WebSphere® software

IBM

Implementing an ESB using IBM WebSphere Message Broker V6 and WebSphere ESB V6 on z/OS

Configuration and customization

Qualities of service

Deployment scenarios

Alex Louwe Kooijmans
Egide van Aerschot
Marcel Amrein
Lawrence J. Carnevale
Kirstine Clapperton
Pierangelo Giusta
Niels Nielsen

Redbooks

ibm.com/redbooks

International Technical Support Organization

Implementing an ESB using IBM WebSphere Message Broker V6 and WebSphere ESB V6 on z/OS

April 2007

SG24-7335-00

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

First Edition (April 2007)

This edition applies to WebSphere Message Broker Version 6 for z/OS and WebSphere Enterprise Service Bus for z/OS Version 6.01.

Contents

Notices

Trademarks

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

AIX®	IBM®	System z™
Cloudscape™	IMS™	System z9™
CICS®	OMEGAMON®	SystemPac®
CICSPlex®	RACF®	Tivoli®
DB2 Connect™	Redbooks®	VTAM®
DB2®	Redbooks (logo) ®	WebSphere®
DRDA®	SecureWay®	z/OS®
HiperSockets™	SupportPac™	z9™

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.

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

Microsoft, 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.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Preface

This IBM® Redbooks® publication is designed for IT architects and IT specialists that are dealing with IBM WebSphere® Message Broker and IBM WebSphere Enterprise Service Bus (ESB) solutions.

This book illustrates how to configure an ESB using either IBM WebSphere Message Broker V6 for IBM z/OS® or IBM WebSphere Enterprise Service Bus V6. It makes various traditional z/OS transactions and data available as a Web service through the broker, including IBM DB2® data, batch programs, and IBM CICS/IMS™ transactions.

In this book, there is more focus on WebSphere Enterprise Service Bus installation and customization than on WebSphere Message Broker because the latter has already been documented in other places.

This book describes multiple scenarios that show how to integrate applications using a mix of MQ and SOAP protocols using both WebSphere Message Broker and WebSphere Enterprise Service Bus.

High availability using clustering of brokers in multiple logical partitions (LPARs) on z/OS is also addressed in this book, with details on which feature you can use and configure to improve continuous operation, such as shared queues, shared ports, and Sysplex Distributor.

The team that wrote this IBM Redbooks publication

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

Alex Louwe Kooijmans is a project leader with the ITSO in Poughkeepsie, NY, and specializes in WebSphere, Java™, and service-oriented architecture (SOA) on IBM System z with a focus on integration, security, high availability, and application development. Previously, he worked as a Client IT Architect in the Financial Services sector with IBM in The Netherlands, advising financial services companies on such IT issues as software and hardware strategy and on demand. Alex has also worked at the Technical Marketing Competence Center for IBM System z™ and Linux® in Boeblingen, Germany, providing support to customers starting up with Java and WebSphere on System z. From 1997 to 2002, Alex completed a previous assignment with the ITSO, managing various IBM Redbooks projects and delivering workshops around the world.

Egide van Aerschot holds an engineering degree in Electricity and Nuclear Physics from the University of Leuven, Belgium. He joined IBM in 1967, and was responsible for many computer installations related to Tele Processing and Database Management in Belgium. In 1997, he moved from IBM Belgium to IBM France, where he works as an architect and consultant at the IBM Program Support Center (PPSC) in Montpellier. Since 1997, Egide has specialized in Java and WebSphere applications mainly on z/OS systems, participating in many projects related to the Internet. He is the co-owner of the patent “Methods, systems, program product for transferring program code between computer processes”.

Marcel Amrein is an IT Specialist in the IBM System z WebSphere software Technical sales group in Germany. He has been at IBM for 20 years, 13 years of which he has been working for the IBM Training group instructing customer classes for IBM CICS® and MQ. He has been a member of the development team for worldwide used course material in the area of CICS and MQ implementation and administration. Marcel is a Certified WebSphere MQ System Administrator and Solution Designer. He holds a German Diploma in Communications Engineering from the University of Corporate Education Stuttgart.

Lawrence J. Carnevale is a Consulting IT Specialist at the IBM Design Center in Poughkeepsie, N.Y. USA. He has 27 years of experience in technology. He holds a degree in Computer Science from William Paterson University of New Jersey. His areas of expertise include Business Integration, SOA, and ESB.

Kirstine Clapperton is a Software Engineer at the Hursely Laboratory in the U.K. She has five years of experience in the software engineering field and has been a member of the z/OS WebSphere Message Broker L3 service team for the last three years. She holds a first class honours degree in Computing and Information systems from the University of London and is close to completing her MSc in Software Engineering from the University of Oxford. Her areas of expertise include Integration Middleware Software, primarily WebSphere MQ and WebSphere Message Broker products.

Pierangelo Giusta is a Senior IT Specialist located in Italy. He has been with IBM for 26 years, covering various technical positions, and has been with the IBM Software Group since 1995. He holds a degree in Electronic Engineering from the University of Genova, Italy. His main area of expertise is communication and messaging infrastructure, with special regard to WebSphere MQ, Message Broker, and ESB. In the past, he also worked in the area of IMS.

Niels Nielsen is an Advisory IT Architect in Denmark. He has 22 years of experience in IT (three years with IBM). His areas of expertise include architecting SOA and ESB solutions designed for high availability and high transaction volumes. Previously, he has been architecting software solutions within Customer Relationship Management (CRM), loyalty, and card management, primarily within the oil industry.

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

Paola Bari, Rich Conway, Robert Haimowitz
International Technical Support Organization, Poughkeepsie Center

Paul Dennis
WebSphere MQ for z/OS Development, IBM UK

Carl Farkas
WebSphere on System z Business Integration Consultant, IBM France

Peter Johnson
WebSphere MQ and ESB Development, IBM UK

Ronald Lotter
WebSphere Enablement Team, IBM Raleigh

Wayne Swales
Certified Consulting I/T Specialist, IBM China

Become a published author

Join us for a two-week to six-week residency program! Help write an IBM Redbooks publication dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an e-mail to:

  redbooks@us.ibm.com

- Mail your comments to:

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

1

ESB overview

This chapter discusses what you can expect when you are building an Enterprise Service Bus (ESB). This chapter also discusses the particular components used in the implementation of our ESB.

1.1 What is an ESB

An Enterprise Service Bus is an architecture construct created by combining components to provide a base set of message-driven services as part of an overall infrastructure architecture (see Example 1-1). The ESB, in itself, is not a service-oriented architecture (SOA). The ESB contributes to an SOA by providing functionality, which is used as part of an SOA implementation.

There is a general misunderstanding that an ESB is primarily a Web services facility. The ESB must be able to accommodate the use of Web services as part of its functionality. However, the ESB must not be limited to only providing Web services and must be designed to support the requirements of the enterprise using the ESB. The ESB must provide transport, mediation, routing, and data handling as required to remove the necessity for coupling between requestors and providers.

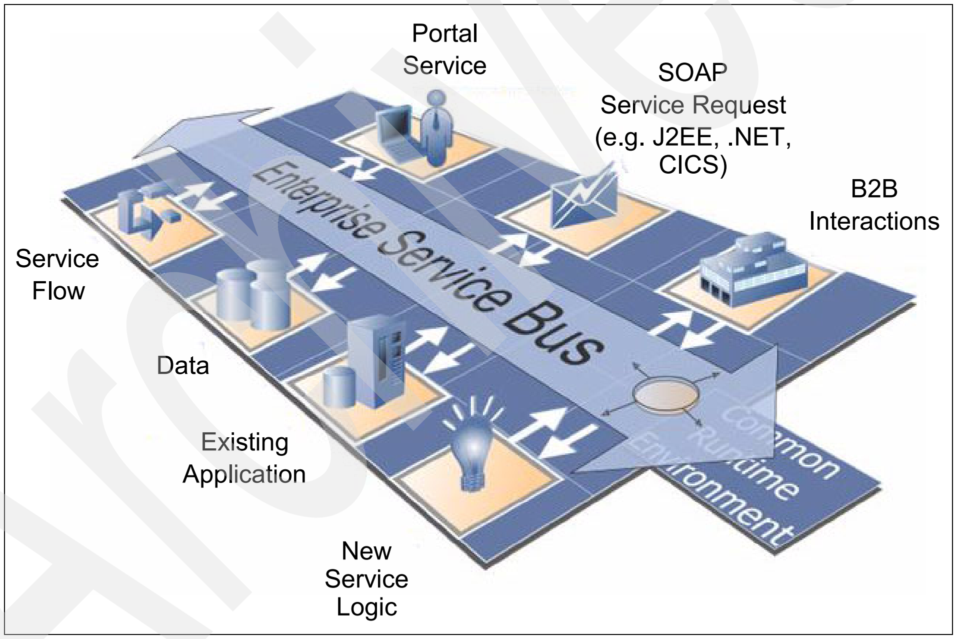


Figure 1-1 Conceptual view of ESB

1.2 Minimum ESB capabilities

This section describes the minimum and extended capabilities that might be addressed when designing an ESB.

The minimum ESB capabilities are an agreed-upon list of capabilities that define an ESB. If a given infrastructure does not meet all these requirements, it cannot be termed an ESB.

The ESB must support at a minimum:

- Communications
- Integration
- Service interaction
- Management

The following sections discuss these requirements in more detail.

1.2.1 Communications

An ESB must supply a communication layer to support service interactions. It must support communication through a variety of protocols. It must provide underlying support for message and event-oriented middleware and integrate with the existing Hypertext Transfer Protocol (HTTP) infrastructure and other Enterprise Application Integration (EAI) technologies.

An ESB must be able to communicate between all these communication technologies through a consistent naming and administration model. The ESB must support at least one messaging style (request/response, pub/sub, and so on) and at least one transport protocol that is or can be made widely available. Particularly in an integrated ESB scenario, the additional ESB must be able to support service interactions provided by the original ESB over one or more available protocols.

1.2.2 Integration

An ESB must support linking to a variety of systems that do not directly support service-style interactions so that a variety of services can be offered in a heterogeneous environment.

This includes existing systems, packaged applications, and other EAI technologies. Integration technologies might be protocols (for example, Java Database Connectivity (JDBC™), File Transfer Protocol (FTP), electronic data interchange (EDI)), or adapters such as the Java 2 Platform, Enterprise Edition (J2EE™) Connector architecture resource adapters or WebSphere business integration adapters.

It also includes service client invocation through client application programming interfaces (APIs) for various languages (such as Java, C+, and C#) and platforms (such as J2EE and Microsoft® .NET), Common Object Request Broker Architecture (CORBA), and Remote Method Invocation (RMI).

1.2.3 Service interaction

An ESB must support SOA concepts for the use of interfaces and support declaration service operations and quality of service requirements.

An ESB must also support service messaging models consistent with those interfaces and be capable of transmitting the required interaction context, such as security, transaction, or message correlation information.

1.2.4 Management

Similar to any other infrastructure component, an ESB must have administration capabilities to enable it to be managed and monitored and, therefore, to provide a point of control over service addressing and naming. In addition, it must be capable of integration into systems management software.

1.3 Extended ESB capabilities

In addition to the minimum ESB capabilities, there is a set of extended ESB capabilities. The detailed requirements of any particular scenario drives extended ESB capabilities, which you can then use to select specific, appropriate products.

In particular, the following types of requirements are likely to lead to the use of more sophisticated technologies, either now or over time:

- Quality of service
- Integration
- Security
- Service level
- Modeling
- Message processing
- Infrastructure intelligence
- Management and autonomic

The following sections discuss these requirements in a little more detail.

1.3.1 Quality of service

An ESB might be required to support service interactions that require different qualities of service to protect the integrity of data mediated through those interactions. This might involve transactional support, compensation, and levels of delivery assurance. These features must be variable and driven by service interface definitions.

1.3.2 Integration

As additional integration capabilities could be supported, the ESB must be capable of connectivity to a wide range of different service providers, using adapters and EAI middleware. They must be capable of data enrichment to alter the service request content and destination on route, and to map an incoming service request to one or more service providers.

1.3.3 Security

An ESB must ensure that the integrity and confidentiality of the services that they carry are maintained. They must integrate with the existing security infrastructures to address the essential security functions, such as:

- Identification and authentication
- Access controls
- Confidentiality
- Data integrity
- Security management and administration
- Disaster recovery and contingency planning
- Incident reporting

Additionally, the ESB must integrate with the overall management and monitoring of the security infrastructure. The ESB might provide security either directly or by integrating with other security components, such as authentication, authorization, and directory components.

1.3.4 Service level

An ESB must mediate interactions between systems supporting specific performance, availability, and other requirements. They must offer a variety of techniques and capabilities to meet these requirements.

An ESB must provide support that enables technical and business service level agreements to be monitored and enforced.

1.3.5 Modeling

An ESB must support the increasing array of cross-industry and vertical standards in both the Extensible Markup Language (XML) and Web services spaces.

The ESB must support custom message and data models. It must also support the use of development tooling and be capable of identifying different models for internal and external services and processes.

1.3.6 Message processing

An ESB must be capable of integrating message, object, and data models between the application components of an SOA. It must also be able to make decisions, such as routing, based on content of service messages, in particular when the services are defined on an integrated ESB.

An ESB must have a mediation model that enables message processing to be customized. The model must also allow sequencing of infrastructure services (for example, security logging and monitoring) around business services invocations.

Mediations can be located close to consumers, providers, or anywhere in the ESB infrastructure transparent to consumers and providers. Mediations can also be chained. An ESB must be able to validate content and format.

1.3.7 Infrastructure intelligence

An ESB must be capable of evolving toward a more autonomic, on demand infrastructure. It must allow business rules and policies to affect ESB functions, and it must support pattern recognition.

1.3.8 Management and autonomic

In addition to basic management capabilities, the ESB must also support the migration to autonomic and On Demand infrastructure by supporting metering and billing, self-healing, and dynamic routing, and the ESB must react to events to self-configure, heal, and optimize. This is a far reaching view of what the ESB might someday evolve to be.

1.4 WebSphere Enterprise Service Bus

WebSphere Enterprise Service Bus is the mediation layer that runs on top of the transport layer within WebSphere Application Server. As such, WebSphere Enterprise Service Bus provides pre-built mediation functions and easy-to-use tools to enable rapid construction and implementation of an ESB as a value-add on top of WebSphere Application Server. From this definition, we realize that WebSphere Enterprise Service Bus for z/OS is a defining element, an integration pillar, that takes advantage of the qualities of z/OS and WebSphere Application Server for z/OS to offer availability as the transport, transformation, and mediation layer.

WebSphere Enterprise Service Bus supports the integration of service-oriented, message-oriented, and event-driven technologies to provide standards-based, messaging infrastructure to companies wanting a fast start to an Enterprise Service Bus. WebSphere Enterprise Service Bus is based on the robust J2EE 1.4 infrastructure and the associated platform services provided by WebSphere Application Server Network Deployment. See Figure 1-2.

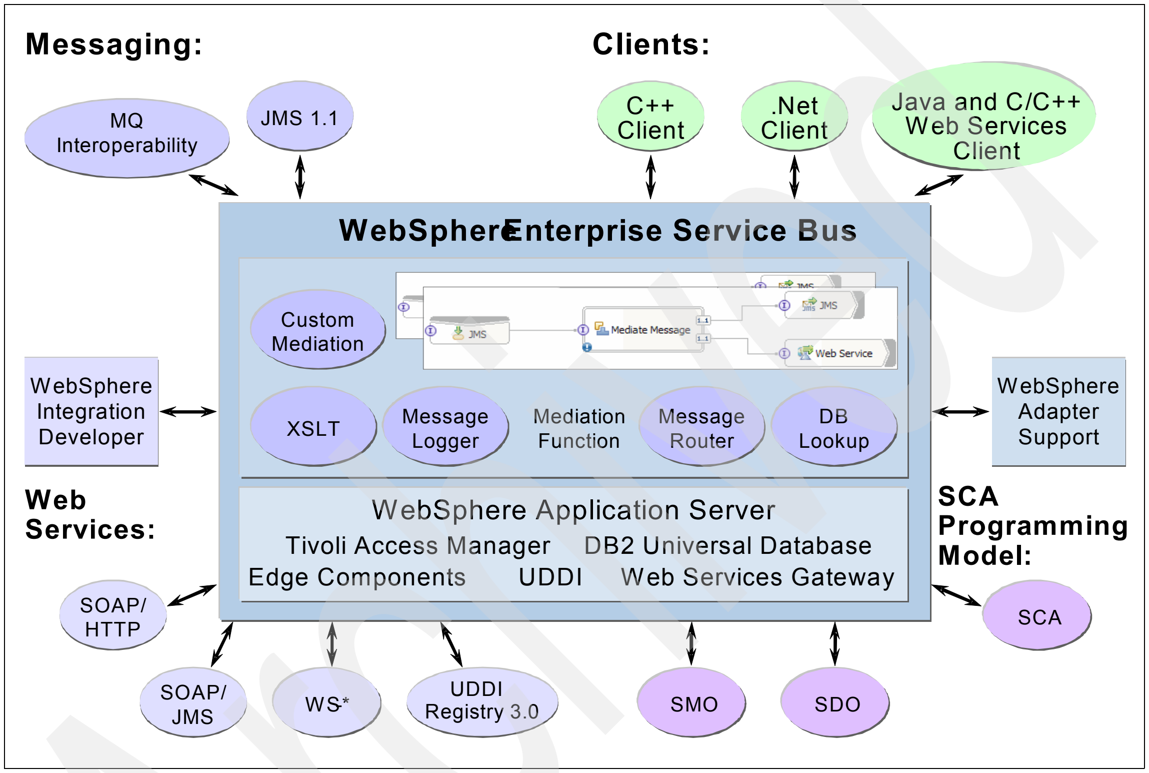


Figure 1-2 WebSphere Enterprise Service Bus components and associations

1.5 WebSphere Enterprise Service Bus capabilities

The following sections provide a summary of the WebSphere Enterprise Service Bus capabilities.

1.5.1 Message routing

WebSphere Enterprise Service Bus manages the flow of messages between Service Component Architecture (SCA)-described interaction endpoints and enables the quality of interaction that these components require. Mediation modules within the ESB handle mismatches between requesters and providers, including protocol or interaction-style mismatches and interface mismatches. In an overall SCA-based solution, mediation modules are a type of SCA module that perform a special role, and therefore have slightly different characteristics than other components that operate at the business level.

1.5.2 Mediation

Mediation components operate on messages exchanged between service endpoints. In contrast with regular business application components, they are concerned with the flow of the messages through the infrastructure and not just with the business content of the messages. Rather than performing business functions, they perform routing, transformation, and logging operations on the messages. The information that governs their behavior is often held in headers flowing with the business messages. The IBM SOA programming model introduces the service message object (SMO) pattern for Service Data Objects (SDOs) to support this pattern.

1.5.3 Service endpoint connectivity

WebSphere Enterprise Service Bus supports advanced interactions between service endpoints on three levels, which are as follows:

- Broad connectivity
- A spectrum of interaction models and qualities of interaction
- Mediation capabilities

The product supports connectivity between endpoints through a variety of protocols and application programming interfaces, such as:

- Java Message Service (JMS) 1.1, provided by WebSphere Application Server, on which WebSphere Enterprise Service Bus is built. Applications can exploit a variety of transports, including TCP/IP, Secure Sockets Layer (SSL), HTTP, and Hypertext Transfer Protocol Secure (HTTPS).
- WebSphere Enterprise Service Bus supports Web services standards that enable applications to make use of Web service capabilities.
  - SOAP/HTTP, SOAP/JMS, and WSDL 1.1
  - UDDI 3.0 Service Registry and Web Services Gateway
  - WS-* Standards, including WS-Security and WS-Atomic Transactions
- For back-end applications (such as SAP®), several IBM WebSphere Adapters (based on the J2EE Connector architecture (J2C)) are available.

Because it is built on WebSphere Application Server, WebSphere Enterprise Service Bus can provide smooth interoperability with other products in the WebSphere portfolio, including IBM WebSphere MQ and IBM WebSphere Message Broker. It can use IBM WebSphere Adapter solutions to leverage existing application assets, and also capture and disseminate business events.

Client interfaces included with WebSphere Enterprise Service Bus further extend this connectivity as follows:

- The message clients for C/C++ and Microsoft .NET enable non-Java applications to connect to WebSphere Enterprise Service Bus using an API similar to the JMS API.
- The Web services client for C++ is similar to the Java API for XML-based Remote Procedure Call (JAX-RPC) and enables users to connect to Web services hosted on WebSphere Application Server from within a C++ environment.

Other features at the connectivity level perform basic protocol conversion between endpoints where the protocol used by the requester to dispatch requests (such as SOAP over HTTP) is different from that of the service provider that handles those requests (such as SOAP over JMS).

1.5.4 WebSphere Enterprise Service Bus components

The following sections provide a description of the WebSphere Enterprise Service Bus components.

Service requestors and providers

In an Enterprise Service Bus, applications that require the services of another application are known as *service requesters*. The corresponding applications that offer services are called *service providers*.

A service requester might send a service request over the bus. Upon receiving the request, a service provider runs the requested service. The service provider might also send a reply back through the bus to the requester. See Figure 1-3.

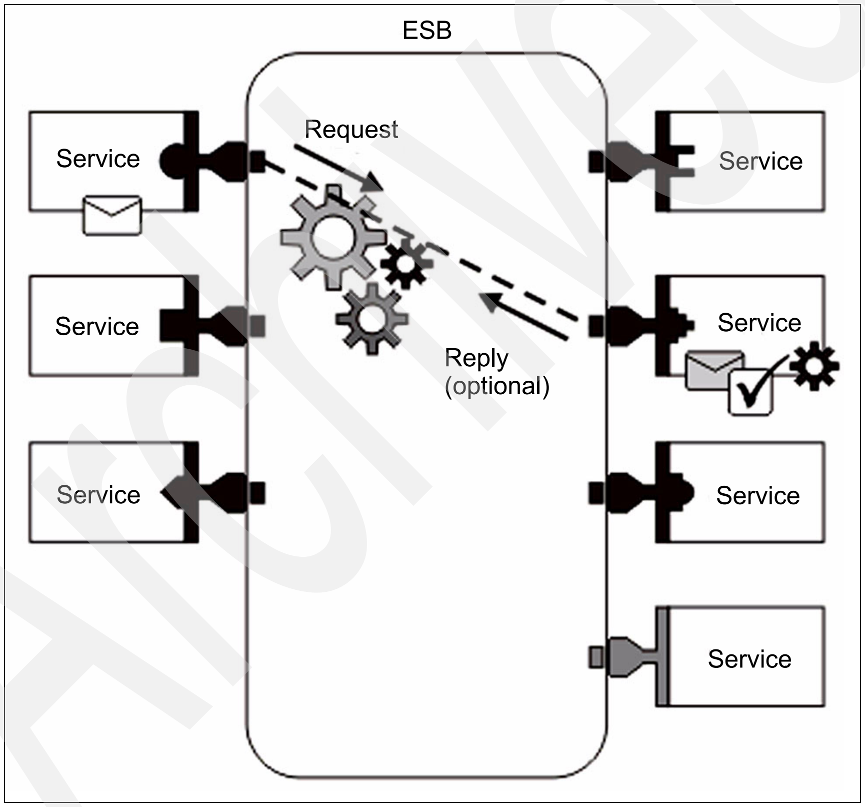


Figure 1-3 A service requester sending a request over the ESB

WebSphere Enterprise Service Bus can interconnect a variety of different requesters and providers using standard protocols that include the following:

- JMS
- SOAP over HTTP Web services
- SOAP over JMS Web services

For common applications (such as SAP) and other protocols, IBM WebSphere Adapters and other industry-standard J2C adapters are available.

WebSphere Enterprise Service Bus supports diverse messaging interaction models to meet your requirements, including the following models:

- Request-reply
- Distribution models for one-way interactions
- Publish/subscribe

Mediation modules

As a request travels between a service requester and provider, the logic that controls what happens to it and how it is routed (often known as the mediating or service interaction logic) can be inserted. This mediating logic is performed by *mediation modules*, which are made up of imports, exports, and flow components.

Imports

Imports let the mediation module reference external services as though they were local services.

Exports

Exports expose a mediation module's external interfaces (or access points) to a client who wishes to use a function of the module as a service.

Flow components

Flow components can carry out the service integration logic, including:

- Routing
- Database lookup
- Database logging
- Structure transformation

Figure 1-4 shows the mediation module forwarding request messages to another module.

