 'Error on GET CONTAINER ' CONTAINER-NAME
           PERFORM GENERIC-ABEND
     END-EVALUATE.


*****************************************************************
*
* BROWSE all containers on the current Channel
*
*****************************************************************
 BROWSE-ALL-CONTAINERS.

     DISPLAY NN 'Containers on channel: List starts.'

* Start the Channel Browse
```

```
     MOVE 'SNIFF003' TO CALL-LABEL.
     EXEC CICS STARTBROWSE
               CONTAINER
               BROWSETOKEN(BROWSE-TOKEN)
               RESP(RESP)
               RESP2(RESP2)
     END-EXEC.

* Browse the next container name
     PERFORM UNTIL RESP NOT = DFHRESP(NORMAL)

        MOVE 'SNIFF004' TO CALL-LABEL
        EXEC CICS GETNEXT
                  CONTAINER(BROWSED-NAME)
                  BROWSETOKEN(BROWSE-TOKEN)
                  RESP(RESP)
                  RESP2(RESP2)
        END-EXEC

        IF RESP = DFHRESP(NORMAL)
        THEN
           MOVE BROWSED-NAME TO CONTAINER-NAME
           PERFORM GET-NAMED-CONTAINER
        END-IF

     END-PERFORM.

     IF RESP NOT = DFHRESP(END)
     THEN
        PERFORM GENERIC-ABEND
     END-IF.

     DISPLAY NN 'Containers on channel: List ends'.

* End the Channel Browse
     MOVE 'SNIFF005' TO CALL-LABEL.
     EXEC CICS ENDBROWSE
               CONTAINER
               BROWSETOKEN(BROWSE-TOKEN)
               RESP(RESP)
               RESP2(RESP2)
     END-EXEC.
     IF RESP NOT = DFHRESP(NORMAL)
     THEN
        PERFORM GENERIC-ABEND
     END-IF.


*****************************************************************
```

```
*                                                                *
* Delete the DFHRESPONSE container                               *
*                                                                *
******************************************************************
 DELETE-DFHRESPONSE.

     MOVE 'SNIFF006' TO CALL-LABEL.
     EXEC CICS ASSIGN
               CHANNEL(CHANNEL-NAME)
               RESP(RESP)
               RESP2(RESP2)
     END-EXEC.

     IF CHANNEL-NAME = 'DFHHHC-V1' THEN
        DISPLAY NN ' in a SOAP header processing program.....'
     ELSE
        EXEC CICS DELETE
                  CONTAINER(PI-DFHRESPONSE)
                  RESP(RESP)
                  RESP2(RESP2)
        END-EXEC
        EVALUATE RESP
           WHEN DFHRESP(NORMAL)
              DISPLAY NN PI-DFHRESPONSE ' container deleted'
           WHEN OTHER
              PERFORM GENERIC-ABEND
        END-EVALUATE
     END-IF.


******************************************************************
*
* Generic abend
*
******************************************************************
 GENERIC-ABEND.


      DISPLAY NN '***** Unable to continue *****'.
      DISPLAY NN 'Unexpected RESP code from EXEC CICS call'.
      DISPLAY NN 'Call label : ' CALL-LABEL.
      DISPLAY NN 'RESP  : ' RESP.
      DISPLAY NN 'RESP2 : ' RESP2.
      IF CALL-DETAILS NOT = SPACES
      THEN
         DISPLAY NN 'Further details:'
                    CALL-DETAILS
      END-IF.

     EXEC CICS ABEND
        ABCODE(UNEXPECTED-RESP-ABCODE)
     END-EXEC.
```

Sample output from the SNIFFER program

Example A-5 shows sample output from the SNIFFER program. The output is sent to CICS transient data queue CESE.

Example: A-5 SNIFFER program - sample output

```
SNIFFER :  *** Start ***
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHFUNCTION
SNIFFER :  Content length   : 00000016
SNIFFER :  Container content: RECEIVE-REQUEST
SNIFFER :  Containers on channel: List starts.
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFH-HANDLERPLIST
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHRESPONSE
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHFUNCTION
SNIFFER :  Content length   : 00000016
SNIFFER :  Container content: RECEIVE-REQUEST
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-SOAPACTION
SNIFFER :  Content length   : 00000002
SNIFFER :  Container content: ""
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-URI
SNIFFER :  Content length   : 00000025
SNIFFER :  Container content: /exampleApp/inquireSingle
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHREQUEST
SNIFFER :  Content length   : 00001007
SNIFFER :  Container content: <soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"xmlns:soapenc="http://
schemas.xmlsoap.org/soap/encoding/"xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><soapenv:Header/><soapenv
:Body><p628:DFH0XCMNxmlns:p628="http://www.DFH0XCMN.DFH0XCP4.Request.com"><p628
:ca_request_id>01INQS</p628:ca_request_id><p628:ca_return_code>0</p628:ca_retur
n_code><p628:ca_response_message>
</p628:ca_response_message><p628:ca_inquire_single><p628:ca_item_ref_req>10</p6
28:ca_item_ref_req><p628:filler1> </p628:filler1><p628:filler2> </p628:filler2>
```

```
<p628:ca_single_item><p628:ca_sngl_item_ref>0</p628:ca_sngl_item_ref><p628:ca_s
ngl_description>
</p628:ca_sngl_description><p628:ca_sngl_department>0</p628:ca_sngl_department>
<p628:ca_sngl_cost>0.0</p628:ca_sngl_cost><p628:in_sngl_stock>0</p628:in_sngl_s
tock><p628:on_sngl_order>0</p628:on_sngl_order></p628:ca_single_item></p628:ca_
inquire_single></p628:DFHOXCMN></soapenv:Body></soapenv:Envelope>
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFH-SERVICEPLIST
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-PIPELINE
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: EXPIPE01
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-USERID
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: CIWS3D
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-TRANID
SNIFFER :  Content length   : 00000004
SNIFFER :  Container content: CPIH
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-WEBSERVICE
SNIFFER :  Content length   : 00000032
SNIFFER :  Container content: inquireSingle
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHWS-APPHANDLER
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: DFHPITP
SNIFFER :  Containers on channel: List ends
SNIFFER :  DFHRESPONSE      container deleted
SNIFFER :  **** End ****
CIWSMSGH:  >=================================<
CIWSMSGH:  Container Name: : DFHWS-WEBSERVICE
CIWSMSGH:  Container content: inquireSingle
CIWSMSGH:  ----------------------------------
CIWSMSGH:  Container Name: : DFHWS-TRANID
CIWSMSGH:  Container content: INQS
SNIFFER :  *** Start ***
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHFUNCTION
SNIFFER :  Content length   : 00000016
SNIFFER :  Container content: SEND-RESPONSE
SNIFFER :  Containers on channel: List starts.
SNIFFER :  >=================================<
SNIFFER :  Container Name   : DFHRESPONSE
SNIFFER :  Content length   : 00001042
SNIFFER :  Container content:
<SOAP-ENV:Envelopexmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"xmln
s:soapenc="http://schemas.xmlsoap.org/soap/encoding/"xmlns:xsd="http://www.w3.o
rg/2001/XMLSchema"xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"xmlns:SO
AP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Body><DFH0XCMNResp
onsexmlns="http://www.DFH0XCMN.DFH0XCP4.Response.com"><ca_request_id>01INQS</ca
_request_id><ca_return_code>0</ca_return_code><ca_response_message>RETURNED
ITEM: REF =0010
</ca_response_message><ca_inquire_single><ca_item_ref_req>10</ca_item_ref_req><
filler1>    </filler1><filler2>
</filler2><ca_single_item><ca_sngl_item_ref>10</ca_sngl_item_ref><ca_sngl_descr
iption>Ball Pens Black24pk
</ca_sngl_description><ca_sngl_department>10</ca_sngl_department><ca_sngl_cost>
002.90</ca_sngl_cost><in_sngl_stock>0</in_sngl_stock><on_sngl_order>0</on_sngl_
order></ca_single_item></ca_inquire_single></DFH0XCMNResponse></SOAP-ENV:Body><
/SOAP-ENV:Envelope>
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFH-HANDLERPLIST
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHFUNCTION
SNIFFER :  Content length   : 00000016
SNIFFER :  Container content: SEND-RESPONSE
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-SOAPACTION
SNIFFER :  Content length   : 00000002
SNIFFER :  Container content: ""
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-URI
SNIFFER :  Content length   : 00000025
SNIFFER :  Container content: /exampleApp/inquireSingle
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFH-SERVICEPLIST
SNIFFER :  Content length   : 00000000
SNIFFER :  Container EMPTY
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-PIPELINE
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: EXPIPE01
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-USERID
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: CIWS3D
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-TRANID
SNIFFER :  Content length   : 00000004
SNIFFER :  Container content: INQS
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-WEBSERVICE
SNIFFER :  Content length   : 00000032
SNIFFER :  Container content: inquireSingle
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-APPHANDLER
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: DFHPITP
SNIFFER :  >================================<
SNIFFER :  Container Name   : DFHWS-OPERATION
SNIFFER :  Content length   : 00000008
SNIFFER :  Container content: DFHOXCMN
SNIFFER :  Containers on channel: List ends
SNIFFER :  **** End ****
```

A.4 Sample XML parser program - MYPARSER

The MYPARSER program shown in Example A-6 is used by the CIWSSECH header processing program to parse the SOAP header in the DFHWS-DFHHEADER container. It uses the COBOL XML PARSE statement to parse through the SOAP WS-Security header that was received in the COMMAREA, extracting the user ID and password. This extracted data is then returned in the COMMAREA for the calling program to use.

Refer to the following publications for more information about XML parsing within COBOL programs:

- *Enterprise COBOL for z/OS V3R3 Programming Guide, SC27-1412*
- *Enterprise COBOL for z/OS V3R3 Language Reference, SC27-1408*

Example: A-6   Sample XML parser program - MYPARSER

```
000100 PROCESS CICS
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID.  MYPARSER
000400
000500 DATE-COMPILED.
000600 ENVIRONMENT DIVISION.
000700 CONFIGURATION SECTION.
000800 SPECIAL-NAMES.
000900 DATA DIVISION.
001000 WORKING-STORAGE SECTION.
001100 01 WS-START.
001200   03 FILLER                   PIC X(44)
001300       VALUE '*** MYPARSER WORKING STORAGE STARTS HERE ***'.
001400   03 RESP                     PIC S9(8) COMP-5 SYNC.
001500   03 RESP2                    PIC S9(8) COMP-5 SYNC.
001600   03 I                        PIC S9(4) COMP-5 SYNC.
```

```
001700   03 J                        PIC S9(4) COMP-5 SYNC.
001800   03 K                        PIC S9(4) COMP-5 SYNC.
001900   03 L                        PIC S9(4) COMP-5 SYNC.
002000   03 WORKAREA                 PIC X(24).
002100*********************************************************
002200* Data items for use by parser
002300*********************************************************
002400   03 IN-ELEM                  PIC X(1) VALUE 'N'.
002500   03 IN-REF-REQ               PIC X(1) VALUE 'N'.
002600   03 PARSE-ERROR              PIC X(1) VALUE 'N'.
002700
002800* The XML document to parse
002900   01 XML-DOCUMENT PIC X(1024).
003000
003100   01 USERNAME-XMLTAG-FOUND  PIC X VALUE 'N'.
003200
003300******************************************************************
003400* Externally referenced data areas
003500******************************************************************
003600 LINKAGE SECTION.
003700 01  DFHCOMMAREA.
003800   02  DFHCOMMAREA-IN          PIC X(1024).
003900   02  CA-RSP REDEFINES DFHCOMMAREA-IN.
004000     05   CA-USER-ID           PIC X(8).
004100     05   CA-PASSWORD          PIC X(8).
004200     05   FILLER               PIC X(1008).
004300
004400******************************************************************
004500* Main line code begins                                          *
004600******************************************************************
004700 PROCEDURE DIVISION.
004800 MAIN-PROCESSING SECTION.
004900***************************
005000*    Validate the commarea
005100***************************
005200     PERFORM INIT-AND-VALIDATE.
005300***********************************************************
005400* Received a valid COMMAREA so invoke the XML parser
005500* invoke the XML parser passing it the XML message
005600* What is the COBOL command used to invoke the XML parser?
005700***********************************************************
005800     XML PARSE XML-DOCUMENT
005900        PROCESSING PROCEDURE XML-HANDLER
006000     END-XML.
006100
006200     PERFORM RETURN-SOAP-LAB-RESPONSE.
006300
006400     EXEC CICS RETURN END-EXEC.
006500 MAIN-PROCESSING-END. EXIT.
```

