

AFFIDAVIT OF NATHANIEL E FRANK-WHITE

1. I am a Records Request Processor at the Internet Archive. I make this declaration of my own personal knowledge.
2. The Internet Archive is a website that provides access to a digital library of Internet sites and other cultural artifacts in digital form. Like a paper library, we provide free access to researchers, historians, scholars, and the general public. The Internet Archive has partnered with and receives support from various institutions, including the Library of Congress.
3. The Internet Archive has created a service known as the Wayback Machine. The Wayback Machine makes it possible to browse more than 450 billion pages stored in the Internet Archive's web archive. Visitors to the Wayback Machine can search archives by URL (i.e., a website address). If archived records for a URL are available, the visitor will be presented with a display of available dates. The visitor may select one of those dates, and begin browsing an archived version of the Web. Links on archived files in the Wayback Machine point to other archived files (whether HTML pages or other file types), if any are found for the URL indicated by a given link. For instance, the Wayback Machine is designed such that when a visitor clicks on a hyperlink on an archived page that points to another URL, the visitor will be served the archived file found for the hyperlink's URL with the closest available date to the initial file containing the hyperlink.
4. The archived data made viewable and browsable by the Wayback Machine is obtained by use of web archiving software that automatically stores copies of files available via the Internet, each file preserved as it existed at a particular point in time.
5. The Internet Archive assigns a URL on its site to the archived files in the format `http://web.archive.org/web/[Year in yyyy][Month in mm][Day in dd][Time code in hh:mm:ss]/[Archived URL]` aka an "extended URL". Thus, the extended URL `http://web.archive.org/web/19970126045828/http://www.archive.org/` would be the URL for the record of the Internet Archive home page HTML file (`http://www.archive.org/`) archived on January 26, 1997 at 4:58 a.m. and 28 seconds (1997/01/26 at 04:58:28). The date indicated by an extended URL applies to a preserved instance of a file for a given URL, but not necessarily to any other files linked therein. Thus, in the case of a page constituted by a primary HTML file and other separate files (e.g., files with images, audio, multimedia, design elements, or other embedded content) linked within that primary HTML file, the primary HTML file and the other files will each have their own respective extended URLs and may not have been archived on the same dates.
6. Attached hereto as Exhibit A are true and accurate copies of browser screenshots of the Internet Archive's records of the archived files for the URLs and the dates specified in the attached coversheet of each printout.





7. Attached hereto as Exhibit B are true and accurate copies of printouts of the Internet Archive's records of the archived files for the URLs and the dates specified in the attached coversheet of each printout.
8. I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct.

DATE: 08/29/2024

Nathaniel Frank-White

Nathaniel E Frank-White

Please see attached
All Purpose
Jurat form
for additional
Notary Events



EXHIBIT A





INTERNET ARCHIVE

waybackmachine

https://www.redbooks.ibm.com/abstracts/sg247206.html

Go

JUL

JUL

AUG

39 captures

23 May 2006 - 19 Apr 2007

2006

2007

2008

About this capture

IBM

Country/region [select]

Redbooks

Search

Home

Business services

IT services

Products

Support & downloads

My IBM

IBM Redbooks > Application Development

IBM Redbooks®

Drafts

Redbooks

Redpapers

Technotes

Redbooks Domains

Residencies

Workshops

Additional Materials

How to order

About Redbooks

Contact us

Mailing list

RSS feeds

Related links

• IBM Publications

• Technical training

• Developers

• IBM Business Partners

• IBM Press books

Redbooks®

Abstract

Implementing CICS Web Services

Download PDF (5.3 MB)

View as HTML

Order Hardcopy

Get Adobe® Reader®

Tips for viewing

Abstract

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, Web services security (WS-Security) 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.

This new version of the book includes a new chapter on basic cryptography concepts and new security scenarios based on the latest WS-Security support provided by CICS TS V3.1. It shows how CICS Web services can be secured using a combination of 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.

Table of Contents

Part 1. Introduction

Chapter 1. Overview of Web services

Chapter 2. CICS support for Web services

Part 2. Web service configuration

Chapter 3. Web services using HTTP

Chapter 4. Web services using WebSphere MQ

Chapter 5. Connecting CICS to the service integration bus

Part 3. Security management

Chapter 6. Elements of cryptography

Chapter 7. Securing Web services

Chapter 8. Security scenarios

Chapter 9. Security scenarios using CICS WS-Security support

Part 4. Transaction management

Chapter 10. Introduction to Web services: Atomic transactions

Chapter 11. Enabling atomic transactions

Chapter 12. Transaction scenarios

Appendix A. Sample handler programs

Appendix B. How the DES, AES, and HMAC algorithms work

Profile

Publish Date

12 December 2006

Last Update

29 January 2007

Rating: Not yet rated

[Rate this book](#)

Author(s)

Mike Eubels

Nigel Williams

Luis Aused Lopez

Robert Herman

Grant Ward Able

Paolo Chieragatti

Tommy Joergensen

Steve Wall

ISBN

0738494534

IBM Form Number

SG24-7206-01

Number of pages

578

E-mail this page

Print this page

About IBM

Privacy

Contact

Terms of use



EXHIBIT B





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 Ebberts

ibm.com/redbooks

Redbooks







International Technical Support Organization

Implementing CICS Web Services

December 2006

SG24-7206-01



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

Second Edition (December 2006)

This edition applies to CICS Transaction Server Version 3.1.

© Copyright International Business Machines Corporation 2006. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.



Contents

Notices	xi
Trademarks	xii
Preface	xiii
The team that wrote this redbook	xiv
Become a published author	xvii
Comments welcome	xvii
Summary of changes	xix
December 2006, Second 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 TCIPSERVICE	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)	158



6.2.3 AES	159
6.3 Public key (or asymmetric) cryptography	160
6.3.1 RSA	161
6.4 Hash functions	164
6.5 Message authentication codes	166
6.6 Digital signatures	168
6.6.1 Using RSA for digital signatures	171
6.6.2 Using DSA for digital signatures	171
6.6.3 Using ECDSA for digital signatures	173
6.6.4 Comparing RSA with DSA for digital signatures	173
6.7 Public key digital certificates	173
6.7.1 tbsCertificate	175
6.7.2 Standard extensions for X.509 V3 digital certificates	180
6.7.3 Certification paths	182
6.8 Certificate revocation lists	184
6.8.1 Extensions for entries in a CRL	186
6.8.2 Extensions for a CRL	187
6.8.3 Security considerations when using digital certificates	188
6.9 Diffie-Hellman key agreement protocol	189
6.10 Transport Layer Security (TLS) 1.0 protocol	191
6.10.1 TLS overview	192
6.10.2 Cipher suites	193
6.10.3 Alert protocol	197
6.10.4 Handshake protocol	198
6.11 Cryptographic hardware	209
6.11.1 CP Assist for Cryptographic Functions (CPACF)	209
6.11.2 Crypto Express2 Feature (CEX2)	209
6.12 Integrated Cryptographic Service Facility	210
6.12.1 Cryptographic hardware requirements for CICS WS-Security	211
Chapter 7. Securing Web services	213
7.1 Traditional CICS security	214
7.1.1 CICS user IDs	214
7.2 Security exposures	215
7.3 Transport security	217
7.3.1 HTTP transport	217
7.3.2 WebSphere MQ transport	224
7.4 SOAP message security	226
7.4.1 CICS and SOAP message security	229
7.4.2 WebSphere and SOAP message security	239
7.5 Comparison of transport versus SOAP message security	241
7.6 Securing CICS Web services using the service integration bus	242
7.7 Identity assertion	244



7.7.1 Trust token model	245
7.7.2 Presumed trust model	245
Chapter 8. Security scenarios	247
8.1 Preparation	248
8.1.1 Software checklist	248
8.1.2 Definition checklist	249
8.2 Basic security configuration	250
8.2.1 Setting up basic security configuration	251
8.2.2 Testing the basic security configuration	251
8.3 Setting the user ID on a URIMAP definition	252
8.3.1 Defining the URIMAP	253
8.3.2 Permitting access to user ID CICSNW	256
8.3.3 Testing user ID on URIMAP resource definition	256
8.4 Enabling SOAP message security with HTTP	257
8.4.1 Configuring the service requester	258
8.4.2 Configuring CICS	265
8.4.3 Testing SOAP message security	270
8.4.4 SOAP fault messages	271
8.5 Enabling SSL/TLS	273
8.5.1 Creating a key ring and certificates on z/OS for CICS	274
8.5.2 Enabling an SSL/TLS connection from WebSphere	276
8.5.3 Configuring CICS support for SSL/TLS	281
8.5.4 Testing SSL/TLS	282
8.6 Enabling SOAP message security with WMQ	284
8.6.1 Configuring CICS to use WMQ	285
8.6.2 Configuring the service requester	287
8.6.3 Header processing program	287
8.6.4 Configuring the service provider	288
8.6.5 Configuring WebSphere MQ for security	290
8.6.6 Testing security with WMQ	291
Chapter 9. Security scenarios using CICS WS-Security support	295
9.1 Preparation	296
9.1.1 Software checklist	296
9.1.2 Definition checklist	297
9.2 Basic security configuration	299
9.2.1 Creating a RACF key ring	299
9.2.2 Specifying the security SIT parameters	300
9.2.3 Testing the basic security configuration	300
9.2.4 Configuring the pipeline	301
9.2.5 Setting a user ID on a URIMAP definition	302
9.3 Basic authentication	302



9.3.1	Configuring the service requester for basic authentication	304
9.3.2	Configuring CICS	311
9.3.3	Testing basic authentication	313
9.4	Certificate and key pair generation	319
9.5	Signing a SOAP message.	330
9.5.1	Configuring the service requester for signature processing	332
9.5.2	Configuring CICS for signature processing.	347
9.5.3	Testing the signature scenario	350
9.6	Encrypting a SOAP message	357
9.6.1	Configuring the service requester for encryption	359
9.6.2	Configuring CICS for encryption	370
9.6.3	Testing the encryption scenario	372
Part 4.	Transaction management.	379
Chapter 10.	Introduction to Web services: Atomic transactions	381
10.1	Beginner's guide to atomic transactions	382
10.1.1	What is a classic transaction.	383
10.1.2	Mapping from classic transactions to WS-Atomic Transaction	387
10.2	WS-Addressing	390
10.2.1	Endpoint references	391
10.2.2	Message information headers.	394
10.2.3	SOAP binding for endpoint references	396
10.3	WS-Coordination	397
10.3.1	Coordination service	398
10.3.2	CreateCoordinationContext.	399
10.3.3	CreateCoordinationContextResponse	401
10.3.4	Register.	404
10.3.5	Register response.	406
10.3.6	Two applications with their own coordinators	406
10.3.7	Addressing requirements for WS-Coordination message types	408
10.4	WS-Atomic Transaction.	408
10.4.1	Completion protocol	410
10.4.2	Two-Phase Commit protocol.	411
10.4.3	Two applications with their own coordinators (continued).	413
10.4.4	Addressing requirements for WS-AT message types	414
10.4.5	CICS TS V3.1 and resynchronization processing.	415
Chapter 11.	Enabling atomic transactions	419
11.1	Enabling atomic transactions in CICS.	420
11.1.1	CICS to CICS configuration	420
11.1.2	More elaborate CICS to CICS configuration	430
11.2	Enabling atomic transactions in WebSphere.	432



Chapter 12. Transaction scenarios	435
12.1 Introduction to our scenarios	436
12.1.1 Software checklist	437
12.1.2 Definition checklist	437
12.2 The simple atomic transaction scenario	439
12.2.1 Setting up CICS for the simple scenario	441
12.2.2 Creating the AtomicClient and ITSO.ORDER table	445
12.2.3 Testing the simple scenario	461
12.3 The daisy chain atomic transaction scenario	482
12.3.1 Setting up CICS for the daisy chain scenario	484
12.3.2 Creating DispatchOrderAtomic and the ITSO.DISPATCH table	486
12.3.3 Testing the daisy chain scenario	489
12.4 Transaction scenario summary	496
Part 5. Appendixes	497
Appendix A. Sample handler programs	499
A.1 Sample message handler program - CIWSMSGH	500
A.2 Sample header processing program - CIWSSECH	505
A.3 Sample handler program - SNIFFER	511
A.4 Sample XML parser program - MYPARSER	522
A.5 Sample header processing program - CIWSSECR	527
A.6 Sample header processing program - CIWSSECS	534
A.7 Sample header processing program - WSATHND	539
Appendix B. How the DES, AES, and HMAC algorithms work	545
B.1 How DES works	546
B.2 How AES works	549
B.3 How the HMAC algorithm of FIPS PUB 198 works	554
Abbreviations and acronyms	557
Related publications	559
IBM Redbooks	559
Other publications	559
Online resources	560
How to get IBM Redbooks	560
Help from IBM	560
Index	561



x Implementing CICS Web Services



Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.



Trademarks

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

@server®	CICS®	Rational®
eServer®	CICSplex®	Redbooks™
Redbooks (logo)  ™	DB2®	RACF®
developerWorks®	IBM®	System z™
ibm.com®	IMS™	System z9™
z/OS®	MQSeries®	Tivoli®
zSeries®	MVST™	VTAM®
z9™	OS/390®	WebSphere®
Candle®	Parallel Sysplex®	

The following terms are trademarks of other companies:

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

NOW, and the Network Appliance logo are trademarks or registered trademarks of Network Appliance, Inc. in the U.S. and other countries.

Enterprise JavaBeans, EJB, Java, Java Naming and Directory Interface, JavaBeans, JavaServer, JDBC, JDK, JSP, JVM, J2EE, Sun, Sun Microsystems, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Internet Explorer, Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.



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.

This book concentrates on implementation specifics such as security, transactions, and availability. The companion redbook *Developing CICS Web Services* (SG24-7126) presents detailed information on developing CICS Web services.



The team that wrote this redbook

This International Technical Support Organization (ITSO) redbook (second 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 redbook. 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 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.

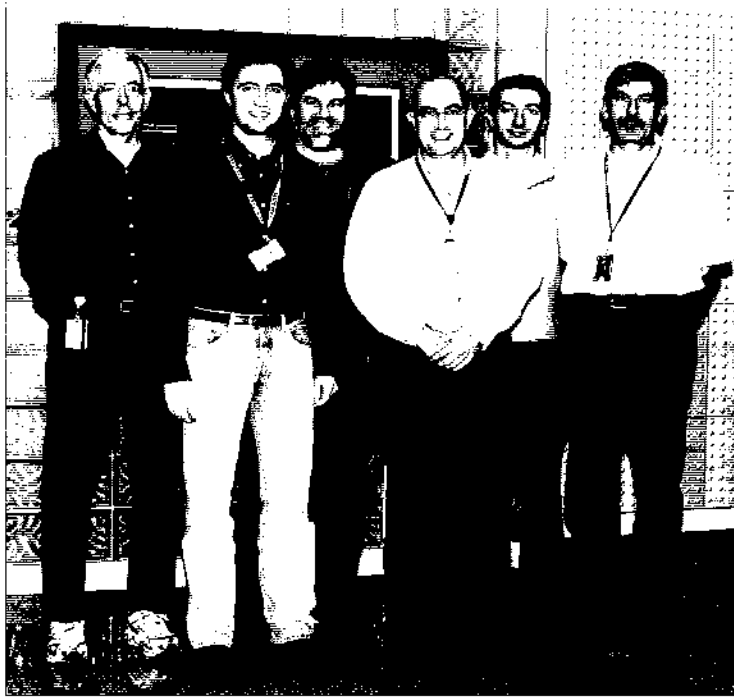
Luis Aused Lopez is an IT Specialist for IBM Global Services in Spain, working in Business Consulting Services (BCS) in the travel and transportation sectors. As an assignee he works in the zSeries Benchmark Center in IBM Montpellier. He has worked for IBM for over ten years. During this time, Luis has 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 and holds a degree in Physics from Complutense University, Madrid, Spain.

Robert Herman is a Senior IT Specialist, Systems Management Integrator with IBM Global Services in Endicott, New York. He has 27 years of experience supporting CICS and related products for a variety of IBM internal and external customer accounts. Bob has worked on several IBM Redbooks including *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.2, SG24-6284*. He holds a Bachelor's degree in mathematics from Grinnell College and a Master's degree in computer science from Iowa State University.

Mike Ebberts is a certified Consulting IT Specialist in the ITSO Poughkeepsie Center. He has spent 33 years with IBM doing technical support and education for mainframe systems.

The first edition was produced by a team of specialists in IBM Montpellier, France. A picture of the first edition team follows.





The team in the foyer at IBM Montpellier

Robert Herman (see biography on previous page).

Luis Aused Lopez (see biography on previous page).

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.

Nigel Williams, project leader (see biography on previous page).

Paolo Chieragatti 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 (missing from the photograph) 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 Redbook 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'll 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.

Summary of Changes
for SG24-7206-01
Implementing CICS Web Services
as created or updated on January 29, 2007.

December 2006, Second Edition

This revision reflects the addition, deletion, or modification of new and changed information described below.

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 9, “Security scenarios using CICS WS-Security support” on page 295 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” on page 545 provides a detailed explanation of how the algorithms work.



xx Implementing CICS Web Services





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.



2 Implementing CICS Web Services



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 schemas 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.

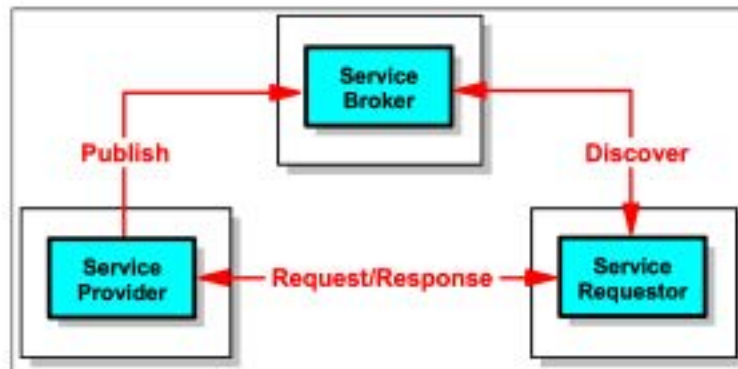


Figure 1-1 Service-oriented architecture components and operations

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.

6 Implementing CICS Web Services



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*, SG24-0303.

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 interoperability 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 1-2 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.

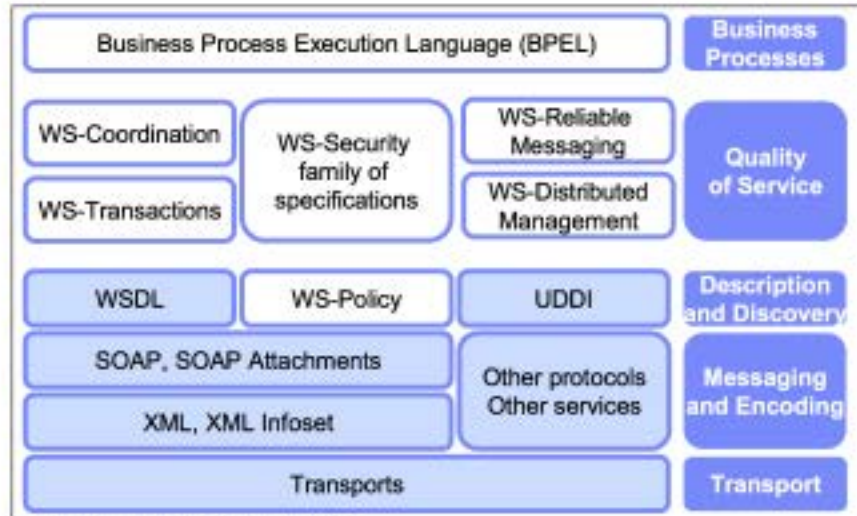


Figure 1-2 Web services standards

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 1-3 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.