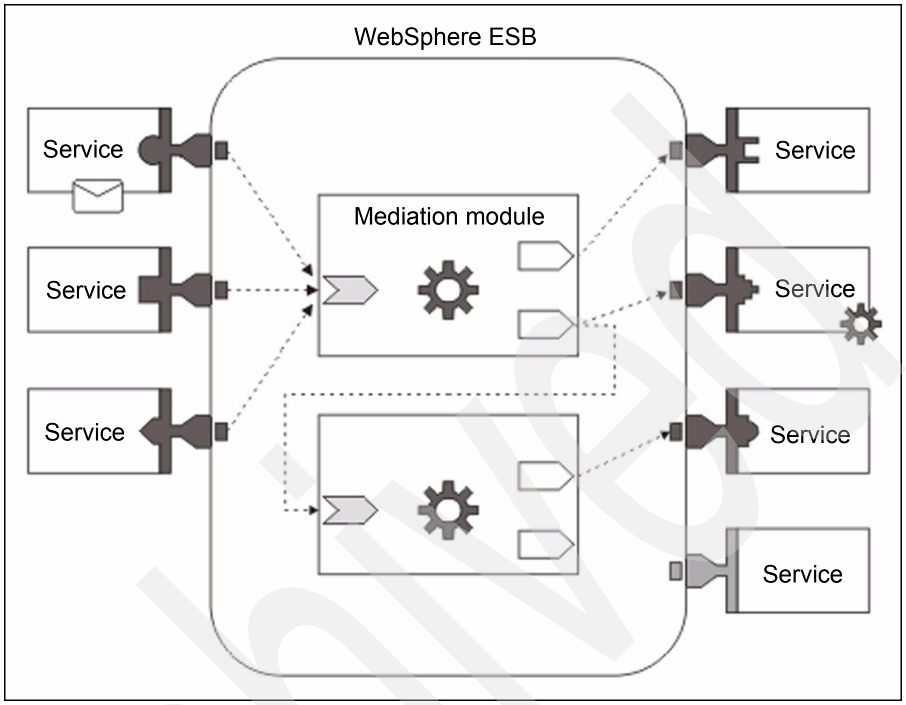


Figure 1-4 Mediation module forwarding request messages to another mediation module

1.5.5 WebSphere Enterprise Service Bus development of services

WebSphere Integration Developer is the separate development environment for WebSphere Enterprise Service Bus. You can use WebSphere Integration Developer to graphically model and assemble mediation components from mediation primitives, and assemble mediation modules from mediation components.

- If the interfaces for SCA mediation components are not imported, you can use the Simplified Interface Editor to create the interface. You can use this editor to specify and edit interfaces (operations and parameters) of mediation modules.
- You can use the Mediation Flow Editor to map between operations on the endpoints of a mediation and to define the set of mediation flows required for this application. You can use a set of predefined mediation primitives to visually compose a mediation flow.

- You can use the Business Object Editor to construct the messages that are used in mediations.
- You can use other editors to extend the development environment to meet your business requirements, for example:
  - Create and edit custom mediation primitives, and add them to the Mediation Flow Editor.
  - Create and edit message descriptors.

You can develop some service components using other application development tools and then import them into WebSphere Integration Developer for modelling, editing, testing, and packaging for deployment into WebSphere Enterprise Service Bus.

1.5.6 WebSphere Enterprise Service Bus deployment of services

Deploying is the act of enabling your applications in either a test or a production environment. While the concept of deploying is the same for both environments, there are a few differences between the deployment task in each environment.

After developing a service application, best practices state that you must deploy the application on to a test server for testing before committing it to the production requirement. Use WebSphere Integration Developer to deploy the applications into a test environment, and to package a service application as a standard enterprise application package for deployment into WebSphere Enterprise Service Bus.

Use WebSphere Enterprise Service Bus to install and deploy the applications into a production environment. In WebSphere Enterprise Service Bus, you can use the standard WebSphere administrative console, with role-based administration views that simplify the experience for solution administrators to deploy and manage the components of service integration packages.

1.5.7 WebSphere Enterprise Service Bus administration

WebSphere Enterprise Service Bus uses the standard WebSphere administrative console used for WebSphere Application Server and other related products, such as WebSphere Process Server. As a result, many administrative tasks (for example, setting security, viewing logs, and installing applications), and many of the terms used are the same for both WebSphere Enterprise Service Bus and WebSphere Application Server.

If you have installed a WebSphere Enterprise Service Bus stand-alone profile, then the bus environment comprises a single node in its own administrative domain known as a cell. You can use the administrative console to manage applications, buses, servers, and resources within the administrative domain.

Similarly, if you have installed WebSphere Enterprise Service Bus to create a Deployment Manager cell, the bus environment comprises a Deployment Manager node and one or more managed nodes in the same cell, which forms the administrative domain. In addition to the application-related items, you can use the administrative console to set up managed nodes into the cell, and can monitor and control the nodes and the resources on those nodes. You can use the administrative console to manage items at a cell, node, or server scope.

1.5.8 WebSphere Application Server

The WebSphere Enterprise Service Bus mediation modules must be deployed on a WebSphere Enterprise Service Bus-enabled application server. Requester and provider applications can run on the same or any Java-based or J2EE-based application server that supports Web services, .NET, and so on.

WebSphere Enterprise Service Bus offers JMS messaging and best-in-class interoperability with other WebSphere MQ and Broker products. It also provides a comprehensive clients package for connectivity to C, C++, .NET Messaging, and C and C++ Web service clients.

Its SCA programming model enables ease of use, while Web services support allows you to incorporate leading-edge capabilities.

WebSphere Enterprise Service Bus is built on top of *WebSphere Application Server Network Deployment* and inherits its qualities of service, its world class clustering, failover, scalability, and security.

It also includes a number of key features, including UDDI as a services registry, the Web services Gateway, Tivoli® Access Manager, DB2, and Edge Components.

Figure 1-5 shows the WebSphere Enterprise Service Bus built on top of WebSphere Application Server Network Deployment.

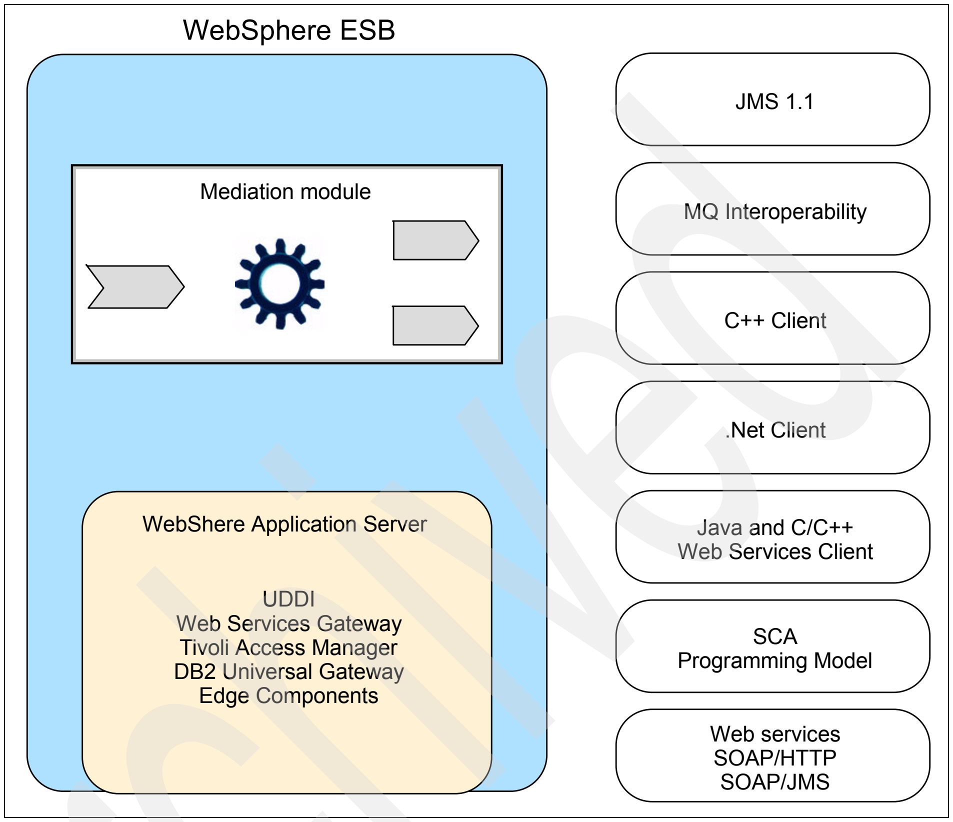


Figure 1-5 WebSphere Enterprise Service Bus built on top of WebSphere Application Server Network Deployment

1.6 WebSphere Message Broker

WebSphere Message Broker is a powerful information broker that allows both business data and information, in the form of messages, to flow between disparate applications and across multiple hardware and software platforms. Business rules can be applied to the data that is flowing through the message broker in order to route, store, retrieve, and transform the information.

1.6.1 WebSphere Message Broker editions

There are three editions of the WebSphere Message Broker product, as described in the following sections.

WebSphere Event Broker

WebSphere Event Broker is used for the distribution and routing of messages from disparate applications. It can distribute information and data, which is generated by business events in real time, to people, applications, and devices throughout an enterprise. WebSphere Event Broker has support for multiple transport protocols and extends the flow of information in an organization beyond point-to-point, utilizing flexible distribution mechanisms such as publish/subscribe and multicast.

WebSphere Message Broker

WebSphere Message Broker contains all the functionality of WebSphere Event Broker and extends it to include additional capabilities to enable storage, complex transformation, intelligent routing, and enrichment of data flowing through the broker. Detailed capabilities of the product are described in the following sections and are based upon the functional capabilities of the WebSphere Message Broker specifically. This is the version of the Broker that is most commonly used today.

Rules and Formatter Extension

WebSphere Message Broker with *Rules and Formatter Extension* includes the Rules and Formatter Extension from New Era Of Networks, which provides Rules and Formatter nodes and associated runtime elements. These support the functionality supplied with earlier releases of WebSphere MQ Integrator. This component is now provided by Sybase. The functionality provided by the Rules and Formatter Extension is not discussed any further in this book.

1.6.2 WebSphere Message Broker capabilities

The primary capabilities of WebSphere Message Broker are message routing, message transformation, message enrichment, and publish/subscribe. Together, these capabilities make WebSphere Message Broker a powerful tool for business integration.

Message routing

WebSphere Message Broker provides connectivity for both standards based and non-standards based applications and services. The routing can be simple point-to-point routing or it can be based on matching the content of the message to business rules defined to the broker.

WebSphere Message Broker contains a choice of transports that enables secure business to be conducted at any time and any place, providing powerful integration using mobile, telemetry, and Internet technologies. WebSphere Message Broker is built upon WebSphere MQ and therefore supports the same transports. However, it also extends the capabilities of WebSphere MQ by adding support for other protocols, including real-time Internet, intranet, and multicast endpoints.

WebSphere Message Broker supports the following transports:

- WebSphere MQ Enterprise Transport
- Web Services Transport
- WebSphere MQ Real-time Transport
- WebSphere MQ Multicast Transport
- WebSphere MQ Mobile Transport
- WebSphere MQ Telemetry Transport
- JMS Transport (HTTP and HTTPS)

In addition to the supplied transports, the facility exists for users to write their own input nodes to accept messages from other transports and formats as defined by the user.

Message transformation and enrichment

Transformation and enrichment of in-flight messages is an important capability of WebSphere Message Broker, and allows for business integration without the requirement of any additional logic in the applications themselves.

You can transform messages between applications to use different formats, for example, transforming from a custom format in an existing system to XML messages that you can use with a Web service. This provides a powerful mechanism to unify organizations, because business information can now be distributed to applications that handle completely different message formats without a requirement to reprogram or add to the applications themselves.

Messages can also be transformed and enriched by integration with multiple sources of data, such as databases, applications, and files. This allows for any type of data manipulation, including logging, updating, and merging. For the messages that flow through the broker, you can store business information in databases or you can extract them from databases and files and add to the message for processing in the target applications.

You can perform complex manipulation of message data using the facilities provided in the Message Broker Toolkit, such as ESQL and Java.

In WebSphere Message Broker, message transformation and enrichment is dependant upon a broker understanding the structure and content of the incoming message. Self-defining messages, such as XML messages, contain information about their own structure and format. However, before other messages, such as custom format messages, can be transformed or enhanced, a message definition of their structure must exist. The Message Broker Toolkit contains facilities for defining messages to the message broker. These facilities are discussed in more detail in the following sections.

Publish/subscribe

The simplest way of routing messages is to use point-to-point messaging to send messages directly from one application to another. Publish/subscribe provides an alternative style of messaging in which messages are sent to all applications that have subscribed to a particular topic.

The broker handles the distribution of messages between publishing applications and subscribing applications, and also applications publishing on or subscribing to many topics, where more sophisticated filtering mechanisms can be applied.

An improved flow of information around the business is achieved through the use of publish/subscribe and the related technology of multicast. These flexible distribution mechanisms move away from hardcoded point-to-point links.

1.6.3 WebSphere Message Broker components

WebSphere Message Broker is comprised of two principle parts: a *development environment* for the creation of message flows, message sets, and other message flow application resources, and a *runtime environment*, which contains the components for running those message flow applications that are created in the development environment.

1.6.4 WebSphere Message Broker development environment

The development environment is where the message flow applications that provide the logic to the broker are developed. The broker uses the logic in the message flow applications to process messages from business applications at runtime. In the Message Broker Toolkit, you can develop both message flows and message sets for message flow applications.

Application Development perspective

The *Application Development perspective* is the part of the Message Broker Toolkit that is used to design and develop message flows and message sets. It contains editors that create message flows, and transformation code, such as ESQL or Java, and define messages.

This perspective is also used for the deployment of message flows and message sets to brokers in the defined broker domains.

The Administration perspective also contains tools for creating WebSphere Message Broker archive (bar) files. Bar files are used to deploy message flow application resources, such as message flows and message sets.

The Administration perspective also contains tools to help test message flows. These tools include Enqueue and Dequeue for putting and getting messages from WebSphere MQ queues.

Configuration Manager

The *Configuration Manager* is the interface between the Message Broker Toolkit and the brokers in the broker domain. The Configuration Manager stores configuration details for the broker domain in an internal repository, providing a central store for resources in the broker domain.

The Configuration Manager is responsible for deploying message flow applications to the brokers. The Configuration Manager also reports back on the progress of the deployment and on the status of the broker. When the Message Broker Toolkit connects to the Configuration Manager, the status of the brokers in the domain is derived from the configuration information stored in the Configuration Manager's internal repository.

Message flows

Message flows are programs that provide the logic that the broker uses to process messages from business applications. Message flows are created by connecting *nodes* together, with each node providing some basic logic. A selection of built-in nodes is provided with WebSphere Message Broker for performing particular tasks. These tasks can be combined to perform complex manipulations and transformations of messages.

A choice of methods is available for defining the logic in the message flows to transform data. Depending on the different types of data or the skills of the message flow application developer, the following options are available:

- Extended Structured Query Language (ESQL)
- Java
- Extensible Style-sheet Language for Transformations (XSLT)
- Drag-and-drop mappings

The nodes in the message flows define the source and the target transports the message, any transformations and manipulations based on the business data, and any interactions with other systems such as databases and files.

Message sets

A *message set* is a definition of the structure of the messages that are processed by the message flows in the broker. As mentioned previously, in order for a message flow to be able to manipulate or transform a message, the broker must know the structure of the message. You can use the definition of a message to verify the message structure and to assist with the construction of message flows and mappings in the Message Broker Toolkit.

Message sets are compiled for deployment to a broker as a message dictionary, which provides a reference for the message flows to verify the structure of messages as they flow through the broker.

1.6.5 WebSphere Message Broker runtime environment

As shown in Figure 1-6, the runtime environment is the set of components that are required to deploy and run message flow applications in the broker.

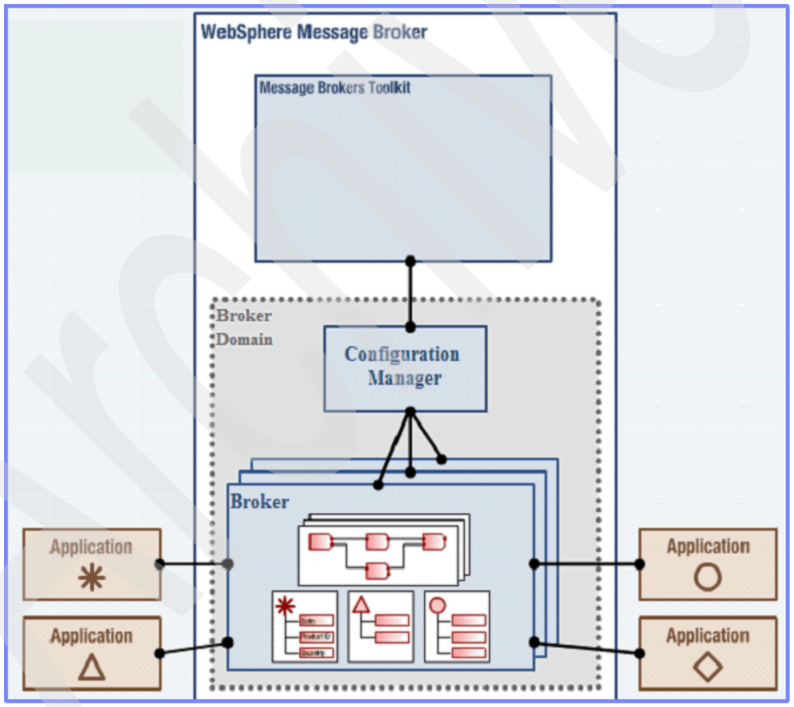


Figure 1-6 Runtime environment

Broker

The broker is a set of application processes that host and run message flows. When a message arrives at the broker from a business application, the broker processes the message before passing it on to one or more other business applications. The broker routes, transforms, and manipulates messages according to the logic that is defined in their message flow applications.

A broker uses WebSphere MQ as the transport mechanism both to communicate with the Configuration Manager, from which it receives configuration information, and to communicate with any other brokers to which it is associated.

Each broker has a database in which it stores the information that it has to process messages in at runtime.

Execution groups

Execution groups enable message flows within the broker to be grouped together. Each broker contains a default execution group. You can create additional execution groups, if they are given unique names within the broker.

Each execution group is a separate operating system process and, therefore, the contents of an execution group remain separate from the contents of other execution groups within the same broker. This can be useful for the isolating pieces of information in regard to security, because the message flows execute in separate address spaces or as unique processes.

Message flow applications are deployed to a specific execution group. To enhance performance, the same message flows and message sets can be running in different execution groups.

Broker domain

Brokers are grouped together in *broker domains*. The brokers in a single broker domain share a common configuration that is defined in the Configuration Manager. A broker domain contains one or more brokers and a single Configuration Manager. It can also contain a *User Name Server*. The components in a broker domain can exist on multiple machines and operating systems, and are connected together with WebSphere MQ channels. A broker might belong to only one broker domain.

User Name Server

A *User Name Server* is an optional component that is required only when publish/subscribe message flow applications are running, and where extra security is required for applications to be able to publish or subscribe to topics. The User Name Server provides authentication for topic-level security for users and groups that are performing publish/subscribe operations.

2

WebSphere Message Broker configuration & customization

This chapter describes the configuration and customization of WebSphere Message Broker for z/OS that we implemented in order to support the sample application scenarios.

We started from a basic installation of WebSphere Message Broker and WebSphere MQ products on two logical partitions (LPARs) of a z/OS sysplex environment, which is documented to show the names and interactions of the components involved.

This chapter also describes and discusses, in detail, the guidelines and options to configure WebSphere Message Broker for high availability, which, based on the z/OS sysplex full redundancy concept, is primarily managed by WebSphere MQ shared queues and Sysplex Distributor.

2.1 Introducing the base installation

The WebSphere MQ and WebSphere Message Broker products were installed by following the instructions provided in the appropriate product installation manuals.

This section briefly refers to the installation steps and describes our reasoning for any deviations from the default settings provided with the products.

Figure 2-1 shows the components of our Enterprise Service Bus (ESB) solution and their names.

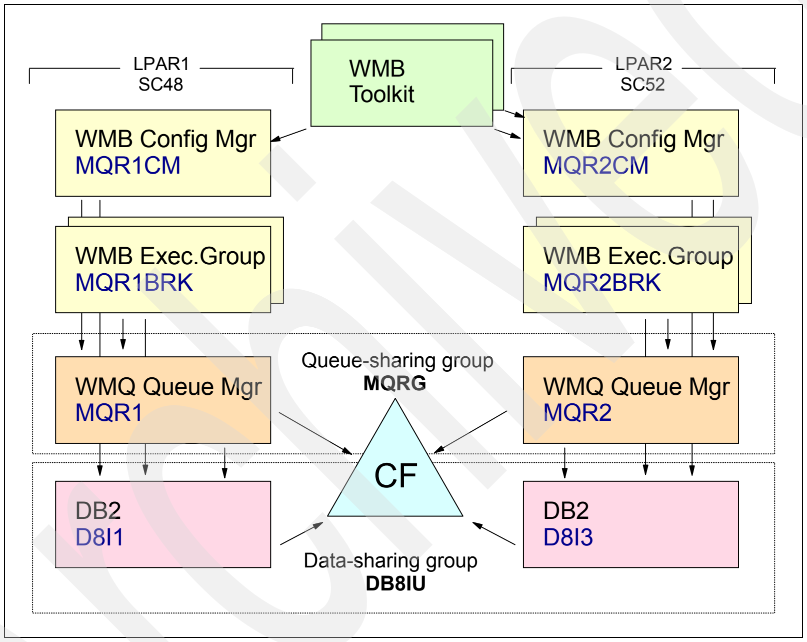


Figure 2-1 WebSphere Message Broker and WebSphere MQ product installations

2.1.1 WebSphere MQ installation and basic configuration

The WebSphere MQ installation and basic configuration is described as follows:

- WebSphere MQ for z/OS has been installed into our z/OS environment using SMP/E, as per the instructions provided in the IBM publication titled *WebSphere MQ for z/OS V6.0 Program Directory*, GI10-2584. The installation included the Client Attachment Feature.

  Our target library high level qualifier is MQ600 and our WebSphere MQ libraries are named MQ600.SCSQ*.

- The Queue Manager configuration followed the guidelines provided by Chapter 2, “Customizing your Queue Managers”, of *WebSphere MQ for z/OS System Setup Guide V6.0*, SC34-6583. The installation specific settings and naming conventions that were applied to our configuration are shown in Table 2-1. Some parameters might have to be reworked for a production environment.

Table 2-1 WebSphere MQ customization: names and special setting

Names	**Special settings**
Queue Manager names	MQR1, MQR2
Command prefix	-MQR1, -MQR2
Queue-sharing group name	MQRG
QSG application structures	MQRGAPPLICATION1 (only one)
Local Listener ports used	1414- by both Queue Managers
Page data sets	five (5), see CSQ4PAGE sample JCL
Log data sets	two times four (2*4) for dual logging, see CSQ4BSDS sample JCL
Log archive	switched off by startup parameter OFFLOAD=NO

Make note of the following:

– All system queues have been defined with the standard attributes provided by the product.

  The system queues to support Java Message Service (JMS) publish/subscribe, the Message Broker, and the server-connection channel for WebSphere Application Server access, which have all been defined with their standard names and attributes. These were provided by the product supplied definitions, namely members CSQ4INSJ and CSQ4INSR in the SCSQPROC library.

– In order to support the CICS MQ Bridge, which is used to run the sample application in two of our application scenarios, the `SYSTEM.CICS.BRIDGE.QUEUE` has been defined to trigger the MQ Bridge monitor task (CKBR). This was configured by the sample definition input in SCSSQPROC(CSQ4CKBM). Backout requeuing has been activated by setting the `BOTRESH` value to 0 (zero) and by defining the requeue queue, `SYSTEM.CICS.BRIDGE.QUEUE`, to the queue managers as a local queue.

– Queue sharing group related settings are documented in 2.2.9, “MQ shared queues” on page 50.

2.1.2 WebSphere Message Broker installation and basic configuration

WebSphere Message Broker for z/OS has been installed using the SMP/E, as per the instructions provided in IBM publication *WebSphere Message Broker for z/OS V6.0 Program Directory, Program Number* 5655-M74. The target library high level qualifier is BIP and the WebSphere Message Broker target libraries are named BIP.V6R0M0.**.

Message Broker configuration

The brokers were created using the guidelines provided by the chapter titled "Creating a broker on z/OS", (ae22400_) in the WebSphere Message Broker online documentation.

All the suggested steps were followed in the following specified order:

1. Information required to create a broker.
2. Creating the broker PDSE.
3. Creating the broker directory on z/OS.
4. Customizing the broker component data set.
5. Customizing the broker JCL.
6. Creating the environment file.
7. Prime DB2.
8. Creating the broker component.
9. Copy the started task to the procedures library.

Table 2-2 shows the applied installation settings and naming conventions.

Table 2-2 WebSphere Message Broker component information: broker1, broker2

Names	Broker 1	Broker 2
Broker name	MQR1BRK	MQR2BRK
Started Task userID	MQR1BRK	MQR2BRK
Associated Queue Mgr	MQR1	MQR2
Associated Configuration Mgr	MQR1CM	MQR2CM
Home directory	/u/mqr1brk	/u/mqr2brk
Component directory	/var/wmqi/MQR1BRK	/var/wmqi/MQR2BRK
Broker JCL Dataset	MQR1BRK.BIPCNTL	MQR2BRK.BIPCNTL
Broker DB2 storage group	MQR1STOR	MQR2STOR
Broker DB2 database	DMQR1BRK	DMQR2BRK

From SDSF, start the component, for example, /S MQR1BRK.