```

*****************************************************************
* The following section is executed as a callback routine
* out of the XML PARSE statement above.
* What is the name of the register we need to check on the
* EVALUATE statement?
*****************************************************************
 XML-HANDLER SECTION.
     EVALUATE XML-EVENT
        WHEN 'START-OF-ELEMENT'
******************************************
* check if we have an element of interest
* i.e. Username and Password
******************************************
           IF XML-TEXT = 'wsse:Username'
              MOVE 'Y' TO IN-ELEM USERNAME-XMLTAG-FOUND
           ELSE
              if XML-TEXT = 'wsse:Password'
                 MOVE 'Y' TO IN-REF-REQ
              END-IF
           END-IF
        WHEN 'CONTENT-CHARACTERS'
************************************************
* If we are in an element we are interested in,
* then extract its value
************************************************
           IF IN-ELEM = 'Y'
              PERFORM EXTRACT-USER-ID
           ELSE
              IF IN-REF-REQ = 'Y'
                  PERFORM EXTRACT-PASSWORD
              END-IF
           END-IF
        WHEN 'END-OF-ELEMENT'
           CONTINUE
        WHEN 'START-OF-DOCUMENT'
           CONTINUE
        WHEN 'END-OF-DOCUMENT'
           CONTINUE
        WHEN 'VERSION-INFORMATION'
           CONTINUE
        WHEN 'ENCODING-DECLARATION'
           CONTINUE
        WHEN 'STANDALONE-DECLARATION'
           CONTINUE
        WHEN 'ATTRIBUTE-NAME'
           CONTINUE
        WHEN 'ATTRIBUTE-CHARACTERS'
           CONTINUE
        WHEN 'ATTRIBUTE-CHARACTER'
           CONTINUE
        WHEN 'START-OF-CDATA-SECTION'
           CONTINUE
        WHEN 'END-OF-CDATA-SECTION'
           CONTINUE
        WHEN 'CONTENT-CHARACTER'
           CONTINUE
        WHEN 'PROCESSING-INSTRUCTION-TARGET'
           CONTINUE
        WHEN 'PROCESSING-INSTRUCTION-DATA'
           CONTINUE
        WHEN 'COMMENT'
           CONTINUE
        WHEN 'EXCEPTION'
*****************************************************************
* Exception handling paragraph
* XML code 52 means that the code page being used does not match
* the one specified in the document. This is expected with non
* EBCDIC data, as it has been translated to EBCDIC since we
* received it.
*****************************************************************
           IF XML-CODE = 52 MOVE ZERO TO XML-CODE
           ELSE
***************************************
* An exception occurred during parsing
***************************************
             MOVE 'Y' TO PARSE-ERROR
           END-IF
        WHEN OTHER
           MOVE 'Y' TO PARSE-ERROR
     END-EVALUATE.
 XML-HANDLER-END. EXIT.
 XML01-SUBROUTINES SECTION.
****************************************************************
* The parser returns everything between the <xmltag> and
* </xmltag> tags, including all whitespace (SP, CR, LF, NL, HT)
* These are removed before saving the xmltag content.
****************************************************************
 EXTRACT-USER-ID.
     MOVE ZERO TO I J.
     MOVE LENGTH OF XML-TEXT TO L.
     IF L > LENGTH OF WORKAREA
            MOVE LENGTH OF WORKAREA TO L.
     MOVE XML-TEXT TO WORKAREA(1 : L).
     INSPECT WORKAREA(1 : L) CONVERTING X'0D251505' TO SPACES.
     INSPECT WORKAREA(1 : L) TALLYING I FOR LEADING SPACES.
     SUBTRACT I FROM L.
     INSPECT WORKAREA(I + 1 : L) TALLYING J FOR ALL SPACES.
     SUBTRACT J FROM L.
     IF L > LENGTH OF CA-USER-ID
            MOVE LENGTH OF CA-USER-ID TO L.
     MOVE WORKAREA(I + 1 : L) TO CA-USER-ID(1 : L).
     MOVE 'N' to IN-ELEM.

 EXTRACT-PASSWORD.
     MOVE ZERO TO I J.
     MOVE LENGTH OF XML-TEXT TO L.
     IF L > LENGTH OF WORKAREA
            MOVE LENGTH OF WORKAREA TO L.
     MOVE XML-TEXT TO WORKAREA(1 : L).
     INSPECT WORKAREA(1 : L) CONVERTING X'0D251505' TO SPACES.
     INSPECT WORKAREA(1 : L) TALLYING I FOR LEADING SPACES.
     SUBTRACT I FROM L.
     INSPECT WORKAREA(I + 1 : L) TALLYING J FOR ALL SPACES.
     SUBTRACT J FROM L.
     IF L > LENGTH OF CA-PASSWORD
            MOVE LENGTH OF CA-PASSWORD TO L.
     MOVE WORKAREA(I + 1 : L) TO CA-PASSWORD(1 : L).
     MOVE 'N' to IN-REF-REQ.

*******************
* Return to caller
*******************
 RETURN-RESPONSE.
     IF PARSE-ERROR = 'Y'
**********************************************
*       Bad news, got an XML parser Exception
**********************************************
        MOVE 'EXCEPTION' to CA-RSP
     ELSE
       IF USERNAME-XMLTAG-FOUND = 'N'
*******************************************************
*          Ignore, it was not for us...
*          Bad news, did not find the expected XML TAG
*******************************************************
           MOVE 'NOT FOUND' to CA-RSP
       END-IF
     END-IF.

**************************************************
* Initialise and Validate the COMMAREA
**************************************************
 INIT-AND-VALIDATE.
***********************
*    Acceptable size ?
***********************
     IF EIBCALEN > 150
     THEN
***********
*      yes
***********
       MOVE DFHCOMMAREA-IN(1:EIBCALEN) to XML-DOCUMENT
       MOVE SPACES TO CA-USER-ID, CA-PASSWORD
     ELSE
**********
*      no
**********
       INITIALIZE DFHCOMMAREA(1:EIBCALEN)
       MOVE 'BAD COMMAREA LENGTH !' to DFHCOMMAREA
       EXEC CICS RETURN END-EXEC
     END-IF.
```

A.5 Sample header processing program - CIWSSECR

The CIWSSECR program in Example A-7 is used to insert a `<wsse:Security>` header into a SOAP message.

It determines, by obtaining the data from DFHFUNCTION, if is it has been invoked in a SEND-REQUEST phase. It also determines if it has been invoked for the correct destination, from the data in DFHWS-URI. It then obtains the user ID that the transaction is currently executing under and inserts this into the WS-Security header. This header is then written into the DFHHEADER container and control is returned to the calling program.

Example: A-7 Sample header processing program - CIWSSECR

```
******************************************************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CIWSSECR
******************************************************************
*                                                                *
* This program:-                                                 *
* 1. Checks for SEND-REQUEST function, else returns              *
* 2. Checks for correct URI from DFHWS-URI                       *
* 3. Obtains user ID from DFHWS-USERID container                 *
* 4. Puts completed WS-Security header into the                  *
*    DFHHEADER container                                         *
******************************************************************

 AUTHOR.      CHIEREGATTI.
 DATE-COMPILED.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
```

```
SPECIAL-NAMES.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-START.
  03 FILLER                   PIC X(44)
      VALUE '*** CIWSSECR WORKING STORAGE STARTS HERE ***'.
  03 NN                       PIC X(10) VALUE 'CIWSSECR '.
  03 WS-RESP                  PIC S9(8) COMP-5 SYNC.
  03 WS-RESP2                 PIC S9(8) COMP-5 SYNC.
  03 WS-HEAD-PTR              USAGE IS POINTER.
  03 WS-HEAD-LEN              PIC S9(8) COMP-4.
  03 WS-URI-PTR               USAGE IS POINTER.
  03 WS-URI-LEN               PIC S9(8) COMP-4.
  03 WS-FUNC-LEN              PIC S9(8) COMP-4 VALUE 16.
  03 WS-PIPE-LEN              PIC S9(8) COMP-4 VALUE 8.
  03 WS-GETMAIN-PTR           USAGE IS POINTER.
  03 WS-GETMAIN-LEN           PIC S9(8) COMP-4.
  03 WS-FAULT-STRING          PIC X(40) value spaces.
  03 WS-FAULT-CODE            PIC S9(8) COMP-4.
  03 WS-SOAP-LEVEL            PIC S9(8) COMP-4.
    88 WS-SOAP-11  VALUE 1.
    88 WS-SOAP-12  VALUE 2.
    88 WS-NOT-SOAP VALUE 10.
  03 WS-FUNC-AREA             PIC X(16).
  03 WS-PIPE-AREA             PIC X(8).
  03 WS-NOT-AUTH              PIC X(40)
              VALUE 'Not authorized to place order.'.
  03 WS-AUTH-FAILED           PIC X(40)
              VALUE 'Authorization failed for order request.'.
  03 WS-WSSE-HEADER.
     05 WS-WSSE-01 PIC X(15) VALUE '<wsse:Security '.
     05 WS-WSSE-02 PIC X(26) VALUE 'soapenv:mustUnderstand="1"'.
     05 WS-WSSE-03 PIC X(26) VALUE ' xmlns:wsse="http://docs.o'.
     05 WS-WSSE-04 PIC X(26) VALUE 'asis-open.org/wss/2004/01/'.
     05 WS-WSSE-05 PIC X(26) VALUE 'oasis-200401-wss-wssecurit'.
     05 WS-WSSE-06 PIC X(18) VALUE 'y-secext-1.0.xsd">'.
     05 WS-WSSE-07 PIC X(20) VALUE '<wsse:UsernameToken>'.
     05 WS-WSSE-08 PIC X(15) VALUE '<wsse:Username>'.
     05 WS-WSSE-09 PIC X(08) VALUE 'WEBASZ  '.
     05 WS-WSSE-10 PIC X(16) VALUE '</wsse:Username>'.
     05 WS-WSSE-11 PIC X(21) VALUE '</wsse:UsernameToken>'.
     05 WS-WSSE-12 PIC X(16) VALUE '</wsse:Security>'.

01 MYPARSER-COMLEN            PIC S9(4) COMP-4.
01 WS-PSW.
     05 WS-WSSE-A PIC X(30)
                        VALUE '<wsse:Password Type="http://do'.
     05 WS-WSSE-B PIC X(30)
                        VALUE 'cs.oasis-open.org/wss/2004/01/'.
      05 WS-WSSE-C PIC X(30)
                         VALUE 'oasis-200401-wss-username-toke'.
      05 WS-WSSE-D PIC X(30)
                         VALUE 'n-profile-1.0#PasswordText">'.
      05 WS-WSSE-E PIC X(8) VALUE 'REDBOOKS'.
      05 WS-WSSE-F PIC X(16)  VALUE '</wsse:Password>'.
 01 CA-PARSER-RSP.
    03 CA-USER-ID             PIC X(8).
    03 CA-PASSWORD            PIC X(8).

 LINKAGE SECTION.
 01 WS-HEAD-AREA.
    02 FILLER    PIC X OCCURS 1024 DEPENDING ON WS-HEAD-LEN.

 01 WS-URI-AREA.
    02 FILLER    PIC X OCCURS 256 DEPENDING ON WS-URI-LEN.

 01 WS-GETMAIN-AREA.
    02 FILLER    PIC X OCCURS 1024 DEPENDING ON WS-GETMAIN-LEN.

******************************************************************
* Main line code begins                                          *
******************************************************************
 PROCEDURE DIVISION.
 MAIN-PROCESSING SECTION.

     PERFORM GET-PIPELINE.

     PERFORM GET-DFHFUNCTION.

     PERFORM GET-DFHWS-URI.

     EVALUATE WS-URI-AREA(1:22)
       WHEN 'jms:/queue?destination'
         IF WS-FUNC-AREA = 'SEND-REQUEST '
            PERFORM GET-USERID
            PERFORM GET-SOAP-HEADER
            PERFORM PUT-SOAP-HEADER
         END-IF
       WHEN OTHER
         CONTINUE
     END-EVALUATE.

     EXEC CICS RETURN END-EXEC.

 MAIN-PROCESSING-END. EXIT.

****************************************************
*     ****************************************     *
*              SUBROUTINES FOLLOW                   *
*     ****************************************      *
*****************************************************

******************************************************************
* Retrieve the PIPELINE name ftom DFHWS-PIPELINE container
******************************************************************
 GET-PIPELINE.
     EXEC CICS
        GET CONTAINER('DFHWS-PIPELINE')
        INTO(WS-PIPE-AREA)
        FLENGTH(WS-PIPE-LEN)
        RESP(WS-RESP)
     END-EXEC.


******************************************************************
* Retrieve the function type ftom DFHFUNCTION container
******************************************************************
 GET-DFHFUNCTION.
     EXEC CICS
        GET CONTAINER('DFHFUNCTION')
        INTO(WS-FUNC-AREA)
        FLENGTH(WS-FUNC-LEN)
        RESP(WS-RESP)
     END-EXEC.

**************************************************
* Check for correct length. Create SOAP fault if
* not correct. If for incoming request then we
* do the work, else we exit immediately.
**************************************************
     IF WS-FUNC-LEN NOT = 16
        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
        PERFORM FAULT-MESSAGE
     END-IF.

******************************************************************
* Retrieve the URI from the DFHWS-URI container
******************************************************************
 GET-DFHWS-URI.
     EXEC CICS
        GET CONTAINER('DFHWS-URI')
        SET(WS-URI-PTR)
        FLENGTH(WS-URI-LEN)
        RESP(WS-RESP)
     END-EXEC.

*******************************************
```

```
* Copy the input container to our storage
******************************************
     IF WS-URI-LEN > 0
        SET ADDRESS OF WS-URI-AREA TO WS-URI-PTR
        MOVE WS-URI-LEN TO WS-GETMAIN-LEN

        EXEC CICS GETMAIN
            SET(WS-GETMAIN-PTR)
            FLENGTH(WS-GETMAIN-LEN)
        END-EXEC

        SET ADDRESS OF WS-GETMAIN-AREA TO WS-GETMAIN-PTR
        MOVE WS-URI-AREA TO WS-GETMAIN-AREA
        SET WS-URI-PTR TO WS-GETMAIN-PTR
        SET ADDRESS OF WS-URI-AREA TO WS-URI-PTR
     ELSE
        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
        PERFORM FAULT-MESSAGE
     END-IF.