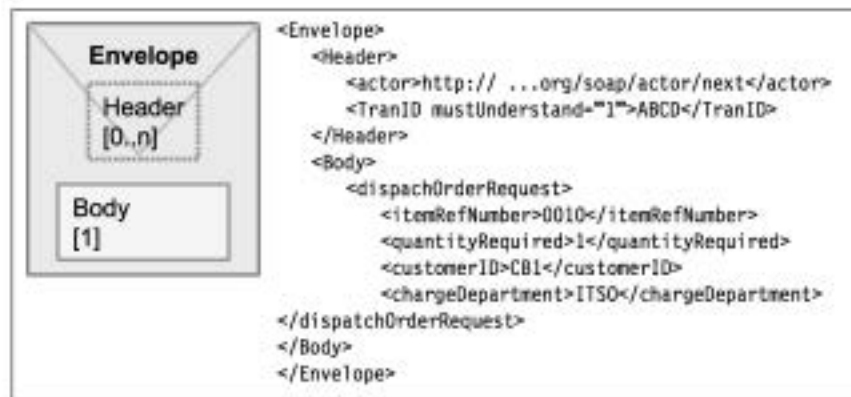


Figure 1-3 Example of a simple SOAP message

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>
  <trans:TransID mustUnderstand="1">ABCD</trans:TransID>
</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 TransID 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 predefined 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 1.1 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 fault codes, 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 <i>message-oriented</i> style. This is a very flexible communication style that provides the best interoperability. The message body is any legal XML as defined in the <code>types</code> section of the WSDL document. |
| RPC | The <i>remote procedure call</i> 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 <i>SOAP encoding</i> enables marshaling/unmarshalling of values of data types from the SOAP data model. This encoding is defined in the SOAP 1.1 standard. |
| Literal | The <i>literal</i> 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 1-4 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 operation bindings. Each of these operation bindings is composed of one protocol extension, where the action and style are defined, and one to three message 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 operation 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.



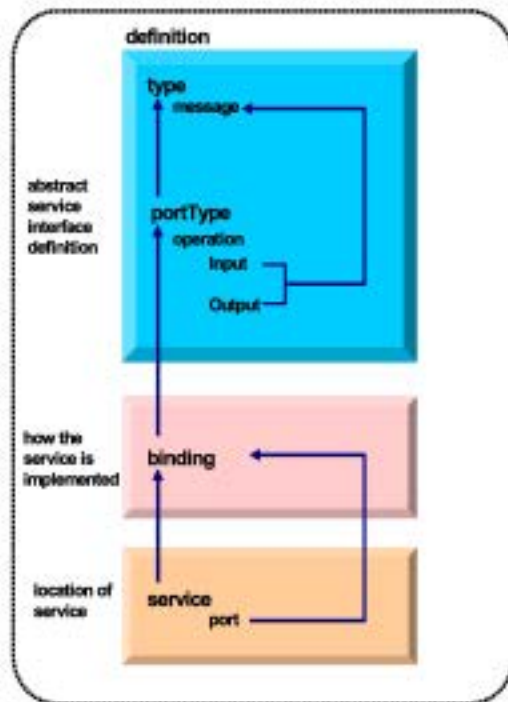


Figure 1-4 WSDL elements and relationships

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:resns="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"/>

```

22 Implementing CICS Web Services



```

</message>
<message name="dispatchOrderRequest">
  <part element="reqns:dispatchOrderRequest" name="RequestPart"/>
</message>
<portType name="dispatchOrderPort">
  <operation name="dispatchOrder">
    <input message="tns:dispatchOrderRequest" name="DFH0X0DSRequest"/>
    <output message="tns:dispatchOrderResponse" name="DFH0X0DSResponse"/>
  </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="DFH0X0DSRequest">
      <soap:body parts="RequestPart" use="literal"/>
    </input>
    <output name="DFH0X0DSResponse">
      <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 <i>this namespace</i> (tns) prefix is used as a convention to refer to the current document. Do not confuse it with the XSD <i>target namespace</i> , 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)/>
  </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 <i>input</i> message only.
Request-response	The port receives a message and sends a correlated message. There is an <i>input</i> message followed by an <i>output</i> message.
Solicit-response	The port sends a message and receives a correlated message. There is an <i>output</i> message followed by an <i>input</i> message.
Notification	The port sends a message. There is an <i>output</i> 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 `DFH0X00SRequest` and the output message `DFH0X00SResponse`. 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="DFH0X00SRequest"/>
    <output message="tns:dispatchOrderResponse" name="DFH0X00SResponse"/>
  </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 DFHOX00SRequest and the output message DFHOX00SResponse, 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="DFHOX00SRequest">
```



```

        <soap:body parts="RequestPart" use="literal"/>
    </input>
    <output name="DFHXODSResponse">
        <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.

32 Implementing CICS Web Services





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 *webbind file*. When your application runs, CICS uses the *webbind 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 TCPIP SERVICE 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.

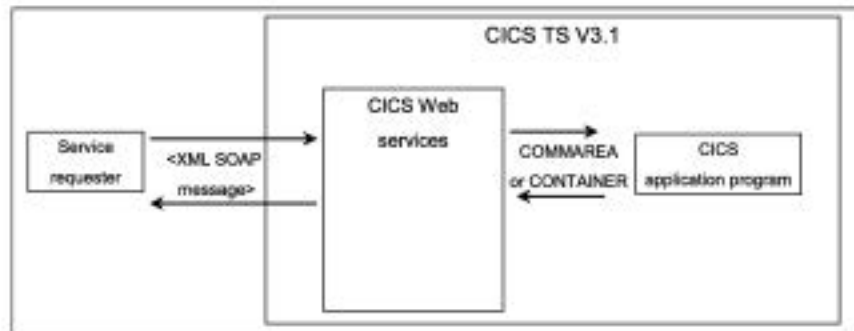


Figure 2-1 CICS as a service provider

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 redbook 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 TCPIP SERVICE 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 3c.

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.

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 TCPIPSEVRDEF 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.

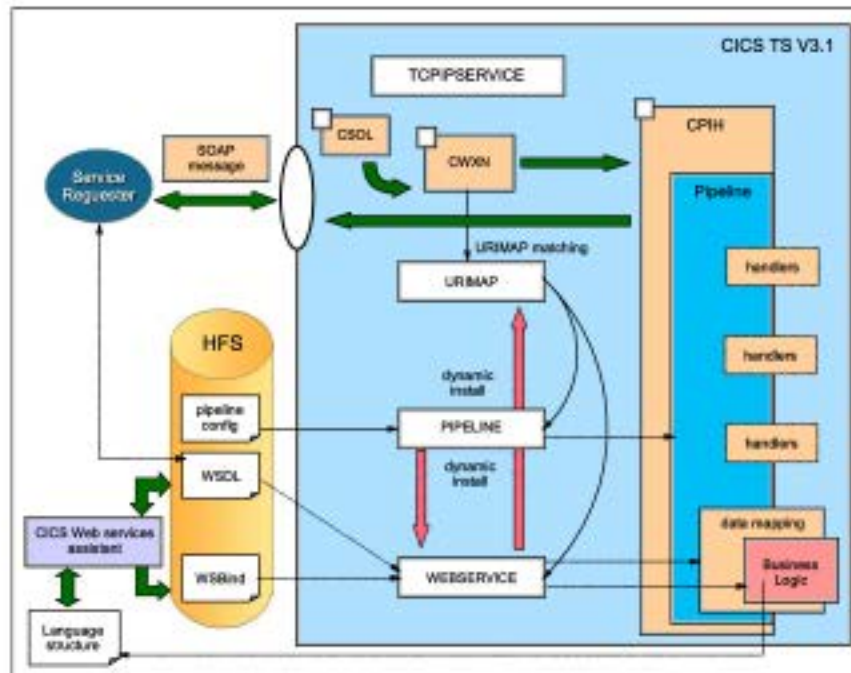


Figure 2-2 Web service run-time service provider processing

The CICS-supplied sockets listener transaction (CSOL) monitors the port specified in the TCPISERVICE resource definition for incoming HTTP requests. When the SOAP message arrives, CSOL attaches the transaction specified in the TRANSACTION attribute of the TCPISERVICE 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.



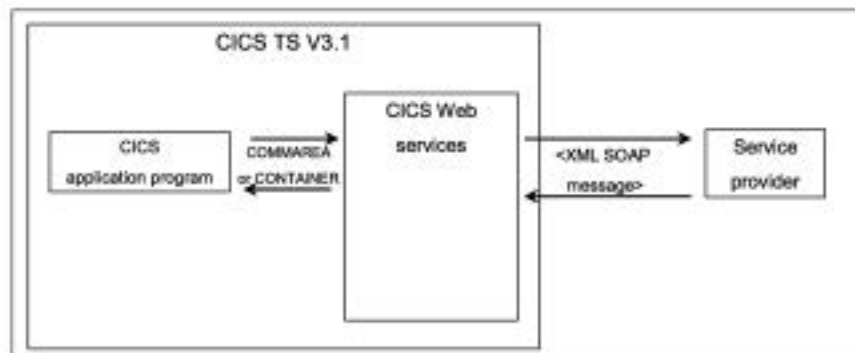


Figure 2-3 CICS as a service requester

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 1 to 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: 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.



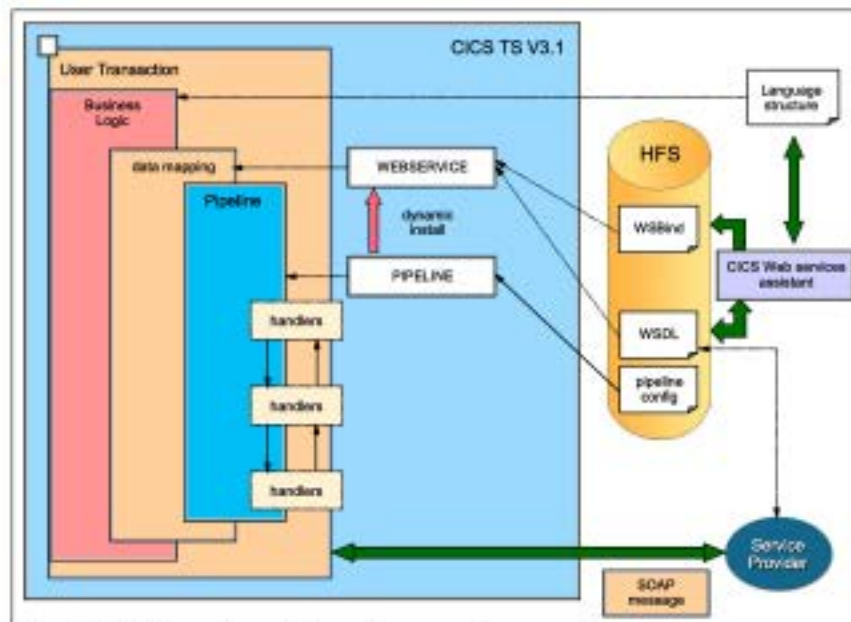


Figure 2-4 Web service run-time service requester processing

When the service requester issues the EXEC CICS INVOKE WEBSERVICE command, CICS uses the information found in the wbind 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).

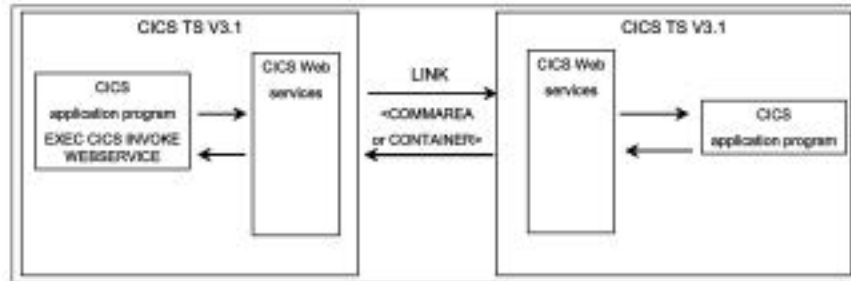


Figure 2-5 Invoking a CICS Web service using local optimization

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 TCPIP SERVICE 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.

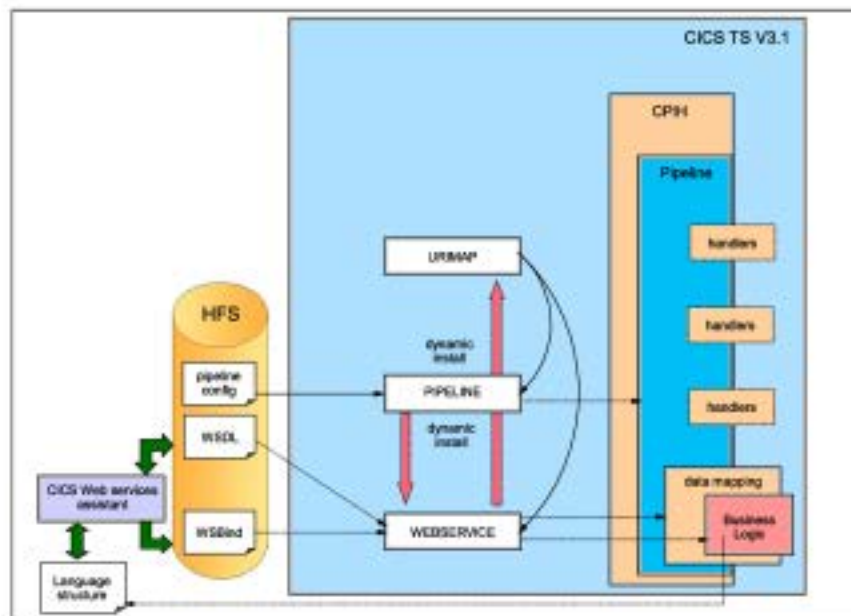


Figure 2-6 URIMAP resource relationships

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.

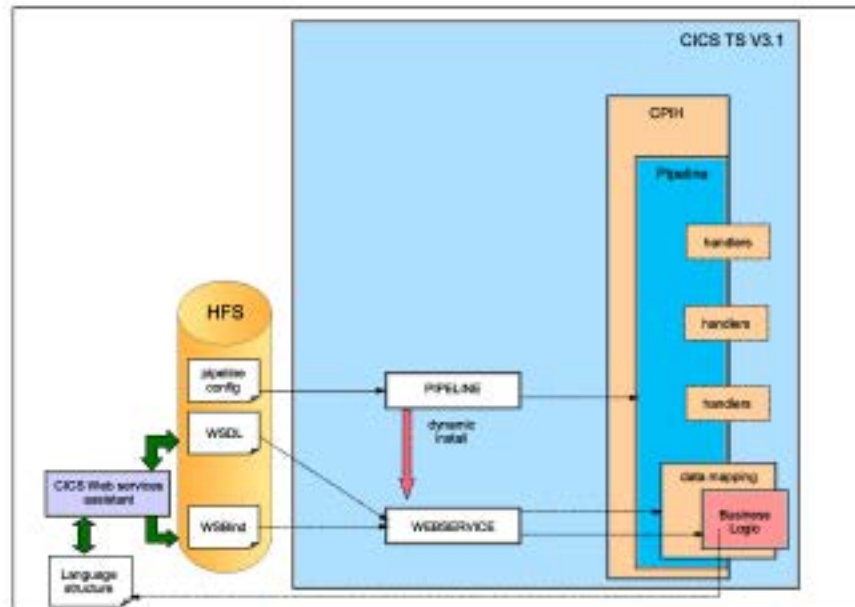


Figure 2-7 PIPELINE resource relationships

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 TCPIP SERVICE 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 `basicssoap11provider.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/http/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/http/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.

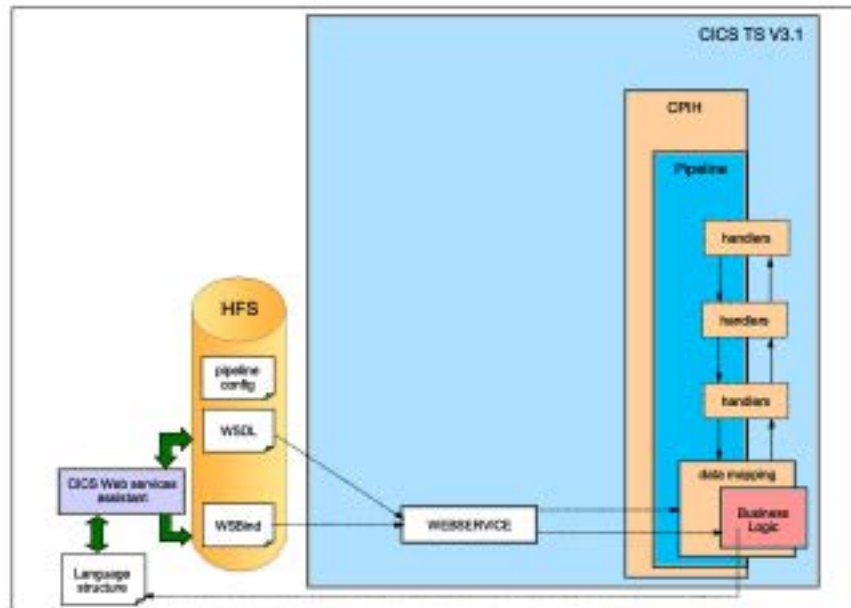


Figure 2-8 WEBSERVICE resource relationships

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 CFMT INQUIRY 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 TCPIP SERVICE

A TCPIP SERVICE 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 TCPIP SERVICE 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 TCPIP SERVICE command

2.4.5 Resources checklist

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



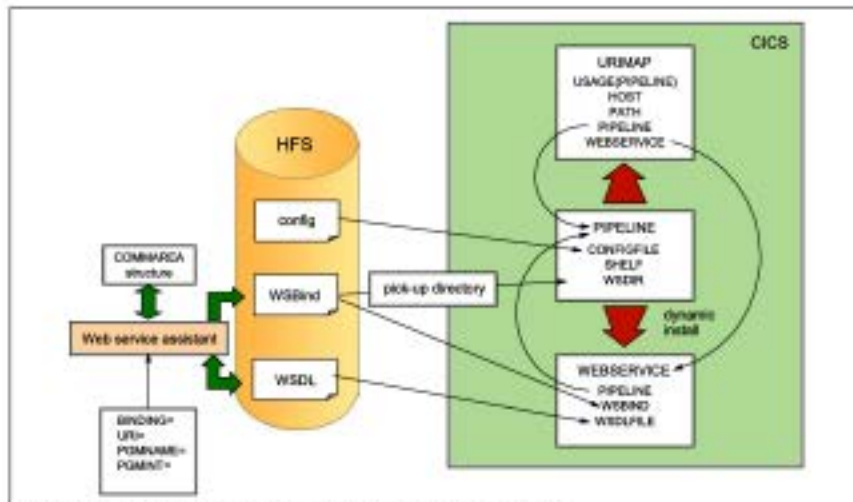


Figure 2-9 CICS Web services resource interrelationships

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.					