Note: The STDOUT section of the Broker joblog displays the results of the Customization Verification Program, which is automatically executed when the component is started.

Configuration Manager

The Configuration Managers were created using the guidelines provided by the chapter titled "Creating a Configuration Manager on z/OS", (ae24280_) in the WebSphere Message Broker online documentation.

All the suggested steps were followed in the following specified order:

1. Installation information: Configuration Manager
2. Component information: Configuration Manager
3. Creating the Configuration Manager PDSE
4. Creating the Configuration Manager directory on z/OS
5. Customizing the Configuration Manager component data set
6. Creating the environment file
7. Customizing the Configuration Manager JCL
8. Creating the Configuration Manager component
9. Copying the started task to the procedures library

Table 2-3 summarizes the applied installation specific settings and naming conventions.

Table 2-3 WebSphere Message Broker component information: configmgr1, configmgr2

Names	**configmgr1**	**configmgr2**
Configmgr name	MQR1CM	MQR2CM
Started Task userID	MQR1CM	MQR2CM
Associated Queue Mgr	MQR1	MQR2
Associated Broker	MQR1BRK	MQR2BRK
Home directory	/u/mqr1cm	/u/mqr2cm
Component directory	/var/wmqi/MQR1CM	/var/wmqi/MQR2CM
Configmgr JCL Dataset	MQR1CM.BIPCNTL	MQR2CM.BIPCNTL

From SDSF, start the component, for example, /S MQR1CM.

Note: The STDOUT section of the Configuration Manager joblog displays the results of the Customization Verification Program, which is automatically executed when the component is started.

Creating access control lists within the Configuration Manager

WebSphere Message Broker uses *access control lists (ACLs)* to govern which users and groups can manipulate objects within the Configuration Manager and Message Broker Toolkit. There are four different access levels that can be granted for a user or group, which are Full, View, Deploy, and Edit. The topic “ACL permissions”, (ap12520_) in the WebSphere Message Broker online documentation describes which permissions you can apply to each object and summarizes the actions that the user or group can perform.

Edit and submit the MQR2CM.BIPCNTL(BIPCRACL) member to create the necessary ACLs for your Configuration Manager.

2.2 WebSphere Message Broker or WebSphere MQ configuration for high availability

The high availability features provided by a z/OS sysplex are, in many cases, the decisive criterion for running the ESB components on z/OS.

This section discusses the special configuration settings of WebSphere MQ and WebSphere Message Broker that exploit z/OS sysplex functions to provide a high availability environment.

The basic approach for a z/OS sysplex configuration to provide high availability is the idea that within the sysplex there are multiple components with equivalent functionality. If one of the components fail, its duplicate component takes over the function of the failed one with little or no impact to applications and users.

The worst case scenario is the failure of an entire LPAR, with all the services (address spaces) currently running on the system, coming down. In such a case, equivalent systems on an available LPAR can take over the workload and serve the applications. Generally speaking, there are two modes of operation to handle this type of situation. Either duplicate systems have to be started to take over from the failing ones, or the equivalent systems and applications are continuously active on two or multiple LPARs. In the second scenario, the load of a failing LPAR is redirected to the surviving systems. They absorb the new work in addition to the loads they are already running. You can also see the first mode often referred to as “Active/Passive” and the second as “Active/Active”.

High availability considerations in an application integration environment usually address three types of resources, which are as follows:

- Network based access to systems and services with a network interface
- Access to queues
- Access to databases

High availability from a communications point of view means that a service, which is invoked through the network, continues to be available even if certain components or the complete LPAR fail. In a z/OS sysplex environment, this type of high availability is provided by the use of *Dynamic virtual IP addresses (DVIPA)* in conjunction with the Sysplex Distributor component of the TCP/IP part of the z/OS Communication Manager.

High availability for queue-based processes of all kinds are provided by the WebSphere MQ queue sharing group function, also known as *shared queues*. This section discusses which queues are to be shared and which ones are not, and it also covers different failure scenarios where high availability is provided by the shared queues.

This section does not discuss the concept of highly available data stored in databases in much detail. Take this for granted by presuming the existence and use of a DB2 data sharing group environment where the critical business data is stored in shared databases.

The following sections provide a brief introduction to both the TCP/IP for z/OS Sysplex Distributor and the WebSphere MQ for z/OS queue sharing group function. The purpose here is to explain their basic capabilities to readers who are not yet familiar with these concepts. The subsequent discussions of specific configuration options are based on a number of typical application scenarios, namely the ones we ran as examples, which are described in Chapter 4, “Sample scenarios” on page 195.

2.2.1 An outline of Dynamic VIPA and Sysplex Distributor

The following sections describe frequently used terms and functions related to high availability options with respect to networking. These are predominantly excerpts from the IBM Redbooks publication *TCP/IP Tutorial and Technical Overview,* GG24-3376. For more details, refer to this book.

Virtual IP address

The *virtual IP address (VIPA)* is a network layer address that, unlike a real IP address, does not map to a hardware adapter. Instead, it is always available as a connection endpoint on the path from the z/OS TCP/IP stack to the Internet or an intranet.

In case of a hardware adapter failure, a secondary path over another adapter to the VIPA address is automatically selected by the router software.

Because a VIPA address does not map a hardware adapter, VIPA addresses are defined in a z/OS TCP/IP profile. They are called static VIPA addresses, because they, along with their subsequent definitions, belong to the TCP/IP stack.

Dynamic virtual IP address

In contrast to the static VIPA defined for only one TCP/IP stack, the *dynamic VIPA (DVIPA)* can be moved to another TCP/IP stack if the primary TCP/IP stack fails.

Thus, applications accessed via this DVIPA might be reached (if they are brought up) in another system with the same DVIPA address. If the user reconnects to the application, he is unaware of the change.

Distributed virtual IP address

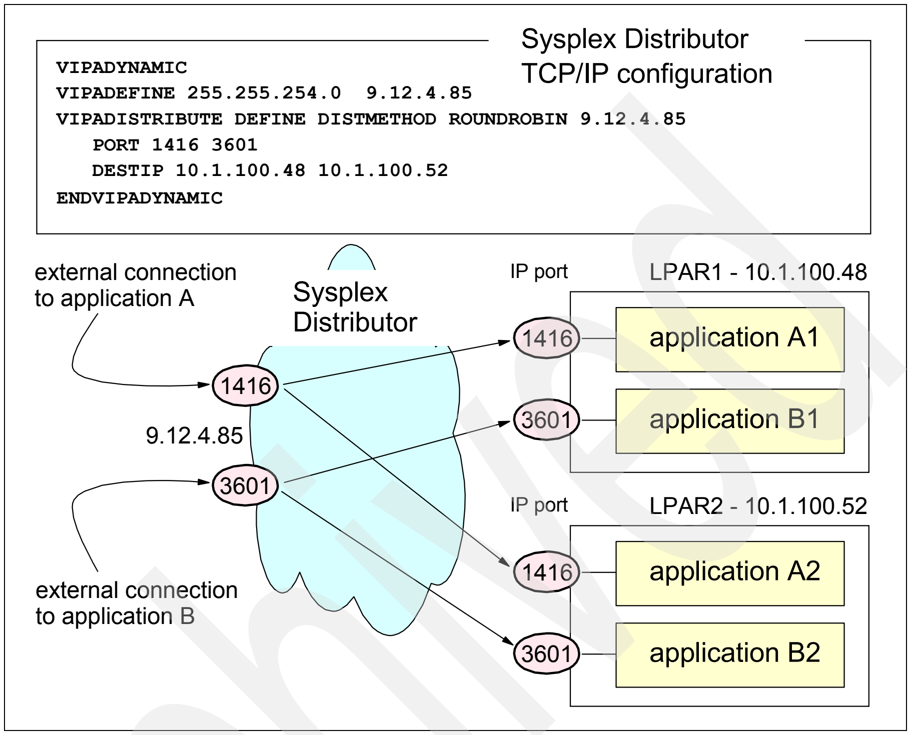
A distributed DVIPA is a special type of DVIPA. It can distribute connections within a sysplex to another TCP/IP stack on the same or another operating system image. It is used by the Sysplex Distributor to forward connections to selected target systems where application servers are located.

Sysplex Distributor

The *Sysplex Distributor* is a TCP/IP function that is defined in the TCP/IP profile. The Sysplex Distributor environment consists of several systems (LPARs) within the sysplex. All TCP/IP stacks are connected via the cross coupling facility to the other systems and via IP-links to the physical network routers.

The TCP/IP configuration profile contains information about DVIPAs and its association to TCP/IP applications. The sysplex distributor has information about which target systems this DVIPA might be distributed.

Figure 2-2 shows a simple but complete set of Sysplex Distributor settings that are part of the TCP/IP configuration, commonly referred to as the TCP/IP profile.



```
VIPADYNAMIC
VIPADEFINE 255.255.254.0  9.12.4.85
VIPADISTRIBUTE DEFINE DISTMETHOD ROUNDROBIN 9.12.4.85
   PORT 1416 3601
   DESTIP 10.1.100.48 10.1.100.52
ENDVIPADYNAMIC
```

Figure 2-2 Sample Sysplex Distributor configuration

The notable facts on the configuration shown are as follows:

- Applications A and B are network enabled systems, such as MQ, CICS, or IMS, running on z/OS. From the network point of view, these are considered applications.
- There are equivalent instances of these systems on both LPARs, named A1, A2 and B1, B2, respectively.
- Each system is listening on a particular IP port. In our example, the port number is 1416 for the A systems and 3601 for the B systems. For the simple configuration that we have used, it is important that both instances of a particular system use the same port number.
- The Sysplex Distributor definitions shown define a new DVIPA with IP address 9.12.4.85, which is used to distribute ports 1416 and 3601 to the target systems (LPARs) with XCF addresses 10.1.100.48 and 10.1.100.52.

- The overall effect of this configuration is that an external connection that specifies 9.12.4.85:1416 as its target, is distributed between port 1416 on LPAR1 and LPAR2, ending up in a connection with application A1 or A2. The same is true for a connection that addresses 9.12.4.85:3601 and application B1 and B2.
  - "Distributed" in this context means that Sysplex Distributor is going to choose either or both targets, if both are available. The system that initiates the connection cannot determine which of the target systems it is going to be connected.
  - Based on the ROUNDROBIN option used in the definition, connections are going to be directed to both targets in turn, as long as both are available.
  - If just one instance of a target system is available, then all external requests are going to go there.

  Note: An automatic backup mechanism in terms of network connectivity is provided for the two instances of an application or system. As long as at least one target system of each type is active, external connection requests can be served.

You can check the effectiveness of Sysplex Distributor settings by means of the display command D TCPIP,,N,VDPT, as shown in Example 2-1.

Example 2-1 Displaying distributed VIPA settings

```
D TCPIP,,N,VDPT
EZZ2500I NETSTAT CS V1R7 TCPIP 318
DYNAMIC VIPA DESTINATION PORT TABLE:
DEST IPADDR     DPORT DESTXCF ADDR    RDY TOTALCONN  WLM TSR FLG
9.12.4.85       03601 10.1.100.48     001 0000000024 01  100 R
9.12.4.85       03601 10.1.100.52     001 0000000025 01  100 R
9.12.4.85       01416 10.1.100.48     001 0000000005 01  100 R
9.12.4.85       01416 10.1.100.52     001 0000000005 01  100 R
```

The display result shows that a DVIPA 9.12.485 has been set up to distribute port numbers 03601 and 01416 between two systems (IP stacks) with internal destination addresses 10.1.100.48 and 10.1.100.52.

A 001 value displayed in the RDY column indicates a listener is active on the specified port and system.

Note that there are additional configuration options available for the Sysplex Distributor, depending on how you want to use it. For our purposes, we are going to use the simple distribution and backup options described here.

2.2.2 An outline of WebSphere MQ queue sharing groups

Our purpose here is to provide a brief outline of the MQ queue sharing group function for those not yet familiar with it. A comprehensive introduction to the concept of queue sharing groups and the facilities used to implement it is provided in Chapter 2, “Shared queues and queue sharing groups” of the IBM publication *WebSphere MQ for z/OS Concepts and Planning Guide V6.0*, GC34-6582.

Shared queues

Queue sharing group, which is the official name of the function, and “shared queues” are commonly used to denote the sysplex support function of WebSphere MQ for z/OS.

A *shared queue* is a local queue that is not owned by a particular queue manager, but by a group of queue managers that are operating within a z/OS sysplex and configured as a *queue sharing group (QSG)*. The QSG is identified by a four character name.

There might be more than one queue sharing group within a sysplex, but a particular queue manager can only be member of a single queue sharing group.

Figure 2-3 shows shared queues in an MQ queue sharing group.

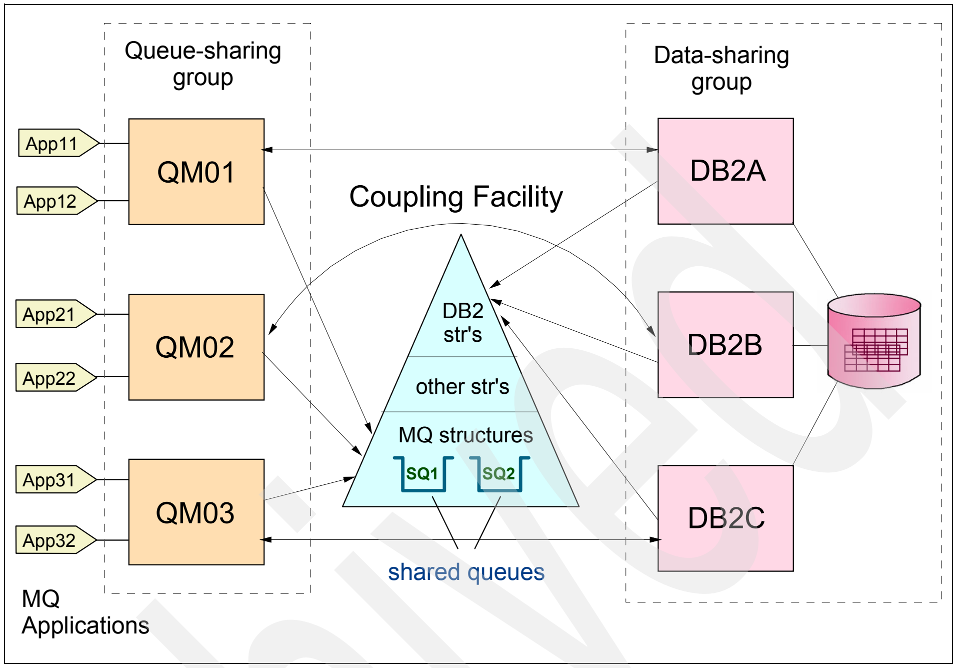


Figure 2-3 Shared queues in an MQ queue sharing group

In order to use the WebSphere MQ queue sharing group function, a DB2 data sharing group environment must exist and the set of Coupling Facility (CF) list structures must be defined for use by MQ. For each DB2 data sharing group, there is one and only one database that supports the WebSphere MQ QSG. This database, with a predefined number of tables spaces and tables, is created as part of the basic setup of the WebSphere MQ queue sharing group function.

The shared queue *messages*, which means the application data, are physically stored in either a Coupling Facility list structure, or for messages over 63 KB. About 1 KB of the message is stored in the CF and the remaining in a DB2 shared database. For shared queues, the CF structure to be used for storing messages has to be specified at queue definition time. Queue managers connect to the particular CF structure when they act on the first application request that references a shared queue. Thus, any queue manager that is active within the queue sharing group might put (write) messages to, or get (read) messages from, a shared queue at any time.

The shared queue *definition* is held in the WebSphere MQ shared repository, which is made up of a set of DB2 tables. The shared queues' attributes are available for both read and update actions to all queue managers within a queue sharing group.

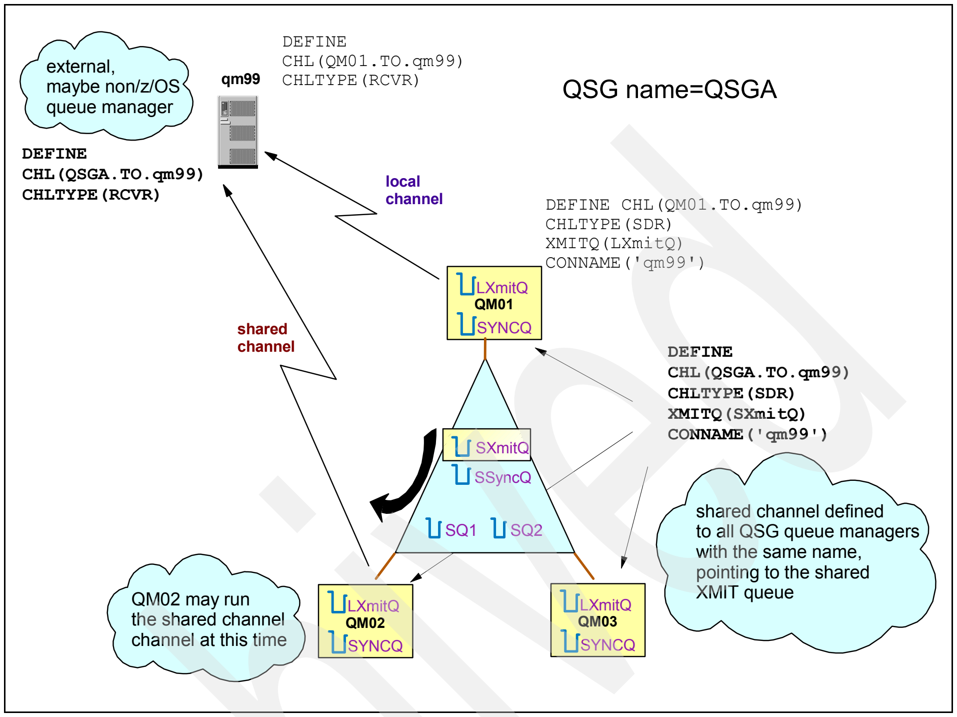
The MQ queue sharing group and the DB2 data sharing group are equivalent functions, whose range is within the scope of the z/OS sysplex. However, a queue manager can only connect to a DB2 within the same z/OS image (LPAR). Similarly, local z/OS MQ applications can only connect to a queue manager within the same z/OS image, although in the case of batch applications, they might use the QSG name instead of the queue manager name within the MQCONN call.

Queue sharing group communication

In order to provide full sysplex support with mutual backup functionality at any time, you need more than set of queue managers with access to shared queues. The queue sharing group concept provides a single system image of the queue managers that make up a queue sharing group to the "outside world". This is achieved by particular enhancements to MQ communication:

- To an external queue manager or an MQ client, the queue sharing group appears as a single queue manager with the name of the queue sharing group.
- *Shared channels* are implemented by creating channels with the same name on each of the queue mangers within the QSG. *Group listeners* are used for inbound QSG communication and are created with the same port number on each of the QSG queue managers. These are in addition to the listeners defined for each of the individual queue managers.
- *QSG outbound communication* is concerned with channels sending messages from within the queue sharing group to a non-QSG queue manager and is based on the following principles:
  - Shared sending channels work on a common shared transmission queue.
  - Only one sending channel is active at any one time.
  - With a triggered sender channel assumed, the trigger message is created within all the QSG queue managers, but only one of them actually takes the channel and begins communication with the external partner.
  - In the case of a channel failure, another instance of this sender channel takes control and restarts the channel based on channel restart information maintained in a shared channel synchronization queue. The same is true when the channel is regularly stopped, possibly by a command or based on a disconnect interval, and restarted again.

Figure 2-4 shows the general setup for QSG outbound communication.


external, maybe non/z/OS queue manager
qm99
QSG name=QSGA
local channel
shared channel
LXmitQ
QM01
SYNCQ
SXmitQ
SSyncQ
SQ1
SQ2
LXmitQ
QM02
SYNCQ
LXmitQ
QM03
SYNCQ
QM02 may run the shared channel channel at this time
shared channel defined to all QSG queue managers with the same name, pointing to the shared XMIT queue

```
DEFINE
CHL(QM01.TO.qm99)
CHLTYPE(RCVR)
```

```
DEFINE
CHL(QSGA.TO.qm99)
CHLTYPE(RCVR)
```

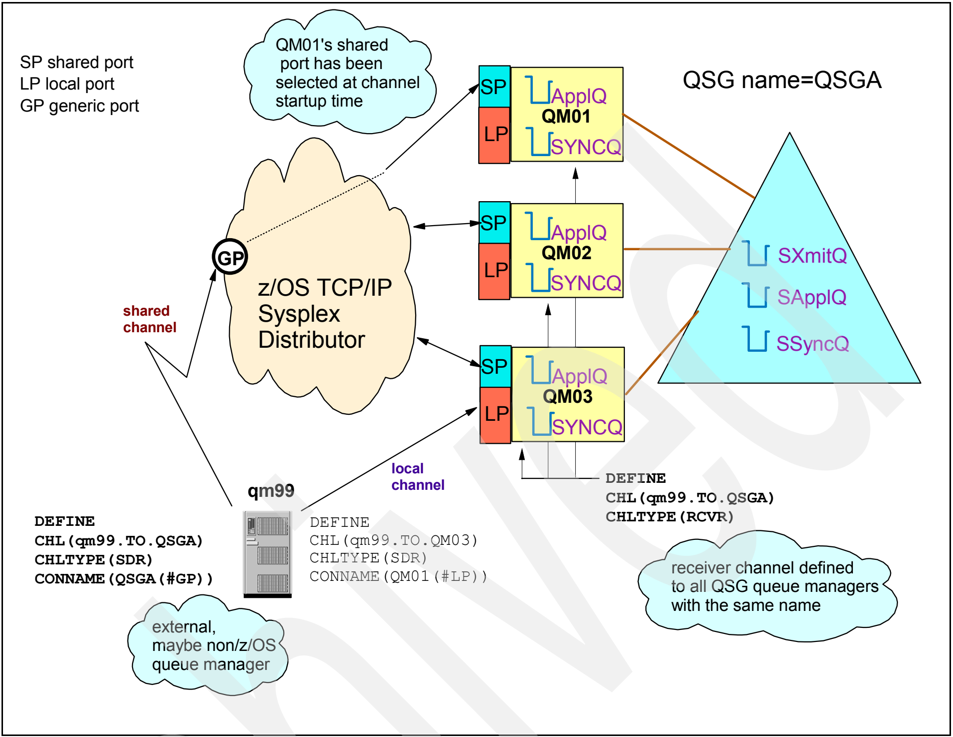
```
DEFINE CHL(QM01.TO.qm99)
CHLTYPE(SDR)
XMITQ(LXmitQ)
CONNAME('qm99')
```

```
DEFINE
CHL(QSGA.TO.qm99)
CHLTYPE(SDR)
XMITQ(SXmitQ)
CONNAME('qm99')
```

Figure 2-4 General setup for QSG outbound communication

- For *QSG inbound communication*, mainly receiver channels, each of the QSG queue managers starts their group listener on a different port of the "regular" queue manager listener. The "regular" listener is used for channels that specifically address the individual queue manager. The group listener ports, on all the QSG queue managers, have to be mapped for dynamic distribution by the TCP/IP sysplex distributor to one separate address and port. This port is called the *group port* or the *generic port*.
  - External partner queue managers define their sender channels to point to this group port. It is the network distribution function that determines which queue manager's group listener a newly started channel is going to connect.
  - This selection always takes place at channel startup time.
  - Only queue managers with an active group listener are eligible for connection.
  - After running, the channel is held by the selected queue manager until it is stopped.

Figure 2-5 shows the general setup for QSG inbound communication.


SP shared port
LP local port
GP generic port
QM01's shared port has been selected at channel startup time
QSG name=QSGA
SP
LP
AppliQ
QM01
SYNCQ
AppliQ
QM02
SYNCQ
AppliQ
QM03
SYNCQ
SXmitQ
SApplQ
SSyncQ
GP
z/OS TCP/IP Sysplex Distributor
shared channel
local channel
qm99
receiver channel defined to all QSG queue managers with the same name
external, maybe non/z/OS queue manager

```
DEFINE
CHL(qm99.TO.QSGA)
CHLTYPE(SDR)
CONNAME(QSGA(#GP))
```

```
DEFINE
CHL(qm99.TO.QM03)
CHLTYPE(SDR)
CONNAME(QM01(#LP))
```

```
DEFINE
CHL(qm99.TO.QSGA)
CHLTYPE(RCVR)
```

Figure 2-5 General setup for QSG inbound communication

- In the case of a failure, the sending side is responsible for reconnecting. When the reconnection occurs, the Sysplex Distributor takes over and selects one of the currently active group listeners. Similar to sending channels, channel restart information comes from the shared channel synchronization queue.

An equivalent function of the Sysplex Distributor mapping of inbound IP requests, called IBM VTAM® Generic Resources, exists for VTAM.