**************************************
* Retrieve the <wsse:Security> header
**************************************
 GET-SOAP-HEADER.
     EXEC CICS
        GET CONTAINER('DFHHEADER')
        SET(WS-HEAD-PTR)
        FLENGTH(WS-HEAD-LEN)
        RESP(WS-RESP)
     END-EXEC.

**********************
* Retrieve the user ID
**********************
 GET-USERID.
     EXEC CICS
        GET CONTAINER('DFHWS-USERID')
        INTO(WS-WSSE-09)
        FLENGTH(LENGTH OF WS-WSSE-09)
        RESP(WS-RESP)
     END-EXEC.

***********************************
* Write the <wsse:Security> header
***********************************
 PUT-SOAP-HEADER.
     EXEC CICS
        PUT CONTAINER('DFHHEADER')
        FROM(WS-WSSE-HEADER)
        FLENGTH(LENGTH OF WS-WSSE-HEADER)
        RESP(WS-RESP)
     END-EXEC.
     DISPLAY NN '>=================================<'
     DISPLAY NN 'Container Name: : DFHHEADER        '.
     DISPLAY NN 'Content length: '    LENGTH OF WS-WSSE-HEADER.
     DISPLAY NN 'Container content: ' WS-WSSE-HEADER.
     DISPLAY NN '----------------------------------'.
**

*****************************************************************
* The user ID has been verified. We will now set the contents of
* DFHWS-USERID container with this value.This will cause the
* business logic (back-end program/s) to be executed with this
* user ID, using it's access rights.
*****************************************************************
 SET-USER-ID.
     EXEC CICS
         PUT CONTAINER('DFHWS-USERID')
         FROM(CA-USER-ID)
         FLENGTH(length of CA-USER-ID)
         DATATYPE(DFHVALUE(CHAR))
         RESP(WS-RESP)
     END-EXEC.

*****************************************************************
* We detected that the ca_request_id field specifies an invalid
* request. This is a CLIENT error.
*****************************************************************
 FAULT-MESSAGE SECTION.
************************
* Generate a SOAP Fault
************************
     EXEC CICS
         GET CONTAINER('DFHWS-SOAPLEVEL')
         INTO(WS-SOAP-LEVEL)
         FLENGTH(WS-HEAD-LEN)
         RESP(WS-RESP)
     END-EXEC.

***********************************************
* Perform the XML PARSE and verify the user ID
***********************************************

 PARSE-AND-VERIFY.
************************************************************
*  The SOAP body XML data can now be parsed by MYPARSER
*    LINK to the XML parser, it will return the CA-USER-ID
*                                           and CA-PASSWORD
*                                           or  'EXCEPTION'
*                                           or  'NOT FOUND'
*                                           or  'BAD COMMA'
************************************************************
     EXEC CICS LINK PROGRAM('MYPARSER')
                    COMMAREA(WS-HEAD-AREA)
                    LENGTH(MYPARSER-COMLEN)
     END-EXEC.
     IF WS-HEAD-AREA(1:9) = 'EXCEPTION' or
                          'NOT FOUND' or
                          'BAD COMMA'
******************************************************
*  Error found during the XML PARSE program execution
******************************************************
         MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
         PERFORM FAULT-MESSAGE
     ELSE
         MOVE WS-HEAD-AREA to CA-PARSER-RSP
************************************
*  OK, data found, so do the VERIFY
************************************
         EXEC CICS VERIFY
            PASSWORD(CA-PASSWORD)
            USERID(CA-USER-ID)
            RESP(WS-RESP)
         END-EXEC
***************
*  Succesful ?
***************
         IF WS-RESP = DFHRESP(NORMAL)
            PERFORM SET-USER-ID
         ELSE
            MOVE WS-NOT-AUTH TO WS-FAULT-STRING
            PERFORM FAULT-MESSAGE
         END-IF
     END-IF.
************************************
* MOVE CORRECT VERSION OF FAULTCODE
************************************
     IF WS-SOAP-11 MOVE DFHVALUE(CLIENT) TO WS-FAULT-CODE
     ELSE
        MOVE DFHVALUE(SENDER) TO WS-FAULT-CODE
     END-IF

     EXEC CICS SOAPFAULT CREATE
               FAULTSTRING(WS-FAULT-STRING)
               FAULTSTRLEN(LENGTH OF WS-FAULT-STRING)
               FAULTCODE(WS-FAULT-CODE)
     END-EXEC.
```

```
031200 FAULT-MESSAGE-END. EXIT.
```

A.6 Sample header processing program - CIWSSECS

Program CIWSSECS in Example A-8 is similar to CIWSSECH (see Appendix A.2, “Sample header processing program - CIWSSECH” on page 533) except that no password verification is done.

Example: A-8 Sample header processing program - CIWSSECS

```
000100  PROCESS CICS
000200 IDENTIFICATION DIVISION.
000300 PROGRAM-ID.  CIWSSECS
000400*****************************************************************
000500*                                                               *
000600* This program:-                                                *
000700* 1. Checks for RECEIVE-REQUEST function, else returns          *
000800* 2. Obtains data from DFHHEADER container                      *
000900* 2. Uses the MYPARSER program to extract the CA-USER-ID        *
001100*    field from the XML message contained within DFHHEADER     *
001200* 2. Puts CA-USER-ID into DFHWS-USERID container                *
001600*****************************************************************
001700
001800 AUTHOR.      Grant Ward Able.
001900 DATE-COMPILED.
002000 ENVIRONMENT DIVISION.
002100 CONFIGURATION SECTION.
002200 SPECIAL-NAMES.
002300 DATA DIVISION.
002400 WORKING-STORAGE SECTION.
002500 01 WS-START.
002600   03 FILLER                    PIC X(44)
002700       VALUE '*** CIWSSECS WORKING STORAGE STARTS HERE ***'.
002800   03 WS-RESP                   PIC S9(8) COMP-5 SYNC.
002900   03 WS-RESP2                  PIC S9(8) COMP-5 SYNC.
003000   03 WS-HEAD-PTR               USAGE IS POINTER.
003100   03 WS-HEAD-LEN               PIC S9(8) COMP-4.
003200   03 WS-URI-PTR                USAGE IS POINTER.
003300   03 WS-URI-LEN                PIC S9(8) COMP-4.
003400   03 WS-FUNC-LEN               PIC S9(8) COMP-4 VALUE 16.
003500   03 WS-GETMAIN-PTR            USAGE IS POINTER.
003600   03 WS-GETMAIN-LEN            PIC S9(8) COMP-4.
003700   03 WS-FAULT-STRING           PIC X(40) value spaces.
003800   03 WS-FAULT-CODE             PIC S9(8) COMP-4.
003900   03 WS-SOAP-LEVEL             PIC S9(8) COMP-4.
004000     88 WS-SOAP-11  VALUE 1.
004100     88 WS-SOAP-12  VALUE 2.
```

```
     88 WS-NOT-SOAP VALUE 10.
   03 WS-FUNC-AREA             PIC X(16).
   03 WS-NOT-AUTH              PIC X(40)
               VALUE 'Not authorized to place order.'.
   03 WS-AUTH-FAILED           PIC X(40)
               VALUE 'Authorization failed for order request.'.

 01 MYPARSER-COMLEN            PIC S9(4) COMP-4.
 01 CA-PARSER-RSP.
    03 CA-USER-ID             PIC X(8).
    03 CA-PASSWORD            PIC X(8).

 LINKAGE SECTION.
 01 WS-HEAD-AREA.
    02 FILLER    PIC X OCCURS 1024 DEPENDING ON WS-HEAD-LEN.

 01 WS-URI-AREA.
    02 FILLER    PIC X OCCURS 256 DEPENDING ON WS-URI-LEN.

 01 WS-GETMAIN-AREA.
    02 FILLER    PIC X OCCURS 1024 DEPENDING ON WS-GETMAIN-LEN.

******************************************************************
* Main line code begins                                          *
******************************************************************
 PROCEDURE DIVISION.
 MAIN-PROCESSING SECTION.

***********************************
*  Receive the SOAP Body namespace
***********************************
     PERFORM GET-DFHFUNCTION.

***********************************
*  Receive the SOAP Body namespace
***********************************
     PERFORM GET-DFHWS-URI.
     IF WS-URI-AREA(1:4) NOT = 'wmq:'
        EXEC CICS RETURN END-EXEC
     END-IF.

***********************************
*  Receive the SOAP Body namespace
***********************************
     PERFORM GET-SOAP-HEADER.

************************************************************
*  The SOAP body XML data can now be parsed by MYPARSER
*    LINK to the XML parser, it will return the CA-USER-ID
```

```
009300*                                         or  'EXCEPTION'
009400*                                         or  'NOT FOUND'
009500*                                         or  'BAD COMMA'
009600*****************************************************************
009700     MOVE SPACES TO CA-PARSER-RSP
009700     EXEC CICS LINK PROGRAM('MYPARSER')
009800                    COMMAREA(WS-HEAD-AREA)
009900                    LENGTH(MYPARSER-COMLEN)
010000     END-EXEC.
010100     IF WS-HEAD-AREA(1:9) = 'EXCEPTION' or
010200                         'NOT FOUND' or
010300                         'BAD COMMA'
010400***********************************************************
010500*  Error found during the XML PARSE program execution
010600***********************************************************
010700         MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
010800         PERFORM FAULT-MESSAGE
010900     ELSE
011100***************************************
011200*  OK, data found, so set the user ID
011300***************************************
011000         MOVE WS-HEAD-AREA to CA-PARSER-RSP
012300         PERFORM SET-USER-ID
012800     END-IF.
012900******************************
013000*  and it is the end, bye bye
013100******************************
013200     EXEC CICS RETURN END-EXEC.
013300
013400 MAIN-PROCESSING-END. EXIT.
013500
013600*****************************************************
013700*     ****************************************     *
013800*              SUBROUTINES FOLLOW                  *
013900*     ****************************************     *
014000*****************************************************
014100
014200*****************************************************************
014300* Retrieve the function type from DFHFUNCTION container
014400*****************************************************************
014500 GET-DFHFUNCTION.
014600     EXEC CICS
014700        GET CONTAINER('DFHFUNCTION')
014800        INTO(WS-FUNC-AREA)
014900        FLENGTH(WS-FUNC-LEN)
015000        RESP(WS-RESP)
015100     END-EXEC.
015200
015300**************************************************
```

```
* Check for correct length. Create SOAP fault if
* not correct. If for incoming request then we
* do the work, else we exit immediately.
*************************************************
     IF WS-FUNC-LEN NOT = 16
        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
        PERFORM FAULT-MESSAGE
     ELSE
        IF WS-FUNC-AREA NOT = 'RECEIVE-REQUEST '
           EXEC CICS RETURN END-EXEC
        END-IF
     END-IF.

*****************************************************************
* Retrieve the URI from the DFHWS-URI container
*****************************************************************
 GET-DFHWS-URI.
     EXEC CICS
        GET CONTAINER('DFHWS-URI')
        SET(WS-URI-PTR)
        FLENGTH(WS-URI-LEN)
        RESP(WS-RESP)
     END-EXEC.

******************************************
* Copy the input container to our storage
******************************************
     IF WS-URI-LEN > 0
        SET ADDRESS OF WS-URI-AREA TO WS-URI-PTR
        MOVE WS-URI-LEN TO WS-GETMAIN-LEN

        EXEC CICS GETMAIN
            SET(WS-GETMAIN-PTR)
            FLENGTH(WS-GETMAIN-LEN)
        END-EXEC

        SET ADDRESS OF WS-GETMAIN-AREA TO WS-GETMAIN-PTR
        MOVE WS-URI-AREA TO WS-GETMAIN-AREA
        SET WS-URI-PTR TO WS-GETMAIN-PTR
        SET ADDRESS OF WS-URI-AREA TO WS-URI-PTR
     ELSE
        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
        PERFORM FAULT-MESSAGE
        EXEC CICS RETURN END-EXEC
     END-IF.

**************************************
* Retrieve the <wsse:Security> header
**************************************
```

```
 GET-SOAP-HEADER.
     EXEC CICS
        GET CONTAINER('DFHHEADER')
        SET(WS-HEAD-PTR)
        FLENGTH(WS-HEAD-LEN)
        RESP(WS-RESP)
     END-EXEC.

*****************************************
* Copy the input container to our storage
*****************************************
     IF WS-HEAD-LEN > 0
        SET ADDRESS OF WS-HEAD-AREA TO WS-HEAD-PTR
        MOVE WS-HEAD-LEN TO WS-GETMAIN-LEN MYPARSER-COMLEN

        EXEC CICS GETMAIN
            SET(WS-GETMAIN-PTR)
            FLENGTH(WS-GETMAIN-LEN)
        END-EXEC

        SET ADDRESS OF WS-GETMAIN-AREA TO WS-GETMAIN-PTR
        MOVE WS-HEAD-AREA TO WS-GETMAIN-AREA
        SET WS-HEAD-PTR TO WS-GETMAIN-PTR
        SET ADDRESS OF WS-HEAD-AREA TO WS-HEAD-PTR
     ELSE
        MOVE WS-AUTH-FAILED TO WS-FAULT-STRING
        PERFORM FAULT-MESSAGE
        EXEC CICS RETURN END-EXEC
     END-IF.