56 Implementing CICS Web Services



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 INVHEQ condition. To help you determine which SOAP level applies to the message, container OFFWS-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,MSCLASS-A
// SET Q='-****'
//CWAAPR03 EXEC DFHLS2WS,
// TMPFILE=QDT.LSYSJCL.QDT
//INPJ1.SYSJCL JOB *
//SLIB=//CICS4.LQ.CICS.SDF4SAMP
//REQMEM=DFH0XCP4
//RESPMEM=DFH0XCP4
//LANG=COBOL
//MMNAME=DFH0XCMY
//UR=example4pp/inquireSingle
//GHINT=COMMAREA
//SRINC=/L/example4pp/wsi/nd/inquireSingle.wsi/nd
//SDL=/L/example4pp/wsi/nd/inquireSingle.wsi/nd
/*
```

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 DFHWS2WS. 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,MSCLASS=A
// SET Q'-'
```



```

//C414PROG EXEC DEFWS21 S,
// THPFILE=AGT.NSYS.ICD.AGT
//CNPJT.SYSJT1 DD *
//SLEB=//CICS4LQ.CICS.SDF4SAMP
REQNM=CPYBK1
RFSPEFH=CPYBK2
LANS=COBOL
PGMNAME=DFHDXCMV
URL=exampleapp/inquireSingle
PGMINT=COMMAREA
WSBIND=/L/exampleapp/wsbnd/inquireSingle.wsbnd
WSDL=/L/exampleapp/wsd/inquireSingle.wsd1
/*

```

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.



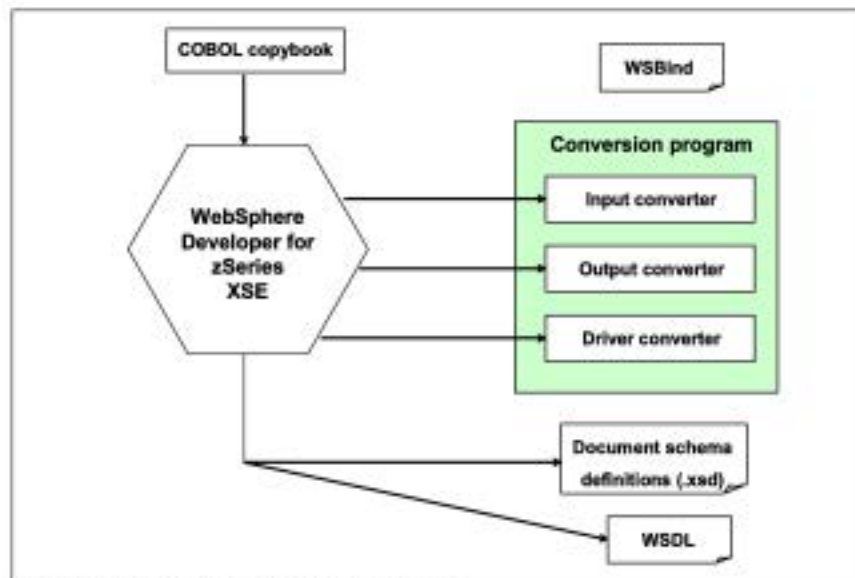


Figure 2-10 WebSphere Developer for zSeries

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.



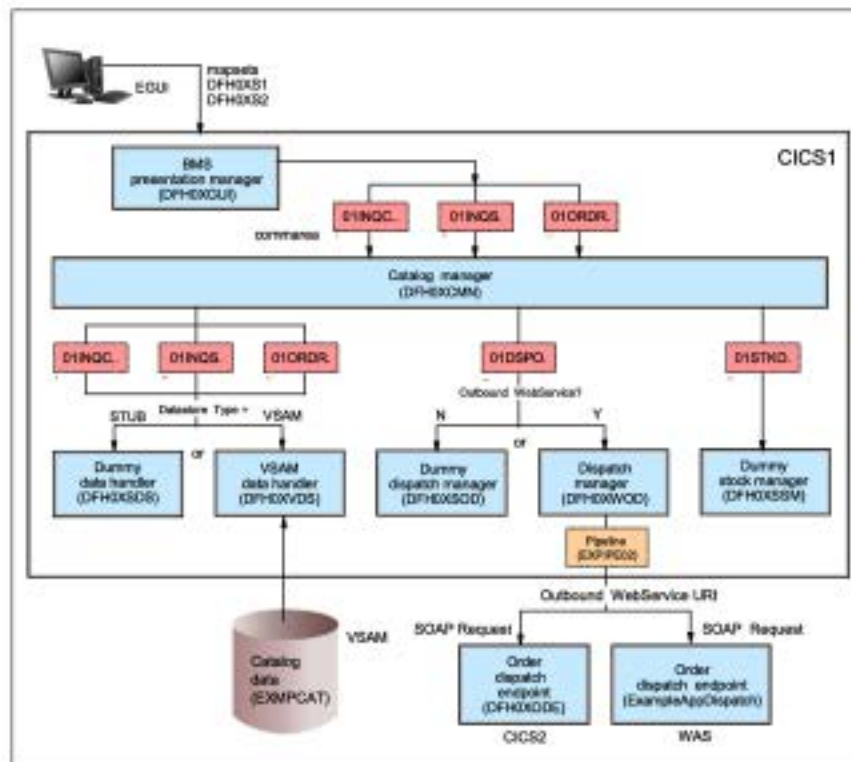


Figure 2-11 Basic catalog manager application structure

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. DFHXOWOD is a service requester that invokes a remote order dispatch endpoint, and DFHXOSOD is a dummy program that simply returns valid responses to its caller. There are two equivalent order dispatch endpoints: DFHIQXODF 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 (DFHXSSM) 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 (DFHXODE).

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



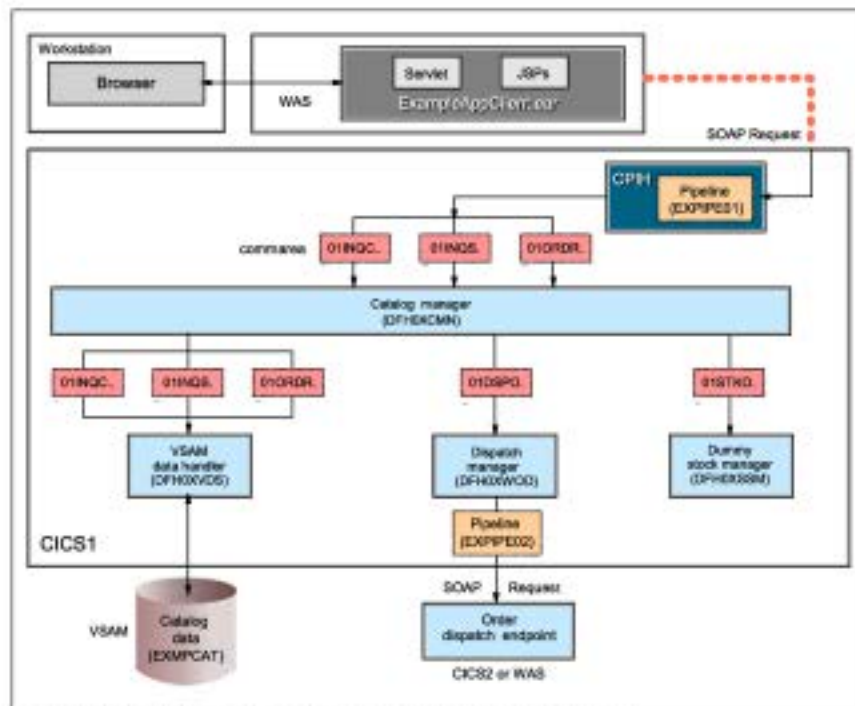


Figure 2-12 Web services catalog manager application structure

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.



72 Implementing CICS Web Services





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

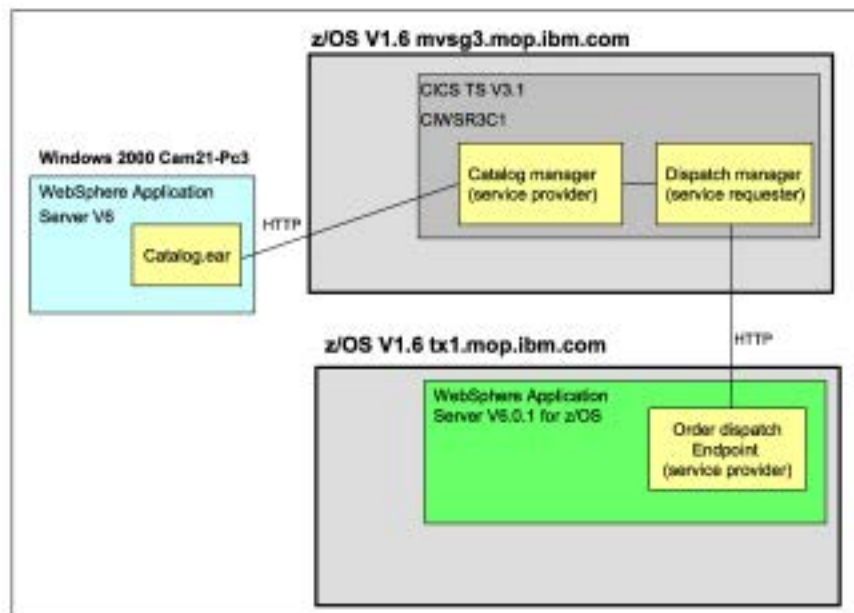


Figure 3-1 Software components: Web services using HTTP transport



We do not provide information on 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) ► CIWSM3GH (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.mcp.ibm.com	lx1.mcp.ibm.com
IP address	0.100.193.167	0.100.193.122
TCP/IP port	13331	13880
Job name	CIWSH3C1	CI1GHS1S



Value	CICS TS	WebSphere Application Server
APPLID	A6POR3C1	
TCPIP SERVICE	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.

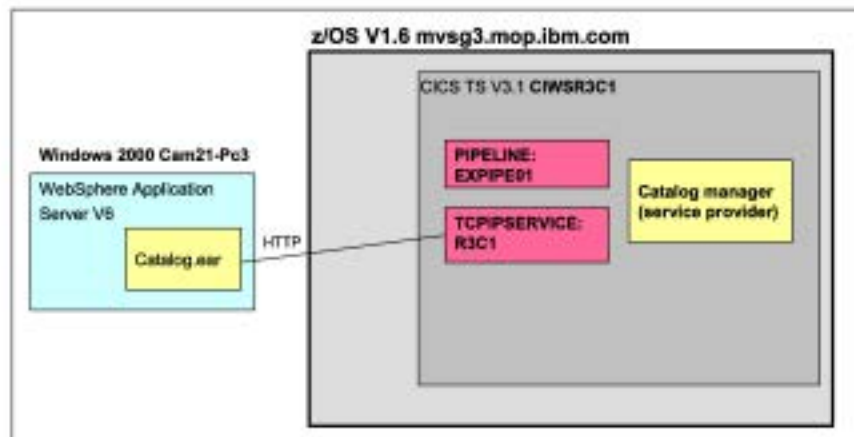


Figure 3-2 CICS as a service provider



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 on 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 TCPIPSEVRVICE 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=AFS
```

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

```
/CICS/R3C1/config  
/CICS/R3C1/shell  
/CICS/R3C1/webmgr/provider  
/CICS/R3C1/webmgr/requester
```

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

Configuring the TCPIPSEVRVICE definition

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

```
CED4 DEFINE TCPIPSEVRVICE(R3C1) GROUP(R3C1)
```

We defined the R3C1 TCPIPSEVRVICE 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      ==>
Ipadddress     ==>
SOcketclose    ==> No                  No | 0-240000 (HHMMSS)
MaxdataLEN     ==> 000032                3-524288
SECURITY
SSI            ==> 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/http/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/http/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 8.2, "Basic security configuration" on page 250.

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 500.

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

```

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.

```

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

```

010300      EXEC CICS
010400          GET CONTAINER('DFHWS-WEBSERVICE')
010500          SET(ADDRESS OF CONTAINER-DATA)
010600          FLENGTH(CONTAINER-LEN)
010700      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

```

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-MESSERVICES = 'placeOrder'
012900          MOVE 'ORDR' TO CA-TRANID
013000          PERFORM CHANGE-CONTAINER THRU END-CHANGE-CONTAINER
013100      END-IF
013200      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

```

013500      CHANGE-CONTAINER.
013600          MOVE CA-TRANID TO CONTAINER-DATA(1:4)
013800          EXEC CICS PUT CONTAINER( DFHWS-TRANID )
013900              FROM( CONTAINER-DATA )
014000              LENGTH( CONTAINER-LEN )
014100          END-EXEC.
015000      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(2301)
```

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) ==>
              ==>
              ==>
              ==>
Wsdire        : /CIWS/R3C1/wsbind/provider/
(Mixed Case) :
              :

```

SYSID=R3C1 APPLID=A6P0R3C1

Figure 3-4 CEDA DEFINE PIPELINE

Tip: The colons in front of Wsdire 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/ITS0_7206_basicsoap12provider.xml)
  Shelf(/CIWS/R3C1/shelf/)
  Wsdir(/CIWS/R3C1/wsbind/provider/)

SYSID=R3C1 APPLID=A6P0R3C1
```

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 dispatcherOrder, inquireCatalog, inquireSingle and placeOrder.

```
INQUIRE WEBSERVICE
STATUS: RESULTS - OVERTYPE TO MODIFY
webs(InquireCatalog          ) Pip(EXPIPEGL)
  Ins Ltr(5605071 ) Prn(CFHGXCM) Con          Tar(20050408)
webs(InquireSingle          ) Pip(EXPIPEGL)
  Ins Ltr(5605023 ) Prn(CFHGXCM) Con          Jdt(20050408)
webs(placeOrder             ) Pip(EXPIPEGL)
  Ins Ltr(5606025 ) Prn(CFHGXCM) Con          Jdt(20050408)

SYSID=3301 APPLID=46P0R3CL
```

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(R3CL) HOST(*)
PAFH(/example4pp/inquireSingle)
  PIPELINE(EXPIPEGL) TRANSACTION(INQS) USAGE(PIPELINE)
  WEBSERVICE(InquireSingle)
CEDA DEFINE URIMAP(INQC) GROUP(R3CL) HOST(*)
PAFH(/example4pp/inquireCatalog)
  PIPELINE(EXPIPEGL) TRANSACTION(INQC) USAGE(PIPELINE)
  WEBSERVICE(InquireCatalog)
CEDA DEFINE URIMAP(ORDR) GROUP(R3CL) HOST(*)
PAFH(/example4pp/placeOrder)
  PIPELINE(EXPIPEGL) 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(Moanalyzer)
  Scheme(Http)
  Redirecttype( None )
  Tcpipservice()
  Host(*)
  Path(/exampleApp/inquireCatalog)
  Transaction(CPIH)
  Converter()
  Program()
  Pipeline(EXPIPE01)
  Webservice(inquireCatalog)
  Userid()
  Certificate()
  Ciphers()
  Templatenamename()

SYSID=R3C1 APPLID=A6P0R3C1
```

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://can21-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\addnat\src\ears\Catalog.ear

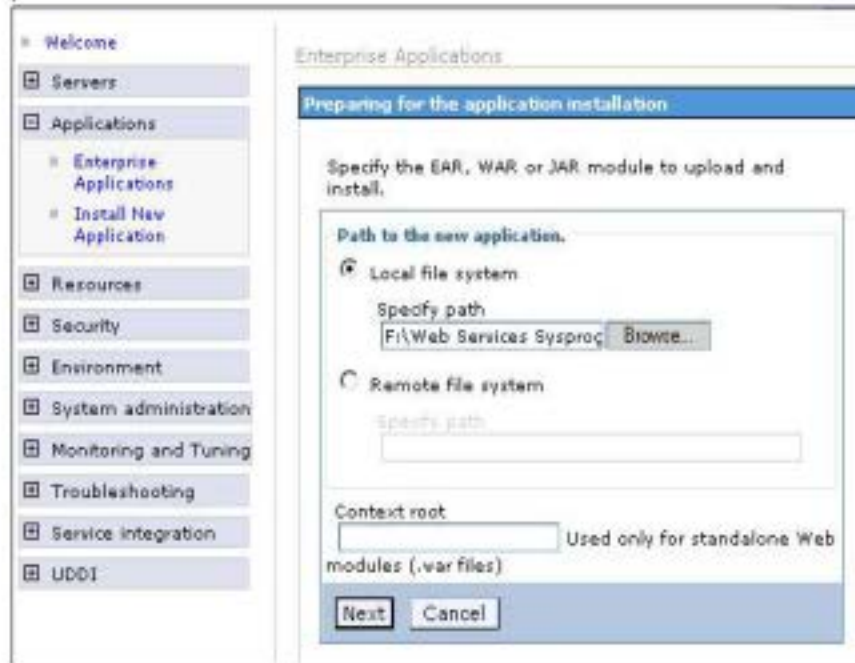


Figure 3-8 WebSphere administrative console - Install new application

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 on 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 to 180 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.

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 on 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://can21-pc11:9080/CatalogWeb/Welcome.jsp`



The window shown in Figure 3-10 was displayed.



Figure 3-10 CICS - Catalog application

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

- ▶ 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	
LIST ITEMS	
INQUIRE	Inquire Catalog Service Endpoint
ORDER ITEM	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
SUBMIT	

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 MCGCF*
Tas(C0000002) Tra(CPIH)          Sus Tas Pri( 001 )
Sta(L ) Use(CICSUSER) Uow(BEC6893FAF509305) Hly(RZC6M011)
Tas(C0000003) Tra(INQC)          Sus Tas Pri( 001 )
Sta(L ) Use(CICSUSER) Uow(BEC6893FD077904F) Hly(RZC6M011)
Tas(C0000007) Tra(CPIH)          Sus Tas Pri( 001 )
Sta(L ) Use(CICSUSER) Uow(BEC6893B1D78940) Hly(RZC6M011)
Tas(C0000008) Tra(INQS)          Sus Tas Pri( 001 )
Sta(L ) Use(CICSUSER) Uow(BEC6893568307606) Hly(RZC6M011)
Tas(C0000002) Tra(CPIH)          Sus Tas Pri( 001 )
Sta(L ) Use(CICSUSER) Uow(BEC68932FFC0430B) Hly(RZC6M011)
Tas(C0000003) Tra(ORDR)          Sus Tas Pri( 001 )
Sta(L ) Use(CICSUSER) Uow(BEC68930049906L) Hly(RZC6M011)

SYSID=3331 APPID=AMPCBRC

```

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-WEBSESERVICE and DFHWS-TRANID containers are logged. See Section A.1, “Sample message handler program - CIWSMSGH” on page 500 for more information on the CIWSMSGH program.

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

```

CIWSMSGH: >-----<
CIWSMSGH: Container Name: : DFHWS-WEBSESERVICE
CIWSMSGH: Container content: inquireCatalog
CIWSMSGH: >-----<
CIWSMSGH: Container Name: : DFHWS-TRANID
CIWSMSGH: Container content: INQC
CIWSMSGH: >-----<
CIWSMSGH: Container Name: : DFHWS-WEBSESERVICE
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.