2.2.3 General system layout

The descriptions that follow are applicable to all types of application scenarios. We refer to the overall system configuration that was previously described in 2.1, “Introducing the base installation” on page 24. Figure 2-6 is an extension of Figure 2-1 on page 24. In addition, we now show which components make use of Sysplex Distributor based port distribution. We use CICS as a typical Enterprise Information System (EIS) against which applications that use the service of the ESB execute.

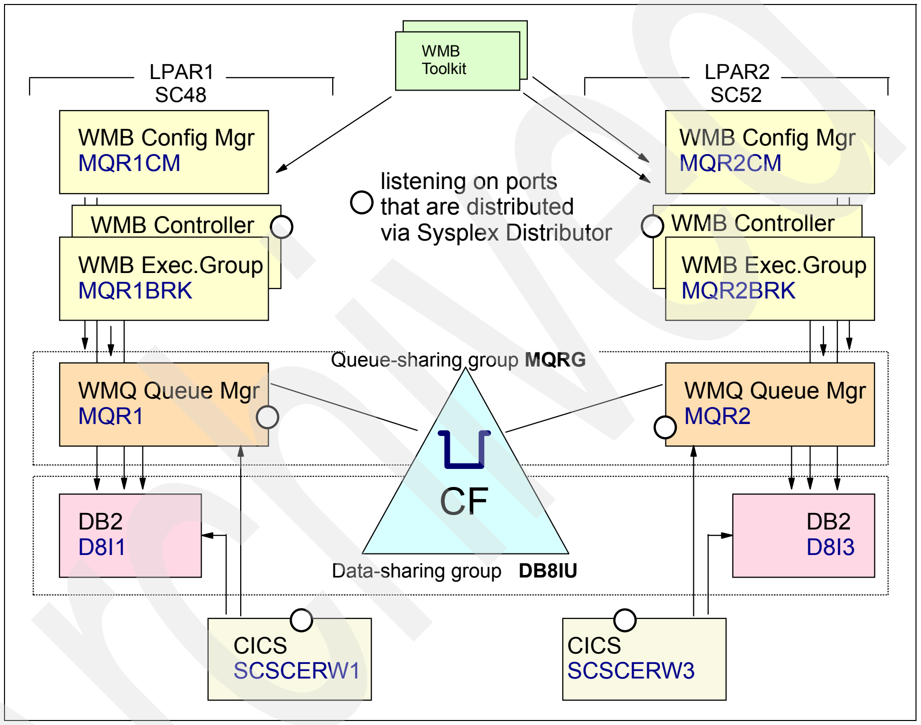


Figure 2-6 General system configuration for high availability

We are going to describe Figure 2-6 from the top down, describing the basic guidelines we followed and the use and behavior of the configuration concepts in case of failures. What is described for our two LPAR configuration might be generalized for a multi-LPAR configuration.

2.2.4 LPAR-based considerations

Sysplex high availability is based on the principle of identical components that are capable of backing up each other. There is an identical system layout on both our LPARs, which enables each of the two LPARs to be a full backup for the other.

As long as the applications permit parallel processing on both LPARs, a similar workload might be executed on both LPARs at any one time.

What happens in case of a failure

If one LPAR fails or is brought down on schedule, the remaining LPAR has to take the full workload. Capacity has to be provided to allow for this action:

- The Sysplex Distributor distributed DVIPA function provides continuous network connectivity to the components on the active LPAR, without the requirement for any changes on behalf of the external process that initiates the connection. Details are provided within the discussion of the individual components of the WebSphere Message Broker.
- WebSphere MQ shared queues and DB2 shared databases ensure continuing access to application data as long as at least one LPAR is up, as discussed in the MQ section that follows.

2.2.5 WebSphere Message Broker components considerations

The following are some considerations applied to the WebSphere Message Broker configuration.

WebSphere MQ queues for Message Broker not shared

The WebSphere Message Broker is made up of the combination of WebSphere Message Broker and WebSphere MQ. WebSphere Message Broker components require access to an associated queue manager that runs in the same z/OS image (LPAR).

The MQ broker queues, named SYSTEM.BROKER.*, are always going to be non-shared local queues to the associated MQ queue manager. Note that the SYSTEM.BROKER.* queues are system queues used by the broker itself. Application queues used in flows might be shared.

One WebSphere Message Broker Configuration Manager or multiple ones

We created WebSphere Message Broker Configuration Managers on both LPARs, so each LPAR forms a separate Broker domain. Thus, in terms of the availability of a Broker Configuration Manager, no special action has to be taken to handle the LPAR failure or shutdown situation. Remember also that the Configuration Manager is not typically considered a resource that must be 100% available. The Configuration Manager is only required for Broker administration, such as deploying new flows. Application flows continue to run regardless of whether the Configuration Manager is available.

We ensure all message flows are deployed to the appropriate broker execution groups identically on both the LPARs. Manual deployment of the message flows poses the potential for error. For a production system, particularly one made up of multiple brokers, automation of the deployment process must be considered.

A single instance of a WebSphere Message Broker Configuration Manager would be sufficient to run the ESB sysplex configuration, with all the brokers in the sysplex included in a single broker domain.

As message flows are deployed to execution groups, the challenge of maintaining equivalent execution groups in both (or multiple) LPARs remains a consideration, although this is really no different than maintaining identical execution groups across two different Configuration Managers (two WebSphere Message Broker domains).

Additional action has to be taken to safeguard the WebSphere Message Broker Configuration Manager’s availability and connectivity in case of a failure of the LPAR that usually runs this WebSphere Message Broker component:

- The WebSphere Message Broker Configuration Manager has to be started in an automated manner on (one of) the other LPARs. This can be achieved by an automation tool, such as IBM Tivoli Workload Scheduler, formerly known as Operations Planning and Control (OPC), or by means of the Automatic Restart Manager (ARM) z/OS sysplex function.
- If the WebSphere Message Broker Configuration Manager is restarted on an LPAR different than the regular one, it must use the same IP address so that the move is transparent to WebSphere Message Broker toolkit instances that connect to this Configuration Manager.

  To ensure this happens, a distributed DVIPA is defined to the sysplex distributor with only one target system being active at a time, as demonstrated in Figure 2-2 on page 31. The TCP/IP profile settings look exactly as shown in Figure 2-2 on page 31.

Distributed DVIPA definitions for WebSphere Message Broker

The Broker's listener port is used to serve inbound Hypertext Transfer Protocol (HTTP) requests that are directed to message flows that use an HTTP input port. By default, the port number used is 7080. If there is not a strong reason to use a different port, then you must use this one.

- To provide high availability, you have to configure this port as a distributed VIPA, as shown in Figure 2-2 on page 31.
  - The broker controllers on both LPARs have to use the same port number for their IP listener.
  - Sysplex Distributor maps network endpoints at which the broker instances are listening to a distributed VIPA. This is the IP address to be used by processes that connect to the broker through the network, for example, processes that connect to an HTTP input node.
  - The part of the TCP/IP profile configuration that serves this purpose is shown in Example 2-2.

  Example 2-2 TCP/IP DVIPA definition portion for WebSphere Message Broker

  ```
  VIPADYNAMIC
  ...
  VIPADISTRIBUTE DEFINE DISTMETHOD ROUNDROBIN 9.12.4.86
     PORT 7080
     DESTIP 10.1.100.48 10.1.100.52
  ...
  ENDVIPADYNAMIC
  ```

  TCP/IP connections to a broker flow are always directed to (address:port) 9.12.4.86:7080, and Sysplex Distributor distributes these requests between the broker instances on either of the two LPARs.

- During normal operation, when both brokers are up on both LPARs, inbound connections to a broker flow are equally distributed between the two LPARs based on the round robin default rule specified within Sysplex Distributor.

HTTP connections in case of failures

If one of the brokers fails, or is stopped, Sysplex Distributor automatically directs all subsequent requests to the surviving broker. Thus, full availability of the appropriate functions is assured, even for connections initiated after the failure has occurred.

In the case of an LPAR failure, the broker and its message flows are going to be terminated. Due to the session-less nature of HTTP connections, the external HTTP client component only becomes aware of the failure when a timeout mechanism indicates that the connection is broken.

The effects on data that might be controlled by MQ, DB2, or another z/OS based resource manager invoked by the broker message flow, is discussed in 2.2.6, “Distributed DVIPA definitions for WebSphere MQ” on page 42.

Based on the control provided by z/OS Resource Recovery Services (RRS), subsequent processing is ensured for committed data that is in a consistent state.

2.2.6 Distributed DVIPA definitions for WebSphere MQ

The high availability concept of MQ Queue Sharing Groups has been introduced in “Queue sharing group communication” on page 35.

MQ group listeners

For each queue manager, you have to start a second listener, known as the *group listener*. This is the network endpoint distributed by the sysplex distributor.

- If queue managers are run in pure QSG mode, without the requirement to ever run a channel that is directed to a specific queue manager, then the “regular listener”, called the *local listener*, does not have to be defined.
- In practice, it is very likely to remain in place and serve those direct connections to the queue managers, for example, non-shared channels.
- Note that the MQ cluster function always uses non-shared channels and requires the local listener to be started in each queue manager.

The command to start the group listener is:

```
START LISTENER INDISP(GROUP) PORT(number)
```

Note: We recommend that you add this command to the MQ channel initiator’s initialization command input dataset, which is determined by the CSQINPX DD statement of the channel initiator startup procedure.

To reiterate, in order to use the Sysplex Distributor distribution function, you have to choose the same port number for each of the two queue managers that run on both the LPARs. Thus, both queue managers can refer to the same CSQINPX data set. Note the following conditions:

- The port number has to be defined and it has to be different from the port number used by the local listener.
- If the previous step is omitted, the default 1414 is going to be used. This might cause a conflict with the local listener port, often defined as the default, 1414.
- In our setup, we used the default port 1414 for the local listeners and introduced port number 14140 for the MQ group listener ports.

The TCP/IP profile section describing our MQ group listeners is shown in Example 2-3.

Example 2-3 TCP/IP DVIPA definition portion for WebSphere MQ group listeners

```
VIPADYNAMIC
...
VIPADISTRIBUTE DEFINE DISTMETHOD ROUNDROBIN 9.12.4.85
   PORT 14140
   DESTIP 10.1.100.48 10.1.100.52
...
ENDVIPADYNAMIC
```

Channels that are to benefit from the queue sharing group's high availability capabilities by addressing the QSG (and not a particular queue manager) as their target have to use (IP address:port) 9.12.4.85:14140 as their endpoint.

They are going to be run by the z/OS queue managers as shared inbound channels.

2.2.7 MQ shared inbound channels

Within the ESB functionality, shared inbound channels are used to receive messages from WebSphere MQ connected applications that run on remote systems, either z/OS or other platforms.

- Connecting to MQ on the sysplex through the global port indicates a connection to the queue sharing group and not to a particular queue manager.
- Note that an external queue manager that connects to the global port is not "aware" of the QSG's shared channels concept and its special behavior. To the external MQ system, it is just a channel with a certain name and a particular IP address and port number.

  Consistent with the Sysplex Distributor definitions previously shown, a queue manager on IBM AIX® could define such a channel, as shown in Example 2-4.

Example 2-4 External sender channel definition addressing the QSG global port

```
DEFINE CHANNEL(QMAIX.TO.MQRG)
   CHLTYPE(SDR) XMITQ(MQRG)
   TRPTYPE(TCP) CONNAME(9.12.4.85(14140))
```

Note: If the channel is to be named according to the common "from-qmgr1.TO.to-qmgr2" convention, then the QSG name is to be used as the to-qmgr2 name. To the external queue manager, the QSG appears as a single queue manager and it is the QSG, not a particular queue manager, that the connection targets.

- Within the MQ queue sharing group, these channels are typically created as *group objects*, which is a special option of maintaining object definitions and attributes within a queue-sharing group.
  - Group object definitions are maintained in a section of the QSG DB2 database, which is called the shared repository.
  - Thus, these definitions are available to each of the QSG queue managers at the same time in the same way.
  - You can create group objects by adding the keyword 'QSGDISP(GROUP)' to the appropriate MQSC DEFINE command, which is only valid in a QSG environment.
  - Apart from the group object, which is just the definition entry in a DB2 table of the QSG shared repository, the DEFINE command is processed by each of the QSG queue managers to create a local object with the specified name and attributes. Thus, an identically named object is created within all queue managers of the QSG with a single DEFINE command.
  - The command to create the receiver channels matching the QMAIX sender channel definition from Example 2-4 on page 43 is shown in Example 2-5.

Example 2-5 Shared receiver channel definition command

```
DEFINE CHANNEL(QMAIX.TO.MQRG)
   CHLTYPE(RCVR) QSGDISP(GROUP)
   TRPTYPE(TCP)
```

Note: The shared nature of the channel is not imposed by its being defined with the QSGDISP(GROUP) option. Instead, it is based on an inbound connection request to the group listener, as opposed to connections to the local listener, which might still be active.

In this case, the dynamic channel status is maintained in DB2 and the recovery information is written to the shared channel synchronization queue (`SYSTEM.QSG.CHANNEL.SYNCQ`). Based on this status, channel recovery and restart information is available to all queue managers in the QSG and any one of them might run the channel, but only one at a time.

- When the channel is started, which is initiated by the remote partner, either of the two queue managers might serve it. Only one of them actually will, and this one QSG queue manager starts communication, receives messages, and puts them to their destination queues as long as the channel remains active.

Shared inbound channel failures

If the channel, the hosting channel initiator, or the relevant queue manager fails, the high availability approach is based on the sending queue manager's behavior upon noticing the failure within a short period of time and subsequently attempting to restart the channel. Communication might be interrupted, but only for a short time, usually a matter of seconds.

- Messages that were part of the current unit of work at the time of the channel failure are backed out. This means that they are (logically) written back to the transmission queue on the sending side and removed from the destination queues at the receiving side of the channel.
  - The external (sending) queue manager performs the backout when it realizes the channel has failed.
  - If a queue manager or channel initiator failure has caused the channel to crash, backout processing for the failed inbound channel is done by a surviving queue manager within the queue-sharing group.

  For the duration of the backout process, no new messages can be transmitted. The time taken to complete this backout activity depends on the amount of data to be processed, for example, the number and length of the messages. These, in turn, depend on application design and the maximal number of messages that can be transmitted between channel synchronization points. This number can be specified as a channel attribute with the `BATCHSZ` (batch size) definition parameter.
- After the sending side has finished its backout processing, it attempts to restart the channel using the same channel name, network address, and port as before.

- The connection request is handled by z/OS Sysplex Distributor. It selects one of the QSG managers, based on the availability of their group listener.
- Obviously, if a queue manager failure or even an LPAR failure caused the channel to terminate, it is going to be the other surviving queue manager that starts serving the channel at this time.
- Channel restart is performed in accordance with MQ channel startup rules and the transmission of messages is resumed.
- The application design determines whether at all, and in what way, applications might be affected by a short-termed channel failure.

  Most applications are not going to be affected at all, although some might experience noticeable delays, such as extended response times. Some applications might terminate because of the way they are designed to react when messages they are waiting for do not show up within a certain period of time.

2.2.8 MQ shared outbound channels

To send messages to MQ connected applications that run on remote systems, irrespective of the platform, shared outbound channels are used by the WebSphere Message Broker queue managers to provide high availability. These typically are *sender* channels that are started automatically based on MQ triggering from the transmission queue.

Unlike shared inbound channels, shared outbound channels do not use special communication functions, but are based solely on MQ internal processing.

- A shared sender channel is one that gets messages for transmission from a shared transmission queue.

> **Note:** The shared nature of the channel is *not* imposed by some definition attribute.
>
> Remember, “shared” means that the channel status is maintained in DB2 and the recovery information is written to the shared channel synchronization queue (SYSTEM.QSG.CHANNEL.SYNCQ). Based on this information, channel recovery and restart information is available to all queue managers in the QSG and any of them might run the channel, but only one at a time.

- The best way to define a shared sender channel, is to use a group definition that ensures that the channel is defined to all QSG queue managers with the same name and identical attributes.

Example 2-6 shows the DEFINE commands for both the shared transmission queue and the associated sender channel, with an imaginary IP address shown for the target queue manager on AIX.

Example 2-6 Shared transmission queue and shared sender channel definitions

```
DEFINE QL(QMAIX) QSGDISP(SHARED)
   USAGE(XMITQ) CFSTRUCT(APPL1)
   TRIGGER TRIGTYPE(FIRST) TRIGDATA(MQRG.TO.QMAIX)
   INITQ(SYSTEM.CHANNEL.INITQ)

DEFINE CHANNEL(MQRG.TO.QMAIX)
   CHLTYPE(SDR) QSGDISP(GROUP)
   TRPTYPE(TCP) CONNAME(9.12.12.167(1419))
```

According to the naming convention, the queue-sharing group name (MQRG) replaces the from-qmgr1 queue manager.

- Regardless of which one of the QSG queue managers places a message on the shared transmission queue, the channel (named by the transmission queue's TRIGDATA attribute) is triggered on all QSG queue managers, MQR1 and MQR2 in our case.
  - Both queue managers's internal channel initiator function attempt to start the named channel.
  - The one that starts first takes it, while the other "ends gracefully".
  - The "channel started" message written to the system log identifies the queue manager that took the channel, which is MQR2 in our sample. It is illustrated as follows:

    ```
    +CSQX500I -MQR2 CSQXRCTL Channel MQRG.TO.QMAIX started
    ```
- The channel is operated by this single queue manager as long as the connection is up. Only when the channel is restarted does the race for taking the channel takes place again. The outcome can be the same queue manager as before, or a different one.

- If at any time during normal operation, you want to determine which of the queue managers is running the channel, the easiest way to do so is to use the DISPLAY CHANNEL function of MQ ISPF, as shown in Figure 2-7.

```
invocation from MQ ISPF main panel:

Action  . . . . . . . . . . 1     0. List with filter   4. Manage
                                  1. List or Display    5. Perform
                                  2. Define like        6. Start
                                  3. Alter              7. Stop
Object type . . . . . . . . CHANNEL      +
Name  . . . . . . . . . . . MQRG*
Disposition . . . . . . . . A  Q=Qmgr, C=Copy, P=Private, G=Group,
                               S=Shared, A=All Action  . . . . . . .

result:

Name                  Type          Disposition   Status
 <>  MQRG*                 CHANNEL       ALL     MQRG
     MQRG.TO.QMAIX         SENDER        COPY    MQR1  INACTIVE
     MQRG.TO.QMAIX         SENDER        COPY    MQR2  RUN
     MQRG.TO.QMAIX         SENDER        GROUP
```

Figure 2-7 Determining within MQ ISPF which queue manager runs a shared channel

Note: Ensure the *Disposition* field specifies “A” (All) in the query! The result shows both the channel copy objects and the group object. The one copy object with a status indication of “RUN” is the active one. The group object cannot have a status.

- Use native MQSC commands to determine which of the QSG queue managers is running the channel, as shown in Figure 2-8.

```
native MQSC command. to be issued against any of the QSG queue
managers:

DIS CHS(MQRG*) CMDSCOPE(*)

Sample response (extract):

CSQN121I -MQR2 'DIS CHS' command responses from MQR2
CSQM420I -MQR2 CHSTATUS(MQRG.TO.QMAIX) CHLDISP(FIXSHARED )
XMITQ(QMAIX) CONNAME(9.12.12.167) CURRENT CHLTYPE(SDR)
STATUS(RUNNING) SUBSTATE(MQGET ) STOPREQ(NO ) RQMNAME(QMAIX) )
```

Figure 2-8 Determining the channel owning queue manager by MQSC command

- The CMDSCOPE(*) command option sends the command to all active queue managers within the QSG.
- Only the queue manager running the channel returns a status, because only active channels have a channel status.

Shared outbound channel failures

If the channel, the hosting channel initiator, or the relevant queue manager fails, then the surviving queue manager actively initiates a takeover of the channel. This usually results in a message being written to the console, as shown in Example 2-7.

Example 2-7 Console messages accompanying shared channel recovery and takeover

```
$HASP395 MQR2CHIN ENDED
CSQM052I -MQR2 CSQMPCRT Shared channel recovery
completed for MQR2, 1 channels found, 1 FIXSHARED, 1 recovered )
...
+CSQX500I -MQR1 CSQXRCTL Channel MQRG.TO.QMAIX started
```

Whether or not the takeover of the channel is accompanied by this or similar console messages depends on the accurate state of the channel at the time of failure. The message indicating that the surviving queue manager has taken the channel (`CSQX500I`) is always issued.

If a shared outbound channel is busy transmitting messages at the time of the failure, the scenario is similar to the one previously described for inbound channels.

Note the following:

- Channel failure causes the current unit of work to be backed out, which means that messages are logically written back to the shared transmission queue. This is the process of shared channel recovery, which in the case of a queue manager or LPAR failure, is performed by the surviving queue manager.
- The external target queue manager becomes aware of the failure and also does a backout. This means that messages that have been received as part of the current batch are not committed, but deleted from the queues where they have been written to.
  - As discussed before with the inbound channel, this process takes a period of time during which no transmission of messages takes place. Again the application design determines whether at all, and in what way, applications might be affected by a short-termed channel failure.
  - The remote receiver channel then enters the INACTIVE state, making it ready to be reinvoked at any time.
- Within the z/OS QSG environment, the surviving queue manager automatically starts its shared channel because of the existence of messages on the shared transmission queue. The connection is again activated and transmission of messages is resumed.

2.2.9 MQ shared queues

To ensure high availability for WebSphere Message Broker data, shared queues have to be used for all MQ queues that are referenced in the WebSphere Message Broker message flows as follows:

- MQ input nodes
- MQ output queues
- MQGET nodes for transferring data between flows.

The use of shared queues ensures that application data can both be read from (GET) and written to (PUT) to the queues used by broker flows, independent of the LPAR on which a flow is executed.

Shared queues definition considerations

Based on the implementation of the MQ shared queues function, there are important characteristics that have to be kept in mind when you are starting to use shared queues.

For example:

- Is there an application affinity where all (or a segment) of the messages sent by an application must all be handled by a single Broker? This is an issue that has to be addressed at application design time.
- Are all the messages put to a particular queue non-persistent, or is there a requirement for persistent messages?
- Is the size of the messages put to a particular queue above or below 63 KB (64524 bytes), including all headers?

It is the design and requirements of the applications making use of the WebSphere Message Broker that provides the answers to these questions and thus imposes the requirements on how the shared queues are to be configured.

These special requirements, based on the persistence and the size of shared queue messages, have to be taken into consideration when defining *Coupling Facility* structure objects within WebSphere MQ.

Planning for and defining Coupling Facility structures

This paragraph aims to provide an outline of what has to be considered when defining CF structures for use by MQ shared queues, as demonstrated by Figure 2-9 on page 52:

- MQ uses CF list structures to store shared queue messages.
- When a shared queue is created, a CF structure has to be specified as a parameter of the QLOCAL definition object.
- CF structures are named as ranges of storage within a Coupling Facility with a limited size, as specified when created using the IXCMIAPU z/OS administrative data utility.
  - After they are created, CF structures are neutral in terms of how and by whom they are going to be used. Access can be controlled by IBM RACF® profiles.
  - In practice, a CF structure is created with a particular usage in mind, which is usually reflected by the name given to it.
  - MQ imposes the rule that CF structure names have to start with the four character name of the QSG that uses them.
  - A structure named <qsg-name>CSQ_ADMIN is mandatory and used for QSG internal administration functions.
  - Structures used for storing messages are called *application structures*. There might be up to 63 application structures per QSG. They can be arbitrarily named, but names have to start with the name of the QSG.

   - This naming rule enforces the fact that CF structures can never be shared among multiple QSGs that might be defined in the same sysplex.

Figure 2-9 shows the definition of CF structures to z/OS and MQ.