****************************************************************
* The user ID has been verified. We will now set the contents of
* DFHWS-USERID container with this value.This will cause the
* business logic (back-end program/s) to be executed with this
* user ID, using it's access rights.
****************************************************************
 SET-USER-ID.
     EXEC CICS
         PUT CONTAINER('DFHWS-USERID')
         FROM(CA-USER-ID)
         FLENGTH(length of CA-USER-ID)
         DATATYPE(DFHVALUE(CHAR))
         RESP(WS-RESP)
     END-EXEC.

****************************************************************
* We detected that the ca_request_id field specifies an invalid
* request. This is a CLIENT error.
****************************************************************
```

```
 FAULT-MESSAGE SECTION.
************************
* Generate a SOAP Fault
************************
     EXEC CICS
         GET CONTAINER('DFHWS-SOAPLEVEL')
         INTO(WS-SOAP-LEVEL)
         FLENGTH(WS-HEAD-LEN)
         RESP(WS-RESP)
     END-EXEC.

************************************
* MOVE CORRECT VERSION OF FAULTCODE
************************************
     IF WS-SOAP-11 MOVE DFHVALUE(CLIENT) TO WS-FAULT-CODE
     ELSE
        MOVE DFHVALUE(SENDER) TO WS-FAULT-CODE
     END-IF

     EXEC CICS SOAPFAULT CREATE
               FAULTSTRING(WS-FAULT-STRING)
               FAULTSTRLEN(LENGTH OF WS-FAULT-STRING)
               FAULTCODE(WS-FAULT-CODE)
     END-EXEC.
 FAULT-MESSAGE-END. EXIT.
```

A.7 Sample header processing program - WSATHND

The WSATHND C program shown in Example A-9 is used to monitor the exchange of registration and protocol service messages between WebSphere Application Server and CICS. It writes messages to the CESO transient data queue.

Example: A-9 Sample header processing program - WSATHND

```
/********************************************************************/
/*                                                                  */
/* This program will perform the following functions when used as  */
/* a  message handler in either the DFHWSATP pipeline or the        */
/* DFHWSATR pipeline:                                               */
/*                                                                  */
/* 1) If MESSAGES_ON is set to 1, it will write the following       */
/*    message:                                                      */
/*        WSAT: REACHED HANDLER - function                          */
/*    where function is the contents of the DFHFUNCTION container   */
/*    (RECEIVE-REQUEST,SEND-RESPONSE,SEND-REQUEST,RECEIVE-RESPONSE, */
/*    PROCESS-REQUEST,NO-RESPONSE,or HANDLER-ERROR). Recall that    */
/*    when the function is NO-RESPONSE, then the handler is being   */
/*    invoked after processing a request, when there is no response */
/*    to be processed.                                              */
/*                                                                  */
/* 2) If the function is not NO-RESPONSE,then:                      */
/*    a) if FULL_MESSAGES_ON is set to 1, it will write the         */
/*       following message:                                         */
/*          WSAT: contents of the DFHREQUEST container              */
/*                                                                  */
/*    b) if MESSAGES_ON is set to 1, it will write the following    */
/*       message:                                                   */
/*          WSAT: ACTION: action                                    */
/*       where action is the character string aaaaaa which comes    */
/*       at the end of the following character string found in the */
/*       DFHREQUEST container:                                      */
/*          http://schemas.xmlsoap.org/ws/2004/10/wscoor/aaaaaa     */
/*       and has one of the following values: RegisterResponse,     */
/*       Prepared,Committed, Register, Prepare, Commit, ReadOnly,   */
/*       Aborted, Abort, Rollback                                   */
/*                                                                  */
/* The messages are written to the CESO transient data queue which  */
/* is normally an extrapartition transient data queue with DDname   */
/* CEEOUT.                                                          */
/*                                                                  */
/********************************************************************/

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

/******************************************************************/
/* The following flags can be set to either 0 or 1.               */
/******************************************************************/
#define MESSAGES_ON       1
#define FULL_MESSAGES_ON   1

void check_response(char* identifier);

int main(char* argv[])
{
   /*****************/
   /*** VARIABLES ***/
   /*****************/

   /* Pointer to COMMAREA */
   char* commptr;

   /* Container names */
   char* cont_function = "DFHFUNCTION     \0";
```

```
char* cont_request  = "DFHREQUEST      \0";

/* Contents and length of DFHFUNCTION container */
char* function;
long int functionLen;

/* Contents and length of DFHREQUEST container */
char* message;
long int messageLen;

/* log message */
char* entryLogMsg;

char action[] = "                \0";

char* state1 =
     "http://schemas.xmlsoap.org/ws/2004/10/wscoor/RegisterResponse";
char* state2 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/Prepared";
char* state3 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/Committed";
char* state4 = "http://schemas.xmlsoap.org/ws/2004/10/wscoor/Register";
char* state5 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/Prepare";
char* state6 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/Commit";
char* state7 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/ReadOnly";
char* state8 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/Aborted";
char* state9 = "http://schemas.xmlsoap.org/ws/2004/10/wsat/Abort";
char* state10= "http://schemas.xmlsoap.org/ws/2004/10/wsat/Rollback";

/***************/
/*** PROGRAM ***/
/***************/

EXEC CICS ADDRESS EIB(dfheiptr) COMMAREA(commptr);

/*****************************************************************/
/* Store contents of DFHFUNCTION container in function variable */
/*****************************************************************/
EXEC CICS GET CONTAINER(cont_function) NODATA
              FLENGTH(functionLen);
check_response("WSATHND.getContainer.dfhfunctionLen");

function = (char*) malloc(sizeof(char*) * (1 + functionLen));

EXEC CICS GET CONTAINER(cont_function)
              INTO(function) FLENGTH(functionLen);
check_response("WSATHND.getContainer.dfhfunction");

*(function + functionLen) = '\0';

/*****************************************************/
```

```
/* Display message: WSAT: REACHED HANDLER - function  */
/*****************************************************/

entryLogMsg = (char*) malloc(sizeof(char*) *
                              (strlen("WSAT: REACHED HANDLER - ") +
                               strlen(function) + 1
                              )
                             );
strcpy(entryLogMsg, "WSAT: REACHED HANDLER - ");
strcat(entryLogMsg, function);
if (MESSAGES_ON == 1)
  printf("%s\n", entryLogMsg);

free(entryLogMsg);

/*************************************************************/
/* Store contents of DFHREQUEST container in message variable */
/*************************************************************/

if (!(strncmp(function, "NO-RESPONSE     ", 16) == 0))
{
    EXEC CICS GET CONTAINER(cont_request) NODATA
                  FLENGTH(messageLen);
    check_response("WSATHND.get.requestLen");

    message = (char*) malloc(sizeof(char*) * (messageLen+1));

    EXEC CICS GET CONTAINER(cont_request)
                  INTO(message) FLENGTH(messageLen);
    check_response("WSATHND.get.request");

    *(message+messageLen) = '\0';

    if (FULL_MESSAGES_ON == 1)
    {
      printf("WSAT:%s:\n", message);
    }

    /****************/
    /* ACTION STATE */
    /****************/


    if (strstr(message, state1) != NULL)
        strcpy(action, "RegisterResponse\0");
    else if (strstr(message, state2) != NULL)
        strcpy(action, "Prepared        \0");
    else if (strstr(message, state3) != NULL)
        strcpy(action, "Committed       \0");
        else if (strstr(message, state4) != NULL)
            strcpy(action, "Register        \0");
        else if (strstr(message, state5) != NULL)
            strcpy(action, "Prepare         \0");
        else if (strstr(message, state6) != NULL)
            strcpy(action, "Commit          \0");
        else if (strstr(message, state7) != NULL)
            strcpy(action, "ReadOnly      \0");
        else if (strstr(message, state8) != NULL)
            strcpy(action, "Aborted       \0");
        else if (strstr(message, state9) != NULL)
            strcpy(action, "Abort         \0");
        else if (strstr(message, state10) != NULL)
            strcpy(action, "Rollback      \0");

        if (MESSAGES_ON == 1)
        printf("WSAT: ACTION:%s:\n", action);

        free(message);
    }

    EXEC CICS RETURN;
    check_response("WSATHND.return");

    return 0;
}


/* Check response of CICS commands - ensure no errors occur */
void check_response(char* identifier)
{
    if ((dfheiptr->eibresp + dfheiptr->eibresp2) != 0)
    {
        char* errStart = "ERROR: ";
        char* errEnd   = ". See stdout for respcodes";
        char* fullMsg;

        if (dfheiptr->eibresp == DFHRESP(NORMAL))
           printf("NORMAL Response found");
        if (dfheiptr->eibresp == DFHRESP(CCSIDERR))
           printf("CCSIDERR Response found");
        if (dfheiptr->eibresp == DFHRESP(CHANNELERR))
           printf("CHANNELERR Response found");
        if (dfheiptr->eibresp == DFHRESP(CONTAINERERR))
           printf("CONTAINERERR Response found");
        if (dfheiptr->eibresp == DFHRESP(INVREQ))
           printf("INVREQ Response found");
        if (dfheiptr->eibresp == DFHRESP(LENGERR))
           printf("LENGERR Response found");
        printf("Error: %s\n", identifier);
        printf("Error resp1: %d\n", dfheiptr->eibresp);
        printf("Error resp2: %d\n", dfheiptr->eibresp2);

        fullMsg = (char*) malloc((strlen(errStart) +
                                  strlen(identifier) +
                                  strlen(errEnd) + 1) *
                                  sizeof(char*));

        strcpy(fullMsg, errStart);
        strcat(fullMsg, identifier);
        strcat(fullMsg, errEnd);

        EXEC CICS SEND TEXT FROM(fullMsg)
                       LENGTH(strlen(fullMsg));

        free(fullMsg);