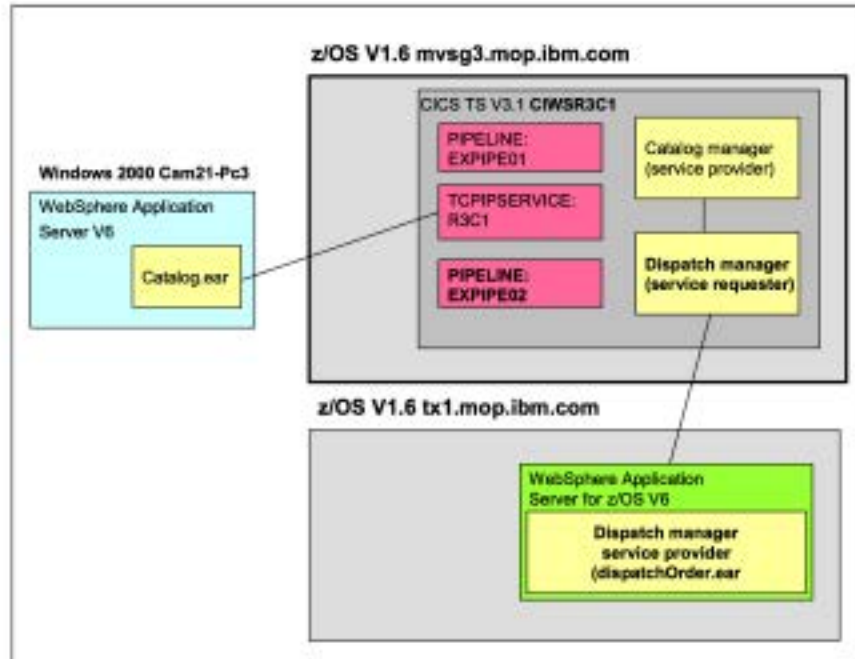


Figure 3-13 CICS as service requester

Figure 3-13 shows the TCPIP SERVICE R3C1 used for inbound HTTP Web service requests. Note that a TCPIP SERVICE 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(R3CL)
```

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

```
SYSCATPG TO MODIFY                                CICS RELEASE = 0640
CEDA DEFINE Pipeline EXPIPE02 ;
Pipeline      : EXPIPE02
Group        : R3CL
Description   --> PIPELINE DEFINITION FOR DISPATCH GGER REQUESTER
Status        --> Enabled           Enabled | Disabled
ConfigFile    --> /CICS/R3CL/config/IT50_7206_basicsoap11requester.xml
(Pixed Case) -->
              -->
              -->
              -->
Shelf         --> /CICS/R3CL/shelf
(Pixed Case) -->
              -->
              -->
WebDir        : /CICS/R3CL/webdir/requester/
(Pixed Case)  :
              :
```

Figure 3-14 CEDA DEFINE PIPELINE command

- We set the CONFIGFILE attribute to:
/CICS/R3CL/config/IT50_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=A6P0R3C1
```

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 ORDER transaction to 30 seconds:

```
CEDA ALTER TRANSACTION(ORDER) 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

```
DFHP10401 12/15/2005 15:51:12 A6P0R3C1 ORDR The CICS pipeline HTTP transport
mechanism failed to send a response or receive a request because the connection
was closed.
DFHP10503 12/15/2005 15:51:12 A6P0R3C1 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.

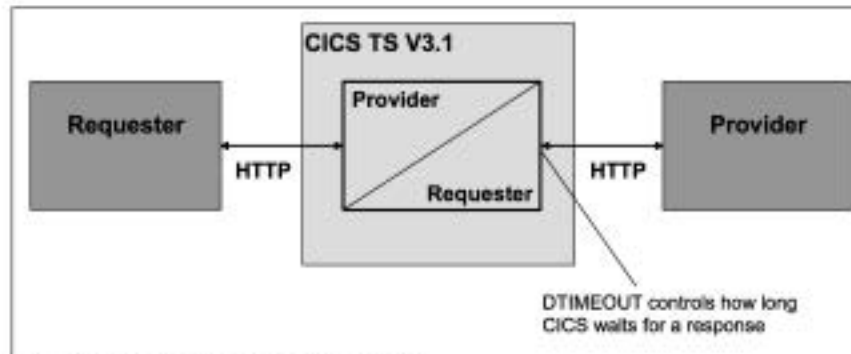


Figure 3-16 CICS timeout considerations

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

```
DFHP10504 12/15/2005 15:55:33 A6P0R3C1 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 ==> DFH0XSOS
      Data Store VSAM ==> DFH0XVDS
      Order Dispatch Stub ==> DFH0XSOD
Order Dispatch WebService ==> DFH0XMOD
      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/cics/cics31/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\address\src\catalog
App\c\ation\Configuration\part
F:\Web Services Sysprog\LAN book\address\src\catalog Application\Configuration\
part>ftp mvsg3.sop.ibm.com
Connected to 9.100.193.167.
220-#P01 IBM FTP CS V1R6 at MVSS3.sop.ibm.com, 17:59:13 on 2008-11-24.
220 Connection will close if idle for more than 2 minutes.
User (9.100.193.167:(none)): CICSUJ
331 Send password please.
Password:
230 CICSUJ is logged on. Working directory is "/CICSUJ".
ftp>cd /usr/lpp/cics/cics31/samples/webservices/client
250 EPS directory /usr/lpp/cics/cics31/samples/webservices/client is the
current working directory
ftp>get ExampleAppDispatchOrder.ear
200 Port request OK.
174 Sending data set
/usr/lpp/cics/cics31/samples/webservices/client/Example
AppDispatchOrder.ear
250 Transfer completed successfully.
ftp> 50523 bytes received in 0.03Seconds 1607.13Kbytes/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://cx1.sop.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\addmaf\src\ears\dispatchOrder.ear

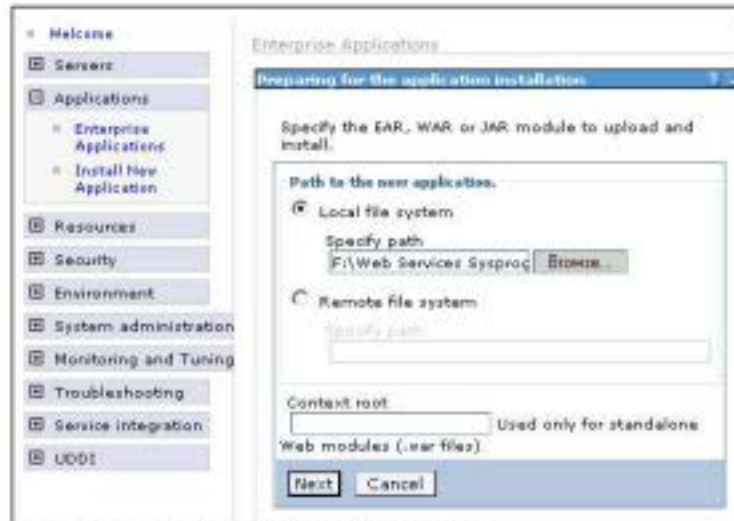


Figure 3-18 WebSphere Administrative console

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**.



Figure 3-19 CICS - Catalog application order window

We received a message back saying "ORDER SUCCESSFULLY 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.

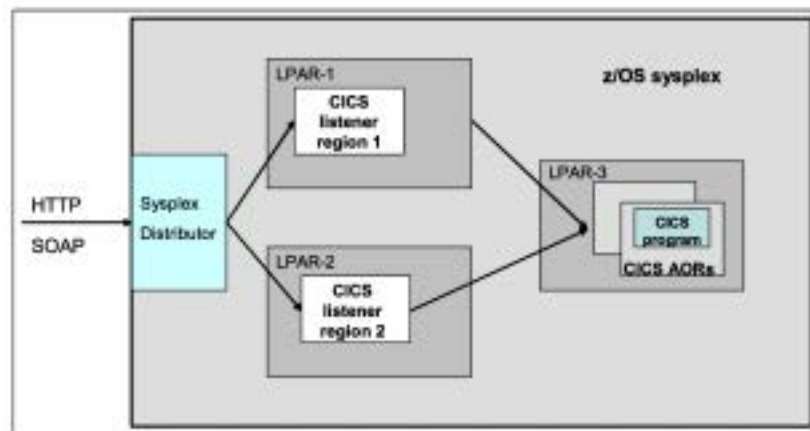


Figure 3-20 High scalability and availability configuration

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.

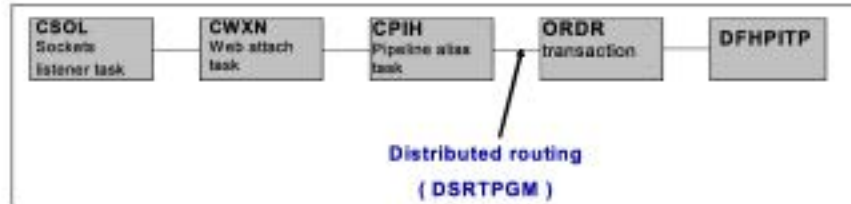


Figure 3-21 Web service provider - Distributed routing

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.

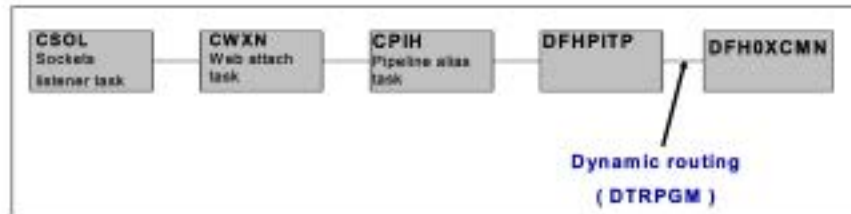


Figure 3-22 Web service provider - Dynamic routing

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.



Figure 3-23 CICS - Catalog application INVREQ

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



```

        Database Type ==> VSAM                STLB|VSAM
        Outbound WebService? ==> YES          YES|NO
        Catalog Manager ==> OFF|OFFCMN
        Data Source Stub ==> OFF|OFFSDS
        Data Source VSAM ==> OFF|OFFVDS
        Order Dispatch Stub ==> OFF|OFFSDO
        Order Dispatch WebService ==> OFF|OFFWUJ
        Stock Manager ==> OFF|OFFSSM
        VSAM File Name ==> EXMPCAT
        Server Address and Port ==>
        Outbound WebService URI ==> https://tcl.wop.ibm.com:13880/exampleApp/dispatch
                                   ==> hOrder
                                   ==>
                                   ==>
                                   ==>

APPLICATION CONFIGURATION DISPLAYED

PF          3 END                                L2 ENCL

```

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: SKYED190E: File not found /services/dispatchOrder."

Example 3-16 Sample DispatchOrder CICS trace

```

PC 0031 PUIS EVENT - REQUEST CNT - TASK-00187 6F MM-000E ICH-.B0C3/00A30A8 RFI-0753F4F7
TIME 12:11.58.41/525962 INTERVAL 00.0000000/00 -002137-
1-0000 30534F41 5020454E 563A456C 76656C67 70652070 60606E7D 3A534F41 5020454C
                                     *<SCA?--ENV:Envelope xmlns:SOAP-ENV=
0020 56302268 7474703A 2F2F7363 68656C6C 732E786D 6C736F6C 70215F72 572F736F
                                     *?="https://schemas.xmlsoap.org/soap
0040 61702F65 6C76656C 6F70652F 22203E3C 534F415D 20454E56 3A426F64 723E3C64
                                     *op/Envelope?*" ><SOAP-ENV:Body><sc
0060 69737061 7461664F 72646572 57657173 64737470 79606C6F 7A3D2768 7474703A
                                     *dispatchOrderRequest xmlns="https
0080 2F2F7363 7265786 516D703E 66417070 26645973 70617461 68417264 56722652
                                     *"/></www.exampleApp.dispatchOrder.*
00A0 65717565 72742361 6F6D223E 3C697462 63626566 6572656E 6D624375 6D626572
                                     *requestion"><iterReferenceNumber"

```



```

00C0 3C31303C 27E974E5 6D526566 6572656E 63654C75 6D626572 3C3C7175 516E7469
                                *<IC>/<!--referenceNumber--><!-- *
00E0 74795265 7173E972 55643E31 3C2F7175 616E7469 74795265 7173E972 55643E3C
                                *tyRequ re>]</quantityRequ re>*>
0100 63757374 6F6D6572 49643F54 6F6D6D79 7070703C 7F637573 746FAD65 7749643F
                                *customerId>10000 </customerId>
0120 3C3C3636 72676544 5573B172 746E6569 747F4954 534F7070 7070703C 55643E3C
                                *changeDepartment>1150 </change
0140 67654465 70617274 63656E74 3C3C2-64 69737061 74636D4- 72646572 52557175
                                *deptDepartment></dispatchOrderRec-
0160 6573743C 3C2F334F 41502045 4C563A12 6F64793C 3C2F534F 41502045 4C563A45
                                *es></SOAP-ENV:Body></SOAP-ENV:
0180 6E76656C 6F70653E
                                *type>*>
P1 0A32 P115 EVENT - REQUESTER CNT - TASK-CC162 4F 404-008C TCR-18003700A00048 RFT-9753F472
                                TIME-17:13:56.4175376594 INTF00A-000000000607 <002142>
1-0000 4577726F 77703430 343A70B3 5756453C 31393045 34704664 6C657059 6F74720B
                                *Error 404: SRVE0190E: File not f
0020 6F756E64 3A202173 55727669 636572- 64697370 6174636D 4F726465 7255
                                *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 CFSF.

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="UTF-8" ?>
<requester_pipeline xmlns="http://www.ibm.com/software/http/cics/pipeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/software/http/cics/pipeline
    requester.xsd">
  <service>
    <service_handler_list>
      <cics_soap_1.1_handlers>
        <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 511. Example 3-18 shows the containers in the requester pipeline as listed by SNIFFER. The container of interest is DFIIWS-URI:

<http://tx1.mop.ibm.com:13880/example.asp/dispatchOrder>

Example 3-18 sample SNIFFER output

```

SNIFFER : *** Start ***
SNIFFER : >
SNIFFER : Container name : DFHFJUNCTION
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-URLS
SNIFFER : Content length : 00000009
SNIFFER : Container content: xmlns:SOAP_ENV="http://schemas.xmlsoap.org/soap/
SNIFFER : envelope/"
SNIFFER : ~~~~~
SNIFFER : Container name : DFIIWS-SOAPLOC
SNIFFER : Content length : 00000004
SNIFFER : Container content:
SNIFFER : ~~~~~
SNIFFER : Container name : DFH-HEADER LIST
SNIFFER : Content length : 00000000
SNIFFER : Container EMPTY
SNIFFER : ~~~~~
SNIFFER : Container name : DFRESPONSE
SNIFFER : Content length : 00000000
SNIFFER : Container EMPTY
SNIFFER : ~~~~~
SNIFFER : Container name : DFHFJUNCTION
SNIFFER : Content length : 00000016
SNIFFER : Container content: SEND REQUEST
SNIFFER : >

```



```

SNCHFFER : Container name      : DFH-SERVICEPLIST
SNCHFFER : Content length      : 00000000
SNCHFFER : Container EMPTY
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-JSFRIE
SNCHFFER : Content length      : 00000008
SNCHFFER : Container content: CICSUSER
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-IRAMIG
SNCHFFER : Content length      : 00000004
SNCHFFER : Container content: 0P0P
SNCHFFER : *****
SNCHFFER : Container name      : DFHREQUEST
SNCHFFER : Content length      : 00000000
SNCHFFER : Container EMPTY
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-REPLY
SNCHFFER : Content length      : 00000793
SNCHFFER : Container content: <SOAP-ENV:Body><dispatchOrderRequest
xmlns="http://www.exampleApp.com:dispatchOrder.Request.com"><itemRefere
nceNumber=10><itemReferenceNumber><quantityRequired></quantityRequired><cust
omerId>Tommy </customerId><changeDepartment>ITSO
</changeDepartment></dispatchOrderRequest></SOAP-Env:Body>
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-URI
SNCHFFER : Content length      : 00000756
SNCHFFER : Container content:
http://tx1.mop.ibm.com:13880/exampleApp/dispatchOrder
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-SOAPACTION
SNCHFFER : Content length      : 00000002
SNCHFFER : Container content: ""
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-OPERATION
SNCHFFER : Content length      : 00000756
SNCHFFER : Container content: dispatchOrder
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-PIPELINE
SNCHFFER : Content length      : 00000000
SNCHFFER : Container content: SOAPPIPE02
SNCHFFER : *****
SNCHFFER : Container name      : DFHWS-DATA
SNCHFFER : Content length      : 00000023
SNCHFFER : Container content: 0010001Tommy ITSO
SNCHFFER : Containers on channel: list ends
SNCHFFER : In a SOAP header processing program.....
SNCHFFER : **** 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.

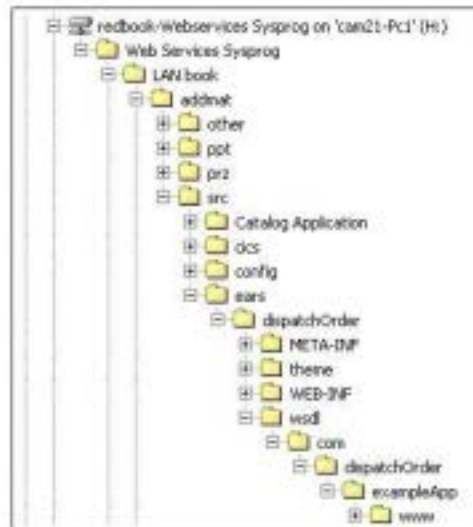


Figure 3-25 ExampleAppDispatchOrder path

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">
```



```

      xsi:type="xsd:string" 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="UFHCOCSSecRest">
          <soap:body parts="RequestPart" use="literal"/>
        </input>
        <output name="UFHCOCSSecResponse">
          <soap:body parts="ResponsePart" use="literal"/>
        </output>
      </operation>
    </binding>
    <service name="dispatchOrderService">
      <port name="dispatchOrderPort" binding="tns:dispatchOrderSoapBinding">
        <soap:address location="http://taxi.wop.ibm.com:11880/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.





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.

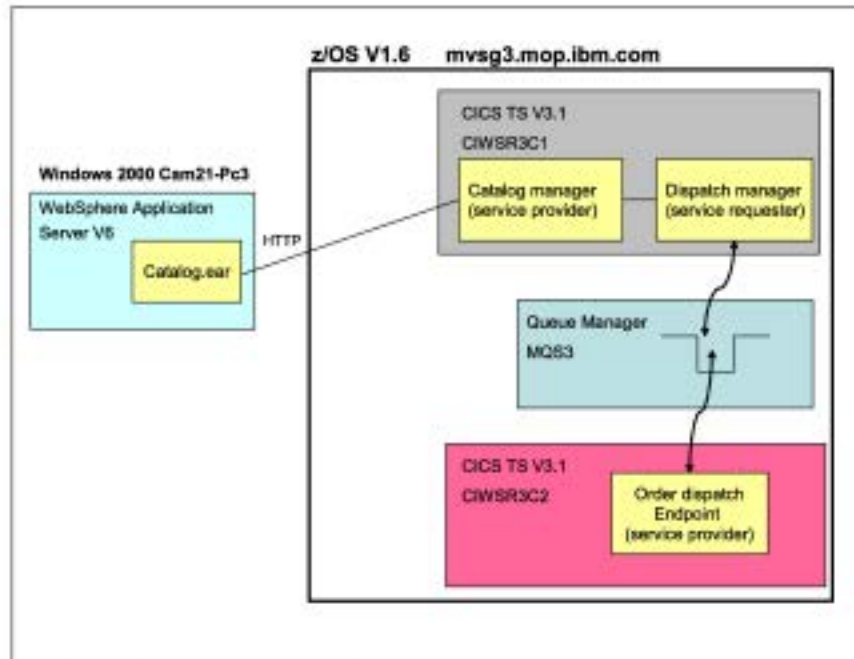


Figure 4-1 Software components: Web services using HTTP and WMQ

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.mcp.ibm.com	mvsg3.mcp.ibm.com
IP address	9.100.193.167	9.100.193.167
TCP/IP port	13301	
Job name	CIWSR3C1	CIWSR3C2
APPLID	A6POR3C1	A6POR3C2
TCPIP SERVICE	R3C1	
Provider PIPELINE	EXPIPE01	EXPIPE03
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 checklist

Value	Queue manager MQS3
Queues	VSG3.R3C2.PIPE3.REQUEST VSG3.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