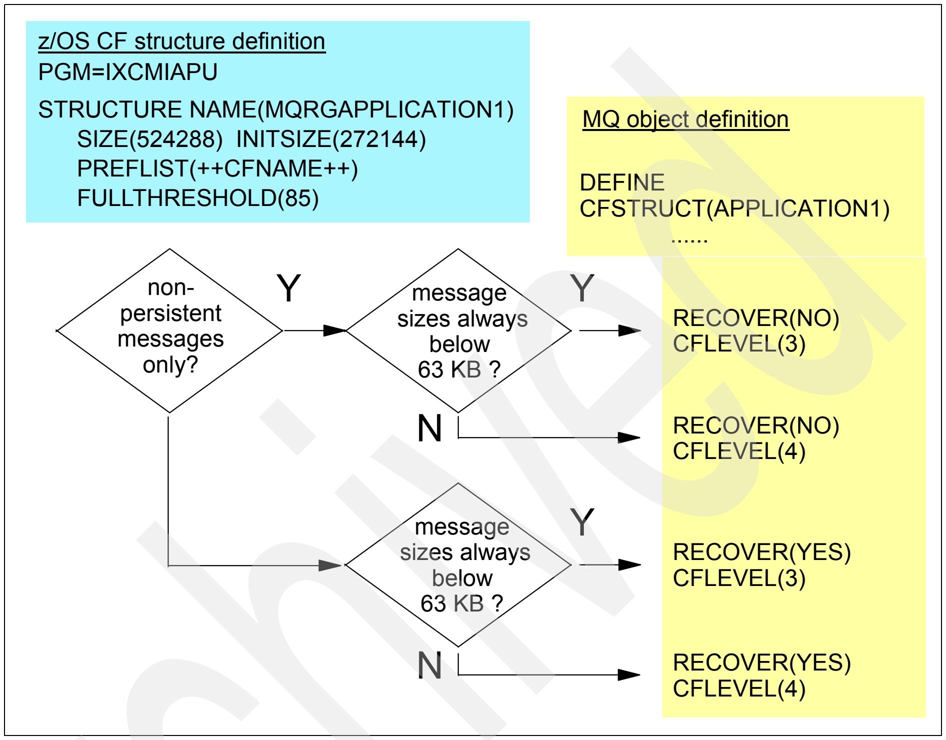


Figure 2-9 Defining CF structures to z/OS and MQ

- A CF structure has to be defined to MQ as “being recoverable or non-recoverable”, because MQ performs all recovery-related activity. Therefore, the structure itself “is not aware” of being recoverable or not.
   - You can use only recoverable structures to store persistent messages, and recoverable structures must be defined to MQ by means of the following command:

     ```
     DEFINE CFSTRUCT(<name>) RECOVER(YES)
     ```

   - Persistent messages can only be written to structures that have been defined with RECOVER(YES).
   - All MQ references to a CF structure, including the name of the CFSTRUCT objects, omit the first four characters, which is the QSG name.

    Thus, a CF structure created with the real name MQRGAPPLICATION1 has to be defined to MQ by a DEFINE CFSTRUCT(APPLICATION1) command, as shown in Figure 2-9 on page 52.

- The second important option to be aware of is the size of messages put to a shared queue.
  - Shared queue messages that do not exceed 63 KB in length, including all headers, are fully stored in the appropriate coupling facility structure.
  - For messages that exceed this length limit, the data part of the message, also referred to as the message payload, is stored in a DB2 table as a binary large object (BLOB). Only the message descriptor (MQMD) and some other control information is stored in the CF structure.
    - The DB2 environment is created as part of the initial setup of the queue-sharing group function. Large message (greater than 63 KB) support for shared queues has been introduced by WebSphere MQ for z/OS Version 6. Before that, these messages could not use the shared queues function.
    - To enable a CF structure to host large messages, it has to be defined to MQ with the CFLEVEL(4) attribute, as shown in Figure 2-9 on page 52. An attempt to write a message greater than 63 KB in size to a queue that uses a CFLEVEL(3) defined structure fails at MQPUT time.

Defining shared queues

As previously mentioned, continuous high availability support for queues using WebSphere Message Broker message flows requires that all the MQ queues used by the flows are created as shared queues.

At queue definition time, you have to determine the CF structure, which is to serve that shared queue, as shown in Example 2-8.

Example 2-8 MQSC command to define a shared queue

```
DEFINE QLOCAL(TRADER.SAVEDATA)
   QSGDISP(SHARED) CFSTRUCT(APPLICATION1)
   MAXMSGL(32000) SHARE
```

Some facts to be aware of:

- QSGDISP(SHARED) is the attribute that makes the new queue a shared queue.
- The CFSTRUCT attribute is required to identify the CF structure that is used to store this queue’s messages.
  - A maximum of 512 shared queues might point to a single CF structure.

    - A particular queue’s messages cannot spread over multiple CF structures.
- At queue definition time, the existence and availability of the named CF structure is not checked. Therefore, the definition might succeed, but the queue might not be usable.
- A common mistake for beginners is to code the CFSTRUCT attribute with the full name of the CF structure, including the leading four characters representing the name of the QSG. As the existence and availability of the real CF structure referenced is not checked at queue definition time, the command might succeed with a new shared queue object being created. But any attempts to open the queue fails, because MQ tries to connect to a non-existent CF structure. This is true for any incorrect CFSTRUCT specification.
- The CFSTRUCT attribute of a shared queue definition object cannot be altered. If it is incorrect or the queue is to be associated with a different CF structure, the object has to be deleted and redefined.
- The maximum message length (MAXMSGL) does not have to be specified with the queue definition, but it is best practice to specify one that is below or equal to 63 KB (64,512) for shared queues using structures with CFLEVEL(3), which are structures that do not support large messages. If an attempt is then made to write a larger message to that queue, MQ rejects it at the queue level.
- The SHARE parameter, as opposed to the QSGDISP(SHARED) parameter shown, has nothing to do with the queue being a shared queue, but it allows multiple processes to open the queue for reading messages (MQOPEN for input) at the same time.

  These processes, which in our case are message flows running in WebSphere Message Broker execution groups, might run in the same LPAR or in different ones. The NOSHARE option has to be coded instead, if the application (WebSphere Message Broker message flow) does not allow for parallel processing. The queue would be a shared queue, with read (MQGET) access only allowed from one message flow at a time. This might be of interest for backup purposes: one broker could pick up the responsibility for an input queue if the other one is not available.
- The definition command might be entered at any of the QSG queue managers. As soon as the shared queue has been successfully defined, it is equally available to all QSG queue managers.

Conclusions and best practices

The following are a list of conclusions and best practices:

- Check the requirements of your MQ applications and the WebSphere Message Broker message flows to see if they require persistent messages and if the messages can be greater than 63 KB in size.
- Because structure might have to be managed differently, you might want to separate queues for persistent and non-persistent messages. This decision is taken during application design.
- Review the requirement for large messages. Check if there is a way to split large messages into smaller ones. Large messages, which have their data portion stored in DB2, experience considerable GET and PUT performance degradation compared to smaller messages, which are completely held in the Coupling Facility.
- Start with just one CF structure defined for each type required, which include one for non-recoverable/small messages and one for recoverable/small messages and so on.
- Be sure that your CF structures never fill up to their physical limit:
  - A full CF structure is a severe situation that is likely to affect a large number of applications. They would no longer be able to write messages to any of the shared queues hosted by that structure.
  - Limit the amount of data that can be stored in a shared queue at any one time by using the MAXDEPTH queue attribute.

    This limits the number of messages that can be written to a particular queue. Setting this parameter to a reasonable value reliably avoids the situation where a CF structure might fill up as a result of problems. For example, a particular application that erroneously writes an excessive number of messages or experiences a delay with processing messages.
  - If the queue MAXDEPTH limit is reached, then only the application that has stored more messages on that queue than it is supposed to is affected. The CF structure provides enough space for other applications to continue to work as usual.
  - A production environment must certainly be using software to monitor queue usage. The IBM Tivoli OMEGAMON® XE for Messaging product is an example of an excellent product offering this capability.

High availability provided by shared queues

The feature provided by shared queues is based on the fact that they remain available for both write (PUT) and read (GET) accesses as long as at least one QSG queue manager is up. In addition to providing high availability, shared queues also provide a form of automatic load-balancing among the participating brokers and their underlying queue managers. The brokers naturally share the load.

In our simple configuration, if two LPARs with an WebSphere Message Broker configured on both and with shared queues is used by WebSphere Message Broker message flows, then as long as at least one of the QSG queue managers is available, the following tasks are possible:

- WebSphere Message Broker flow input queues remain available for putting messages, both from a local z/OS based application or from a remote MQ application connected to the QSG through a shared channel.
- Flow input queues remain available to the appropriate message flows, independent of the LPAR in which they execute. Therefore, if a flow is stuck for whatever reason on one LPAR, it remains startable and executable on the other.
- Queues used for intermediate storage of data, for transfer to a different message flow, remain available for both the originating and the target message flows, independent of which LPAR the flows are executed on.
- Flow output queues remain available to z/OS based applications that access them, independent of the LPAR on which both the flow executed and the target applications run.

Failures

Shared queues are fully controlled by MQ, with a dependence on DB2 data-sharing group availability to MQ.

- If an LPAR goes down, shared queues are fully available to the components on the active LPAR. New messages can be put to them and “old messages”, which are the ones that had been written to the shared queues before the time of the failure, can, without restrictions, be read and processed by active WebSphere Message Broker flows and applications using the ESB on the surviving LPAR.

  As has already been pointed out by the preceding sections, network based requests that might invoke an HTTP based message flow are directed to the other LPAR. The same is true for inbound MQ channels from external queue managers. To external applications, the surviving LPAR provides the full ESB support.

- If a *broker component* fails, this is “not perceived” by the MQ shared queue function. Shared queues are available to any WebSphere Message Broker message flow that uses them to write messages to and read messages from on whatever LPAR the appropriate WebSphere Message Broker execution group runs.
- If a *queue manager* fails, no MQ service is available within that LPAR. This includes the access to shared queues. The WebSphere Message Broker Agent Controller brings down the WebSphere Message Broker execution groups which stops its HTTP listeners.

Note: As the MQ service is crucial to WebSphere Message Broker execution, this scenario is very close to the failure of the complete LPAR. It only makes a difference to applications that execute on the same LPAR where the assumed WebSphere MQ/WebSphere Message Broker failure takes place. This could be batch programs, CICS transactions that use the MQ Bridge, or IMS transactions using the MQ adapter (IMS MQ Bridge applications are not affected in this way).

To these applications, there is no MQ service available, therefore they do not have access to the shared queue messages.

If application failures are not acceptable, effort has to be taken by means of a system automation facility, to prevent z/OS-based applications that depend on the local MQ support, which means the applications that locally connect to the queue manager, from starting during the time MQ is not available in an LPAR.

- If an instance of *DB2* fails, both the queue manager and broker that were connected to that DB2 continue to run, but with restricted functionality and an overall undefined level of service:
  - Active MQ shared channels continue working, but are unable to stop. Channels that were inactive are unable to start. This is because MQ maintains the channel status within DB2 tables and access is required at channel startup and stop. Thus, MQ communication might be down within a short period of time.
  - Shared queues that are served by CFLEVEL(3) CF structures, those that are restricted to message length less than 63 KB, continue to be available for both read and write operations.
  - Accesses to shared queues that are served by CFLEVEL(4) CF structures fail, if trying to access a large message, as do direct calls to DB2 via Database, Java, and Compute nodes within WebSphere Message Broker message flows. Small messages on the CFLEVEL(4) structure are available.

Note: The decision on how to handle a failure of DB2 has to be made based on the overall use of the system and the nature of applications.

If MQ is exclusively used by the ESB environment, a good choice might be to automatically bring down MQ if DB2 fails. This brings the WebSphere Message Broker execution groups down as well and prevents additional message flows from starting on this LPAR.

- The shared queues' high availability contribution to *failing WebSphere Message Broker message flows* is:
  - If a flow is forced to terminate because a system component fails (possibly the entire LPAR), all changes applied to shared queues are backed out. The same is true for changes to DB2 data that have been performed within the flow.
  - As part of the backout process, the initial message that provided the input to an MQ input node based message flow is backed out to the shared queue. From here, it is again available to the message flow either on the same or another LPAR. Furthermore:
    - In the case of a pure application failure, this happens immediately and the flow might restart right away.
    - In case of an LPAR failure, a backout cannot be done before the LPAR is up again, because the backout process is controlled by the z/OS Resource Recovery Service, which is LPAR based. The shared flow input queue remains active, however, and messages that have already been put there before the LPAR failure, or after, are able to be processed by flows on the surviving LPAR.
  - Message flows that have been initiated through an HTTP input node are also backed out and any changes that were made to MQ and DB2 are undone. As HTTP is not a transactional protocol, the initial HTTP input message is lost and the application that sent it is returned in a “connection broken” condition instead of a reply. It has to handle the restart of the particular flow.

2.2.10 Sample application scenarios for high availability

This section discusses the sample application scenarios for high availability provided by shared queues and distributed DVIPAs. The following two simple scenarios demonstrate how flexible WebSphere Message Broker components that make use of the MQ shared queues are, and how the sysplex distributor distributed DVIPA functions interact in a z/OS sysplex environment.

Scenario 1: MQ in - MQ out

As shown in Figure 2-10, assume a flow with both an MQ input and MQ output node. Activity is described from top to bottom.

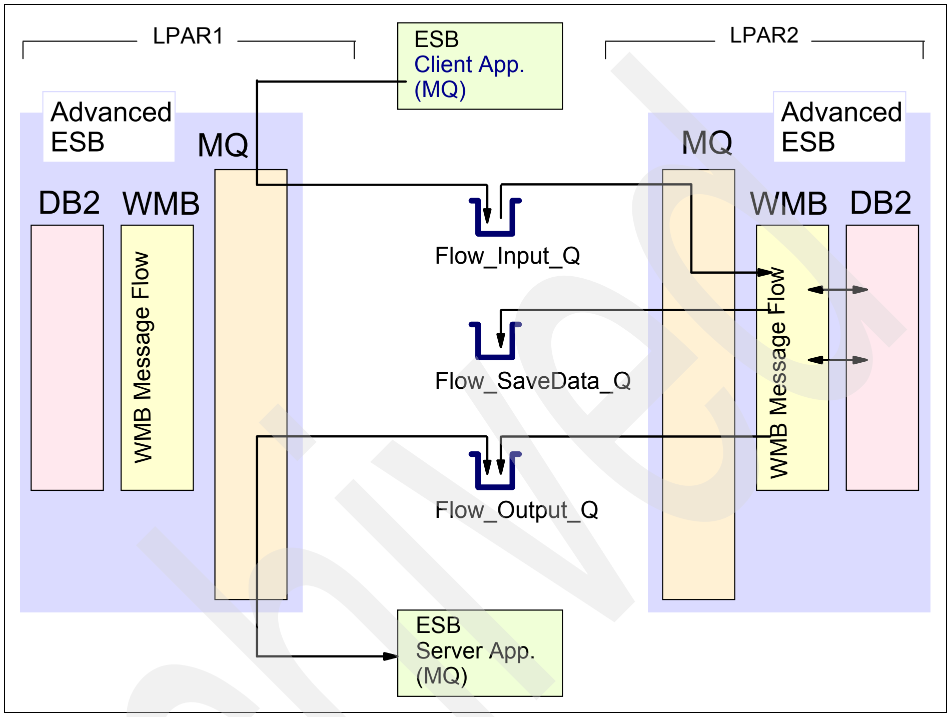


Figure 2-10 General MQ-to-MQ flow scenario using shared queues

The following steps discuss the process illustrated in Figure 2-10:

1. The application that invokes the ESB might be a z/OS based one, for example, a CICS transaction. In this case, it uses the MQ service of the queue manager in the same LPAR.

   If the application is executing on an external system, the initial message is received through a shared inbound MQ channel that might be served at this time by either of the two queue managers.

2. We assume that it was the LPAR1 queue manager that put the input MQ message to the shared Flow_Input_Q queue.

   As the Flow_Input_Q is a shared queue and the appropriate message flow is started within both brokers, it might be picked up from either side. Our example assumes that it is picked up by the LPAR2 message flow.

3. After it is started, the message flow is completely run within that one execution group, in our example, within LPAR2.

Apart from the MQ input node, the WebSphere Message Broker message flow uses at least two MQ output nodes. One is used to save the name of the MQ application reply-to queue, Flow_SaveData_Q in the sample, and one to input the output message in the target application. It might access shared DB2 data by means of appropriate nodes.

4. The Flow_Output_Q might be another shared queue on z/OS if the request message is going to be processed by a z/OS based component, such as a CICS transaction, or it could be a remote queue to the QSG, in which case the message is physically written to a shared transmission queue and asynchronously sent to the target queue manager via a shared sender channel.

Typically, there is going to be a second flow that serves the MQ reply message:

- From the WebSphere Message Broker point of view, this is a completely equivalent scenario, with the only difference being that the Flow_SaveData_Q has to be read-accessed from within an MQGet Node.
- The two flows serving the MQ request and MQ reply are completely independent from the broker point of view. Both flows might therefore start and be executed on different LPARs.

Note: With requests originating from within a z/OS application, there is an important issue with the name of the reply-to queue manager appearing in the request messages. The queue manager serving the ESB client application (MQR1 in our case) puts its own name to the reply-to-qmgr MQMD field. The ESB server application directs its reply message accordingly to that same queue manager. But the other QSG queue manager (MQR2) might be the one through which the WebSphere Message Broker message flow puts the final reply message.

To make sure that the message is immediately put to the (shared) queue and MQR2 does not attempt to forward it to MQR1, all queue managers must have their SQQMNAME attribute set to “IGNORE”.

Scenario 2: MQ in - HTTP out

As shown in Figure 2-11, assume a mediation flow that is initiated by an MQ message but transferred to a Web services (SOAP) request via HTTP to interface to the target application. This implies a synchronous call of the target application and therefore, the reply message is handled within the same flow.

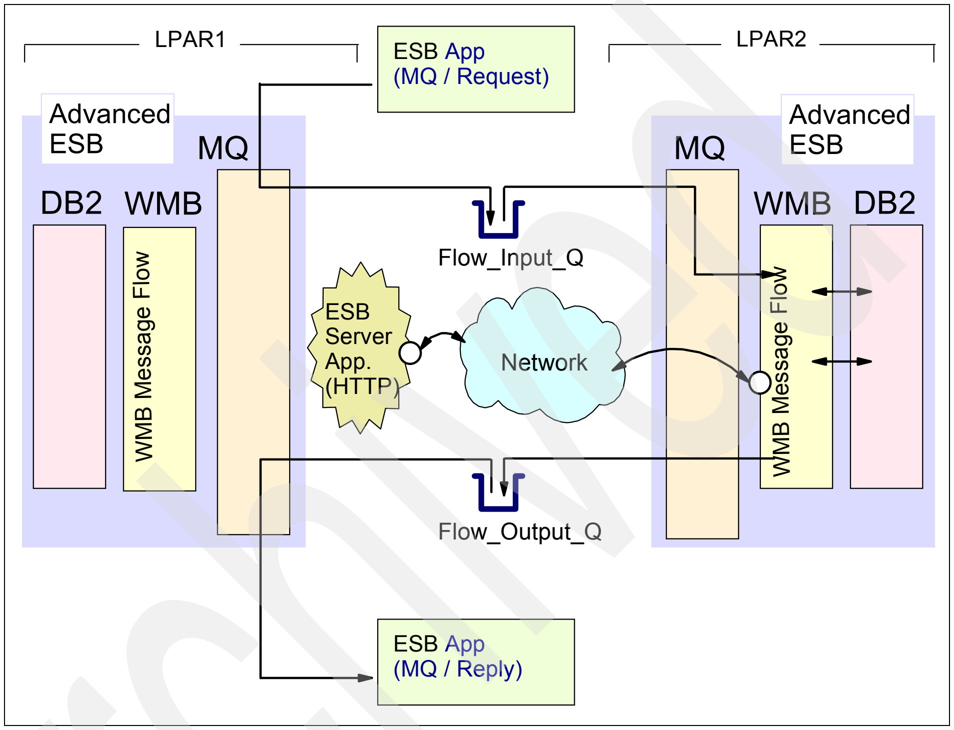


Figure 2-11 General MQ-to-HTTP flow scenario including MQ reply

The following steps discuss the process illustrated in Figure 2-11:

1. Similar to the fist scenario, the initial MQ message is sent to the shared Flow_Input_Q. The queue manager performing this MQPUT operation might be either of the two; we assume it is on LPAR1.
2. The WebSphere Message Broker message flow starts, either on LPAR1 or LPAR2. We choose LPAR2 in the scenario to highlight the fact that, even at this point, the LPAR involved might change.

   As the target application, marked *ESB server app. (HTTP)* in the figure, is going to be invoked synchronously, the MQ reply-to queue specified by the origin application is held within the flow and there is no requirement for an additional queue node within the flow. But the flow might access DB2 data by means of some ESQL or JavaCompute node. This happens within the LPAR where the WebSphere Message Broker flow started.

3. From there, the Web services call is initiated by use of a HTTP request node. The target of this call might be a Web services-enabled application that runs anywhere, possibly within the z/OS sysplex, or on another machine.
4. After the synchronous HTTP response arrives, it is further processed by the flow that is still active on LPAR2, as assumed previously. Again, additional mediation and transformation activities might be involved. Finally, an MQ reply message is created and put to the Flow_Output_Q.
5. As the Flow_Output_Q is a shared queue, it is going to be delivered to the application component through either of the two QSG queue managers so that MQGET receives it.

Note: The “switches” between the two LPARs do not necessarily take place. The two preceding samples have been chosen to demonstrate that at any point within the processing, where a shared queue is used for forwarding data, there is the flexibility to run the next step on the other LPAR. Take note of the following:

- As long as the WebSphere Message Broker components on one LPAR are up, application requests can be placed, either by MQ, or by Web services via HTTP.
- In the case of an MQ input, the mediation flow itself might execute on a different LPAR.
- In the case of an MQ output, the application that is the target of the original request via the ESB, might again access its data on either LPAR, independent from where the input was processed and the flow was executed.

This is a very significant contribution towards performance, throughput optimization and above all, towards high availability.

Moreover, it makes clear the advantages in terms of availability and flexibility offered by a queue based (asynchronous) interface compared to one based on the (synchronous) Web services.

On the other hand, it is important to realize that not every broker application can make use of this “sharing” capability. It is important that in the above scenario the application must be one where there is no “affinity” between the messages. For example, if the broker on LPAR2 handles the first part of the flow, and it saves some state information that is required for the second flow, then this information also has been made available to the broker on LPAR1. Shared broker variables, for example, would not be available.

2.3 Use of z/OS Automatic Restart Manager

Automatic Restart Manager (ARM) is a z/OS subsystem that ensures that you can restart appropriately registered applications and subsystems after a failure.

You can either restart applications and subsystems on the image on which they failed, or in the case of image failure, on another available image in the sysplex. ARM also covers application restart dependencies where applications require other applications or subsystems to be present before they restart.

Users of ARM might control both when and how a batch job or started task is to be restarted, as follows:

- When (conditions for restart)
  - Restart only on ABENDS (for example, application development).
  - Restart on ABENDS and on system failure (for example, production systems).
  - Never restart.
- How (A batch job JCL or start command will be used):
  - Persistent (Use the same JCL or command text.)
  - JOB (Use a different JCL to start a batch job.)
  - STC (Use a different command text to start a Started Task.)

The ARM policy is defined by the Administrative Data Utility IXCMIAPU and stored in the ARM couple data set.

Parameters that might be specified in order to control restart processing include:

RESTART_ORDER	This parameter applies to all restart groups and specifies the order in which elements in the same restart group are to become ready after they are restarted.
RESTART_GROUP	Identifies related elements that are to be restarted as a group if the system on which they are running fails.

2.3.1 Using ARM with WebSphere Message Broker

All components of the WebSphere Message Broker, WebSphere Message Broker V6, WebSphere MQ V6, and DB2, are fully ARM-enabled, and it is simple for an operator to assign an ARM classification and name to the broker.

Make note of the following:

- When the broker is started or stopped, it performs the necessary registration with ARM, enabling it to be restarted as defined in the ARM policy by the z/OS systems programmer.
- The Configuration Manager and User Name Server components are also ARM enabled.

The particular use of ARM functionality to control WebSphere Message Broker components restart is a decision that you have to make with the particular installation and the special characteristics of the applications in mind.

The most likely approach is the one we followed throughout our samples and discussions, which is based on the redundancy of servers and z/OS images (LPARs). Following this approach:

- WebSphere Message Broker components are always to be restarted as persistent, which means in the same LPAR in which they failed and with the same JCL. This is the simplest restart rule.
- They are never restarted on a different LPAR, because at any time, the components on the active LPAR can provide the full WebSphere Message Broker functionality. Accessibility of the service from outside is ensured by the Sysplex Distributor DVIPA function for network (TCP/IP) based requests, and MQ shared queues for messaging (MQ and JMS) based requests.

2.3.2 Recommendations and sample ARM settings for WebSphere Message Broker

The following are the set up recommendations:

1. Define RESTART_GROUPs for each queue manager instance that also contains the WebSphere Message Broker components using that queue manager and any CICS or IMS subsystems that connect to that queue manager instance.

   If you use a subsystem naming convention, you might be able to use the “?” and “*” wild-card characters in your element names to achieve the above with minimum definition effort.
2. Specify TERMTYPE(ELEMTERM) for your channel initiators to indicate that they are going to be restarted only if the channel initiator has failed and the z/OS image has not failed.

3. Specify TERMTYPE(ALLTERM) for your queue managers and the broker to indicate that they are going to be restarted if either the queue manager has failed, or the z/OS image has failed.
4. Specify RESTART_METHOD(BOTH, PERSIST) for both queue managers, channel initiators and broker components. This tells ARM to restart using the JCL it saved (after resolution of system symbols) during the last startup. It tells ARM to do this irrespective of whether the individual element failed or the z/OS image failed.

Special rules, in terms of ARM policies, might be applied in respect to the sequence in which the systems have to be restarted within an LPAR, which is quite obvious from the interdependencies between the systems.