        exit;
    }
}
```

B

How the DES, AES, SHA-1, and HMAC algorithms work

Some readers may want or need a more detailed explanation of one or more of the following:

- How the DES algorithm works
- How the AES algorithm works
- How the SHA-1 algorithm works
- How the HMAC algorithm of FIPS PUB 198 works

In this appendix we attempt to provide an overview of this information that does not require an understanding of advanced mathematics.

B.1 How DES works

DES makes some use of permutations. A permutation of a set of elements is an arrangement of the elements of the set in some order. For example, there are six permutations of the set {1,2,3}: 123, 132, 213, 231, 312, and 321. The permutation 123 and the permutation 132 are considered to be different permutations because the order of the elements is different.

A sketch of the DES enciphering computation is given in Figure B-1. The 64 bits of the input block to be enciphered are first subjected to a permutation. In DES the permuted input has bit 58 of the input as its first bit, bit 50 of the input as its second bit, and so on as specified in FIPS Pub 46-3 with bit 7 of the input as its last bit. Figure B-1 shows the leftmost 32 bits of the permuted input as L_0, and the rightmost 32 bits of the permuted input as R_0. The permuted input block is then the input to a complex key-dependent computation described in "The cipher function f" on page 575. The output of that computation, called the preoutput, is then subjected to a permutation that is the inverse of the initial permutation.

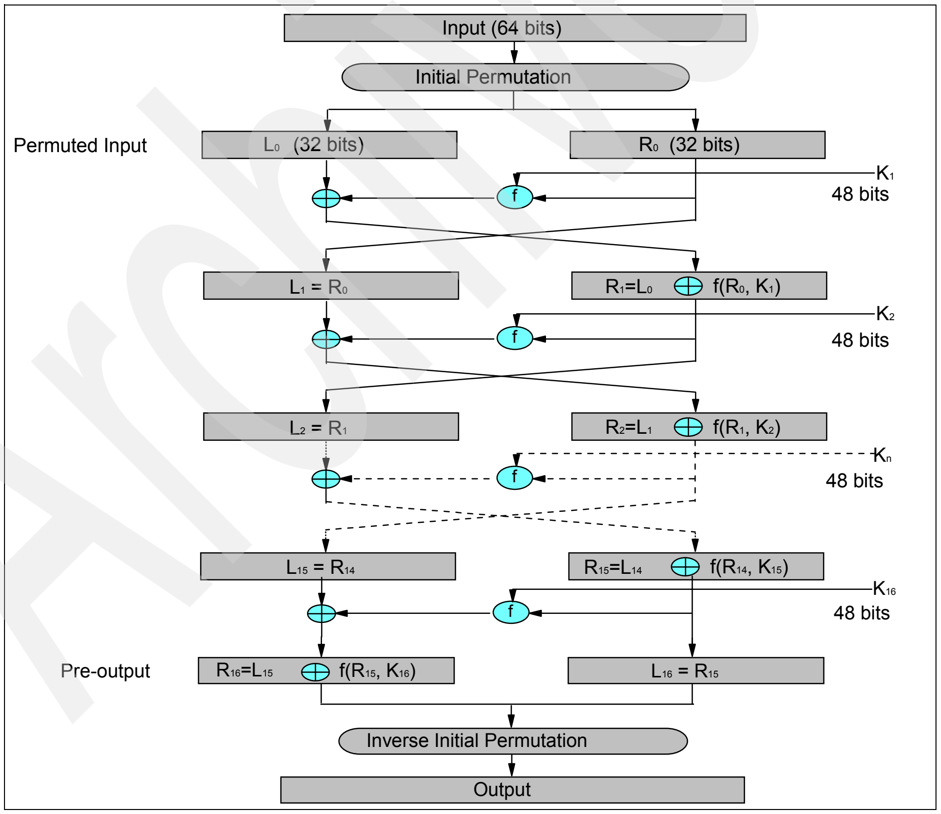


Figure B-1 The DES algorithm

The computation that uses the permuted input block as its input to produce the preoutput block consists, but for a final interchange of blocks, of 16 iterations of the calculation:

$$L_n = R_{n-1}$$

$$R_n = L_{n-1} \oplus f(R_{n-1}, K_n)$$

R_{n-1} represents the rightmost 32 bits of the output from the previous iteration. K_n represents a permuted selection of 48 bits chosen from the 64-bit key by the key schedule algorithm described in "Key schedule algorithm" on page 576. Thus the cipher function f operates on two blocks, one of 32 bits and one of 48 bits; it produces a block of 32 bits, which is then XORed with L_{n-1} to produce R_n. (XOR is a bit-by-bit exclusive OR operation in which $1 \oplus 1 = 0, 1 \oplus 0 = 1, 0 \oplus 1 = 1, 0 \oplus 0 = 0$. XOR is equivalent to a bit-by-bit addition modulo 2.) The preoutput block is then $R_{16}L_{16}$.

The cipher function f

A sketch of the calculation of f(R,K) is given in Figure B-2.

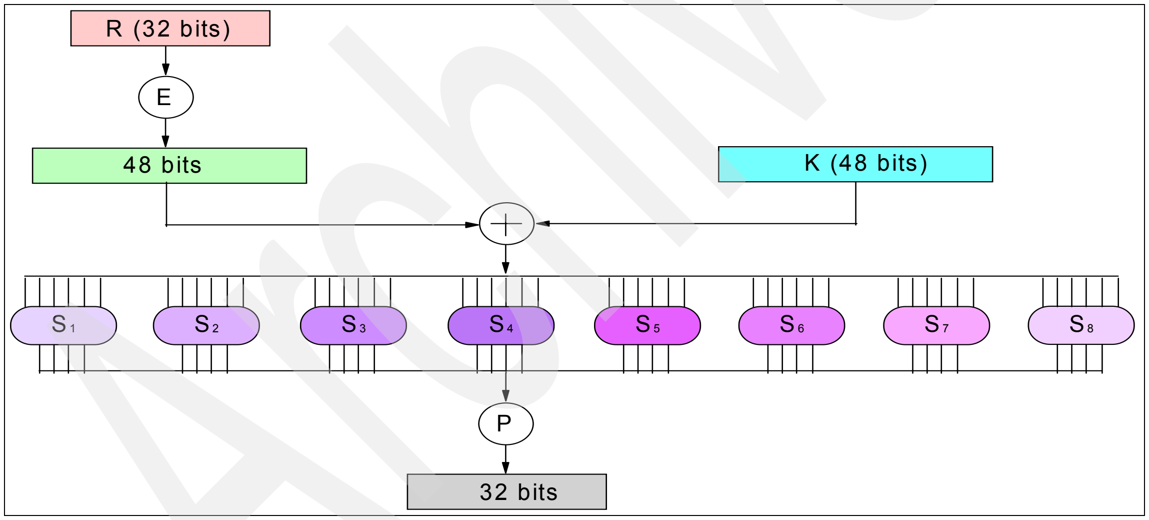


Figure B-2 The cipher function f of the DES algorithm

E denotes a function that takes a block of 32 bits as input and produces a block of 48 bits as output. The first three bits of E(R) are the bits in positions 32, 1, and 2 of R, the last 2 bits of E(R) are the bits in position 32 and 1 of R, and the remaining bits of E(R) are chosen from the bits of R according to a table specified in FIPS Pub 46-3.

Each of the unique selection functions S_1, S_2,...,S_8 takes a 6-bit block of input and yields a 4-bit block as output as specified in FIPS Pub 46-3.

The 8 blocks of 4 bits each are consolidated into a single block of 32 bits, which forms the input to the permutation function P. P yields a 32-bit output from a 32-bit input by permuting the bits of the input block, again as specified in FIPS Pub 46-3. The output is then the output of the cipher function f for the inputs R and K.

Key schedule algorithm

Figure B-3 shows a sketch of the key schedule calculation of the DES algorithm. The bits of the key are numbered 1 through 64. The bits of C_0 are respectively bits 57,49, 41, 33, 25,..., 44, and 36 of the key. The bits of D_0 are respectively bits 63, 55, 47, 39, 31,..., 12, and 4 of the key. Permuted Choice 1 does not select any of the eight parity bits of the key for either C_0 or D_0.

C_1 and D_1 are obtained from C_0 and D_0 respectively by one left shift. C_2 and D_2 are obtained from C_1 and D_1 respectively by one left shift. C_9 and D_9 are obtained from C_8 and D_8 by one left shift, and C_{16} and D_{16} are obtained from C_{15} and D_{15} by one left shift. In all other cases C_n and D_n are obtained from C_{n-1} and D_{n-1} by two left shifts. In all cases, by *one left shift* is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3, ..., 28, 1.

Permuted Choice 2 selects the 48 bits of K_n from the 56 bits of C_nD_n as specified by a table in FIPS Pub 46-3. The first bit of K_n is the 14th bit of C_nD_n, the second bit is the 17th, and so on with the 47th bit of K_n being the 29th bit of C_nD_n, and the 48th bit of K_n being the 32nd bit of C_nD_n.

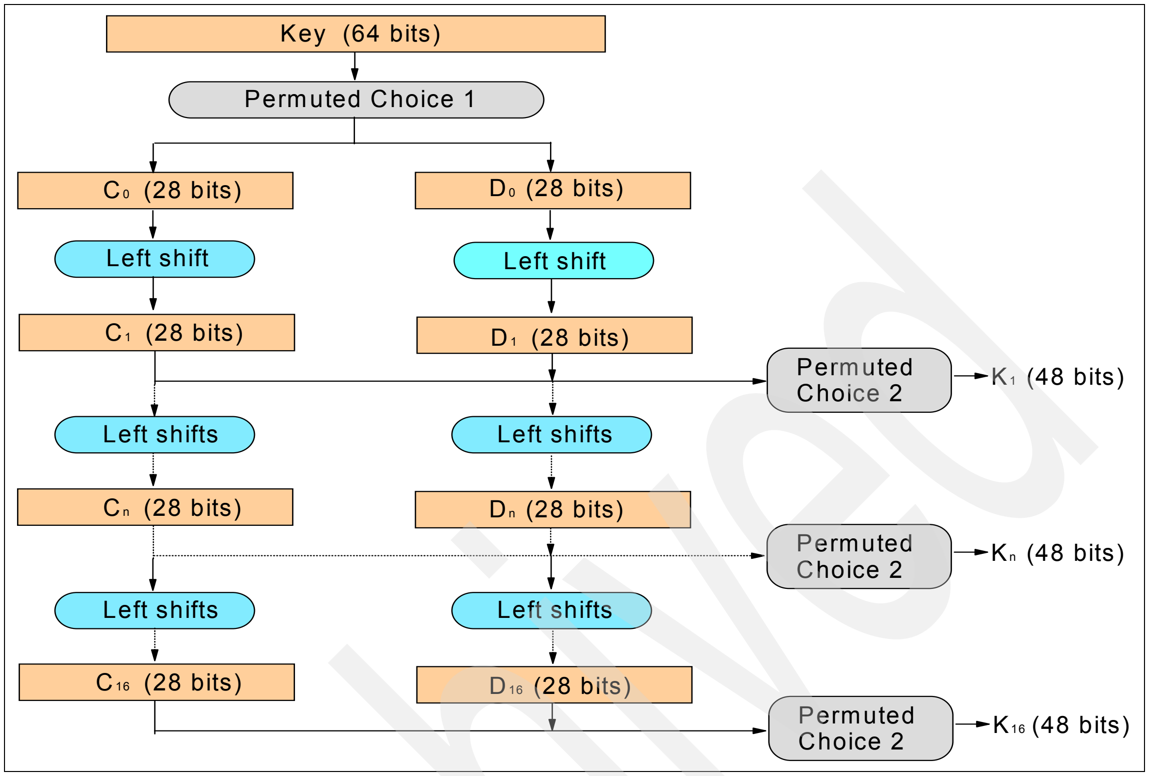


Figure B-3 The key schedule calculation of the DES algorithm

B.2 How AES works

The AES algorithm requires an initial set of 4 words of key data, and each of the Nr rounds requires 4 words of key data. Thus the algorithm requires 4 + 4*Nr = 4(Nr+1) words of key data. Consequently the algorithm specifies a Key Expansion routine that accepts the cipher key as input and uses it to generate a key schedule containing 4(Nr+1) words as output. The first Nk words of the key schedule are the first Nk words of the cipher key. For example, if we specify the following 128-bit cipher key:

2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

then the Key Expansion routine generates a key schedule containing 4(10 + 1) = 44 words, of which the first 4 words are as follows:

w_0 = 2b7e1516 w_1 = 28aed2a6 w_2 = abf71588 w_3 = 09cf4f3c

Example B-1 shows pseudo code that describes how AES encrypts data. The pseudo code uses four variables:

- The variable *in* is an array of 16 bytes that contains the 128 bits to be encrypted.
- The variable *out* is an array of 16 bytes that contains the data after it has been encrypted.
- The variable *w* is an array of 4*(Nr + 1) words; it contains the key schedule that the Key Expansion routine generated.
- The variable *state* is a rectangular array having 4 rows and 4 columns of bytes that represent intermediate encryption results.

Example: B-1 Pseudo code for AES encryption

```
Cipher( byte in[16], byte out[16], word w[4*(Nr+1)])
begin
    byte state[4,4]

    state = in

    AddRoundKey(state, w[0,3] )

    for round = 1 step 1 to Nr-1
       SubBytes(state)
       ShiftRows(state)
       MixColumns(state)
       AddRoundKey(state, w[4*round, 4*round+3])
    end for
    SubBytes(state)
    ShiftRows(state)
    AddRoundKey(state, w[4*Nr, 4*Nr+3])

    out = state
end
```

The input array, *in*, is copied into the *state* array according to the scheme shown in Example B-2.

Example: B-2 state = in