```
//STEP_LIB 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=CICS.CICS.LSFRLOAD,DISP=SHR
//          DD DSN=CEE.SCECCICS,DISP=SHR
//          DD DSN=CEE.SCEERJN,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
 - INITPAHM=(CSOCPAHM='SN=MQS3.'N=1,IQ=VSG3.H3C'.IN IQ')



- We updated the SIT parameters on CICS region CIW5R3C2:
 - MOCOMN=YES
 - INITPARM=(CSQCPARM='SN=MQS3.*N=1,IQ=VSG3.R3C2.IN TQ')
- 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:


```
CFDA ADD GROUP(CSQCAT1) TO LIST(L1STR3C1)
CEDA ADD GROUP(CSQK301) TO LIST(L1STR3C1)
```
- 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:


```
CE04 ADD GROUP(CSQCAT1) TO LIST(L1STR3C2)
CFD4 ADD GROUP(CSQK301) TO LIST(L1STR3C2)
```

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

```
//CHQUEUE JOB 1,CICS,TIME=1440,NOTIFY=SYSOUT,REGION=RM,
//          CLASS=A,MSGCLASS=X,MSCLEVEL=(1,1)
//
//CSQJTL EXEC PGM=CSQJTL,PARM='MQS3'
//STEP_1B DD DSN=MQM.SCSQLOAD,DISP=SHR
//          DD DSN=MQM.SCSQANLE,DISP=SHR
//          DD DSN=MQM.SCSQJTL,DISP=SHR
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSJN DD *
//SYNAMP DDNAME(CMDINP)
//
//CMDINP DD *
*
DEFINE Q_3CA.(VSG3.R3C2.PIPE3.REQUEST) -
DESCR( QUEUE SOAP [INCOMING REQUEST] ) -
PROCESS(VSG3.R3C2.PROCESS) -
TRIGGER
TRIGTYPE(FIRST) -
CNCIQ( VSG3.R3C2.INITQ ) -
*
DEFINE Q_3CA.(VSG3.R3C2.PIPE3.RESPONSE) -
```



```
DFSCR( QUFUF 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(CPIQ)
```

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 CPIQ (the SOAP MQ inbound listener transaction) which means that this transaction will be started in CICS when a service request arrives. CPIQ 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 DFHDODE using WMQ.



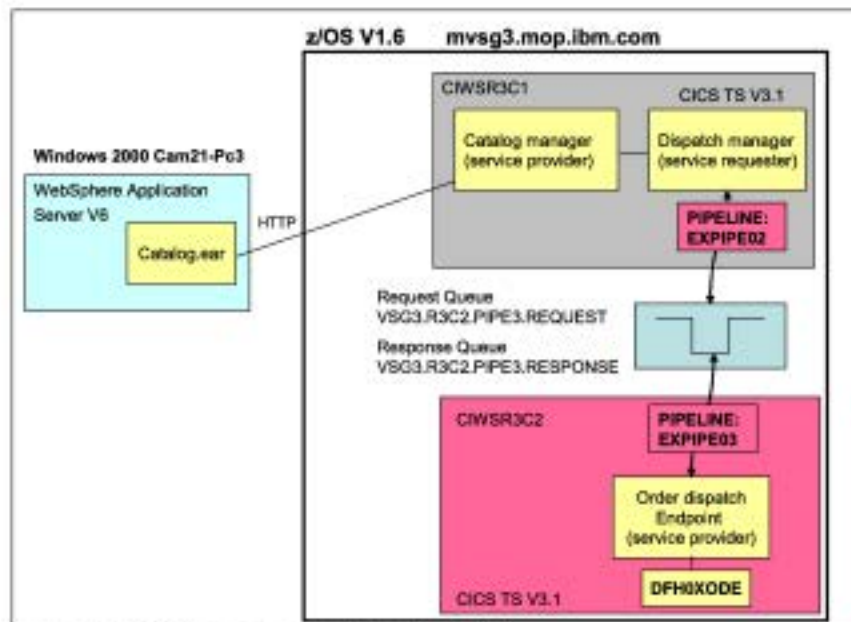


Figure 4-2 CICS as a service provider using WMQ

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.



```
/CICS/R3C2/config
/CICS/R3C2/shell
/CICS/R3C2/wsbinding/provider
```

We copied the dispatchOrderEndpoint.wsbind file from the CICS-supplied directory /usr/lib/cics/cicsts31/samples/webservices/wsbinding/provider to our wsbind directory /CICS/R3C2/wsbinding/provider.

The config directory is used for the pipeline configuration file that we create in a subsequent step, while CICS uses the shell 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 shell 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(DISF)
```

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     --> /CICS/R3C2/config/basicsoap12provider.xml
  (Fixed Case)  -->
               -->
               -->
  Shell         --> /CICS/R3C2/shelf
  (Fixed Case)  -->
               -->
               -->
  Module        : /CICS/R3C2/wsbmwd/provider
  (Fixed Case)  :
               :

SYSID-R3C2 APPLID-A6PC03C2
```

Figure 4-3 CEDA DEFINE PIPELINE EXPIPE03

- We set CONFIGFILE to the name of our pipeline configuration file.
/CICS/R3C2/config/basicsoap12provider.xml
- We set SHELF to the name of the shelf directory.
/CICS/R3C2/shelf



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

/C:\WS\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(/C:\WS\R3C2/config/bas (cscs)2provider.xml)
  Shell(/C:\WS\R3C2/shell1/)
  Wsdir(/C:\WS\R3C2/wsbind/provider/)

                                     SYSID=31C2 APPLID=6MPSR3C2

```

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, dispatcherOrderEndpoint) is taken from the wsbind file.

```

INQUIRE WEBSERVICE
STATUS: RESULTS - OVERTYPE TO MODIFY
  webs(dispatcherOrderEndpoint ) Pip(EXPIPE03)
  Ins Len (6432310 ) Pro(DFHOX00E) Con
                                     Def120051209:

                                     SYSID=31C2 APPLID=6MPSR3C2

```

Figure 4-5 CEMT INQUIRE WEBSERVICE

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



```

[INQUIRE URIMAP
RESULT - OVERTYPE TO MODIFY
  UriMap(1419210)
  Usage(Pipe)
  EnableStatus( Enabled )
  AnalyzerStatus(Analyzer)
  Scheme(Http)
  RedirectType( None )
  TcpService()
  Host(*)
  Path(/exampleApp/dispatchOrder)
  Transaction(IPID)
  Converter()
  Program()
  Pipeline(EXECPE03)
  WebService(dispatchOrderEndpoint)
  Header()
  Certificate()
  Ciphers()
  Templaterane())

```

SYSCC=3112 APPLID=A6PCR3C2

Figure 4-6 CFMT 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 DPMOXW00, 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.



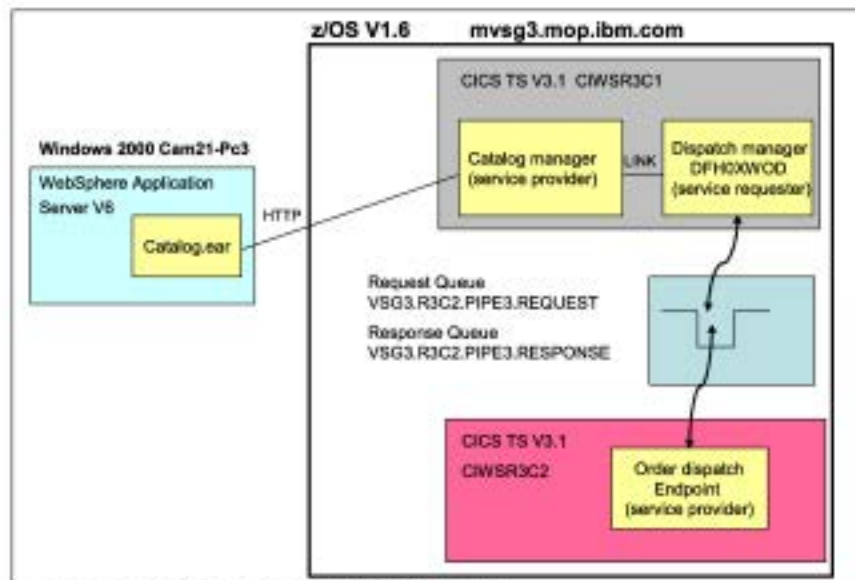


Figure 4-7 CICS as a service provider using WMQ

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 ==> DFH0XSOS
      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
                        ==> ST0MQS3&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=/exampleApp/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 on 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.



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(CKT1)          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(ZC10WAIT)

SYSID=R3C2 APPLID=A6P0R3C2
```

Figure 4-10 CEMT INQUIRE TASK



Figure 4-10 shows the in-flight transactions:

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

After ending the EDI session, we received the ORDER SUCCESSFULLY PLACED response in the browser.

We noted the SYSPRINT messages by the CICSWSM5GH message handler for CICS region CICSWSR3C2 (Example 4-5).

Example 4-5 CICS CICSWSR3C2 - SYSPRINT

```
CICSWSM5GH: =====
CICSWSM5GH: Container Name: : DFHWS-WERSERVICE
CICSWSM5GH: Container content: dispatchOrderEndpoint
CICSWSM5GH: -----
CICSWSM5GH: Container Name: : DFHWS-IRANJD
CICSWSM5GH: Container content: DISP
CICSWSM5GH: -----
CICSWSM5GH: Container Name: : DFHWS-URI
CICSWSM5GH: Container content: WMQ:VS63.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 CICS Plex. 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.

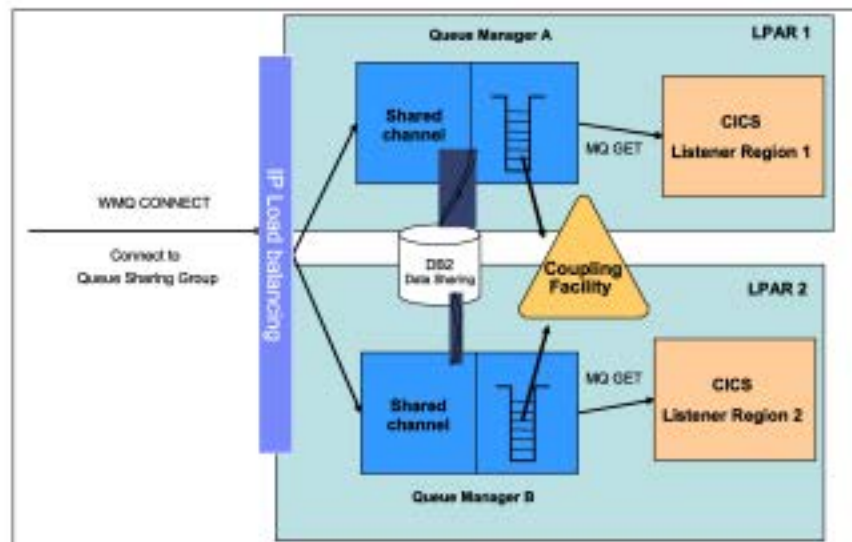


Figure 4-11 High availability configuration for WMQ

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 or 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 full 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 on configuring high availability with WMQ refer to *WebSphere MQ in a z/OS Parallel Sysplex Environment*, SG24-6864.





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 requestors 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.

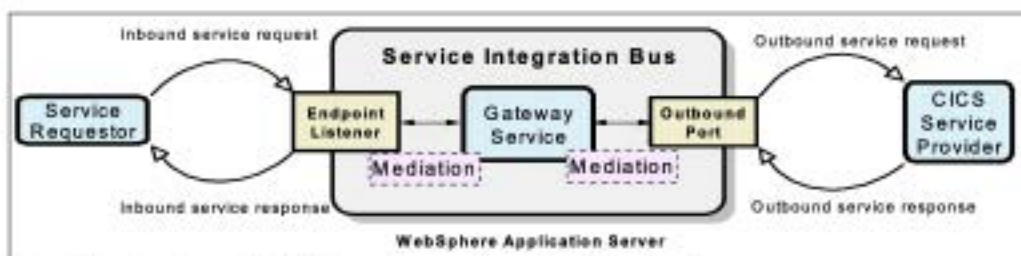


Figure 5-1 Exposing a CICS Web service over the service integration bus

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; I 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, not the service requesters. This service location independence enables a more flexible configuration and eases the task of service administration.
- ▶ **Secure 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



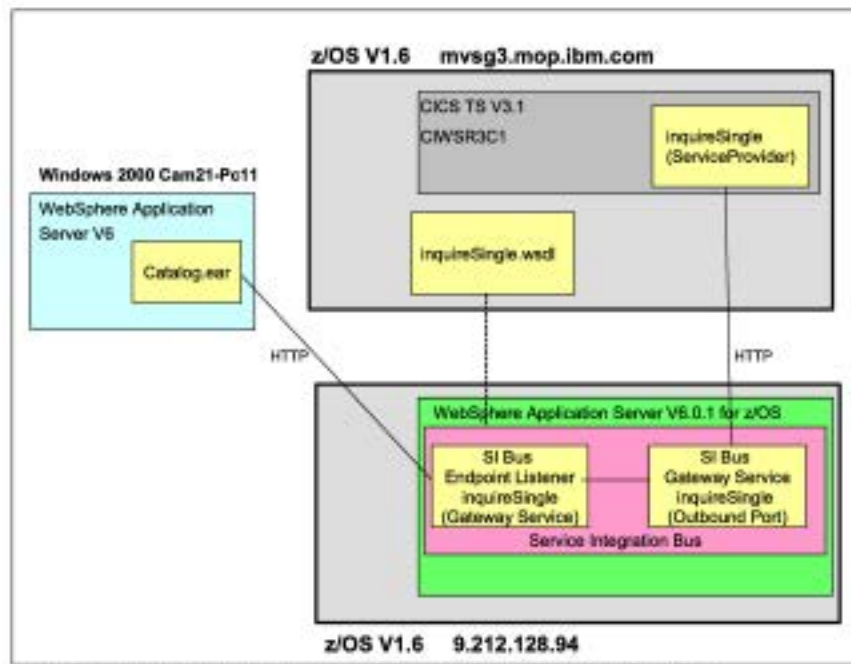


Figure 5-2 The catalog application accessed via a service integration bus

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	
TCPIP SERVICE	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 `TCPIPERVICE` R3C1 that we configured in "Configuring the `TCPIPERVICE` 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 choose 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 `TCPIPERVICE`. 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

```
<?xml version="1.0"?>
<service name="DFH0XCMNSERVICE">
  <port binding="tns:DFH0XCMNHTTPSoapBinding" name="DFH0XCMNPort">
    <soap:address
      location="http://mvs03.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
  CCGA View Unmap: INQSINGW !
    Unmap      : INQSINGW
    Group      : ASGA
    Description :
    Status     : Enabled           Enabled | Disabled
    Usage      : Server           Server | Client  Pipeline
  UNIVERSAL RESOURCE IDENTIFIER
    Scheme     : HTTP             HTTP | HTTPS
    HOST       : *
    (Lower Case) :
    Path       : /exampleApp/inquireSingle.wsd!
    (Mixed Case) :
    :
    :
  ASSOCIATED CICS RESOURCES
    Implementation :
  + Analyzer      : No           No | Yes

SYSID=0001 APPLID=66P0H3CL

```

Figure 5-3 UHIMAP for serving inquireSingle.wsd! (page 1 of 2)

```

OBJECT CHARACTERISTICS                                     CICS RELEASE = 0640
  CCGA View Unmap: INQSINGW !
  Converter      :
  Transaction    :
  Program       :
  Pipelining     :
  Webservices    : (Mixed Case)
  SECURITY ATTRIBUTES
    HSETC       :
    Ciphers     :
    Certificate  : (Mixed Case)
  STATIC DOCUMENT PROPERTIES
    MediaType    : text/xml
    (Lower Case) :
    CharSet     : iso-8859-1      (Mixed Case)
    HOSTCodePage : 037
    TemplateLang :
    (Mixed Case) :
  + HostFile     : /u/exampleApp/inquireSingle.wsd!