In general, the desired sequence of starting systems after a system initial program load (IPL), possibly the result of planned maintenance, or an LPAR failure, is the following:

1. DB2
2. MQ
3. Broker
4. Application environments, such as CICS, IMS, and WebSphere Application Server

Set up examples

Both WebSphere MQ and WebSphere Message Broker can automatically register as an ARM element during their startup phase (subject to ARM availability). They de-register during shutdown, unless requested not to.

For WebSphere MQ, the use of ARM is mandatory. At startup, the queue manager determines whether ARM is available. If it is, WebSphere MQ registers using the name SYSMQMGRssid, where ssid is the four character queue-manager name, and SYSMQMGR is the element type.

Each MQ channel initiator address space determines whether ARM is available and if so, it registers with the element name SYSMQCHssid, where ssid is the queue manager name, and SYSMQCH is the element type.

To WebSphere Message Broker components, the use of ARM is optional and has to be configured within the set of environment variables, also called the component profile, that describe each component's startup options.

Sample settings are shown in Example 2-9.

Example 2-9 ARM specifications within broker component profile files

```
SBIBSAMP(BIPBPROF) for the Broker component:

MQSI_USE_ARM=YES
MQSI_ARM_ELEMENTNAME=MQSIMQR1BRK
MQSI_ARM_ELEMENTTYPE=MQSIWMB

SBIBSAMP(BIPCPROF) for the Configuration Manager component:

MQSI_USE_ARM=YES
MQSI_ARM_ELEMENTNAME=MQSIMQR1CM
MQSI_ARM_ELEMENTTYPE=MQSIWMB

SBIBSAMP(BIPUPROF) for the User Name Server component:

MQSI_USE_ARM=YES
MQSI_ARM_ELEMENTNAME=MQSIMQR1UNS
MQSI_ARM_ELEMENTTYPE=MQSIWMB
```

Example 2-10 shows the JCL used to invoke the z/OS IXCMIAPU administrative data utility with control input that defines a restart sequence to ARM, referring to DB2, WebSphere MQ, WebSphere Message Broker, and CICS as previously mentioned, to be restarted in the specified sequence.

Example 2-10 IXCMIAPU invocation with ARM definitions

```
//IXCMIAPU EXEC PGM=IXCMIAPU,REGION=2M
 //SYSPRINT DD SYSOUT=*
 //SYSIN    DD *
   DATA TYPE(ARM)
 DEFINE POLICY NAME(ARMPOL1) REPLACE(YES)

 RESTART_ORDER
   LEVEL(1)
     ELEMENT_NAME(<DB2name>)
   LEVEL(2)
     ELEMENT_NAME(<SYSMQQMGRMQR1>)
   LEVEL(3)
     ELEMENT_NAME(MQSIMQR1BRK)
   LEVEL(4)
     ELEMENT_NAME(SYSCICS_<CICSRegion_name>)

     RESTART_GROUP(DEFAULT)
     TARGET_SYSTEM(*)
     FREE_CSA(600,600)
       ELEMENT(*)
         RESTART_ATTEMPTS(3,300)
         RESTART_TIMEOUT(900)
         READY_TIMEOUT(900)
  RESTART_GROUP(MQ)
     TARGET_SYSTEM(*)
    FREE_CSA(600,600)
    ELEMENT(SYSMQMGRssid*)
        RESTART_ATTEMPTS(3,300)
     RESTART_TIMEOUT(900)
     READY_TIMEOUT(900)
     TERMTYPE(ELEMTERM)           <------ this depends on the
configuration - you might need to restrict the restart only in case of
address space failure
     RESTART_METHOD(ELEMTERM,PERSIST)
```

For an exhaustive discussion of setting up ARM policies, refer to the chapter about ARM in the IBM Redbooks publication *ABCs of System Programming Volume 5*, SG24-5655.

2.3.3 Additional systems management considerations

Additional automated operational interventions might be activated based on the particular concept applied and the requirements of applications using the services of the WebSphere Message Broker.

The following considerations apply to most of the real implementations of WebSphere Message Broker on z/OS. Some of these have already been touched on in the failure scenarios in “High availability provided by shared queues” on page 56.

Handling application environment failures

The best way of achieving this depends mainly on the configuration of these environments (CICS, IMS, and WebSphere Application Server) and the special requirements of these applications.

In the desirable case of redundant application environments on multiple LPARs, restart in place is the right way to handle these systems in the case of a failure.

If, for example, CICS applications execute within a network of cloned CICS regions that are interconnected and use dynamic routing functions, so-called application owning regions (AOR) are to simply be restarted in place, because they are backing up each other within the sysplex.

Example 1: Connecting to a CICS based Web services application

If CICS regions are targets of HTTP requests issued from within WebSphere Message Broker flows, at least one CICS region with an appropriate HTTP listener activated has to be regularly started on each LPAR and the listeners have to use a distributed VIPA port controlled by Sysplex Distributor to ensure continuous availability in case one of the regions fail, or the entire LPAR goes down.

If this can be achieved, a restart-in-place rule is sufficient and appropriate for the HTTP/Web services entry CICS regions.

An appropriate configuration is shown in Figure 2-12.

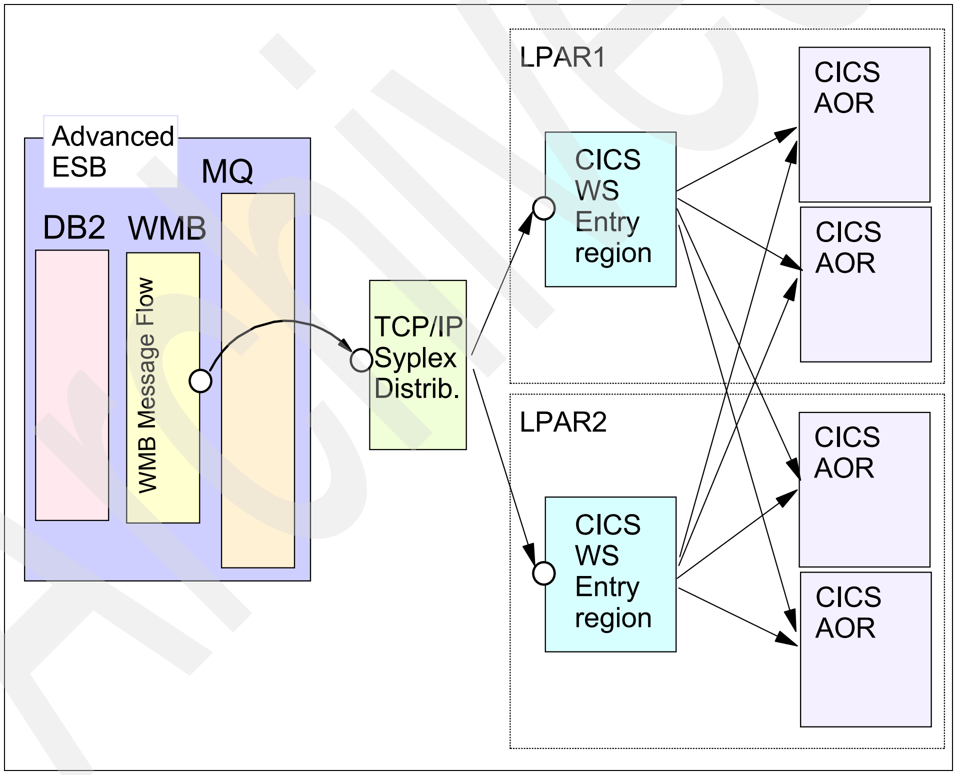


Figure 2-12 CICS configuration connected via HTTP to the WebSphere Message Broker.

Example 2: Connecting to a CICS through the MQ CICS Bridge

If CICS applications are targeted by WebSphere Message Broker message flows through means of an MQ queue using the CICS Bridge, then there are two configuration options available:

1. You might follow the same concept as shown in "Example 1: Connecting to a CICS based Web services application" on page 68, with one region per LPAR where requests are entering the CICSplex. All sysplex CICS AORs can be reached from any of these entering regions by means of CICS distributed program links to the target system.
2. CICS administrators might well decide to have AORs directly connected to MQ, as shown in Figure 2-13.

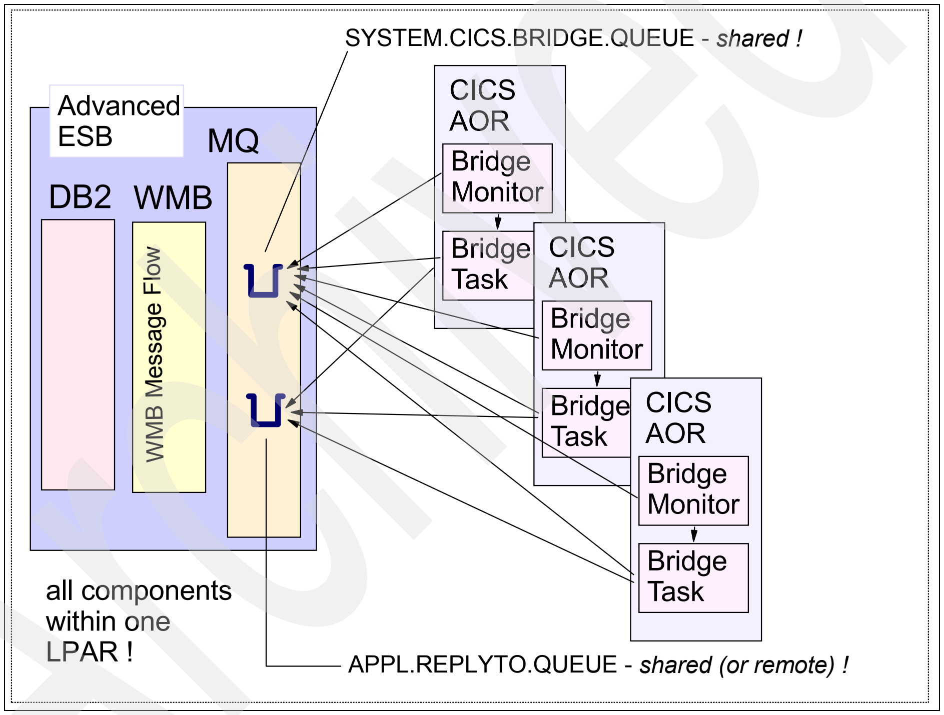


Figure 2-13 CICS configuration connected through the MQ CICS Bridge

In any case, both the CICS Bridge queue (by default, SYSTEM.CICS.BRIDGE.QUEUE) and any other MQ queues accessed by the CICS application, have to be defined as shared queues to the MQ QSG in order to maintain high availability.

Appropriate configuration rules have to be applied to IMS and WebSphere Application Server.

Handling a WebSphere Message Broker failure

This probably requires the least effort, no matter what application configuration has been chosen. No message flows can start on the LPAR on which the broker has failed; all are executed on the active LPAR. No further actions have to be taken apart from restarting the broker components in place as soon as possible.

Handling an MQ queue manager failure

The basic restart rule is to bring up the queue manager in place as soon as possible.

WebSphere Message Broker message flows do not have to specifically be considered, because the WebSphere Message Broker execution groups are stopped when MQ goes away and these groups are automatically restarted when it comes up again.

However, depending on the application configuration, additional considerations might be noted, such as:

- Are there CICS transactions designed to use MQ that are not invoked through the CICS MQ Bridge or started based on MQ triggering? In this case, these CICS applications would fail because no MQ is available within the same LPAR. If this is the case, CICS administrators have to decide how to handle or prevent this situation.
- If the failure of that CICS transactions is not acceptable, then one solution could be to disable all the transactions that require the MQ service at the time the MQ comes down. After the transactions are identified, this can be achieved, for example, by means of the CICSPlex® System Monitor (CPSM) Real Time Analysis function.

Equivalent considerations apply to IMS applications that make use of the IMS MQ adapter.

Handling a DB2 failure

The basic restart rule is to restart DB2 in place as soon as possible. Because both MQ and the broker components continue running, though without functions that require DB2 availability, additional actions might be appropriate.

As a general rule, we state the following:

1. If the MQ serving the WebSphere Message Broker is exclusively used as part of the WebSphere Message Broker, then the best way to handle the DB2 failure is to automatically bring down the queue manager when DB2 fails.

This can be done by some system automation, either based on a DB2 failure message, or on the following MQ message, which states that MQ has lost contact to DB2:

```
CSQ5014I -MQR1 CSQ5MONR Connection to D8I1 lost, DB2 terminated
abnormally
```

2. Restart both systems, DB2 first, followed by MQ.

This ensures that all WebSphere Message Broker message flows execute on the healthy LPAR and it reduces the problem to the failing MQ, which has been covered previously.

Note: This point also makes clear that, from a recovery scenario point of view, it is highly desirable that the queue manager, which is configured as part of the WebSphere Message Broker environment, must not be used for other purposes.

There might, however, be installations with a low application dependency on DB2 that decide to keep MQ and WebSphere Message Broker running, even during the time that DB2 is unavailable in that LPAR.

In this case, application failures have to be accepted and handled. One option is to identify application objects, such as CICS, IMS transactions, and WebSphere Message Broker message flows, and specifically disable or stop these for the duration of DB2 being unavailable on that LPAR. This has to be done by some product specific systems management tools.

2.4 Use of the z/OS Workload Manager

The Workload Manager (WLM) is the component of the z/OS operating system that dynamically manages resource allocation to the multiple workloads that execute at the same time within one z/OS image, or across multiple images. WLM is a major contributor to the IBM System z platform and z/OS operating system's ability to run multiple independent workloads in a controlled manner at the same time.

The purpose of workload management is to balance the available system resources to meet the demands of z/OS subsystems work managers, such as CICS, Batch, TSO, UNIX® System Services, and Web server, in response to incoming work requests. WLM tries to achieve the requirements of the workloads (response time) as described in service level agreements (SLAs), by attempting the appropriate distribution of resources without over committing them. Equally important, WLM maximizes system use (throughput) to deliver maximum benefit from the installed hardware and software platform.

The SLA contains agreed upon values for the following:

- Average transaction response time
- Distribution of response time
- System availability

All currently supported releases of z/OS require the use of WLM goal mode. This component of z/OS allows the system to classify all work in the sysplex, assigning a “service class” to each item of work.

WLM service classes

There is usually one service definition per sysplex. The service definition itself contains one or more service policies, only one of which is going to be active at any point in time in the sysplex. The service definition is a named collection of performance goals and capacity bounds. It is composed of workloads that consist of management classes and resource groups.

The service class is assigned by WLM based on classification rules in the WLM policy. These rules are supplied by the z/OS systems programmers. The service class has two important attributes:

- The goal of the work
- The importance of the work

Each piece of work in the sysplex has a service class. This way, z/OS can ensure that the work that is the most important (has an importance value of 1) meets its goal wherever possible. If all importance-1 work meets its goals, then lower importance work is managed. This management is completely automatic. WLM ensures that work from the network is routed to the system with the most capacity. It moves initiators from a heavily loaded system to another more lightly loaded system.

For a detailed discussion of z/OS WLM, refer to the IBM Redbooks publication *System Programmer's Guide to: Workload Manager*, SG24-6472.

2.4.1 Using WLM with WebSphere Message Broker

The following sections discuss how the WebSphere Message Broker can benefit using WLM.

Goal-oriented resource allocation

When a Message Broker V6 execution group address space starts, you can assign to it a Workload Manager service class. System programmers can assign different goals (typically response time) to this service class through WLM configuration choices.

The ability to assign WLM service classes to message processing workload has two significant benefits:

- As work passes through subsystems that are WLM enabled, the service class can be maintained.
- Resources such as CPU and IO “follow” users' work requests to make sure these requests meet the goals set by WLM, where possible.

WLM classification means that in the event of resource constraint, at peak time for example, high priority workloads can receive the appropriate resource to ensure that critical workloads are completed at the expense of less important work.

It is possible to assign different Execution Group address spaces to different WLM classes so that CPU and IO resource allocation can be prioritized appropriately for different classes of work performed within a single message broker.

Workload scaling

z/OS sysplex potentially brings vast processing capacity to Message Broker V6. Not only are IBM System z9™ processors extremely powerful in their own right, but up to 32 can be configured in a single z/OS image. Moreover, up to 32 z/OS images can participate in a sysplex configuration. Both WebSphere MQ and WebSphere Message Broker V6 are designed to exploit all these capabilities without any design changes to message flows.

Message flows have a built-in multiprocessing capability. Each instance of a message flow can execute on a separate thread (TCB), allowing the broker to easily exploit multiple CPUs. If a flow is unable to process enough messages due to a CPU constraint, it is simple to assign more instances to the message flow, which dynamically increases the number of eligible CPUs available for processing. No outages or changes in flow design are required.

When you have to deploy flows across multiple execution groups, for example, to overcome the 2 GB virtual storage limit on z/OS address spaces, it is easy to assign a message flow (including any additional instances) to these new address spaces. Again, no design changes are required; the flow is just assigned and becomes active automatically without a broker restart.

As described previously, it is simple to scale message flows across the sysplex by deploying to multiple brokers within the sysplex broker domain. Again, this reconfiguration process is dynamic and does not require restart. Similarly, you can remove brokers, execution groups, and message flow instances as required.

Workload isolation

In some scenarios, message processing workloads must be separate from each other. For example, regulatory requirements might mandate that infrastructure groups that are processing workloads from third parties must keep those workloads separate. While you can assign a new broker to each party, it is simpler and just as effective to assign separate execution group address spaces to different types of work. From a storage isolation perspective, these execution groups are completely separate and storage in one address space is not visible to another execution group. Failures are isolated to execution groups. In the event of an execution group failure, the execution group is restarted without affecting other execution groups owned by the broker.

2.4.2 Sample WLM settings for Broker and MQ

This section discusses some sample WLM settings for WebSphere Message Broker and WebSphere MQ.

Configuring WLM for the WebSphere Message Broker

The recommendation is to add the started task names of the Brokers and User Name Server (named MQR%BRK and MQR%UNS in the sample), either to the (high prioritized) Started Task Control (STC) default service class, which is named SYSSTC, or to an “aggressive velocity class”, which has a goal slightly beneath that of DB2 and transaction managers, such as CICS, IMS, and WebSphere Application Server.

The WebSphere Message Broker Configuration Manager might be assigned to the OMVS subsystem type, which usually specifies a less aggressive performance goal, because its activities are not performance critical.

Example 2-11 displays sample WLM settings for WebSphere Message Broker within the SYSSTC service class.

Example 2-11 Setting for WebSphere Message Broker within the SYSSTC service class

```
* Subsystem Type STC -
Last updated by user ZAZULAK on 2006/09/04 at 21:59:19
Classification:
    Default service class is SYSSTC
    Default report class is OTHER
Qualifier  Qualifier     Starting       Service  Report
```

```
  # type       name           position       Class    Class
  - ---------- -------------- ---------      -------- --------
  1 TN         MQR%BRK*                     SYSSTC   MQBR
  1 TN         MQR%UNS                      SYSSTC   MQBR
```

Configuring WLM for WebSphere MQ

We recommend that both the queue manager and the channel initiator started tasks are assigned to the (high prioritized) Started Task Control (STC) default service class named SYSSTC, or to an “aggressive velocity class”. You have to add the started task names of both (MQR%MSTR and MQR%CHIN in our case) to the Started Task Control (STC) subsystem types.

Example 2-12 displays sample WLM settings for WebSphere Message Broker within the VELO80 service class.

Example 2-12 Settings for WebSphere Message Broker within VELO80 service class

```
* Subsystem Type STC -
  Last updated by user ZAZULAK on 2006/09/04 at 21:59:19
Classification:
    Default service class is SYSSTC
    Default report class is OTHER
      Qualifier  Qualifier      Starting       Service  Report
    # type       name           position       Class    Class
    - ---------- -------------- ---------      -------- --------
    1 TN         MQR%MSTR                      VEL80    ITSOWN
    1 TN         MQR%CHIN                      VEL80    ITSOWN
```

3

Installing WebSphere ESB on z/OS

The WebSphere Enterprise Service Bus (ESB) function runs on top of a WebSphere Application Server. In this chapter, we assume that you are familiar with the WebSphere Application Server installation and that you understand the configuration concepts *Stand alone (BASE)* and *Network Deployment (ND)*. The objective of the WebSphere ESB part of this book is to guide you to an ND installation where one cluster, composed of two Application Servers, is enabled for ESB participation. We recommend that you refer to the appropriate Infocenters of the products for additional reading at:

`http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp`

The installation and configuration phases used during our setup are as follows:

- Verification of software prerequisites on z/OS
- Installation of the distribution libraries
- Installation and configuration of a WebSphere ESB WebSphere Application Server in a stand-alone configuration
- Installation and configuration of a WebSphere ESB WebSphere Application Server in an ND configuration
- Example of Web Services via WebSphere ESB to IMS
- Connection of WebSphere ESB to an MQ queue-sharing group

3.1 Prerequisites verification

The following section is a summary of the prerequisites for an ESB installation. It is important that you verify all the prerequisites are in place to avoid being impacted in the following phases.

Hardware requirements

You need hardware that supports z/OS 1.4 or z/OSe 1.4 or later.

Software requirements

You need software that supports z/OS 1.4 or z/OSe 1.4 or later with the following features installed, enabled, and configured:

- z/OS Communications Server (TCP/IP) or equivalent
- z/OS UNIX System Services and the Hierarchical File System (HFS)
- IBM SecureWay® Security Server (RACF) or equivalent security management product
- System Secure Sockets Layer (SSL) security required when using SSL
- Resource Recovery Services (RRS)
- WebSphere Application Server Version 6.02 for z/OS (in our installation, we used level 13.)

3.2 WebSphere ESB basic installation considerations

WebSphere Application Server V6 and WebSphere ESB are distributed on media, and have to be loaded onto DASD before you can use them. This is what we call “SMP” work.

If the product delivery package is an IBM Custom-Built Product Delivery Option (CBPDO), you must use SMP/E to unload the product code onto the z/OS system. Refer to the Program Directory as a guidance through the process. The Program Directory contains information concerning the material and procedures associated with the installation of the product code. You can download the Program Directory in PDF format from the WebSphere Enterprise Service Bus for z/OS download page at:

`http://www.ibm.com/software/integration/wsesb/library/infocenter/`

If the product delivery package is an IBM SystemPac® or ServerPac, you can copy the SMP/E data sets that correspond to the CustomPac service level on to the z/OS system.

WebSphere ESB has to be configured on top of WebSphere Application Server, as shown in Figure 3-1.

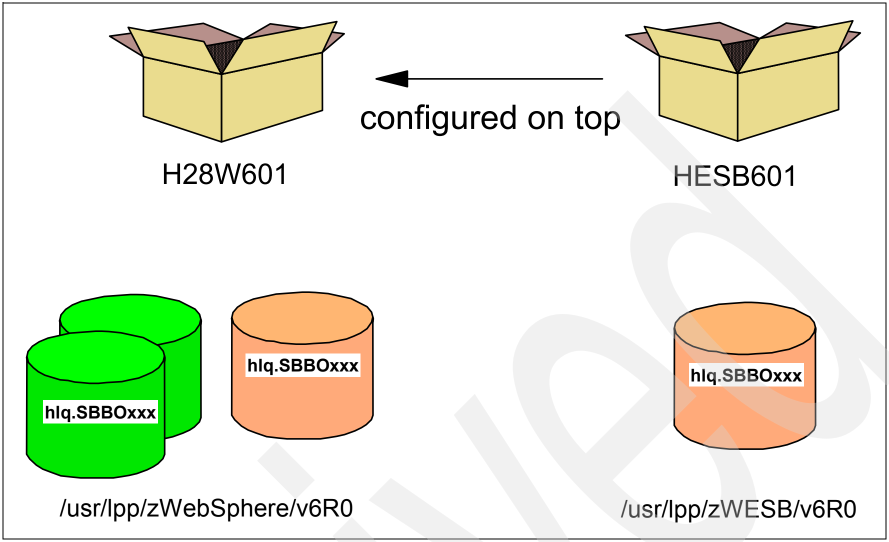


Figure 3-1 WebSphere ESB configured on top

This phase of the install assumes that the correct version of WebSphere Application Server for z/OS has been installed and customized. The SMP/E install process checks the version level of WebSphere Application Server for z/OS to ensure version compatibility. If the version of WebSphere Application Server for z/OS is not at the level required to support WebSphere Enterprise Service Bus, then you have to take the appropriate actions to update the version of WebSphere Application Server for z/OS. For information about how to apply service to WebSphere Application Server for z/OS, refer to Applying product maintenance in the WebSphere Application Server for z/OS information center on the Web at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/welcome_nd.html

Product data set contents

The WebSphere Enterprise Service Bus for z/OS product data sets are divided into target data sets (used during product customization and execution) and distribution libraries.

The WebSphere Enterprise Service Bus for z/OS target libraries are as follows:

- wesb_hlq.SBSBEXEC CLIST scripts
- wesb_hlq.SBSBJCL JCL for installation jobs

The WebSphere ESB for z/OS distribution libraries are as follows:

- wesb_hlq.ABSBANT HFS files
- wesb_hlq.ABSBEBCD HFS files (EBCDIC)
- wesb_hlq.ABSBEXEC CLISTs used by the Customization Dialog
- wesb_hlq.ABSBJCL JCL for installation jobs

After successfully unloading the product code from the install media on to the system, the WebSphere administrator implements phase 2 of the installation by running the install/config scripts.