```
s[0,0] = in[0]   s[0,1] = in[4]  s[0,2] = in[8]   s[0,3] = in[12]
s[1,0] = in[1]   s[1,1] = in[5]  s[1,2] = in[9]   s[1,3] = in[13]
s[2,0] = in[2]   s[2,1] = in[6]  s[2,2] = in[10]  s[2,3] = in[14]
s[3,0] = in[3]   s[3,1] = in[7]  s[3,2] = in[11]  s[3,3] = in[15]
```

For example, if the data to be encrypted is:

32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

then the initial value of the *state* array would be as shown in Table B-1.

Table B-1 state = in

32	**88**	**31**	**e0**
43	5a	31	37
f6	30	98	07
a8	8d	a2	34

Note that the first column of the *state* array contains the first 32-bit word of the input, the second column of the *state* array contains the second word of the input, and so forth. The *state* can hence be interpreted as a one-dimensional array of 32-bit words (columns): w0, w1, w2, w3.

The encryption operations are then conducted on this *state* array. After an initial Round Key addition, the *state* array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first Nr-1 rounds because it does not include the Mix Columns() transformation. The final state is then copied to the output.

The individual transformations - SubBytes(), ShiftRows(), MixColumns(), and AddRoundKey() - which make up the round function that processes the state array are described in the following sections.

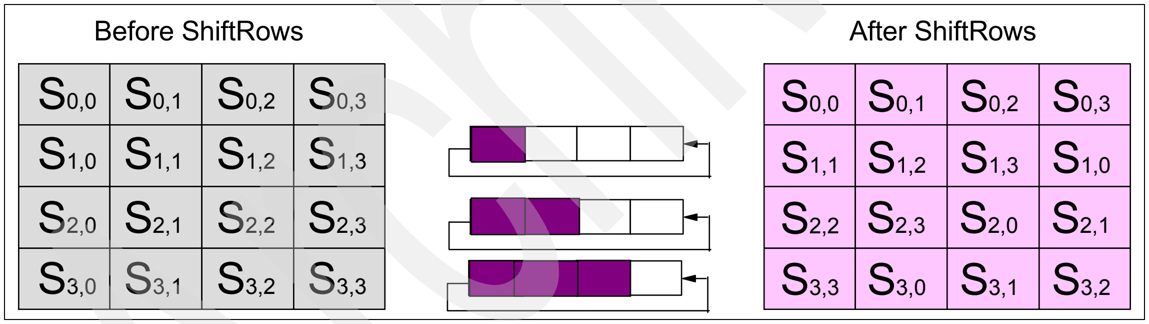
SubBytes() transformation

The SubBytes() transformation is a substitution that operates independently on each byte of the *state* array using the substitution table set forth in FIPS Pub 197. The substitution table contains 16 rows and 16 columns. If, for example, the $s_{1,1}$ element of the *state* array contained the byte 53, then the substitution value would be found at the intersection of the row with index '5' and the column with index '3' in the substitution table. This would result in 53 being replaced with ed.

The creators of AES used complex mathematics to create the substitution table in such a way that it would have an inverse substitution that could be used in a decryption operation.

ShiftRows() transformation

In the ShiftRows() transformation the bytes in the second row of the *state* array are cyclically shifted left by one column. The bytes in the third row are cyclically shifted left by two columns. The bytes in the last row are cyclically shifted left by three columns. The first row is not shifted. See Figure B-4.



Before ShiftRows
After ShiftRows
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,0}$
$S_{2,2}$ $S_{2,3}$ $S_{2,0}$ $S_{2,1}$
$S_{3,3}$ $S_{3,0}$ $S_{3,1}$ $S_{3,2}$

Figure B-4 ShiftRows() cyclically shifts the last three rows in the state array

MixColumns() transformation

The MixColumns() transformation, shown in Figure B-5, replaces the four bytes in column c of the state array, namely $\{s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}\}$, with four new bytes $\{s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}\}$ whose values are computed according to the following equations:

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c})$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

In these equations $a \oplus b$ represents the XOR operation, which is equivalent to addition modulo 2, and $x \bullet y$ represents the multiplication of two elements in the finite field GF(2^8). It is beyond the scope of this book to explain the mathematics of finite fields. However, we will note that the creators of AES chose the coefficients in the preceding equations to have certain specific values in the finite field GF(2^8) so that the MixColumns() transformation would have an inverse transformation that could be used in a decryption operation.

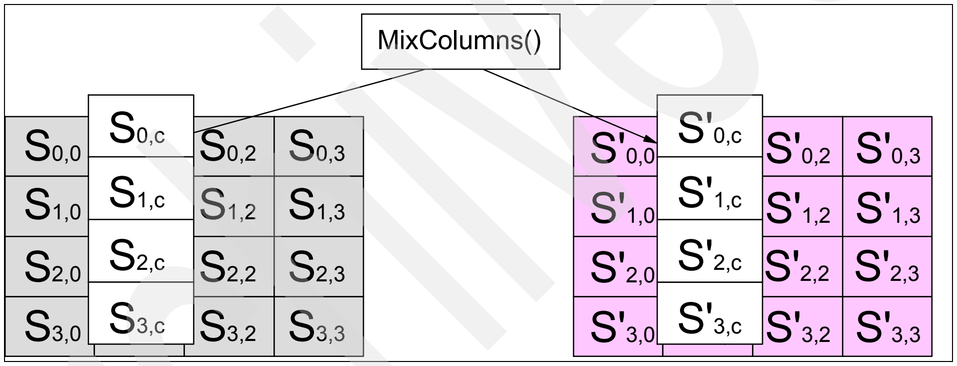

MixColumns()
$S_{0,0}$ $S_{0,c}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,c}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,c}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,c}$ $S_{3,2}$ $S_{3,3}$
$S'_{0,0}$ $S'_{0,c}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{1,c}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{2,0}$ $S'_{2,c}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,c}$ $S'_{3,2}$ $S'_{3,3}$

Figure B-5 MixColumns() operates on the state array column-by-column

AddRoundKey() transformation

Figure B-6 shows that the AddRoundKey() transformation XORs each column of the *state* array with a word from the key schedule. In Figure B-6 l is equal to four times the number of the round. The AddRoundKey() transformation performs an XOR operation between the first column of the state array and word w_l from the key schedule in order to create the first column of the new *state* array. Likewise, it performs an XOR operation between the second column of the *state* array and word w_{l+1} from the key schedule to create the second column of the new state array. Likewise, column three is XORed with w_{l+2}, and column four is XORed with w_{l+3}.

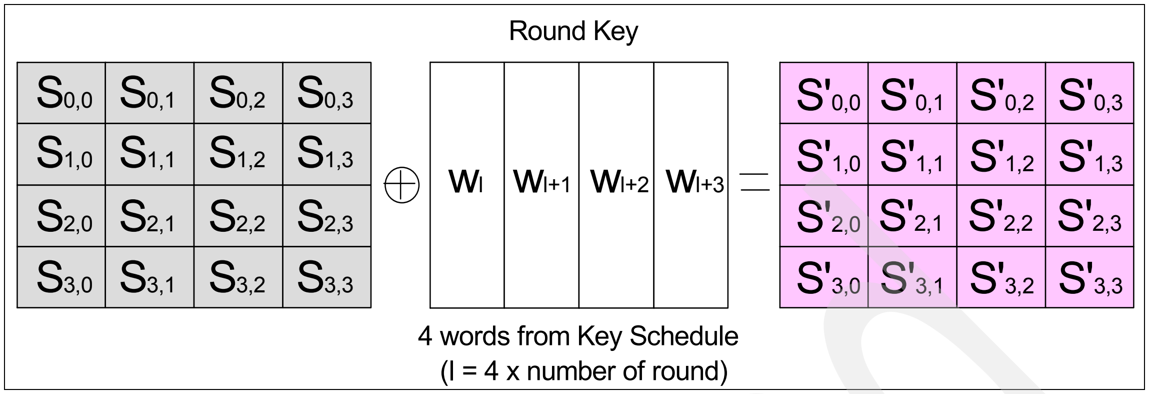

Round Key
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$\oplus$
W_l W_{l+1} W_{l+2} W_{l+3}
$=$
$S'_{0,0}$ $S'_{0,1}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{1,1}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{2,0}$ $S'_{2,1}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,1}$ $S'_{3,2}$ $S'_{3,3}$
4 words from Key Schedule
(l = 4 x number of round)

Figure B-6 AddRoundKey() XORs each column of the state with a word from the key schedule

B.3 How SHA-1 works

SHA-1 can be used to hash a message, *M*, having a length of *L* bits, where *L* is less than 2^{64}; it produces a 160-bit (20 byte) message digest. The message must be preprocessed before the actual hash computation begins. Before we can describe the preprocessing and the computation, we need to define an operation, some functions, and some constants.

B.3.1 Definitions

The **ROTL**^{n}(x) operation, where x is a 32-bit word and n is 0 or a positive integer less than 32, is a circular shift (rotation) of x by n positions to the left.

SHA-1 uses a sequence of functions $f_0, f_1, \ldots, f_{79}$. Each function f_t, where $0 \leq t \leq 79$, operates on three 32-bit words, x, y, and z, and produces a 32-bit word as output. The function $f_t(x,y,z)$ is defined as follows:

- For $0 \leq t \leq 19$:

$$f_t(x, y, z) = (x \wedge y) \oplus ((\neg x) \wedge z)$$

- For $20 \leq t \leq 39$:

$$f_t(x, y, z) = x \oplus y \oplus z$$

- For $40 \leq t \leq 59$:

$$f_t(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$$

- For $60 \leq t \leq 79$:

$$f_t(x, y, z) = x \oplus y \oplus z$$

SHA-1 uses a sequence of eighty constant 32-bit words $K_0, K_1, \ldots, K_{79}$, which are defined as follows:

- For $0 \leq t \leq 19$: 5a827999
- For $20 \leq t \leq 39$: 6ed9eba1
- For $40 \leq t \leq 59$: 8f1bbcdc
- For $60 \leq t \leq 79$: ca62c1d6

B.3.2 SHA-1 preprocessing

Preprocessing consists of three steps:

1. Padding the message, *M*.
2. Parsing the padded message into message blocks.
3. Setting the initial hash value.

B.3.2.1 Padding the message

The purpose of padding the message is to ensure that the length of the padded message is a multiple of 512 bits. Begin the padding by appending the bit “1” to the end of the message, followed by *k* zero bits, where *k* is the smallest, non-negative solution to the equation $L + 1 + k = 448 \bmod 512$. Then append the 64-bit block that is equal to the number *L* expressed using a binary representation.

As an example, consider the (8-bit ASCII) message “abc.” Since this message has length 8 x 3 = 24, the message is padded with a “1” bit, then 448 - (24 + 1) = 423 zero bits, and then the message length become the 512-bit padded message shown in Figure 13-36.

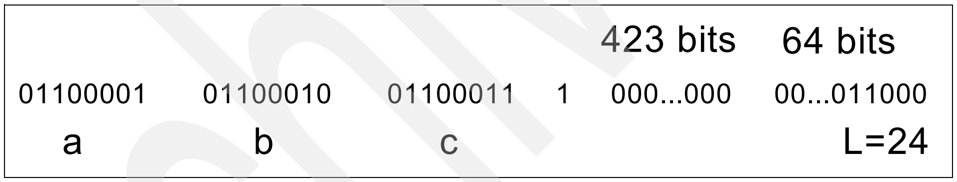


Figure 13-36 Padding a message

As a second example, let the message, *M,* be the 448-bit ($L = 448$) ASCII string

abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq

The message is padded by appending a “1” bit, followed by 511 zero bits, and ending with the hex value 00000000 000001C0 (the 64-bit representation of the length, 448). Note that the final padded message consists of two blocks instead of just one.

B.3.2.2 Parsing the padded message

After a message has been padded, it must be parsed into *N* 512-bit blocks, $M^{(1)}$, $M^{(2)}$,...,$M^{(N)}$. Since the 512 bits of each of these blocks can be expressed as sixteen 32-bit words, the first 32 bits of message block *i* are denoted $M^{(i)}_0$, the next 32 bits are $M^{(i)}_1$, and so on up to $M^{(i)}_{15}$. See Figure 13-37.

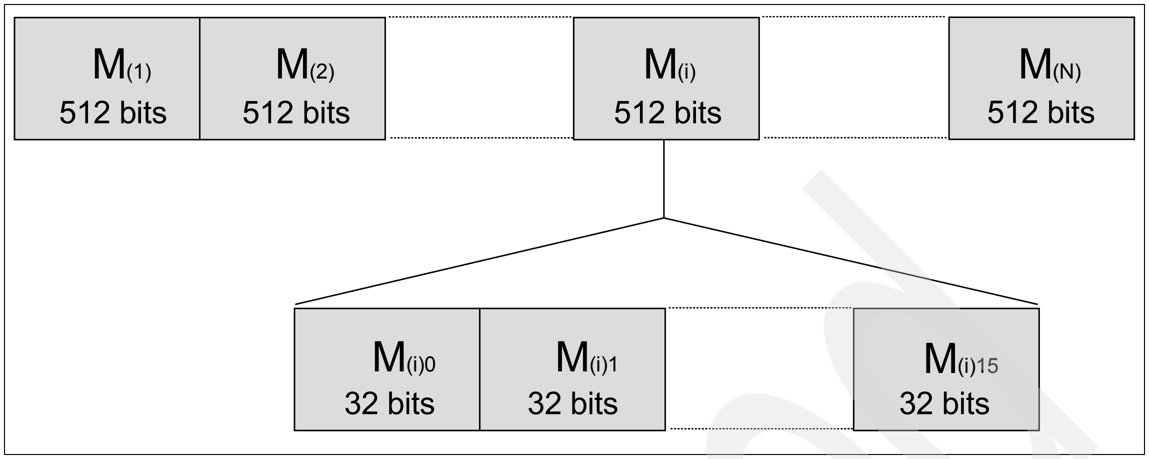


Figure 13-37 Padded and parsed message

The 512-bit blocks are processed sequentially, taking as input the result of the hash so far and the current message block, with the final output being the hash value for the message. We need an initial hash value to start the process.

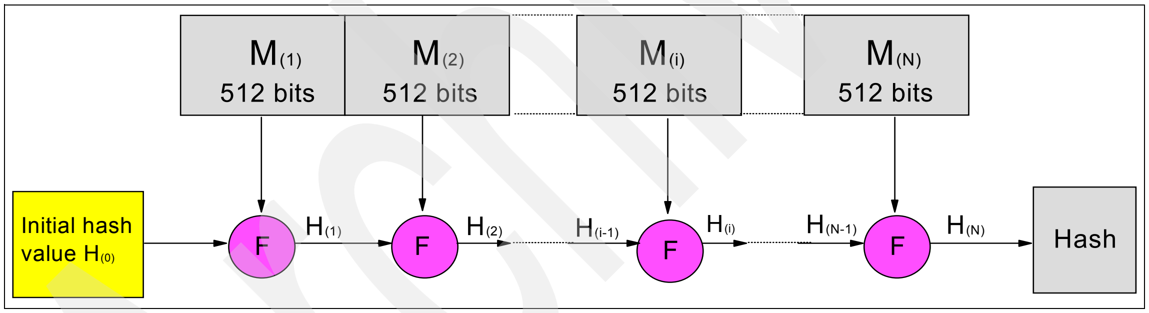


Figure 13-38 SHA-1 as an iterative hash function

B.3.2.3 Setting the initial hash value

For SHA-1 the initial hash value, $H^{(0)}$, consists of the following five 32-bit words:

$H^{(0)}_0$ = 67452301

$H^{(0)}_1$ = efcdab89

$H^{(0)}_2$ = 98badcfe

$H^{(0)}_3$ = 10325476

$H^{(0)}_4$ = c3d2e1f0

B.3.3 SHA-1 hash computation

After preprocessing is completed, each message block, $M^{(1)}$, $M^{(2)}$, ..., $M^{(N)}$, is processed in order. The following steps are performed for each block:

1. Prepare the message schedule $\{W_0, W_1, W_2, ..., W_{79}\}$ of 32-bit words:
   - For $0 \leq t \leq 15$: $W_t = M^{(i)}{}_t$

     For example, $W_0 = M^{(i)}{}_0$, the first 32-bit word of the *i* th message block. Likewise, $W_{15} = M^{(i)}{}_{15}$, the last 32-bit word of the *i* th message block.
   - For $16 \leq t \leq 79$:

     $$W_t = ROTL^1(W_{(t-3)} \oplus W_{(t-8)} \oplus W_{(t-14)} \oplus W_{(t-16)})$$

     For example,

     $$W_{16} = ROTL^1(W_{13} \oplus W_8 \oplus W_2 \oplus W_0)$$

     This is the same as

     $$W_{16} = ROTL^1(M^i{}_{13} \oplus M^i{}_8 \oplus M^i{}_2 \oplus M^i{}_0)$$

     That is, W_{16} is prepared by performing an exclusive OR operation on the first, third, ninth, and fourteenth words of the message block and then performing a circular left shift by one bit position. The result is still a 32-bit word.
2. Initialize the five working variables, *a*, *b*, *c*, *d*, and *e*, with the (*i*-1)st hash value:

   $a = H^{(i-1)}{}_0$

   $b = H^{(i-1)}{}_1$

   $c = H^{(i-1)}{}_2$

   $d = H^{(i-1)}{}_3$

   $e = H^{(i-1)}{}_4$
3. For t = 0 to 79 perform the following:

   $T = ROTL^5(a) + f_t(b,c,d) + e + K_t + W_t$

   $e = d$

   $d = c$

   $c = ROTL^{30}(b)$

   $b = a$

   $a = T$

where T is a temporary 32-bit word and addition (+) is performed modulo 2^{32}. Observe that T is computed by using the (i-1)st hash value (in the form of the variables `a`, b, c, d, and e) and the ith message block (in the form of the words W_t, which ultimately come from the ith message block) as shown in Figure 13-38 on page 585.

4. Compute the ith intermediate hash value $H^{(i)}$:

$$H^{(i)}{}_0 = a + H^{(i-1)}{}_0$$
$$H^{(i)}{}_1 = b + H^{(i-1)}{}_1$$
$$H^{(i)}{}_2 = c + H^{(i-1)}{}_2$$
$$H^{(i)}{}_3 = d + H^{(i-1)}{}_3$$
$$H^{(i)}{}_4 = e + H^{(i-1)}{}_4$$

Again, addition is performed modulo 2^{32}.