SYSID=0001 APPLID=66P0H3CL

```

Figure 5-4 UHIMAP for serving inquireSingle.wsd! (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.



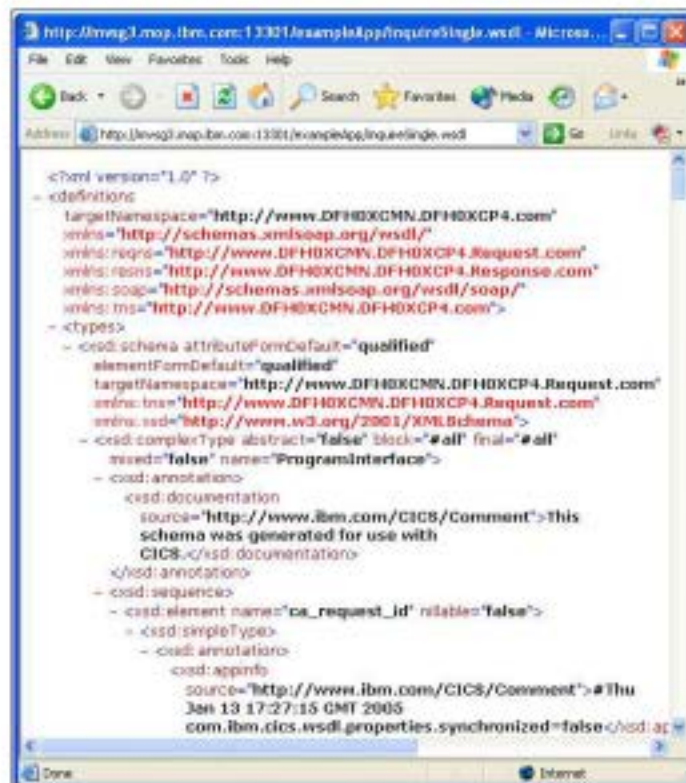


Figure 5-5 inquireSingle.wsdl from Web browser

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 SOAPMTTPChannel1
- ▶ Created an endpoint listener

For information on 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.



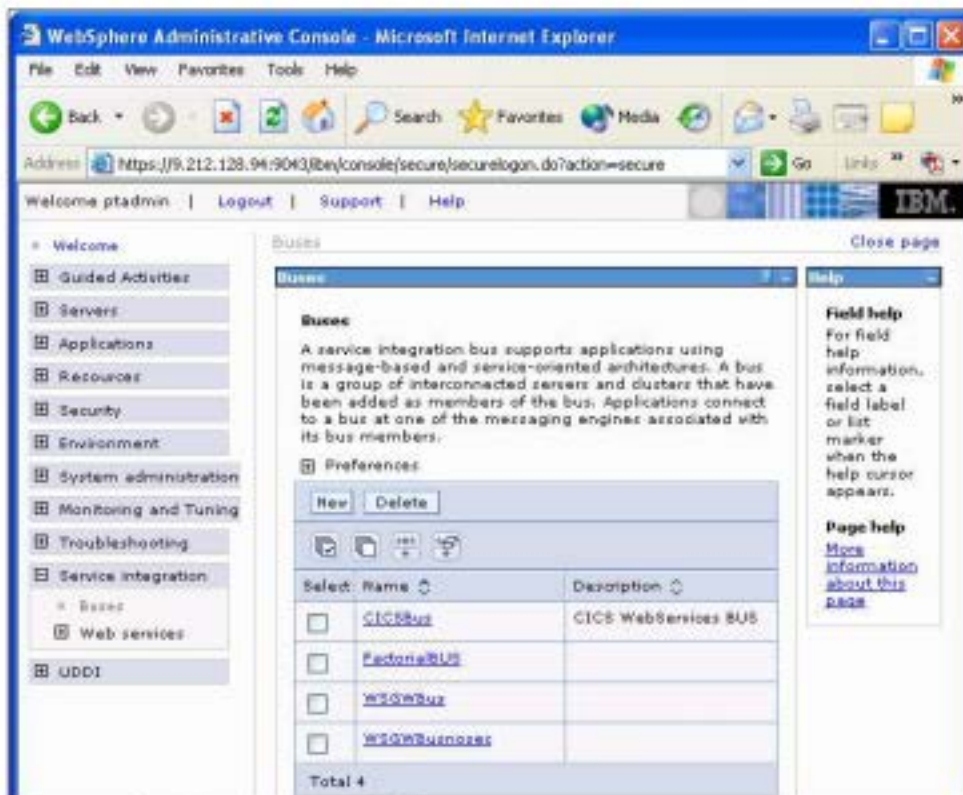


Figure 5-6 Service integration bus CICSBus

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.

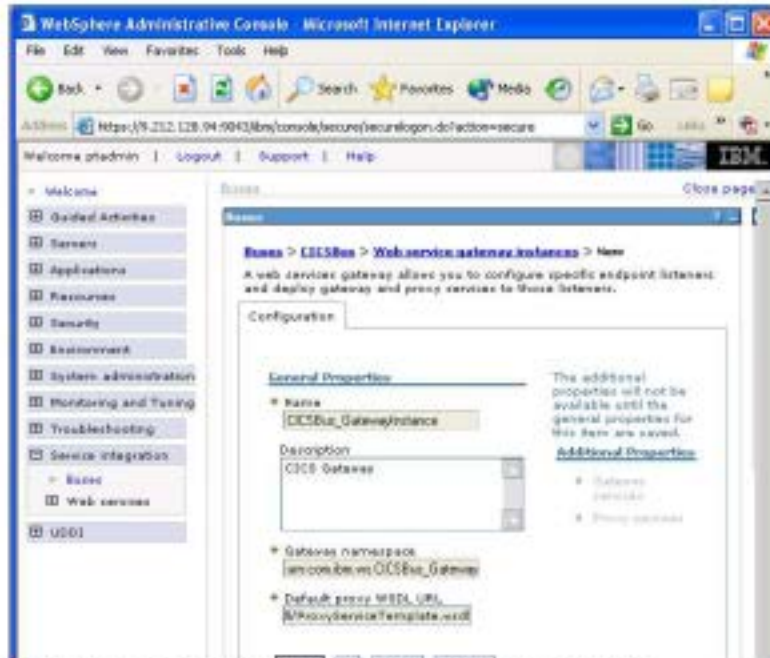


Figure 5-7 Create a gateway instance CICSBus_GatewayInstance

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**.

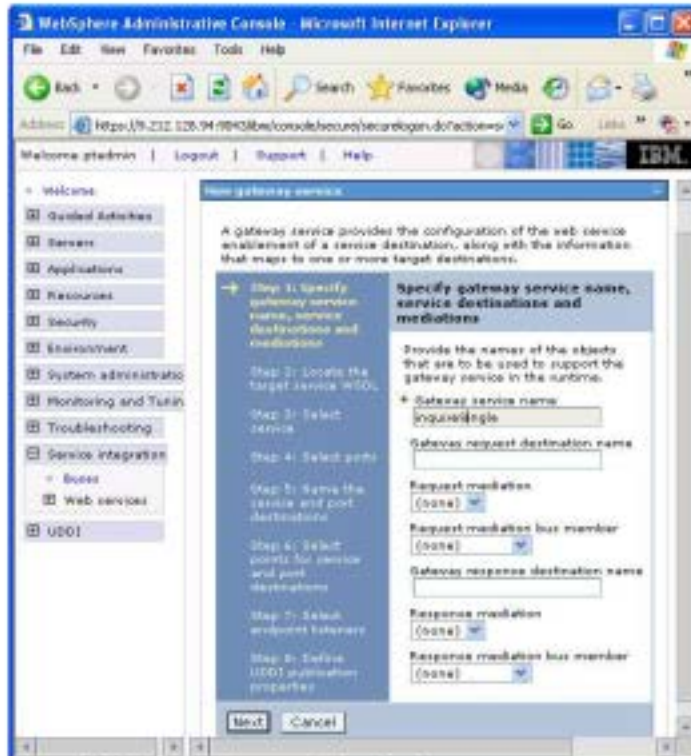


Figure 5-8 Create gateway service - Step 1



- 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).



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 choose 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 choose from:
(<http://www.DFH0XCMN.DFH0XCP4.com>)DFH0XCMNService
We selected the operation and clicked **Next**.



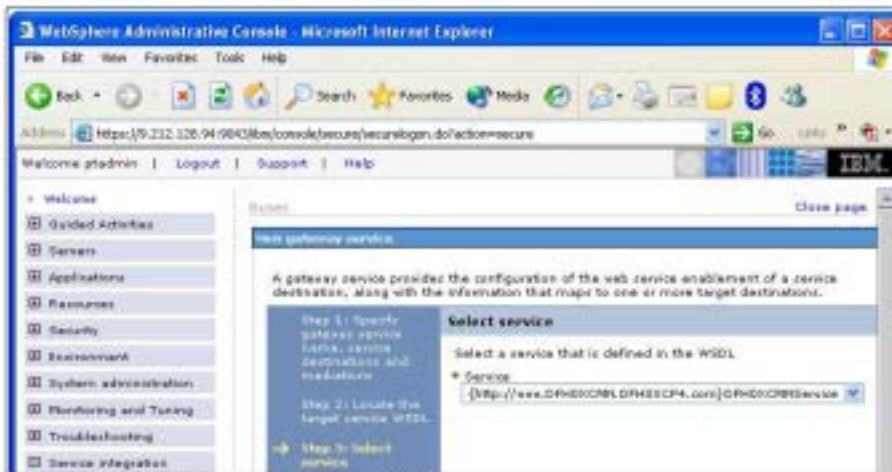


Figure 5-10 Create gateway service - Step 3

- 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).



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).



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.inquireSingleFindings.wsdl
 - CICSBus.inquireSingleNonBound.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/wsgwscaphttp1/soaphttpengine/CICSBus
/inquireSingle/?wsdl" server="SOAP-HTTPChannel1 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-jpl11:9060/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.



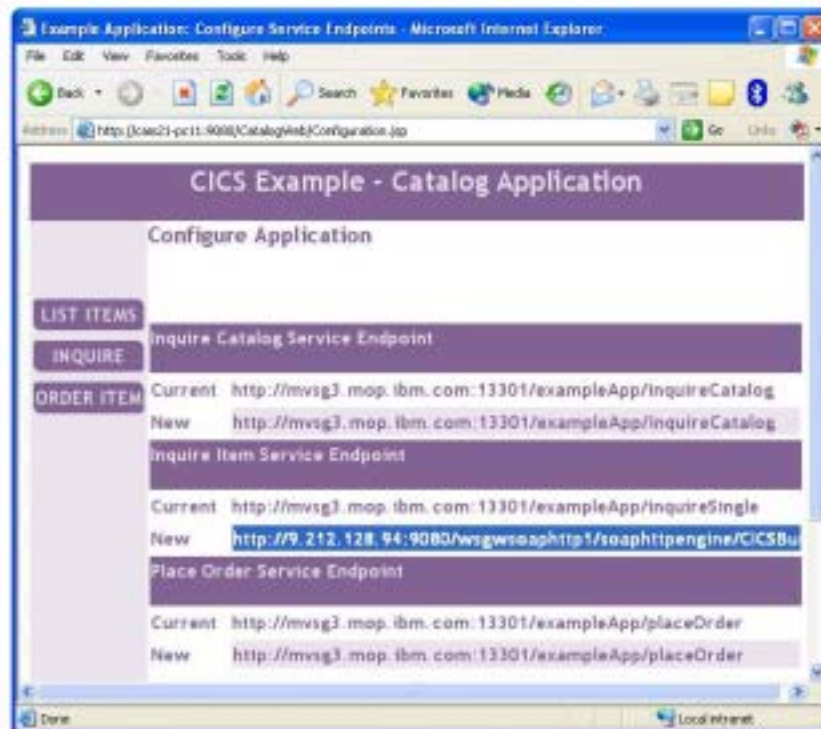


Figure 5-13 Configure Catalog manager application to invoke gateway service

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.





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 if it had been sent directly from the service requester, rather than having been sent over the *SiBus*.



Security management

We begin this section by introducing you to some basic cryptography concepts and to some algorithms 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 include DES, AES, RSA, and SHA-1.

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.



150 Implementing CICS Web Services



Elements of cryptography

If you want to implement WS-Security in CICS, you need 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

You also have to choose between various algorithms (for example, among DES, TripleDES, and AES; or between RSA and DSS). In writing this chapter our main goal is to collect in one place enough information about cryptography for you to be able to read and understand the open standards related to securing Web services, the documentation related to securing Web services in CICS, and the following chapters in this book in which we describe the scenarios we implemented.

At the end of this chapter we provide a brief discussion of the hardware requirements for implementing WS-Security in CICS.



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 *encrypting*.

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.

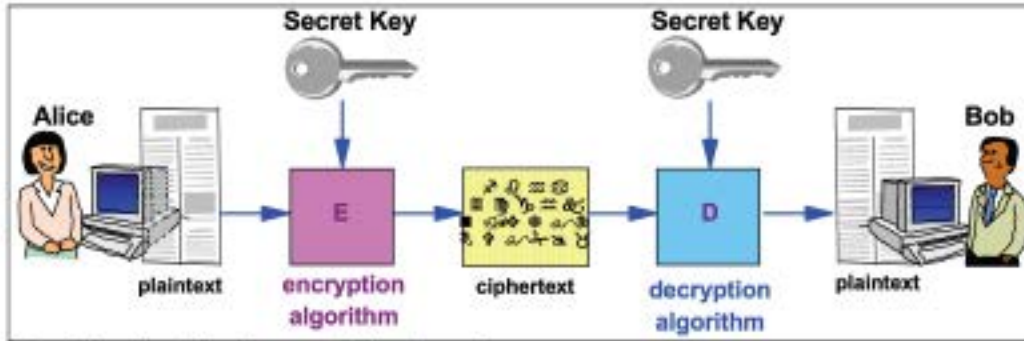


Figure 6-1 Secret key (or symmetric) cryptography

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 6.1, “How DES works” on page 545.

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.

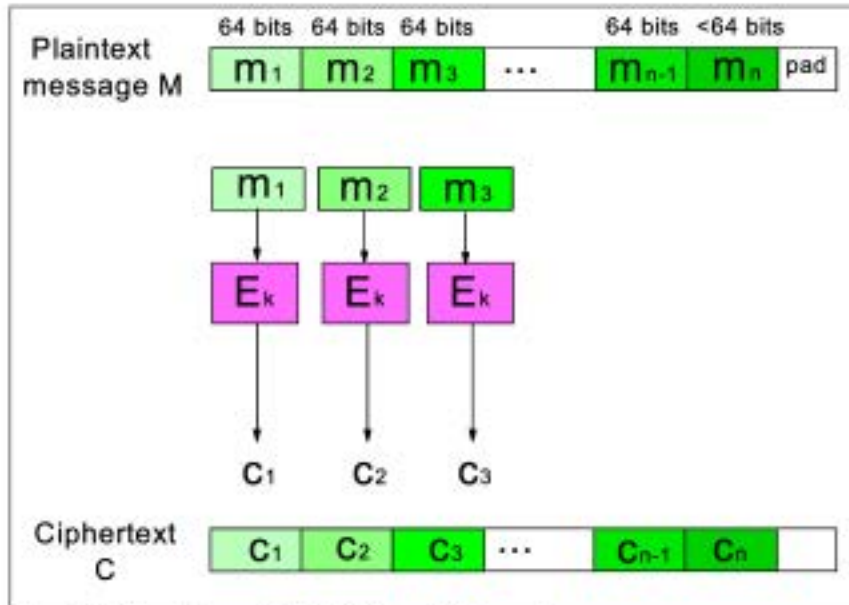


Figure 6-2 Electronic codebook (ECB) mode of operation

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.

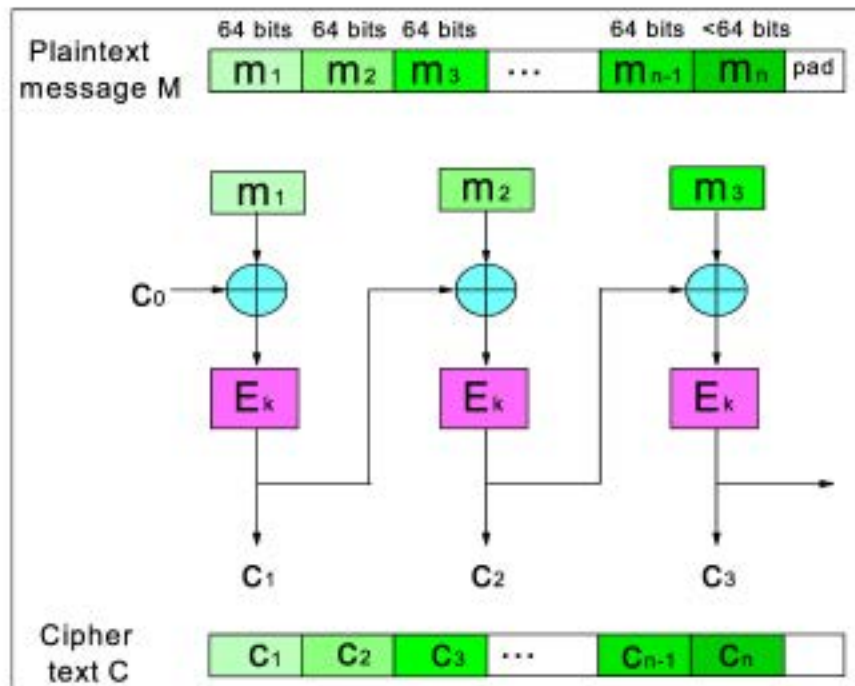


Figure 6-3 Cipher block chaining (CBC) mode of operation

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 decryptor 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 decryptor can reverse the padding process, remove the padding bits and produce the original plaintext. The decryptor 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.

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 26, 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:

$$C = E_{k_3}(D_{k_2}(E_{k_1}(M)))$$

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 6-4.

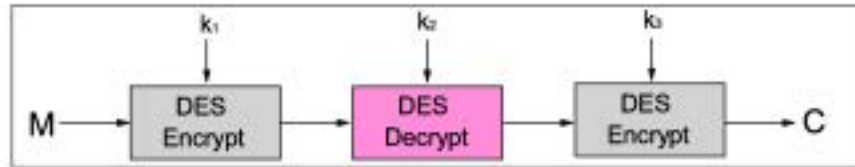


Figure 6-4 Triple DES - EDE

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.
- ▶ k_1 and k_2 are independent, but $k_3 = k_1$.
- ▶ $k_1 = k_2 = k_3$.

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)

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 N_k in the key is thus 4, 6, or 8.

The number of rounds N_r to be performed during the execution of the algorithm depends on the key size. When $N_k = 4$, then $N_r = 10$. If $N_k = 6$, then $N_r = 12$, and when $N_k = 8$, then $N_r = 14$.

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 549.

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 6-5. Public key cryptography is also known as asymmetric cryptography.

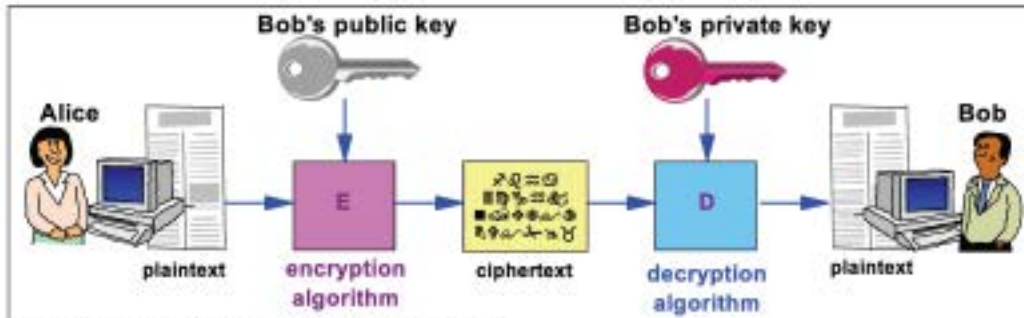


Figure 6-5 Public key (or asymmetric) cryptography

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 RAS 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. 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. 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 $a-b = i \times n$, that is, such that $a = b + (i \times n)$. If a and b are congruent modulo n , we write

$$a \equiv b \pmod{n}$$

For example, $50 \equiv 0 \pmod{5}$ because $(50 - 0)/5 = 10$, an integer. But 50 is not congruent to $1 \pmod{5}$ because $(50-1)/5$ is not an integer. However, $51 \equiv 1 \pmod{5}$.

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, $5^5 \pmod{23} = 8$ because $5^5 = 5 \times 5 \times 5 \times 5 \times 5 = 15,625$ and 15,625 divided by 23 gives a quotient of 679 and a remainder of 8. Also, $3 + 4 = 2 \pmod{5}$ because $3 + 4 = 7$ and 7 divided by 5 gives a quotient of 1 and 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 $(p-1)(q-1)$. The public key is the pair (n, e) , the private key is (n, d) . 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.

In practice, however, 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. This combines the high speed of DES with the key management convenience of the RSA system.

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 6-6.

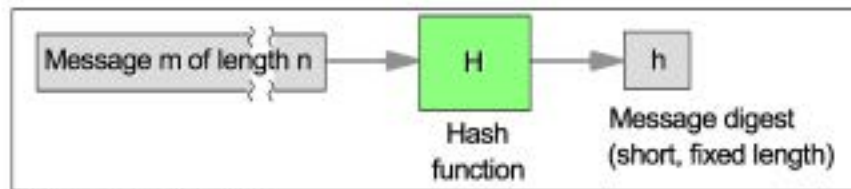


Figure 6-6 A hash function

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.

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.

1. Damgård 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 and returns a shorter, fixed-length output. 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 6-7.



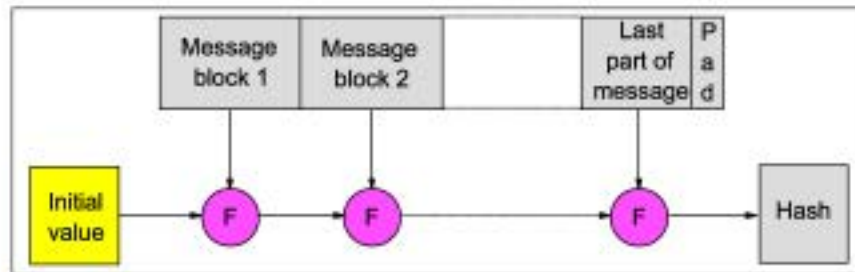


Figure 6-7 Iterative structure for hash functions

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), the algorithm specified in the Secure Hash Standard (FIPS PUB 180), was developed by NIST. SHA-1 is a revision to SHA that was published in 1994; the revision corrected an unpublished flaw in SHA.

The algorithm takes a message of less than 2^{64} bits in length and produces a 160-bit message digest. The algorithm is slightly slower than MD5 but the larger message digest makes it more secure against brute-force collision and inversion attacks.

6.5 Message authentication codes

There are four types of message authentication codes (MACs):

- Unconditionally secure
- Stream cipher-based
- Block cipher-based
- Hash function-based



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 6-8.

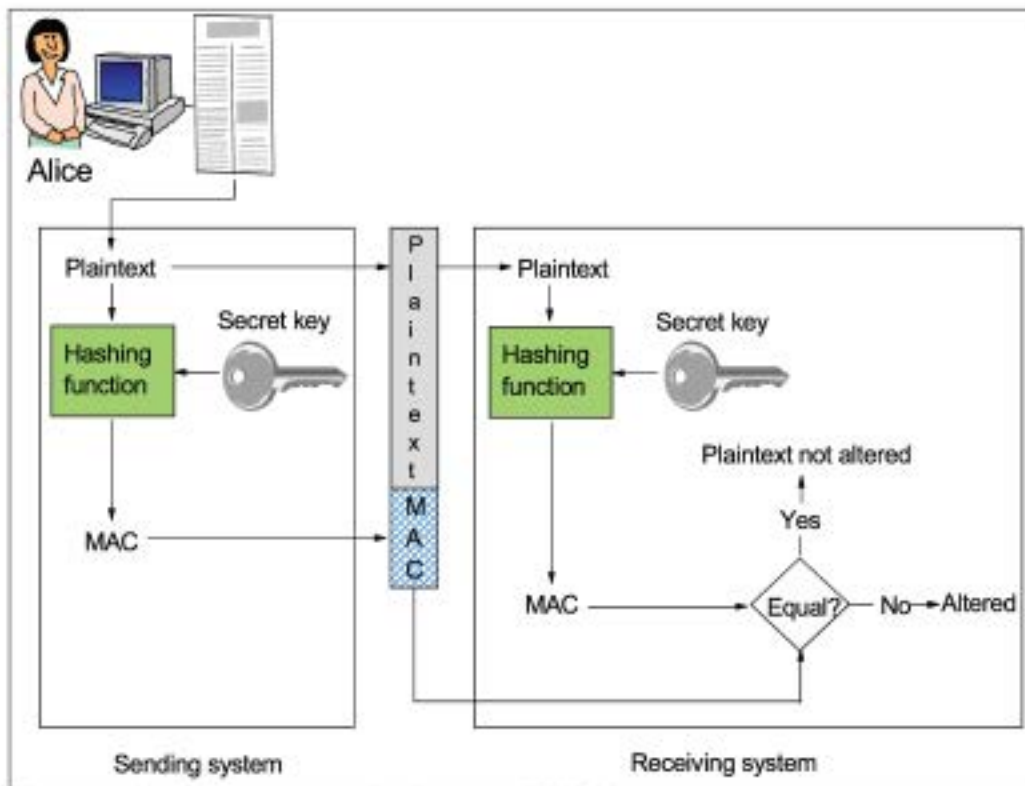


Figure 6-8 Keyed-hash message authentication code (HMAC)



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.

Readers who want to have an overview of the HMAC function described in FIPS PUB 198 should consult B.3, "How the HMAC algorithm of FIPS PUB 198 works" on page 554.

6.6 Digital signatures

When public-key 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 6-9.



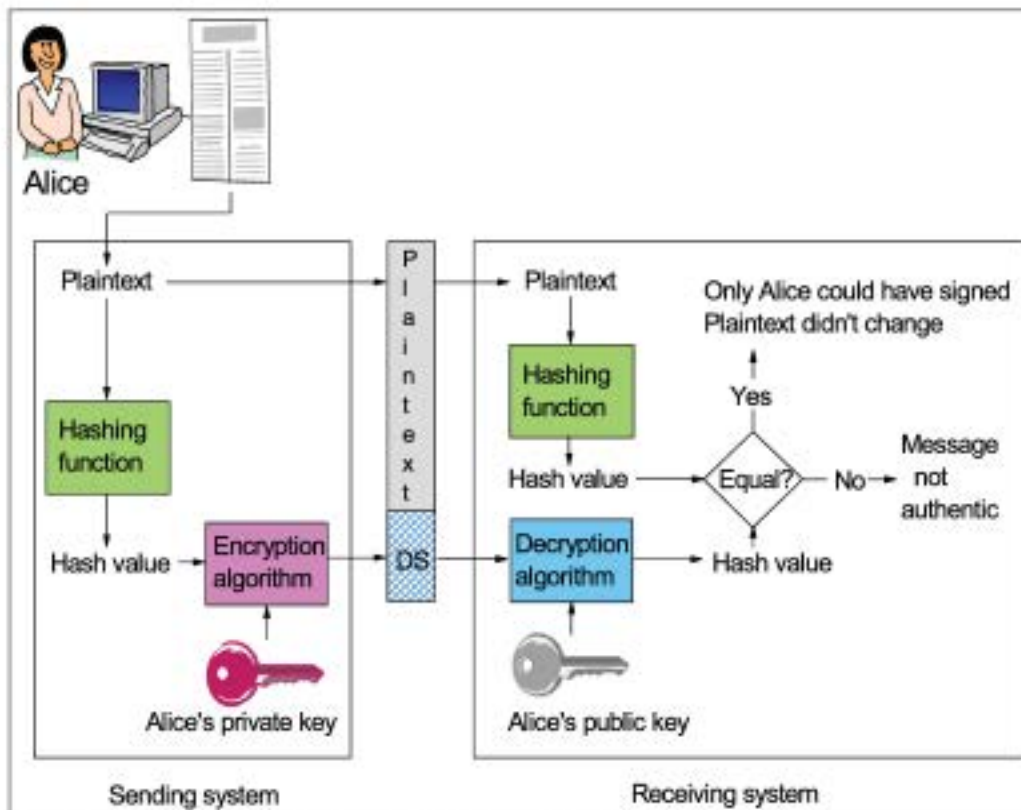


Figure 6-9 Creating and verifying a digital signature in a public key system

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.

Suppose that Alice wishes to keep the contents of the document secret. 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 6-10.



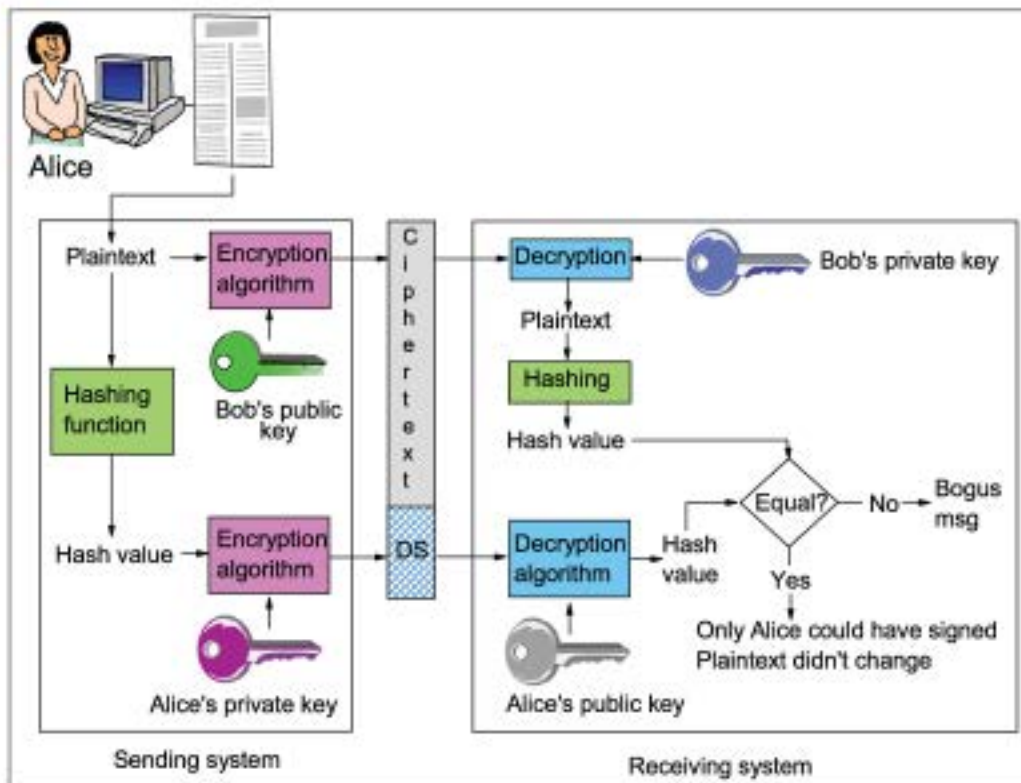


Figure 6-10 Creating and verifying a digital signature while encrypting the message

Alternately, 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.

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 5.7, "Public key digital certificates" on page 173.

6.6.1 Using RSA for digital signatures

You can use the RSA algorithm to compute a digital signature. Suppose 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 because a message will be signed by an individual only once, but the signature may be verified many times.

6.6.2 Using DSA for digital signatures

While the RSA system can be used for both encryption and digital signatures, the Digital Signature Algorithm (DSA) can only be used to provide 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 $2^{1023} < p < 2^{1024}$
Appendix 2 of FIPS PUB 186-2 specifies a method for generating p .
- ▶ q = a prime divisor of $p-1$, where $2^{159} < q < 2^{160}$
Appendix 2 of 186-2 also specifies a method for generating q .
- ▶ $g = h^{(p-1)/q} \bmod p$, where h is any integer with $1 < h < p-1$ such that $h^{(p-1)/q} \bmod p \neq 1$.
- ▶ 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 3 of 186-2 specifies a method for generating both 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, $(k^{-1} \cdot k) \bmod q = 1$ and $0 < k^{-1} < q$. The value of $\text{SHA-1}(M)$ 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. 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.



6.6.3 Using ECDSA for digital signatures

The ANSI X9.62 standard *Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)* specifies a third method of providing digital signatures. This method 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.

6.6.4 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 6.7.1. "tbsCertificate" on page 175.

- **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-itsa-with-sha1

This algorithm uses SHA-1 for the hash function, and it 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. The signature value is encoded as a BIT STRING and included in the signature field as shown in Figure 6-11.



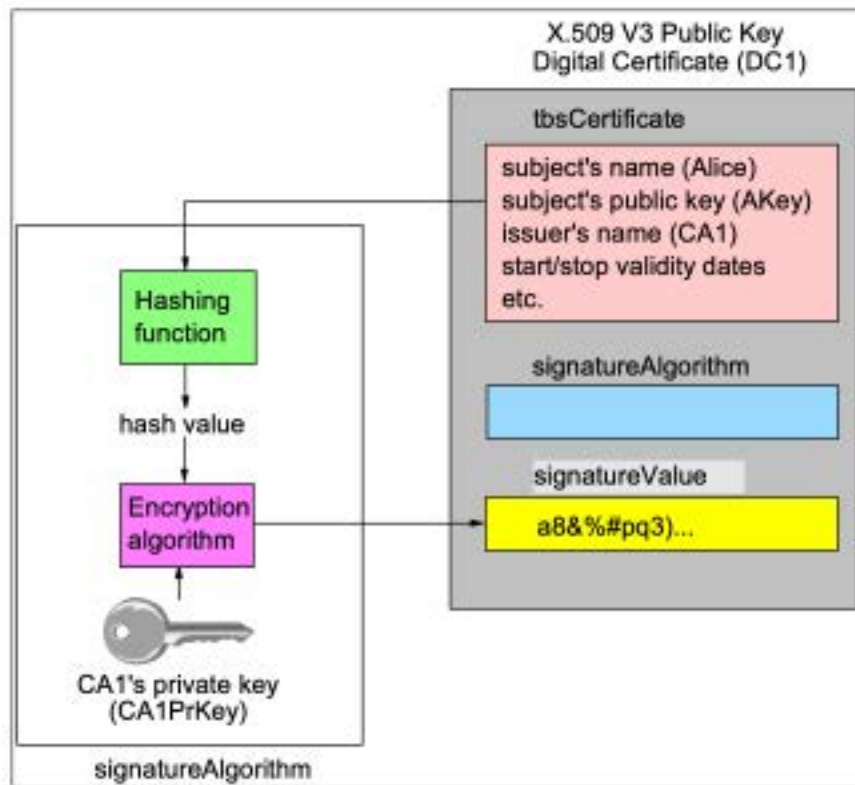


Figure 6-11 X.509 V3 public key digital certificate

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.509 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 each 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 178.

▪ **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 algorithm identifier 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 algorithm identifier is `id-dsa`, the public key must be encoded as the integer y .

– `id-publicNumber`

This identifies the Diffie-Hellman key exchange algorithm. The public key is the integer $y = g^x \bmod p$.

`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 algorithm identifier 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 are 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 entries 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 180.

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 (v1.0)

 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 16:04:32 1998 GMT
Not After : Jul 9 16:04:32 1999 GMT
Subject: C=US, ST=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:d9:31:98:0a: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:56:17:44:ba:75:ch:e6:1c:9c:5h:66:
70:33:52:14:c9:ec:4f:91:51:77:39:de:53:65:17:
15:94:6e:ee:14:d5:6f:d5:ca:b3:47:5e:1b:0e:7b:
c5:cc:2a:6b:c1:90:ca:1b:31:0d:bf:7a:c7:47:77:
8f:a0:21:c7:4c:d0:16:65:90:c1-0f:d7:b8:60:e3:
d2:75:6a:c1:ea:9e:5c:5c:ea:7d:c1:a1:10:bc:h6:
ea:35:1c:9e:27:52:7e:41:0f:
Exponent: 65537 (0x10001)
Signature Algorithm: md5withRSAEncryption
93:5f:8f:5f:c5:af:b7:0a-ab:a5-6d:fb:24:5f:b6:59:5c:9d:
92:2e:4a:1b:8a:ac:7d:90:17:5d:cd:19:f6:aad:ef:63:7f:92:
ad:2f:4b:0f:0a:13:90:ec:2c:0e:43:03:he:f6:ea:8e:9c:67:
d0:a2:40:03:17:el:6a:15:09:79:a9:4b:ed:b7:16:1b:41:72:
0d:19:aa:ad:dd:9a:d7:ab:97:50:65:f5:5e:85:a6:ef:19:c1:
5a:de:9d:ea:03:cd:ca:cc-6d:5d-c1:85:b5:6d:c6:f3:d9:f7:
6f:0e:fc:aa:1f:34:e9:96:6e:6c:cf:f2:ef:9h:hf:de:h5:22:
ea:9f