Note: It is a good idea to put the installed HFS in R/O so that it cannot be altered.

3.3 Configuring a WebSphere ESB stand-alone (Base) installation

After completing the installation, you have to perform some customization tasks to bring additional WebSphere ESB capabilities in to the WebSphere Application Server. WebSphere Enterprise Service Bus complements WebSphere Application Server by introducing enhanced integration capabilities. Services must, by definition, be reusable by a number of different consumers, so that they benefit from reduced connections. This applies to WebSphere Application Server as a consumer or provider. WebSphere Enterprise Service Bus is the mediation layer that runs on top of the transport layer within WebSphere Application Server. WebSphere Enterprise Service Bus provides pre-built mediation functions and easy-to-use tools to enable rapid construction and implementation of an ESB as a value-add on top of WebSphere Application Server.

From this definition, we realize that WebSphere Enterprise Service Bus for z/OS is a defining element, an integration pillar, that requires the qualities of z/OS and WebSphere Application Server for z/OS to offer availability as the transport, transformation, and mediation layer. The modules in charge of performing the operations for WebSphere Enterprise Service Bus are called *mediation modules* and are composed of *mediation flow components* built from *mediation primitives*. These mediation components are built using WebSphere Integration Developer (WID).

Extended reading about WebSphere ESB and mediation can be found in the following IBM publications:

- *Architecting High Availability Using WebSphere V6 on z/OS*, SG24-6850
- *Getting Started with WebSphere Enterprise Service Bus V6*, SG24-7212
- *z/OS Getting Started: WebSphere Process Server and WebSphere Enterprise Service Bus V6*, SG24-7378

Mediations can be simple, or can contain a mediation flow. Mediations are deployed as enterprise applications in an application server, which is defined as a member of the Service Integration Bus (SIB). The deployment format is an *Enterprise ARchive (EAR)*.

The following sections describe the process of building a WebSphere Application Server base node, augment it for WebSphere ESB usage, and finally test the sample application StockQuote.

We build this configuration from scratch. After upgrading the configuration for WebSphere ESB, you can see that some test programs are automatically installed and available. This is true also for the StockQuote application, which comes with the product and uses most of the WebSphere ESB facilities.

3.3.1 Building a base WebSphere Application Server cell

The first step is to have a WebSphere Application Server configured. We assume that you are familiar with the building procedure for a “stand-alone (base)” server via ISPF panels. The configuration we used is represented in Figure 3-2.

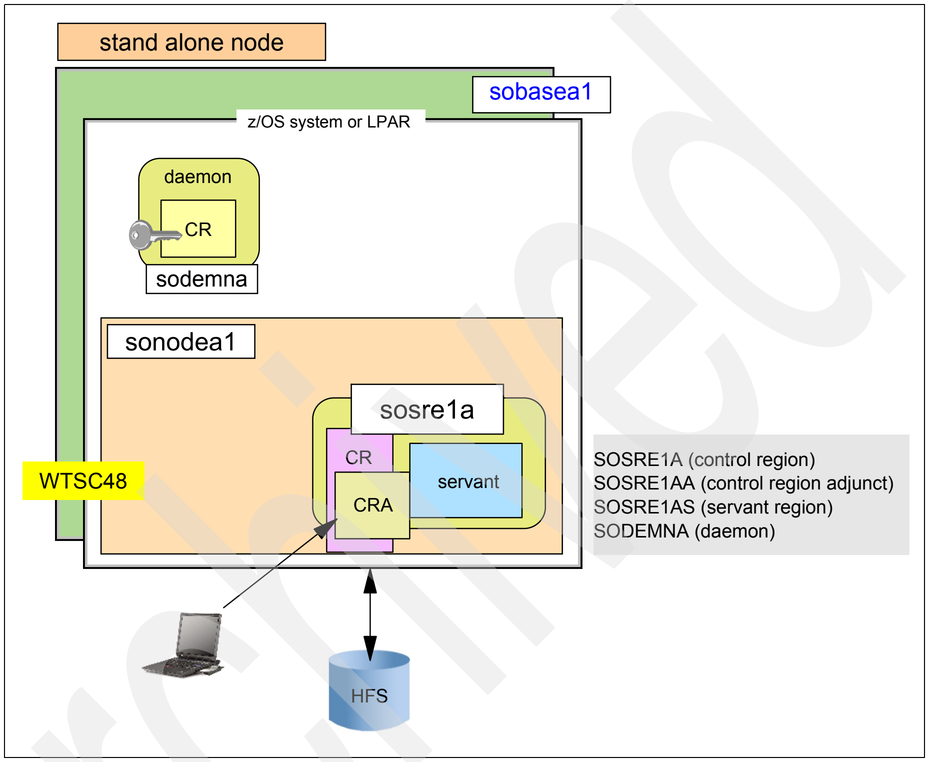


Figure 3-2 Stand-alone node without WebSphere ESB

The configuration shown in Figure 3-2 has no capability yet to execute any WebSphere ESB related applications. Two additional steps have to be completed, *zSMPInstall* and *zWESBConfig*. These are described in the following sections.

3.3.2 Installing WebSphere ESB in the base cell (zSMPInstall)

This step updates the cell's configuration HFS with links to the WebSphere ESB installation HFS and is performed by running the WebSphere Enterprise Service Bus for z/OS install script zSMPInstall.sh. You must run this script from the SMP/E install HFS directory. The default install HFS directory for the product is as follows:

/usr/lpp/zWESB/V6R0/zos.config/bin

The script requires the keyword parameters -smproot and -runtime, where -smproot represents the HFS directory that resulted from the SMP/E installation and where -runtime represents the configuration root ($WAS_Home) of the WebSphere Application Server for z/OS.

An easy way to run these commands is by using a batch JCL executing the batch shell program, as shown in Example 3-1. This makes the task easily repeatable.

Example 3-1 JCL for zSMPInstall in base node

```
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//STDENV DD *
_CBINSTALL=/usr/lpp/zWESBSO/V6R0
_CBCONFIG=/wessoconfig/sobasea1/sonodea1
_ASDIR=AppServer
/*
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   $_CBINSTALL+
   /zos.config/bin/zSMPInstall.sh +
    -smproot $_CBINSTALL +
    -runtime $_CBCONFIG/$_ASDIR +
    -install +
    1> /tmp/INSWESB1.out +
    2> /tmp/INSWESB1.err
//*  STEP Copy - Copy script output back to joblog              */
//*****************************************************************/
//PRINT  EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=WAS601.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/INSWESB1.out'
   BBOHFSWR '/tmp/INSWESB1.err'
//*****************************************************************/
```

Example 3-2 contains a sample of the execution log for this installation step.

Example 3-2 Execution log of zSMPInstall

```
CWPIZ0253I: parsing command arguments...
CWPIZ0254I: parsing arguments complete
CWPIZ0255I: setting up configuration...
CWPIZ0256I: set up configuration complete
CWPIZ0257I: creating the symbolic links...
CWPIZ0259I: creation of symbolic links complete
CWPIZ0260I: doing post install file updates...
CWPIZ0262I: post install updates complete
CWPIZ0263I: running Configuration Manager update...
CWPIZ0264I: Configuration Manager update complete
```

As a result of the script execution, you must have:

- Symbolic links created

  There must be a symbolic link for every file in the install SMP/E read-only HFS directory.

  The following script file must also be created with a symbolic link:

  zWESBConfig.sh in $_CBCONFIG/$_ASDIR/bin/

  In our configuration, we have:

  $_CBINSTALL/zos.config/bin/zWESBConfig.sh

  This file is the shell script used in the second configuration step.

  As you see in the following steps, our configuration is built with a Deployment Manager server and two managed nodes (one called b1 and the other called c1).

  Each server has to be "augmented" with the WebSphere ESB code by running in the *zWESBConfig.sh* script against each server and pass the correct response file.

  In our case, for example, the following was the sequence for the order of processing:

  a. Build a Deployment Manager server.
  b. Run the following against the Deployment Manager server:

     ```
     ./zWESBConfig.sh -response /u/larryc/DmgrDB2.rsp -augment
     ```

  c. Build the b1 managed node.
  d. Run the following against the b1 managed node server:

     ```
     ./zWESBConfig.sh -response /u/larryc/ManagedDB2_b1.rsp -augment
     ```

e. Build the c1 managed node.

f. Run the following against the c1 managed node server:

```
./zWESBConfig.sh -response /u/larryc/ManagedDB2_c1.rsp -augment
```

- Additional symbolic links and properties files created
  - The service product directory, if it does not already exist, must be created. The default name is:

    $_CBCONFIG/$_ASDIR/properties/service/product/WBI
  - The service product backup directory, if it does not already exist, must be created. The default name is:

    _CBCONFIG/$_ASDIR/properties/service/product/WBI/backup
  - The service log directory, if it does not already exist, must be created. The default name is:

    $_CBCONFIG/$_ASDIR/properties/service/logs/WBI
  - The service level properties from the install SMP/E read-only HFS directory has the following default name:

    $_CBINSTALL/productversion/properties/service/product/WBI/service/service-level.properties

    This must be copied into the service product directory. The default name is:

    $_CBCONFIG/$_ASDIR/properties/service/product/WBI/service-level.properties
  - A symbolic link file in the runtime directory has the default name:

    $_CBCONFIG/$_ASDIR/properties/service/product/WBI/service
  - This must be created for the read-only HFS directory service directory. The default name is:

    $_CBINSTALL/properties/service/product/WESB/service/
  - An Installer properties file in the runtime directory must be created. the default name is:

    $_CBCONFIG/$_ASDIR/properties/service/product/WBI/zWESBPostInstaller.properties

3.3.3 Configuring WebSphere ESB as stand-alone (zWESBConfig)

In a WebSphere ESB installation, relational tables are required for the Common Event Infrastructure (CEI). Table spaces, tables, and indexes are going to be defined later, but before executing the *zWESBConfig* script, you must define the database and the storage group, because they are inputs for the script. You can use Cloudscape™ or DB2 as the persistence mechanism. We used DB2 for our configuration.

The following command provides a skeleton for the DB2 definitions, although it has to be adjusted to your environment:

```
$_CBINSTALL/zos.config/wesb_DB_StorGrp.sql
```

This member is provided in an ISO8859 (ASCII) code page. To execute it in a z/OS environment, you have to perform any *one* of the following:

- Convert the member to *Extended Binary Coded Decimal Interchange Code (EBCDIC)* with the following command and execute the command through a utility, such as SPUFI:

  ```
  iconv -f ISO8859-1 -t IBM-1047 wesb_DB_StorGrp.sql > wesb_DB_EBCDIC.sql
  ```

- Move the file to an *American Standard Code for Information Interchange (ASCII)* platform with Universal Database (UDB) or IBM DB2 Connect™ installed, which is connected with the DB2 on z/OS through IBM DRDA®. Execute the command through the DB2 “Command Editor”.

When you run the install script, a skeleton DDL file (*.sql), with definitions for the CEI database and the ESB database (used by the mediation logging primitive) is created and you have to adjust it for your installation. Our sample is shown in Example 3-3.

Example 3-3   DDL for defining database and storage group

```
DROP   DATABASE EVENT ;
DROP   DATABASE EVENTCAT ;
DROP   STOGROUP EVTSTO;
CREATE STOGROUP EVTSTO VOLUMES (WBIUS3,WBIUS4,WBIUS5) VCAT WSDB2;
CREATE DATABASE EVENT
  STOGROUP EVTSTO
  BUFFERPOOL BP0
  INDEXBP BP0;
CREATE DATABASE EVENTCAT
  STOGROUP EVTSTO
  BUFFERPOOL BP0
INDEXBP BP0;
COMMIT;

DROP   DATABASE ESBDB ;
DROP   STOGROUP ESBDBSTO;
CREATE STOGROUP ESBDBSTO VOLUMES (WBIUS3,WBIUS4,WBIUS5) VCAT WSDB2;
CREATE DATABASE ESBDB
  STOGROUP ESBDBSTO
  BUFFERPOOL BP0
  INDEXBP BP0;
COMMIT;
```

Make sure that it runs successfully. After the DB2 definitions have been completed, you are ready to run the second additional step, the product configuration script “zWESBConfig.sh” through the shell command. There are several configuration options that are driven by response files. The content in the response file is used to augment the WebSphere Application Server for z/OS profile with WebSphere Enterprise Service Bus for z/OS configuration data.

Profile augmentation is the process by which a product administrator adds WebSphere Enterprise Service Bus for z/OS configuration data to the WebSphere Application Server for z/OS default profile. You must run the command from the directory /WebSphere/V6R0M0/AppServer/bin and include an argument that points to the response file. The property values that you set in the response file are used to create your product configuration.

In our configuration, we run the “zWESBConfig.sh” script via the batch shell, as shown in Example 3-4. Remember that the identity used on the job must be WSADMIN, or one with root authority. Example A-5 on page 324 shows the script used for our configuration.

Example 3-4 Running the base server config in batch

```
//INST1 EXEC PGM=IKJEFT01,REGION=0M,TIME=1440
//STDENV DD *
_CBINSTALL=/usr/lpp/zWESBSO/V6R0
_CBCONFIG=/wassoconfig/sobasea/sonodea1
_ASDIR=AppServer
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH +
   $_CBCONFIG+
   /$_ASDIR+
   /bin/zWESBConfig.sh +
    -response /u/larryc/standAloneProfileDB2.rsp +
    -runtime $_CBCONFIG/$_ASDIR +
    -augment +
    1> /tmp/CFGWESB1.out +
    2> /tmp/CFGWESB1.err
//*  STEP Copy - Copy script output back to joblog                  */
//******************************************************************/
//PRINT  EXEC PGM=IKJEFT01,REGION=0M
//SYSEXEC  DD  DISP=SHR,DSN=BBS6048.SBBOEXEC
//SYSTSIN  DD *
   BBOHFSWR '/tmp/CFGWESB1.out'
   BBOHFSWR '/tmp/CFGWESB1.err'
//SYSTSPRT DD SYSOUT=*
/*
```

The install command requires the -response keyword parameter to define the response file containing the properties for configuring WebSphere Enterprise Service Bus for z/OS.

Attention: Make sure to verify the following while preparing the response file:

- Be sure that when you enter the value for a property, that you do *not* enter trailing spaces. You can verify this by using "OBROWSE" in "HEX ON" mode on your file. "OEDIT" will not help you because it pads shorter lines with spaces. This can be corrected by using "vi" in Telnet mode or by bringing the file to a workstation.
- During the zWESBConfig run, an SQL is built for defining table spaces, tables, and indexes in the database that was created manually. Those definitions have to be executed manually afterwards. z/OS is not able to implicitly run those definitions. To avoid z/OS from trying to perform this process, we suggest that you specify `dbDefineSQL=false`.
- We found out that the "response file" also contains directives related to the process server. These are not applicable for a WebSphere ESB installation.

Example 3-5 shows some critical statements in the response file for the augmentation of the base node.

Example 3-5 Excerpt of the response file

```
......
######################################################################
Jmsuser=wsadmin
Jmspass=wsadmin
Dbuser=wsadmin
Dbpass=wsadmin
CONFIGSERVER=sosre1a
DBLOCATION=DB8I
######################################################################
augment
######################################################################
#Profile name
profileName=default
######################################################################
# Profile path
profilePath=/wassoconfig/socell/sonodea1/AppServer/profiles/default
######################################################################
#Template path
templatePath=/wassoconfig/socell/sonodea1/AppServer/profileTemplates/de
fault.*
######################################################################
# Cell name
cellName=sobasea1
```

```
######################################################################
# Node name
nodeName=sonodea1
######################################################################
# Server name
serverName=$CONFIGSERVER
######################################################################
# ESB Properties
######################################################################
# Database product name for Enterprise Service Bus
esbDbProduct=DB2UDBOS390_V8_1
######################################################################
# Database name for Enterprise Server Bus
esbDbName=ESBDB
######################################################################
# Database Storage Group for Enterprise Service Bus
esbDbStorageGroup=ESBDBSTO
######################################################################
# Database Schema Name for Enterprise Service Bus
esbDbSqlId=ESBLOG
######################################################################
# Business Process Choreographer Properties  (NOT applicable)
######################################################################
# Database Host Name
dbHostName=wtsc48.itso.ibm.com
######################################################################
# Database Port Number  for type 4 if used
dbServerPort=38100
######################################################################
# Database Connection Location
dbConnectionLocation=$DBLOCATION
######################################################################
# Database Home Classes Directory
dbJDBCClasspath=/usr/lpp/db2/d8ig/jcc/classes
######################################################################
# Database DB2 JCC Properties Directory for type 2
dbJDBCProperties=/usr/lpp/db2/d8ig/jcc/properties
######################################################################
# Create new or use existing database
dbCreateNew=false
######################################################################
# Execute the database table and datasource definitions
dbDefineSQL=false
```

Example 3-6 is an example of a “zWESBConfig” execution and successful completion.

Example 3-6 Output from zWESBConfig.sh

```
CWPIZ0253I: parsing command arguments...
CWPIZ0254I: parsing arguments complete
CWPIZ0255I: setting up configuration...
CWPIZ0256I: set up configuration complete
CWPIZ0265I: augmenting profile(s)...
INSTCONFSUCCESS: Profile augmentation succeeded.
```

3.3.4 Post installation tasks and verification

The following sections describe the steps that you have to perform after the installation has completed. They also describe how you can verify that the installation has been successfully completed and what are the best commands to run SQL.

SQL definitions

Before we can run any ESB related application, we need to execute the DB2 definitions for an additional database, table spaces, tables, and indexes. A sample is available in ceidef.sql, and in ASCII format. You can convert it to EBCDIC using the **`iconv`** command, or by using a DB2 Connect DRDA connection to the DB2 on z/OS. Using DB2 SPUFI to execute the commands in ceidef.sql does not work and is not supported.

You can find the ceidef.sql member in the following directory:

/wassoconfig/socell/sonodea1/AppServer/profiles/default/databases

Execute the statements and verify that it completes successfully.

The definitions contained in the .sql file, generated as a result of the zWESBConfig script, and the Table_esb_DB2UDBOS390_V8_1.sql, located in the same directory, must be executed. These definitions create a table that is used by the "message logger" mediation primitive. You can see a sample in Example 3-7.

Example 3-7 Table_esb_DB2UDBOS390_V8_1.sql

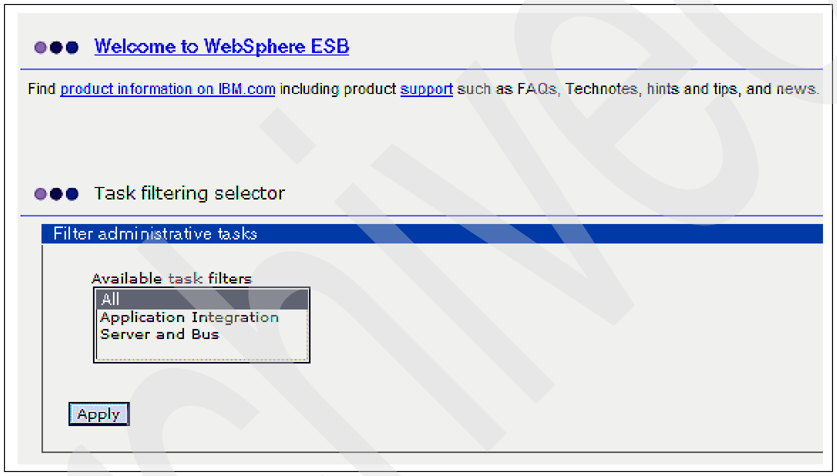
```
-- DB2UDB for z/OS V8.1 schema for Message Logger Mediation

-- ESBDB            DBName
-- ESBDBSTO     StorageGroup
-- ESBLOG              SQLID

-- Create Table spaces
CREATE TABLESPACE ESBTS LOCKSIZE ROW CCSID UNICODE BUFFERPOOL BP0
   IN ESBDB USING STOGROUP ESBDBSTO;
CREATE LOB TABLESPACE ESBCLOB IN ESBDB USING STOGROUP ESBDBSTO;

-- Create Tables
CREATE TABLE ESBLOG.MSGLOG
  (TIMESTAMP TIMESTAMP NOT NULL,
   MESSAGEID VARCHAR(36) NOT NULL,
   MEDIATIONNAME VARCHAR(256) NOT NULL,
   MODULENAME VARCHAR(256),
   MESSAGE CLOB(100000K),
   VERSION VARCHAR(10),
CONSTRAINT PK_MSGLOG PRIMARY KEY (MESSAGEID, TIMESTAMP, MEDIATIONNAME))
   IN ESBDB.ESBTS;

CREATE UNIQUE INDEX ESBLOG.MSGLOG_INDEX_PK
ON ESBLOG.MSGLOG (MESSAGEID, TIMESTAMP, MEDIATIONNAME)
        USING STOGROUP ESBDBSTO;

-- Create AUX CLOB Tables/Index
CREATE AUX TABLE ESBLOG.MESSAGECLOB IN ESBDB.ESBCLOB
         STORES ESBLOG.MSGLOG COLUMN MESSAGE;
CREATE INDEX ESBLOG.MESSAGECLOB_IDX
ON ESBLOG.MESSAGECLOB USING STOGROUP ESBDBSTO;
```

Looking at the Admin console

To be able to view all the elements of the installation, you have to stop and start the server in the base node. When the server comes up, you can see an additional address space (*Control Region Adjunct (CRA)*) is also started. The base node cell is now composed of the following address spaces:

- SOSRE1A (Control Region)
- SOSRE1AA (Control Region Adjunct)
- SOSRE1AS (servant region)
- SODEMNA (daemon)

When you log in to the Admin console, you can see the first change in the Welcome window “Welcome to WebSphere ESB”, as shown in Figure 3-3.


Welcome to WebSphere ESB

Find product information on IBM.com including product support such as FAQs, Technotes, hints and tips, and news.

Task filtering selector

Filter administrative tasks

Available task filters

All
Application Integration
Server and Bus

Apply

Figure 3-3 Admin Welcome window with ESB

Other updates are visible in the left selection pane, where you can see an additional entry System Component Architecture(SCA) Modules under Applications, as shown in Figure 3-4.

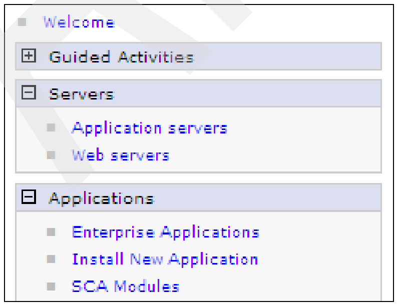
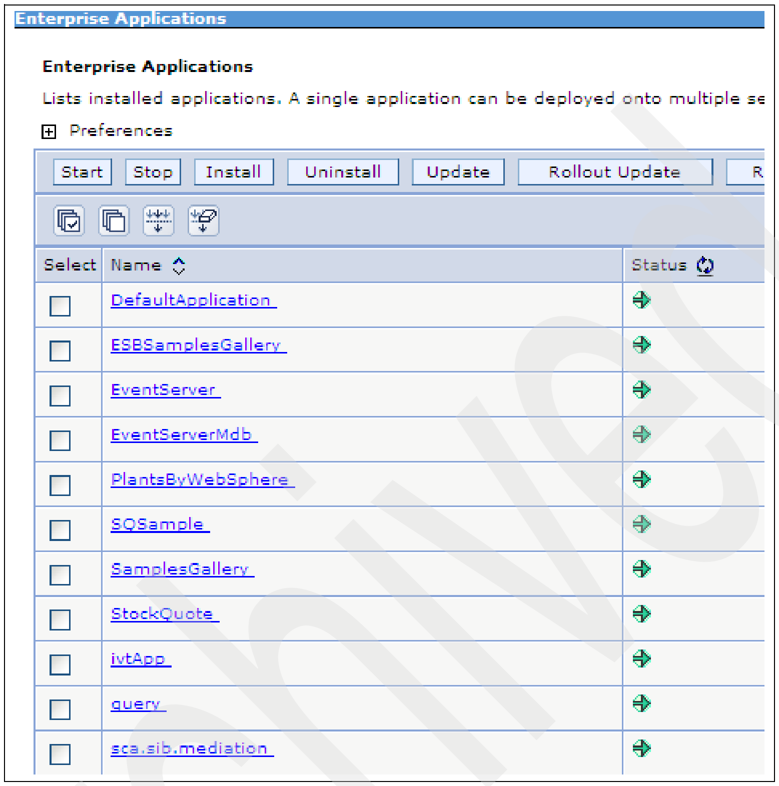


Figure 3-4 Application entry/SCA Modules

A few additional programs have been installed in the (only) SOSRE1A server, as shown in Figure 3-5.