After repeating steps one through four a total of N times (that is, after processing the last 512-bit message block $M^{(N)}$), the resulting 160-bit message digest of the message, M, is

$$H^{N}{}_0 \parallel H^{N}{}_1 \parallel H^{N}{}_2 \parallel H^{N}{}_3 \parallel H^{N}{}_4 \parallel H^{N}{}_5$$

B.4 How the HMAC algorithm of FIPS PUB 198 works

FIPS PUB 198 describes an HMAC using the parameters shown in Table B-2.

Table B-2 Parameters for the HMAC algorithm

Parameter	Description
B	Block size (in bytes) of the input to the hash function
H	Hash function (such as SHA-1)
ipad	Inner pad; the byte x'36' repeated B times
K	Secret key shared between the originator and the intended receiver(s)
K_0	The key K after any necessary pre-processing to form a key of length B
L	Block size (in bytes) of the output of the hash function
opad	Outer pad; the byte x'5C' repeated B times
t	The number of bytes of MAC
text	The data on which the HMAC is calculated; *text* does **not** include the padded key. The length of t*ext* is n bits.

A well-known practice with MACs is to truncate their output; that is, the length of the MAC used is less than the length L of the output of the hash function. When a truncated HMAC is used, the t leftmost bytes of the HMAC computation are used as the MAC. The output length, t, should be no less than four bytes, so $4 \leq t \leq L$.

To compute a MAC over the data '*text*' using the HMAC function, FIPS PUB 198 specifies the performance of the following operation:

$$MAC(text)_t = HMAC(K, text)_t = H((K_0 \oplus opad) \| H((K_0 \oplus ipad) \| text))_t$$

where $a \oplus b$ indicates an Exclusive-OR operation and || indicates a concatenation operation. Table B-3 describes the step-by-step process in the HMAC algorithm.

Table B-3 The HMAC algorithm

Step	Description of step
1	If the length of $K = B$: set $K_0 = K$. Go to step 4.
2	If the length of $K > B$: hash K to obtain a string L bytes long, then append (B-L) zeros to create string K_0 which is B bytes long. That is, $K_0 = \mathrm{H}(K) \parallel 00...00$. Then go to step 4.
3	If the length of $K < B$: append zeros to the end of K to create a string K_0 which is B bytes long. (For example, if K is 20 bytes in length and $B = 64$, then K will be appended with 44 zero bytes X'00').
4	Exclusive-OR K_0 with *ipad* to produce the string $\boldsymbol{K_0} \oplus \boldsymbol{ipad}$. This string has length B.
5	Append the stream of data 'text' to the string resulting from step 4: $(\boldsymbol{K_0} \oplus \boldsymbol{ipad}) \parallel \boldsymbol{text}$
6	Apply the hash function H to the stream generated in step 5: $H((\boldsymbol{K_0} \oplus \boldsymbol{ipad}) \parallel \boldsymbol{text})$.
7	Exclusive-OR K_0 with *opad* to produce the string $\boldsymbol{K_0} \oplus \boldsymbol{opad}$. This string has length B.
8	Append the result from step 6 to step 7: $(\boldsymbol{K_0} \oplus \boldsymbol{opad}) \parallel \boldsymbol{H}((\boldsymbol{K_0} \oplus \boldsymbol{ipad}) \parallel \boldsymbol{text})$
9	Apply the hash function H to the result from step 8: $\boldsymbol{H}((\boldsymbol{K_0} \oplus \boldsymbol{opad}) \parallel \boldsymbol{H}((\boldsymbol{K_0} \oplus \boldsymbol{ipad}) \parallel \boldsymbol{text}))$
10	Select the leftmost t bytes of the result of step 9 as the MAC.

The successful verification of a MAC does not completely guarantee that the accompanying message is authentic; there is a chance that a source with no knowledge of the key can present a purported MAC on the plaintext message that will pass the verification procedure. For example, an arbitrary purported MAC of t bits on an arbitrary plaintext message may be successfully verified with an expected probability of $(1/2)^t$. Therefore, in general, if the MAC is truncated, then its length, t, should be chosen as large as is practical, with at least half as many bits as the output block size, L.

Abbreviations and acronyms

2PC	Two-phase commit
AAT	Application Assembly Tool
AOR	Application-Owning Region
API	Application Programming Interface
APPC	Advanced Program-to-Program Communications
ASCII	American Standard Code for Information Interchange
B2B	Business-to-business
CCI	Common Client Interface
CEX2	Crypto Express2
CICS	Customer Information Control System
CICS TG	CICS Transaction Gateway
CORBA	Common Object Request Broker Architecture
CPACF	CP Assist for Cryptographic Functions
DPL	Distributed Program Link
DSA	Digital Signature Algorithm
EAR	Enterprise Application Archive
EBCDIC	Extended Binary Coded Decimal Interchange Code
ECI	External Call Interface
EIS	Enterprise Information Systems
EJB	Enterprise JavaBeans™
EPI	External Presentation Interface
ESI	External Security Interface
ESM	External Security Manager
EXCI	External CICS Interface
FTP	File Transfer Protocol
GTF	Generalized Trace Facility
GUI	Graphical user interface
HFS	Hierarchical File System
HTML	Hypertext Transfer Protocol
HTTP	Hypertext Markup Language
IBM	International Business Machines
ICSF	Integrated Cryptographic Service Facility
IDE	Integrated development environment
IRC	inter-region communication
ISC	Inter-System Communication
ITSO	International Technical Support Organization
J2C	J2EE Connector Architecture (also known as JCA)
J2EE	Java 2 Enterprise Edition
JAAS	Java Authentication and Authorization Service
JAR	Java Archive
JAX-RPC	Java API for XML-based RPC
JCA	J2EE Connector Architecture (also known as J2C)
JDBC	Java Database Connectivity
JDK™	Java Developer's Kit
JKS	Java Keystore
JMS	Java Messaging Service
JNDI	Java Naming and Directory Interface™
JNI™	Java Native Interface
JSP	JavaServer Page

JSSE	Java Secure Sockets Extension
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
LPAR	Logical Partition
LTPA	Lightweight Third-Party Authentication
LUW	Logical Unit of Work
MRO	Multi Region Operation
PEM	Password Expiration Management
PKA	Public Key Algorithm
PKDS	Private Key Data Set
QoS	Quality of service
RAD	Rational Application Developer
RAR	Resource Adapter Archive
RPC	Remote procedure call
RRMS	Recoverable Resource Management Services
RRS	Resource Recovery Services
RSA	Rivest, Shamir, and Adelman (encryption)
SAF	System Authorization Facility
SAX	Simple API for XML
SDK	Software Development Kit
SNA	Systems Network Architecture
SOAP	Simple Object Access Protocol (also known as Service-Oriented Architecture Protocol)
SQL	Structured query language
SSL	Secure Sockets Layer
TCP/IP	Transmission Control Protocol/Internet Protocol
TLS	Transport Layer Security
TPV	Tivoli® Performance Viewer
UDDI	Universal Description, Discovery, and Integration
UOW	Unit of Work
URI	Uniform resource identifier
URL	Uniform resource locator
URN	Uniform resource name
VSAM	Virtual Storage Access Method
WMQ	WebSphere MQ
WS-A	Web Services - Adressing
WS-AT	Web Services - Atomic Transaction
WS-BA	Web Services - Business Activity
WS-C	Web Services - Coordination
WSDL	Web Services Description Language
WSGW	Web Services Gateway
WS-I	Web Services interoperability
WS-Security	Web Services - Security
XMI	XML metadata interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

For information about ordering these publications, see "How to get IBM Redbooks" on page 594. Note that some of the documents referenced here may be available in softcopy only.

- *Application Development for CICS Web Services,* SG24-7126
- *WebSphere Version 6 Web Services Handbook Development and Deployment*, SG24-6461
- *Web Services Handbook for WebSphere Application Server 6.1,* SG247257
- *Patterns: Service-Oriented Architecture and Web Services*, SG24-6303
- *WebSphere MQ in a z/OS Parallel Sysplex Environment,* SG24-6864
- *WebSphere MQ Security in an Enterprise Environment,* SG24-6814

Other publications

These publications are also relevant as further information sources:

- *Enterprise COBOL for z/OS V3R3 Programming Guide,* SC27-1412
- *Enterprise COBOL for z/OS V3R3 Language Reference,* SC27-1408
- *CICS Web Services Guide V3.1,* SC34-6458
- *CICS TS V3.1 RACF Security Guide,* SC34-6249
- *CICS TS V3.1 Application Programming Reference, S*C34-6434
- ICSF Overview, SA22-7519
- RACF Command Reference, SA22-7687
- *CICS Transaction Server for z/OS RACF Security Guide*, SC34-6454
- *Tour Web Services Atomic Transaction operations: Beginner's guide to classic transactions, data recovery, and mapping to WS-Atomic Transactions,* IBM developerWorks (September 2, 2004)

- *WebSphere MQ for z/OS System Setup Guide V6.0,* SC34-6583
- *WebSphere MQ - Transport for SOAP,* SC34-6651
- *WebSphere MQ Security,* SC34- 6588

Online resources

These Web sites and URLs are also relevant as further information sources:

- CICS library

  `http://www.ibm.com/software/htp/cics/library/`
- WebSphere Application Server library

  `http://www.ibm.com/software/webservers/appserv/was/library/index.html`

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

A

aborted 441
AddRoundKey 581
Advanced Encryption Standard (AES) 161
AES algorithm
  how it works 573
alert protocol 206
APARs
  PK10849 467
  PK16509 465
  PK20393 113
application sample 76
Application Server Toolkit V6.1 (AST) 333
asymmetric cryptography 162
ATM
  application 413
atomic transaction 409
AtomicClient 467–468, 473, 489
auditing 153
authentication 152, 258
  basic 330
authorizaiton 152
Automatic Teller Machine (ATM) 410
availability 126
  configuring for 101

B

basic 239
basic authentication 239, 330
  sample 290, 336
bean-managed transaction 478
block cipher 154
bus
  see service integration bus

C

C language 34
C++ 34
catalog application 122, 467, 489
catalog manager 146
catalog manager application 67
  configuration transaction, ECFG 97, 122, 513
  testing 90, 100, 124, 145
Catalog.ear 87
CatalogAtomic.ear 478
CatalogSec.ear 288
CatalogSec_WS-Security.ear 293, 374, 398
CatalogSec_WS-Security_HTTPS.ear 309
CEDA 119, 471
CEMT 120
certificate 302, 347
certificate revocation 191
certification paths 189
channels 59
CICS 324
  certificate 350
  COBOL programs 66
  configuring 339
  Execution Diagnostic Facility (EDF) 125
  security 220, 236
  service provider 37
    preparation 38
    using HTTP 76–77
    using WMQ 116
  service requester 41
    local optimization 45
    preparation 42
    timeout considerations 96
    using HTTP 94–95
    using WMQ 121
  trace 105
  Web services assistant 34, 62
    DFHLS2WS 63
    DFHWS2LS 64
  Web services support 33
    catalog manager application 67
    development tools 61
    high availability 101, 126
    overview 34
    problem determination 104
    resources 47
    resources checklist 55
  WS-Security Component 256
CICS default user ID 237
CICS region user ID 237
CICSplex 102

CICS-to-CICS 112
cipher 154
  suite 202
Cipher Block Chaining (CBC) 155
CIWSMSGH 80, 93, 118, 528
CIWSSECH 293–296, 533
CIWSSECR 315, 555
CIWSSECS 317, 331, 562
COBOL 34
code page 74, 77
code page conversion 77
commit 411, 441
committed 441
Communications Manager 102
completion protocol 438
confidentiality 152
connection 79
connector 467
container-managed transaction 478
containers 59
  context container 59
  control container 59
  header container 60
  user container 60
CoordinationContext 428
  elements 496
    Address 497
    CoordinationType 497
    Expires 496
    Identifier 496
    ReferenceProperties 497
    RegistrationService 497
  example 495
  sample created by CICS 430
  sample created by WebSphere 431
coordinator 434, 464
CORBA 6
core standards 8
CP Assist for Cryptographic Functions (CPACF) 349