```

Distinguished names

Both the issuer field and the subject field of this Certificate 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 6-12, 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.

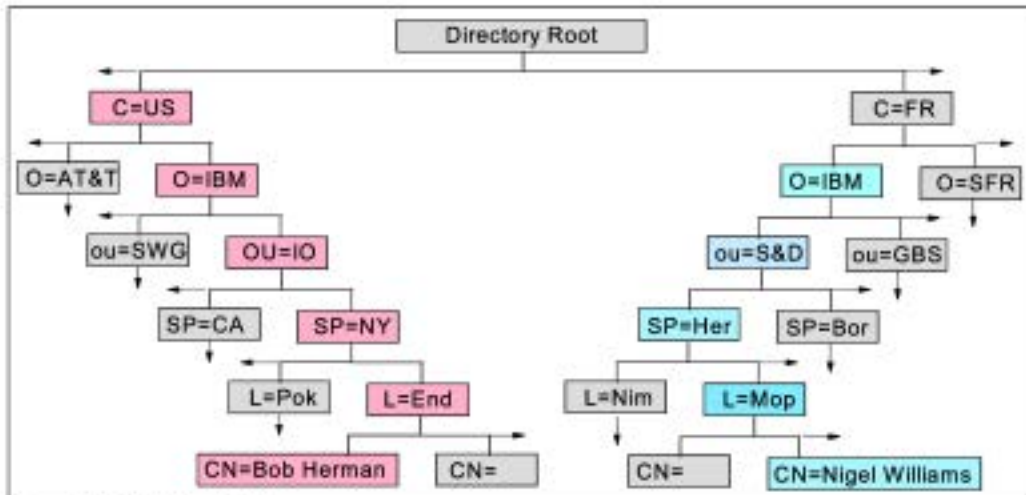


Figure 6-12 Distinguished names



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 6.8, "Certificate revocation lists" on page 184.)

- 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 syntax of the part 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(user-id) GENCERT
      SUBJECTS(
          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') )
      FCIC( | FCST
      KEYUSAGE( HANDSHAKE DATAENCRYPT DCSIGN CERTSIGN)
      ALIAS( IP(numeric-ip-address)
             DDH4IN('internet-domain-name')
             EMail('email-address')
```



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.

6.7.3 Certification paths

In Figure 6-11 on page 175, 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 `tbCertificate` signed with CA1's private key, we must verify the digital signature just like we verified the digital signature in Figure 6-9 on page 169. That is, we must use CA1's public key to proceed as shown in Figure 6-13.



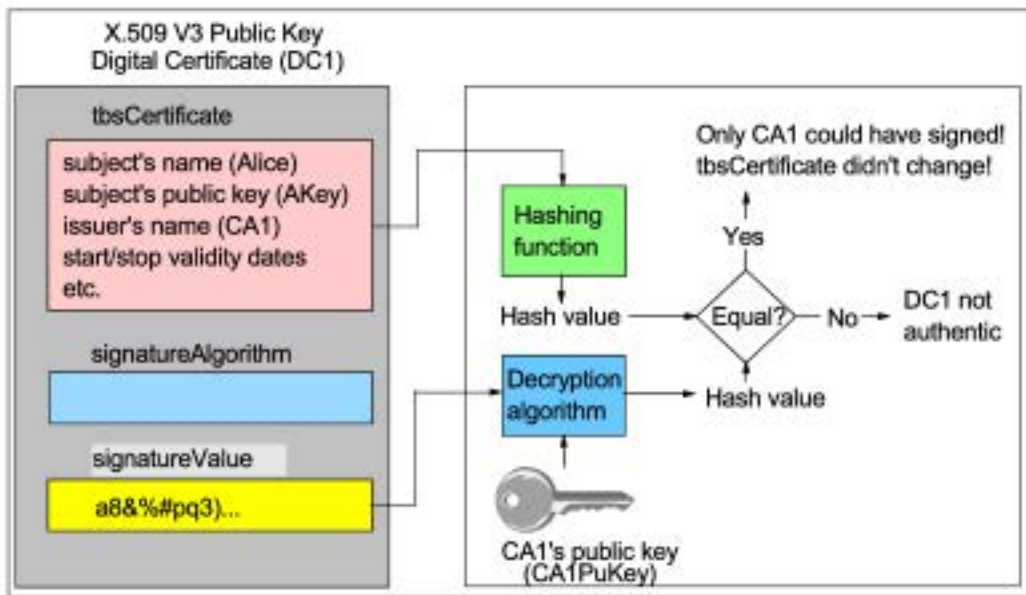


Figure 6-13 Verifying digital certificate

But now we have another problem: 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 6-14.



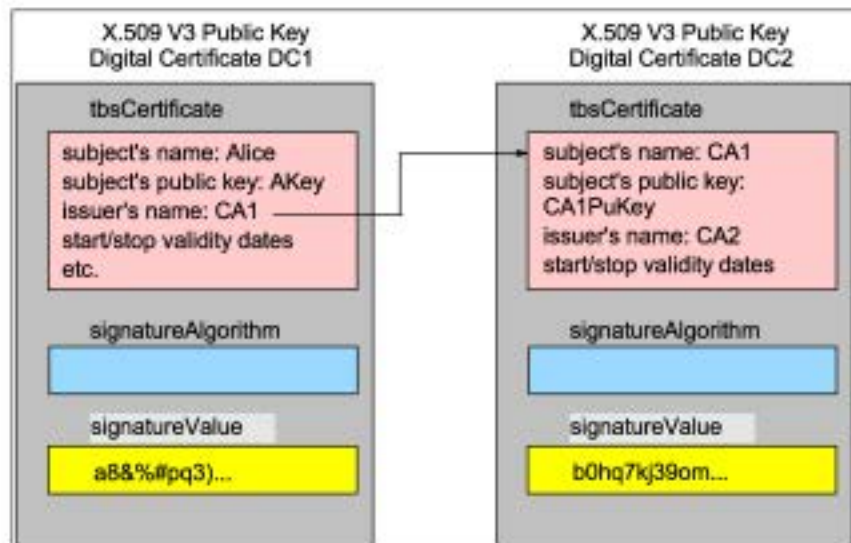


Figure 6-14 Certification path

In general, we need a sequence of n certificates, which satisfies the following conditions:

- For all x in $\{1, 2, \dots, n-1\}$, 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).
- Certificate 1 is the certificate to be validated.
- For all x in $\{1, 2, \dots, n\}$ the certificate is valid at the time in question.

Such a chain 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.

X.509 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 certifying 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:

- **tbisCertList**

The **tbisCertList** 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.509 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 186.

- crlExtensions (optional)

We discuss some of the extensions for CRLs in 6.8.2 "Extensions for a CRL" on page 187.

- 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 tbsCrlList. The ASN.1 DER encoded tbsCrlList is used as the input to the signature function. This signature value is encoded as a B-T-STRING and included in the CRL signatureValue field.

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.

► **freshesCRL**

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 the 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 Diffie-Hellman key agreement protocol

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.

One example of such a protocol is called the Diffie-Hellman key agreement protocol. It 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 and parameter g (usually called a generator) is an integer less than p , with the following property: for every number n between 1 and $p-1$ inclusive, there is a



power k of g such that $r = g^k \bmod p$. 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 $g^b \bmod p$.
- ▶ Bob sends his public value to Alice.
- ▶ Alice computes $(g^b)^a \bmod p$.
- ▶ Bob computes $(g^a)^b \bmod p$.

Since $(g^b)^a \bmod p = g^{ba} \bmod p = g^{ab} \bmod p = (g^a)^b \bmod p$, 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 $5^6 \bmod 23 = 8$.
- ▶ Alice sends her public value 8 to Bob.
- ▶ Suppose Bob chooses a secret integer $b = 15$. He then computes his public value $5^{15} \bmod 23 = 19$.
- ▶ Bob sends his public value 19 to Alice.
- ▶ Alice computes $19^6 \bmod 23 = 2$.
- ▶ Bob computes $8^{15} \bmod 23 = 2$.

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 (or 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 to Bob
- 2 Bob generates a random number b and computes g^b .
- 3 Bob computes the shared secret key $K = (g^a)^b$
- 4 Bob concatenates the exponentials (g^b, g^a) (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 to Alice
- 5 Alice computes the shared secret key $K = (g^b)^a$.
- 6 Alice decrypts (g^b, g^a) using the shared secret key K and verifies Bob's signature using Bob's public key.
- 7 Alice concatenates the exponentials (g^a, g^b) (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. The differences between TLS 1.0 and SSL 3.0 are not dramatic, but they 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. 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, and reassembled, then delivered to higher level clients. See Figure 6-15, which shows a client sending a message to a server.



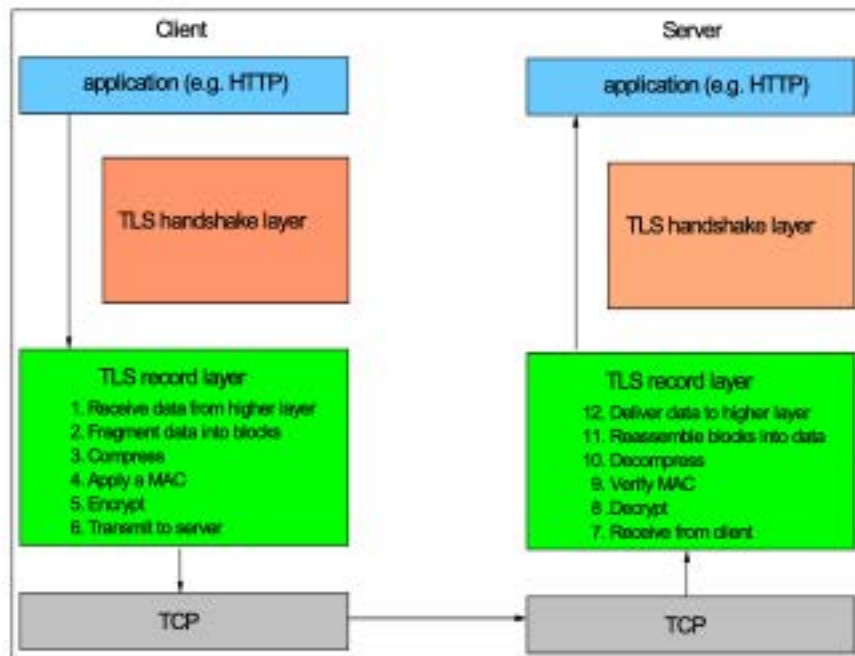


Figure 6-15 Overview of TLS

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. DHF 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 the DH or DHE parameter.

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.

RFC 2246 also assigns a number to each of the 27 cipher suites. The first ten are listed in Table 8-2 on page 196.



Table 6-2 Some TLS 1.0 cipher suite numbers and their meanings

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
06	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
07	TLS_RSA_WITH_IDEA_CBC_SHA
08	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
09	TLS_RSA_WITH_DES_CBC_SHA
0A	TLS_RSA_WITH_3DES_EDE_CBC_SHA

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 4182 *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)*

CICS TS V3.1 added support for the cipher suites added by RFC 3268, which are identified in Table 6-3.



Table 6-3 Two cipher suites added by RFC 3268 and supported by CICS

Number	Name
2F	TLS_RSA_WITH_AES_128_CBC_SHA
35	TLS_RSA_WITH_AES_128_CBC_SHA

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_cancelled` (90)
- `no_negotiation` (100)

RFC 2246 provides a short explanation of each of these descriptions.

The user-cancelled and no-negotiation 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 6-15.
- Allow the client and server to verify that the other peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.



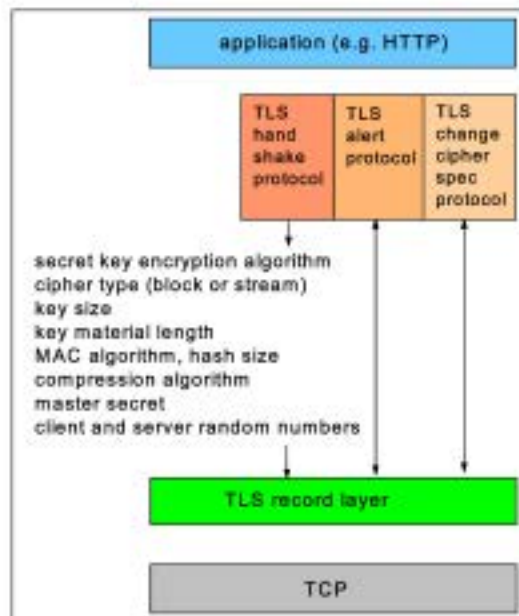


Figure 6-16 Handshake protocol passes security parameters to the record layer

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 6-17.

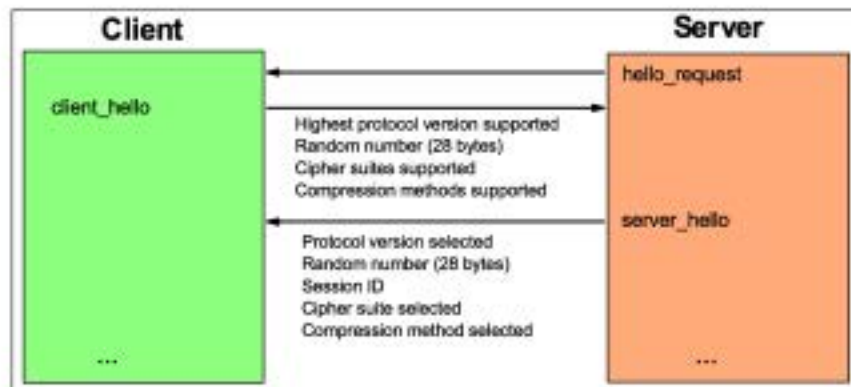


Figure 6-17 Starting a new TLS session(1): Establishing algorithms



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 defines 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.

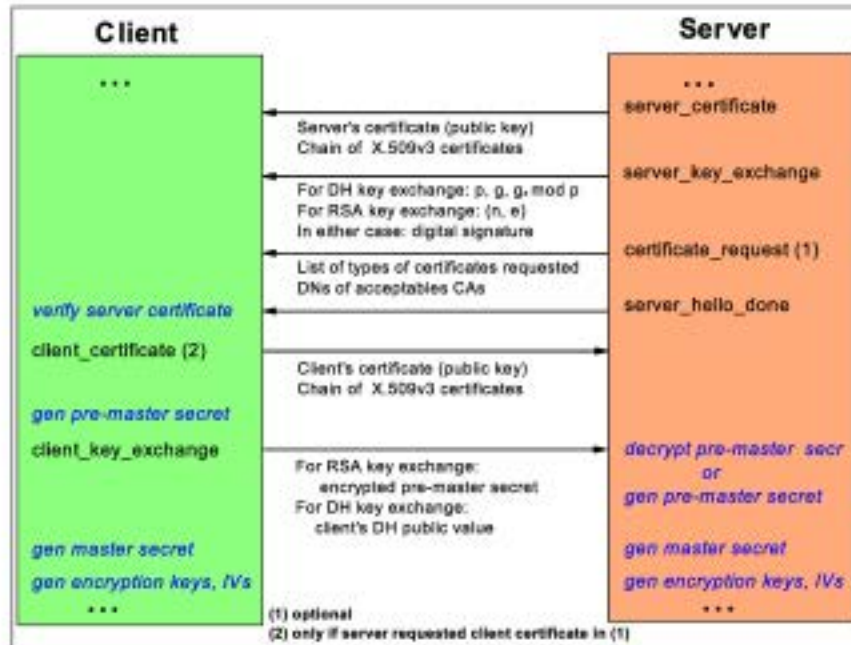
The server will send a `server_hello` message in response to a `client_hello` message when it is able to find an acceptable set of algorithms. If it cannot find such a match, it will respond with a handshake failure alert. The `server_hello` message includes the following parameters:

- ▶ 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 8-16 on page 199, which the TLS Handshake Protocol must pass to the



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 6-18 is to create a pre-master secret known to the communicating parties and not to attackers. Note that the italicized lines in Figure 6-18 represent actions taken by the client and server rather than protocol messages.



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 as shown in Table 6-4.

Table 6-4 Key exchange methods and corresponding certificate key types

Key exchange method	Certificate key type
RSA	RSA public key; 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.
RHE_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 communicate the pre-master secret: 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 $(p\|e)$; 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 $g^s \bmod p$.
- ▶ 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 $g^s \bmod p$. 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's 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 $g^c \text{ mod } p$. 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 `DHE_RSA` or `DHE_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 6-19 on page 206 shows the final phase of starting a new TLS session.



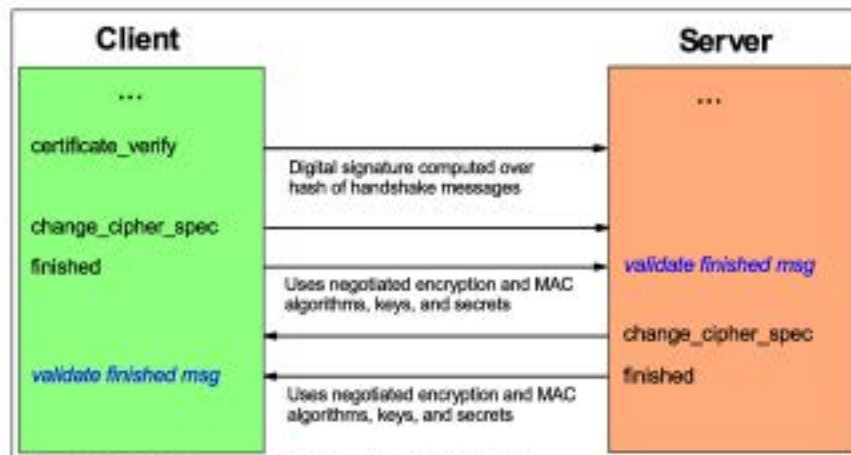


Figure 6-19 Starting a new TLS session (3): Verification

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 6-20 on page 208.



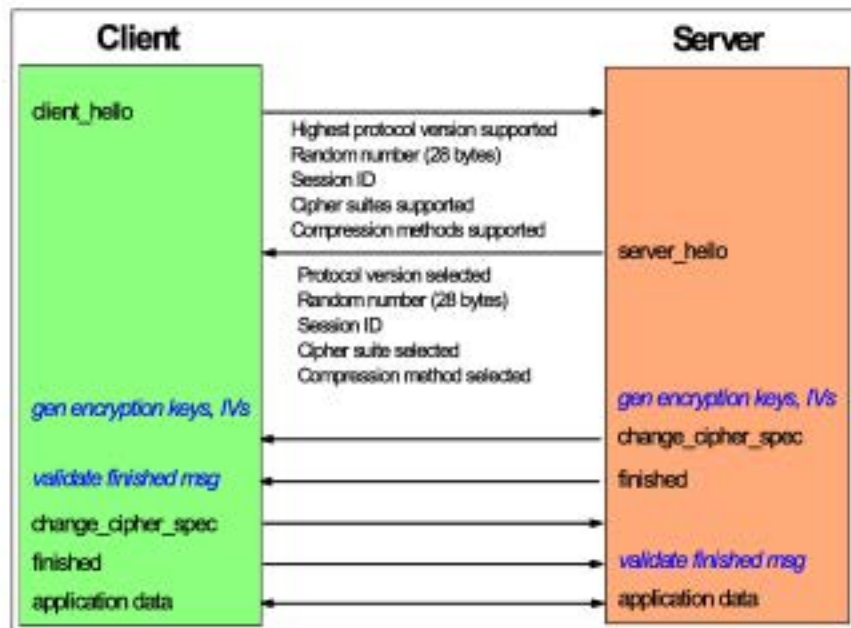


Figure 6-20 Resuming a session

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.



6.11 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 the latest System z hardware.

6.11.1 CP Assist for Cryptographic Functions (CPACF)

CPACF is a set of cryptographic instructions available on all CPs of a z990, z890, z9™-109, z9 Enterprise Class (EC), and z9 Business Class (BC). Use of the CPACF instructions provides improved performance.

CPACF offers a set of symmetric cryptographic functions. 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.

Note: 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.

The CPACF feature supports z/OS applications and subsystems (such as CICS) that use the Integrated Cryptographic Service Facility (ICSF) for cryptographic functions (see 6.12, "Integrated Cryptographic Service Facility" on page 210).

6.11.2 Crypto Express2 Feature (CEX2)

An installation can configure the CEX2 feature as an asynchronous cryptographic coprocessor or accelerator. It is only available on the IBM System z9 109, IBM System z9 EC, and IBM System z9 BC.



- The Crypto Express 2 *Coprocessor* (CEX2C) feature on System z9 enables the user to do the following, using secure keys:

Note: A *secure key* is a key that has been encrypted under another key (usually the master key).

- 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.

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 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

For detailed information on configuring these cryptographic hardware features see IBM Redbook *z9-109 Crypto and TKE V5 Update*, SG24-7123.

Important: The CEX2 feature requires ICSF to be active.

6.12 Integrated Cryptographic Service Facility

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

The CICS support for WS-Security is dependent on these ICSF services and therefore the configuration and startup of ICSF is a requirement for using this support.

Important: ICSF must be configured with cryptographic devices and started in order to use the CICS WS-Security support.

For general information on ICSF see the IBM publication *z/OS Cryptographic Services Integrated Cryptographic Service Facility Overview*, SA22-7519.

6.12.1 Cryptographic hardware requirements for CICS WS-Security

Since CICS relies on ICSF for cryptographic services, and ICSF is dependent on cryptographic devices to perform the actual cryptographic functions, there is a clear dependency on the availability of cryptographic devices when using the CICS WS-Security support. Specific requirements depend on the server that you have.

Table 6-5 presents a list of the ICSF callable services used by the CICS WS-Security support, and a brief summary of each service.

Table 6-5 ICSF callable services used by CICS WS-Security support

ICSF service	Description
CSNBCKM	Multiple clear key import callable service to import a clear single-, double-, or triple-length DATA key that is to be used to encipher or decipher data.
CSNBDEC	Decipher callable service to decipher data in an address space or a data space using the cipher block chaining mode.



ICSF service	Description
CSNBENC	Encipher callable service to encipher data in an address space or a data space using the cipher block chaining mode.
CSNBOWH	One-way hash generate callable service to generate a one-way hash on specified text.
CSNBRNG	Service to generate a random number.
CSNBSYD	Symmetric key decipher callable service to decipher data in an address space or a data space using the cipher block chaining or electronic code book modes.
CSNBSYE	Symmetric key encipher callable service to encipher data in an address space or a data space using the cipher block chaining or electronic code book modes.
CSNDDSG	Digital signature generate callable service to generate a digital signature using a PKA private key.
CSNDDSV	Digital signature verify callable service to verify a digital signature using a PKA public key.
CSNDPKB	Service to build external PKA key tokens containing unenciphered private RSA or DSS keys.
CSNDPKD	Service to decrypt (unwrap) a formatted key value. The service unwraps the key, deformats it, and returns the deformatted value to the application in the clear.

Many of these services require cryptographic hardware to be configured.

Important: On a z9 an optimal cryptographic hardware configuration is a combination of CPACF and CEX2.

The *z/OS 1.8 ICSF Application Programmer's Guide*, SA22-7522-08, documents the cryptographic hardware requirements for the System z9 and other types of servers.