CPIH 41, 93
CPIL 116
credentials 339
Crypto Express2 (CEX2) feature 217, 349
cryptography 151, 153
  hardware 349
  public key 162
  secret key 153
  symmetric 154
cryptosystem 163

D
daisy chain 464
Data 155
Data Encryption Algorithm (DEA) 155
Data Encryption Standard (DES) 155
DB2 table 467
decrypting 153
default user ID, CICS 237
DES algorithm
  how it works 573
development tools 61
DFH$RING 303, 327
DFHLS2WS 34, 63
DFHPIRS 457
DFHPISN1, message handler module 448, 468
DFHRESPONSE container 82
DFHRSURI, URIMAP resource 454
DFHWS2LS 34, 64
DFHWSATH, header processing program 448, 455–457, 468
DFHWSATP, provider pipeline 449–450
DFHWSATR, requester pipeline 449
  PIPELINE definition 452
DFHWSATX, message handler 449, 452
Diffie-Hellman 196
digital signature 153, 176, 358
Digital Signature Algorithm (DSA) 174
dispatchOrder.ear 99
DispatchOrderAtomic.ear 510
distributed routing 102–103
DPL subset 81
dynamic program routing 102, 104
dynamic response 47
dynamic routing 104

E
EAR file 99
eavesdropping 238
ECFG 97, 122, 513
Electronic Codebook (ECB) 155
electronic data interchange (EDI) 4
Elliptic Curve Digital Signature Algorithm (ECDSA) 176
encrypting 153
encryption algorithm 202
endpoint

listener 131
reference 419
enterprise bean 478
error 328
handling 16
error handling 206
ExampleAppClient.ear 87
ExampleAppDispatchOrder 98
ExampleAppDispatchOrder.ear 99, 510
Execution Diagnostic Facility (EDF) 125
Extensible Markup Language
see XML

F
FTP 9, 304

G
gateway
instance 131
configuring 140
service 131, 135
configuring 139, 141
invoking 147
publishing 145
testing 145

H
handler program 527
handshake protocol 202, 207
hardware
cryptography 349
hash algorithm 202
hashing 153, 167
heuristic decision 444
HFS file system 74
high availability 101, 126
with HTTP 101
with WMQ 126
HMAC algorithm
how it works 573
HTTP 7, 36, 73, 237
hub configuration 464

I
IBM Java SDK 305
ibm-webservices-bnd.xmi 287, 332
ibm-webservicesclient-bnd.xmi 287, 332
ibm-webservicesclient-ext.xmi 287, 332
ibm-webservices-ext.xmi 287, 332
identification 152
identity assertion 273
iKeyMan 305
IMS 66
inbound service 131
in-doubt window 443
integrity 152
intermediary 270
intermediate gateway
identity assertion 273
Internet 3
iterated block cipher 154

J
J2EE
application 146
connector 467
J2EE Connector Architecture (JCA) 524
Java SDK 305
Java Transaction API (JTA) 460
JAX-RPC
handler 130–131
JCA 467
JDBC 467
JDBC provider 487
JMS 9, 467
JVM 89

K
key agreement protocol 196
key exchange algorithm 202
key exchange protocol 196
key pair
generation 347
key ring 302
creating 302
key schedule algorithm 576
keystore 305

L
local optimization 45

M
master secret 210
mediation 130–131

message authentication codes (MAC 171
message handler 35, 80, 118, 324
  introduction 57
  SOAP 57
  writing 80
message handler program 331
message information headers 422
mixcolumns 580
mustUnderstand 14, 293, 339, 344, 379, 401
MVS 102
MYPARSER 296, 550

N
namespace 12
Namespaces 12
National Institute of Standards and Technology (NIST) 155
non-repudiation 153
non-terminal message 443
notification message 442

O
one-way message 442
outbound service 131

P
Parallel Sysplex 102
participant 445, 464
performance 332
PIPELINE 49
  attributes
    CONFIGFILE 50
    SHELF 50
    WSDIR 49
  defining 83, 95, 119
pipeline 35, 49, 80, 118, 293, 329
  configuration file 51
    customizing 80
pipeline alias transaction 48, 57, 93
PL/I 34
Port definition 29
prepared 441
presumed abort 411
presumed trust 274
problem 344
  determination 104
protocol transformation 132
public key cryptography 162
  cryptography
    public key 151

R
RACF 348
  key ring 327
Rational Application Developer 293, 308
readonly 441
recoverable 411
Redbooks Web site 594
  Contact us xvii
reference
  parameter 421
  property 421
Register
  request 432
  response 434
remote procedure call (RPC) 17
replay 441
request consumer 268
request generator 268, 387
RequestStream 459
response consumer 268, 367
response generator 268
rollback 411, 441
RPC 17
  style 30
RSA algorithm 163

S
sample application 76, 315, 467
scalability 101
secret 153
secret key cryptography 153
  cryptography
    secret key 151
Secure Sockets Layer (SSL) 237
security
  auditing 153
  authentication 152
  authorization 152
  confidentiality 152
  exposures 237
  HTTP 239
  identification 152
  integrity 152
  non-repudiation 153

prefixing 328
setup 327
SIT parameters 328
SOAP message 239, 285
see WS-Security
token 330
transport-level 239
comparison with SOAP message security 269
WMQ 251, 318
testing 319
self-signed 347
service
broker 5
gateway 135
see gateway service
inbound 131
mediation 130
outbound 131
provider 5
requestor 5
service integration bus 129–130, 139
configuration 132
securing CICS Web services 270
service provider 76
service provider mode 331
service requester
installing 87
timeout considerations 124
service requester setup 332
service-oriented approach 4
service-oriented architecture 3–4
characteristics 5
shiftrows 580
signature
processing 375
SIT parameters 78, 328
DFLTUSER 237
ENCRYPTION 241
KEYRING 242, 309, 328
MAXSSLTCBS 242
SEC 279, 328
SECPRFX 279, 328
SSLCACHE 243
SSLDELAY 242
TCPIP 78
XTRAN 279, 328
XUSER 279, 328
SMTP 9
SNIFFER 106, 468, 539
sample output 547
SOA
see service-oriented architecture
SOAP 8–9
1.1 specification 9
1.2 specification 10
binding 30
body 15, 497
outbound data conversion 41
communication styles 17
document 17
RPC 17
encodings 17
literal 17
SOAP encoding 17
envelope 12–13
fault 16
fault messages 299, 342, 382
headers 14, 496
intermediary 14
introduction 12
message 324, 358
message encryption 385
message handler 35, 57
message security 239, 253, 269
messaging mode 17
MustUnderstand 15
namespaces 12
security headers 312
server 7
Simple Object Access Protocol 9
validation in CICS 53
SOAPFAULT
commands
ADD 58
CREATE 58
generating a SOAP fault in CICS 299
SOCKETCLOSE 79
spoofing 238
SSL/TLS 301
CICS support for 240
creating a key ring 302
enabling CICS support 241, 309
enabling WebSphere support 304
testing 310
with WMQ 253
standards
JAX-RPC 130

JCA 524
JTA 460
SOAP 9
UDDI 10
WS-Addressing 409
WS-Atomic Transaction 409
WS-Coordination 409
WSDL 10
WS-Security 11
WS-Transactions 11
XML 9
static response 47
Station-to-Station (STS) protocol 198
subbytes 580
subkey 154
surrogate 236
surrogate access 341
switch profiles 251
symmetric cryptography 154
Sysplex Distributor 102, 128

T
tampering 238
tbsCertificate 182
TCP/IP 445
load balancing 102
TCPIP, SIT parameter 78
TCPIPSERVICE 55, 78, 309
attributes
AUTHENTICATE 240
PORTNUMBER 79
PROTOCOL 79
terminal message 443
testing 90, 100, 124, 298, 310, 328, 489, 517
throughput 332
token 330
tracing
CICS 105
transactional scope 413
transactions
atomic 417, 429
enabling CICS support 447
business activity 429
classic 410–411
two phase commit 411
transport 111, 269
security 239
Transport Layer Security (TLS) 199
trust token 274
trusted 162
two phase commit (2PC) 411
presumed abort 412
walkthrough 413
with WS-AT 440

U
UDDI 8
Unicode 77
Universal Description, Discovery, and Integration (UDDI) 10
URI 123
URIMAP 47, 120, 135, 280, 330
attributes
USERID 250
defining 86, 135, 281
for serving HFS file 135
setting the user ID 280
UsernameToken 331
UsernameTokenGenerator 291, 337

V
verification 175

W
Waldo 410
Web 325
Web service 53
Web Service Description Language (WSDL) 8
Web services 3, 325
assistant 62
binding file 55
definition 7
introduction 6
properties 7
Web Services Description Language
see WSDL
Web Services Gateway 130
Web Services Interoperability Organization
see WS-I
Web services versus service-oriented architectures 6
Web technologies 3
WEBSERVICE 53
attributes
VALIDATION 53

WSBIND 53
WSDLFILE 53
dynamically installing 85
WebSphere 266
certificate 352
key store 354
MQ 111
service requester
deploying 88, 98–99
using HTTP 87, 98
WebSphere Application Server 324
administrative console 88
connection pooling 89
WebSphere Developer for zSeries 65
WebSphere MQ
see WMQ
WMQ 112
configuration 114
defining queues 115
trigger process 116
shared queues 127–128
WS Binding 308
WS Binding page 291, 337
WS Extension page 289, 335
WS-Addressing 409, 418
CICS Registration Service endpoint 424
endpoint reference 419
specification 418
WSATHND 472, 567
WS-Atomic Transaction (WS-AT) 36, 409, 415, 436
Completion protocol 438
daisy chain test scenario 510
testing 517
enabling CICS support 448–449
enabling WebSphere support 460, 478
resynchronization processing 443
simple test scenario 469
testing 489
specification 428
Two-Phase Commit protocol 439
Durable 2PC 439
Volatile 2PC 439
typical scenario 467
walkthrough 415, 442
WS-Business Activity specification 428
unsupported by CICS 429
WS-Coordination 409
Activation service 426
Coordination service 426
Protocol services 426
Registration service 426
specification 425
walkthrough 434
WSDL 10, 18
binding 19, 27
bindings 30
definition 24
document 19
anatomy 19
message 19, 25
namespaces 23
operation 19, 26
port 19, 29
port type 19, 26
service 19
service definition 29
SOAP binding 30
type 19
types 24
Web Services Description Language 10
WSGW 130
WS-I
basic profile 10
introduction 6
WS-Security 253, 285
CICS support 256
configuration
files 293, 339–341, 378, 400
configuring CICS 293
example 258, 293, 339, 400
message authentication 258
specifications 255
test scenario 298, 341, 378, 400
typical scenario 255
WebSphere client configuration 288, 334
WebSphere configuration files 286, 332
with WMQ 312
WS-Security Component 256

X
X.509 257, 324
XML 7, 9
XML encryption 385
XML Services for the Enterprise (XSE) 65

Z
z/OS 102

Communications Manager 102
Parallel Sysplex 102

IBM

Redbooks

Implementing CICS Web Services

(1.0” spine)
0.875”<->1.498”
460 <-> 788 pages

Implementing CICS Web Services

Redbooks

Configuring and securing Web services in CICS Transaction Server

Connecting CICS to a service integration bus

Enabling atomic Web services

The Web services support in CICS Transaction Server Version 3.1 enables your CICS programs to be Web service providers or requesters. CICS supports a number of specifications including SOAP Version 1.1 and Version 1.2, and Web services distributed transactions (WS-Atomic Transaction).

This IBM Redbooks publication will help you configure the CICS Web services support for both HTTP and WebSphere MQ based solutions. We show how Web services can be used to integrate J2EE applications running in WebSphere Application Server with COBOL programs running in CICS.

It begins with an overview of Web services standards and the Web services support provided by CICS TS V3.1. Complete details for configuring CICS Web services using both HTTP and WebSphere MQ are provided next, along with the steps for using Web services to connect to CICS from a service integration bus. The book then shows how CICS Web services can be secured using a combination of Web Services Security (WS-Security) and transport-level security mechanisms such as SSL/TLS. Finally, it demonstrates how atomic Web services transactions can be configured to allow WebSphere and CICS resource updates to be synchronized.

This book concentrates on implementation specifics such as security, transactions, and availability. The companion redbook *Developing CICS Web Services* (SG24-7126) presents detailed information about developing CICS Web services.

SG24-7206-02 ISBN 0738489042