Enterprise Applications

Enterprise Applications

Lists installed applications. A single application can be deployed onto multiple se

⊞ Preferences

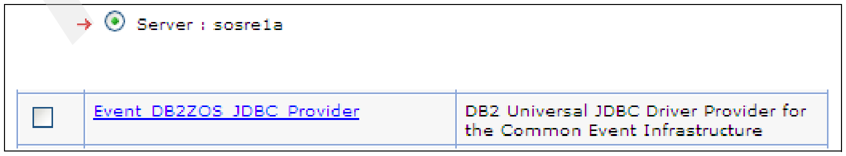
Select	Name	Status
☐	DefaultApplication	➜
☐	ESBSamplesGallery	➜
☐	EventServer	➜
☐	EventServerMdb	➜
☐	PlantsByWebSphere	➜
☐	SQSample	➜
☐	SamplesGallery	➜
☐	StockQuote	➜
☐	ivtApp	➜
☐	query	➜
☐	sca.sib.mediation	➜

Figure 3-5 Installed programs in server SOSRE1A

Besides sample programs, there is one particular Java 2 Platform, Enterprise Edition (J2EE) program (sca.sib.mediation) that acts as the container (runtime environment) of the mediations on the ESB.

Verifying the resources required by WebSphere ESB

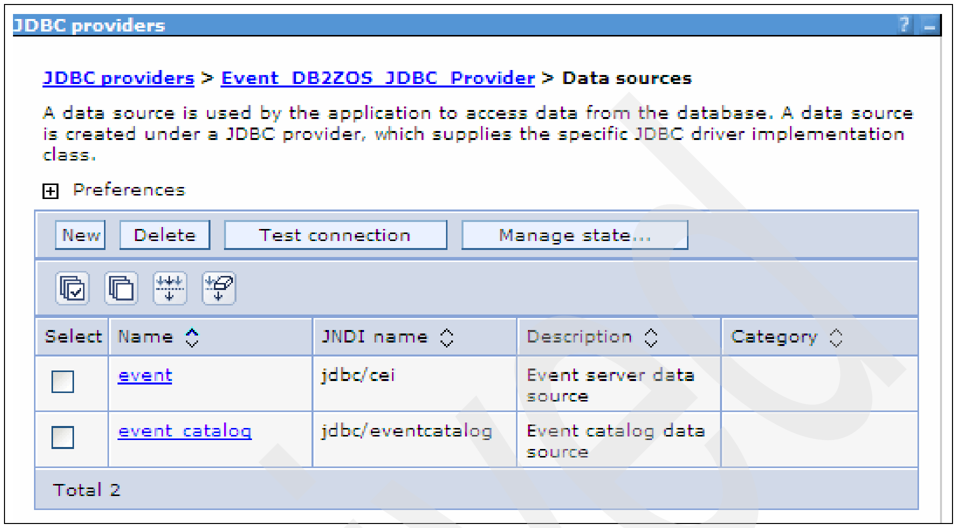
Java Database Connectivity (JDBC) resources are used by the Common Event Infrastructure (CEI). To accommodate this function under DB2 V8, a new JDBC provider has been defined, as shown in Figure 3-6.


Server : sosre1a

☐	Event_DB2ZOS_JDBC_Provider	DB2 Universal JDBC Driver Provider for the Common Event Infrastructure

Figure 3-6 JDBC provider for CEI

Under this provider, two data sources have been set up, both using a Type 2 connection to DB2, as shown in Figure 3-7.


JDBC providers

JDBC providers > Event_DB2ZOS_JDBC_Provider > Data sources

A data source is used by the application to access data from the database. A data source is created under a JDBC provider, which supplies the specific JDBC driver implementation class.

⊞ Preferences

New | Delete | Test connection | Manage state...

Select	Name	JNDI name	Description	Category
☐	event	jdbc/cei	Event server data source	
☐	event catalog	jdbc/eventcatalog	Event catalog data source	

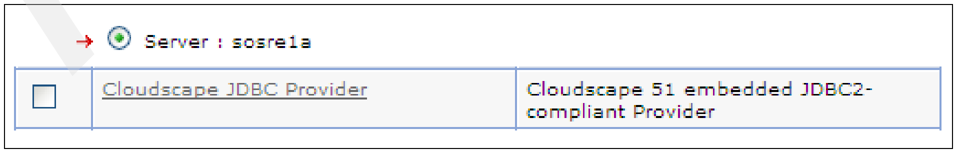
Total 2

Figure 3-7 Data sources for CEI

For each data source, under a JDBC provider, select the **Test connection** function to verify that all things are working. If you receive an error, the following are the most probable causes to check:

- The environment variables pointing to JDBC classes and the native library.
- The STEPLIBs for the DLLs pointed to by the driver shared objects (.so) (Type 2 access).
- The authorizations (the Test connection uses the “Component Managed” alias).
- Verify that the databases, tables, and indexes have been defined correctly, because the Test Connection does not check this.

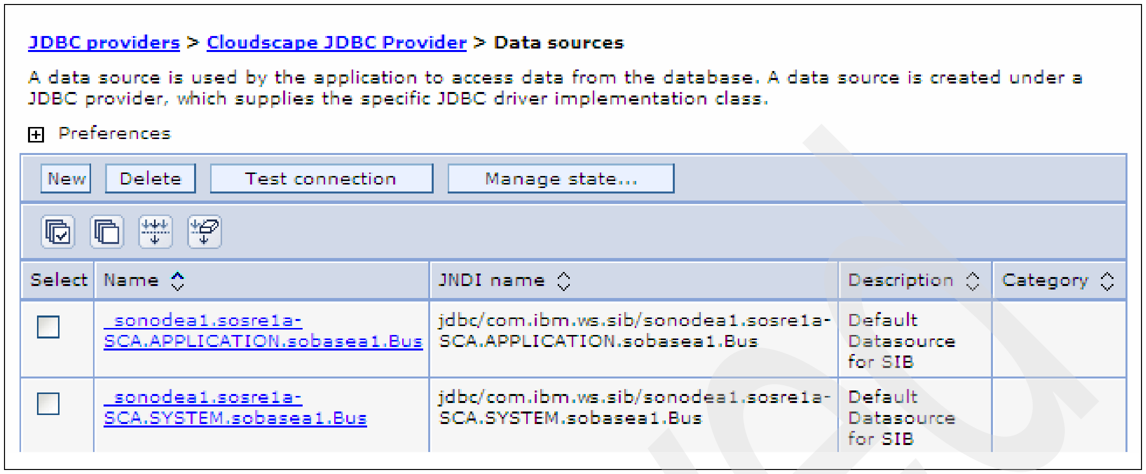
If the problem persists, you might consider switching to a Type 4 connection by changing the DataSource information bundle. Beside DB2, the server can also use CloudScape as its JDBC provider, as shown in Figure 3-8.


Server : sosre1a

☐	Cloudscape JDBC Provider	Cloudscape 51 embedded JDBC2-compliant Provider

Figure 3-8 JDBC provider for CloudScape

Under this provider, we find the data sources shown in Figure 3-9.


JDBC providers > Cloudscape JDBC Provider > Data sources

A data source is used by the application to access data from the database. A data source is created under a JDBC provider, which supplies the specific JDBC driver implementation class.

⊞ Preferences

Select	Name	JNDI name	Description	Category
☐	_sonodea1.sosre1a-SCA.APPLICATION.sobasea1.Bus	jdbc/com.ibm.ws.sib/sonodea1.sosre1a-SCA.APPLICATION.sobasea1.Bus	Default Datasource for SIB	
☐	_sonodea1.sosre1a-SCA.SYSTEM.sobasea1.Bus	jdbc/com.ibm.ws.sib/sonodea1.sosre1a-SCA.SYSTEM.sobasea1.Bus	Default Datasource for SIB	

Figure 3-9   SCA data sources for Cloudscape

Using the sample program

A sample program has been delivered with the WebSphere ESB product called StockQuote. It contains a mediation with an embedded flow, which sits (mediates) between a JMS input request and two Hypertext Transfer Protocol (HTTP) Web Services callouts. In *mediation* terms, we can say that this mediation can be entered by a JMS call (exports JMS) and it calls services via Web Services over HTTP (imports HTTP or Web Services). The mediation module consists of an export that provides an interface for the client service requester, an import that provides interfaces to the external Web Service providers, and a mediation flow component that defines the mediation implementation. The application is deployed as a J2EE application, as shown in Figure 3-10.

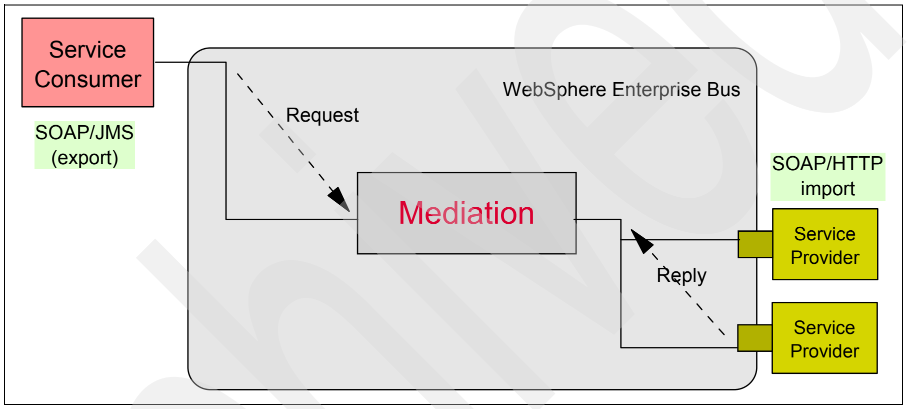


Figure 3-10 StockQuote mediation flow

The mediation module, StockQuote, is built in the assembly editor of WebSphere Integration Developer, and the mediation flow component (StockQuote_MediationFlow) is created with the mediation flow editor of this tool. The StockQuote mediation module consists of the following elements:

- StockQuoteService with a Web Services Description Language (WSDL) interface (StockQuoteService) using a SOAP/JMS Web Service binding so that the servlet front end can connect to the mediation module using JAX-RPC.
- StockQuote_MediationFlow, which contains the mediation flow. In this sample, a flow is implemented, although it is not always required.
- A RealtimeService that has a Web Service binding and an interface that matches the real-time (premium) service. In this sample, a WSDL file (RealtimeService.wsdl) was imported.

- DelayedService that has a Web Service binding and an interface that matches the delayed (standard) service. In this sample, a WSDL file (DelayedService.wsdl) was imported.

Figure 3-11 shows the request flow that defines the mediation logic applied to the message as it flows through the StockQuote_MediationFlow component toward the target service providers. This is the forward direction of the flow.

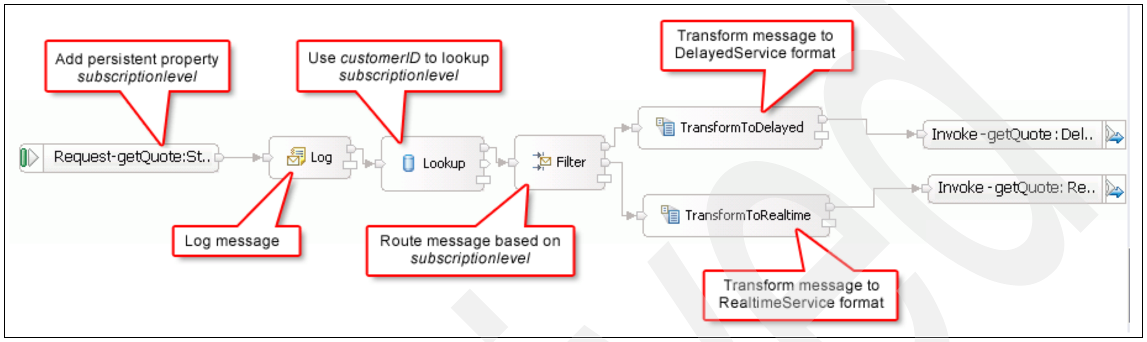

Add persistent property *subscriptionlevel*
Use *customerID* to lookup *subscriptionlevel*
Transform message to DelayedService format
Request-getQuote:St..
Log
Lookup
Filter
TransformToDelayed
Invoke-getQuote:Del..
Invoke-getQuote: Re..
TransformToRealtime
Log message
Route message based on *subscriptionlevel*
Transform message to RealtimeService format

Figure 3-11 Forward request flow of the sample

The request flow is executed from left to right, in the following order:

1. The property "subscriptionLevel" is set in the correlation context of the message so that it is available later in the response flow.
2. The request is logged using a message logger mediation primitive named "Log".
3. A database lookup mediation primitive named "Lookup" uses the customerID element in the message body to determine whether the customer is entitled to the premium or standard service by looking this information up in the supplied customer database. This information is added to the subscriptionLevel property in the correlation context of the message for later use later.
4. The request is routed by a message filter called "Filter", based on the subscriptionLevel information in the correlation context, to either the real-time or delayed stock quote service.
5. The message is transformed on the way to either service by *Extensible Style-sheet Language for Transformations (XSLT)* primitives "TransformToDelayed" and "TransformToRealtime" so that it matches what the service expects.

6. The response from each service is passed through an XSLT mediation primitive (DelayedToStockQuoteService and RealtimeToStockQuoteService) to match the format that is required by StockQuoteService. In addition, the XSLT primitive is also used to map the value of subscriptionLevel in the correlation context to the property qualityOfService in the message.

   The qualityOfService text indicates “Premium” to a response that is returned from the real-time service, and “Standard” to a response that is returned from the delayed service.

   The qualityOfService text is displayed in the client to indicate the service provider that was used.

Figure 3-12 shows the response flow that defines the mediation logic applied to the returning message as it flows through the StockQuote_MediationFlow component from the target service provider toward the client.

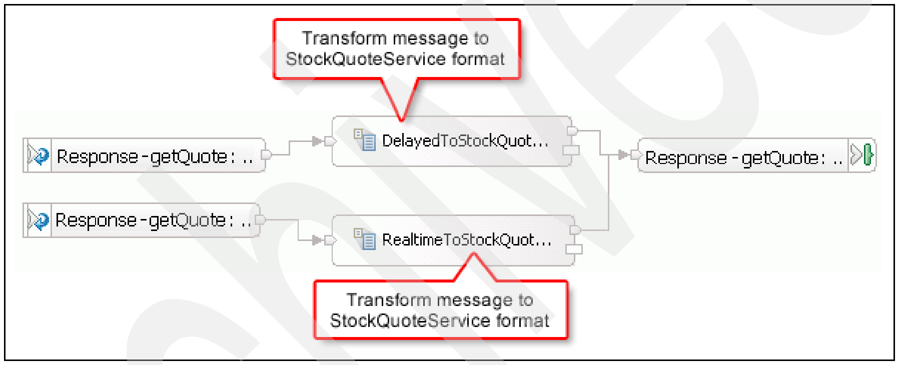


Figure 3-12 Response flow of the sample

This mediation flow built in the WebSphere Integration Developer, uses most of the primitive nodes that are currently available.

Installation considerations of the sample program

The generation of the deployment code for this application prepares an EAR file, StockQuoteApp, composed of several artifacts of servlet and Enterprise JavaBeans™ (EJB™). This application only contains the mediation elements, the driving servlet (browser) interface, and the two Web Services that are deployed in a separate J2EE application called “SOSample”, as shown in Figure 3-13.

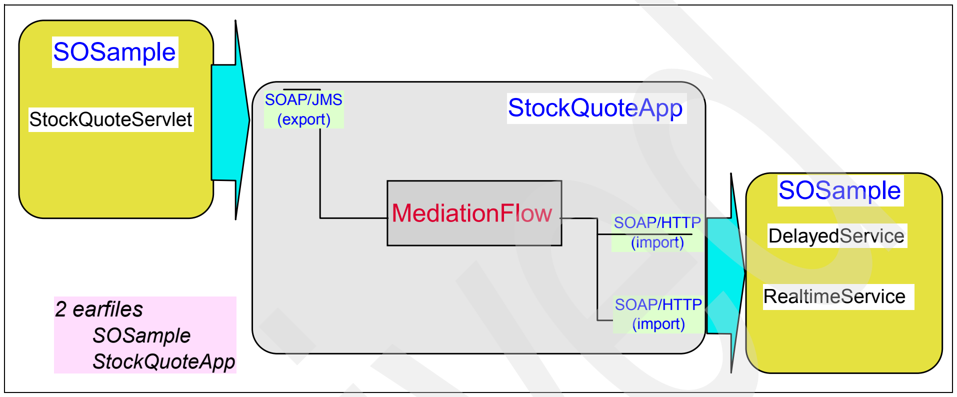


Figure 3-13 J2EE application for StockQuote and Web Service consumer/producer

The breakdown of the StockquoteApp Enterprise Application file is shown in Table 3-1.

Table 3-1 StockQuoteApp breakdown

J2EEAppl StockQuoteApp			
Servlet			
	StockQuoteService_ StockQuoteServiceJmsPort		
EJB		Type	
	MediationFlow	SLSB[1]	Implements the Meditionflow.
	Module	SLSB	Interacts with the soap/http services.
	export.StockQuoteService	SLSB	The real endpoint is implemented in this Session EJB.
	ServiceSIBusMessageBus	MDB[2]	
	MessageDriven_StockQuoteService _StockQuoteServiceJmsPort	MDB	This Message Driven Bean (MDB) receives the input from the Browser via a JMS queue(soap/jms).

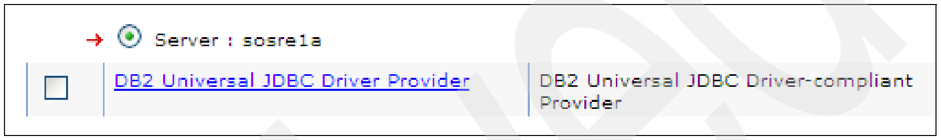
1. SLSB stands for StateLessSessionBean (a stateless session EJB).

2. MDB stands for Message Driven Bean.

To verify the set up for the sample program and to run it, perform the following steps:

1. Log in to the WebSphere Admin console.
2. The program, as delivered, was tested/assembled in a WebSphere Integration Developer environment. It uses resources (JMS queues and database) that have been defined.

   We recommend verifying the definitions of those resources. In the case of JDBC, this can easily done with the knowledge of the JNDI names, as shown in Figure 3-14.



Server : sosre1a

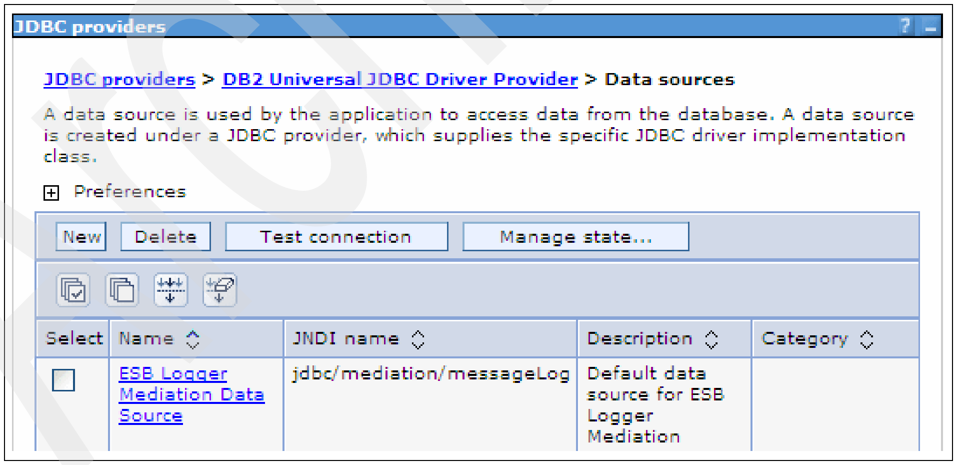
☐	DB2 Universal JDBC Driver Provider	DB2 Universal JDBC Driver-compliant Provider

Figure 3-14 Default DB2 JDBC provider

   Verify that under this DB2 provider entry, you have a definition for a message log data source, referenced by jndiName as follows:

   ```
   jdbc/mediation/messageLog
   ```

   See Figure 3-15.



JDBC providers > DB2 Universal JDBC Driver Provider > Data sources

A data source is used by the application to access data from the database. A data source is created under a JDBC provider, which supplies the specific JDBC driver implementation class.

⊞ Preferences

Select	Name	JNDI name	Description	Category
☐	ESB Logger Mediation Data Source	jdbc/mediation/messageLog	Default data source for ESB Logger Mediation	

Figure 3-15 messageLog data source

3. Test the connection by selecting the **Test Connection** function for each data source, to verify that you can access it. Inside the data source, the table

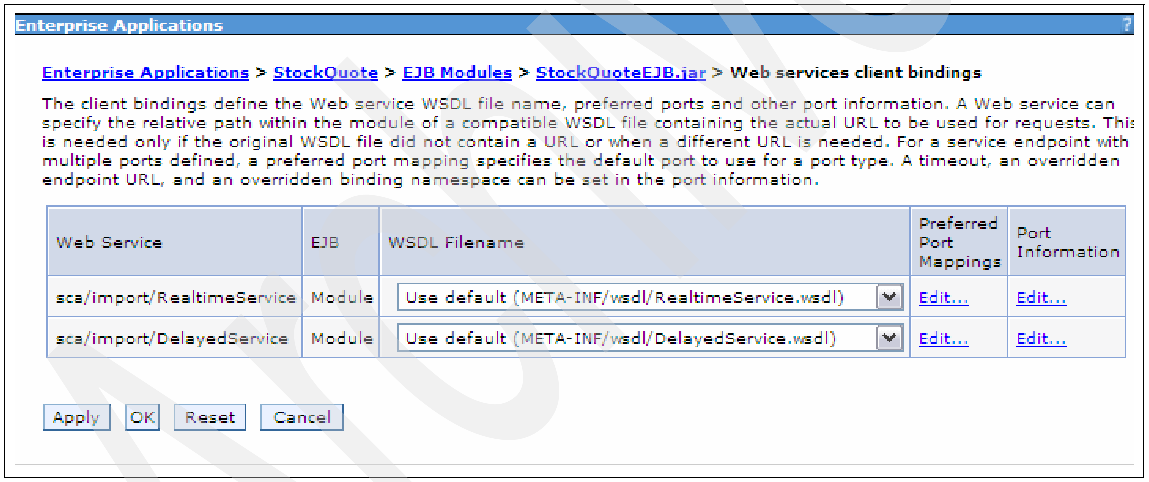
accessed by the "mediation logging" must have been defined, as described earlier.

4. Verify and match the HTTP port number. The calls to the HTTP Web Service were tested on a particular port address, which is probably different from the port address on the server on z/OS. This has to be adjusted.

   If you have to change the port number value of the HTTP protocol, perform the following steps:

   a. In the Admin console, select **Applications**.
   b. Click **StockQuote**. The details of the StockQuote application are displayed.
   c. In the Related Items section, click **EJB Modules**.
   d. Click **StockQuoteEJB.jar**. The details of the StockQuoteEJB.jar file are displayed. In the Additional Properties section, click **Web services client bindings**.

   See Figure 3-16.


Enterprise Applications
Enterprise Applications > StockQuote > EJB Modules > StockQuoteEJB.jar > Web services client bindings
Apply
OK
Reset
Cancel

The client bindings define the Web service WSDL file name, preferred ports and other port information. A Web service can specify the relative path within the module of a compatible WSDL file containing the actual URL to be used for requests. This is needed only if the original WSDL file did not contain a URL or when a different URL is needed. For a service endpoint with multiple ports defined, a preferred port mapping specifies the default port to use for a port type. A timeout, an overridden endpoint URL, and an overridden binding namespace can be set in the port information.

Web Service	EJB	WSDL Filename	Preferred Port Mappings	Port Information
sca/import/RealtimeService	Module	Use default (META-INF/wsdl/RealtimeService.wsdl)	Edit...	Edit...
sca/import/DelayedService	Module	Use default (META-INF/wsdl/DelayedService.wsdl)	Edit...	Edit...

Figure 3-16 Web services client bindings

5. For sca/import/RealtimeService, click **Edit** in the Port Information column. The port information is displayed.

   a. In the Overridden Endpoint URL field, type the following:

```
http://localhost:nnnn/RealtimeService/services/RealtimeServiceSOA
P
```

Here nnnn is the port number used for your default host. This assumes that local host is known as an alias in your installation; otherwise, this also has to be changed.

b. Click **OK**.

For sca/import/DelayedService, click **Edit** in the Port Information column. The port information is displayed.

a. In the Overidden Endpoint URL field, type the following:

```
http://localhost:nnnn/DelayedService/services/DelayedServiceSOAP
```

Here nnnn is the port number used for your default host.

b. Click **OK**.

c. Save your workspace changes to the master configuration and you can try the sample.

6. You can now run the sample program. For additional instructions on how to run the program, refer to *WebSphere Enterprise Service Bus for z/OS V6.0.1 Reference*, SC34-6768.

3.4 Installing WebSphere ESB in a ND node

We use, as a starting point, an existing ND cell. It might already have some application servers, though we are not using these existing ones for our scenario. We are going to define new empty cells/nodes and federate them in the existing cell “socell”. These new nodes will carry WebSphere ESB capabilities. The objective is depicted in Figure 3-17.

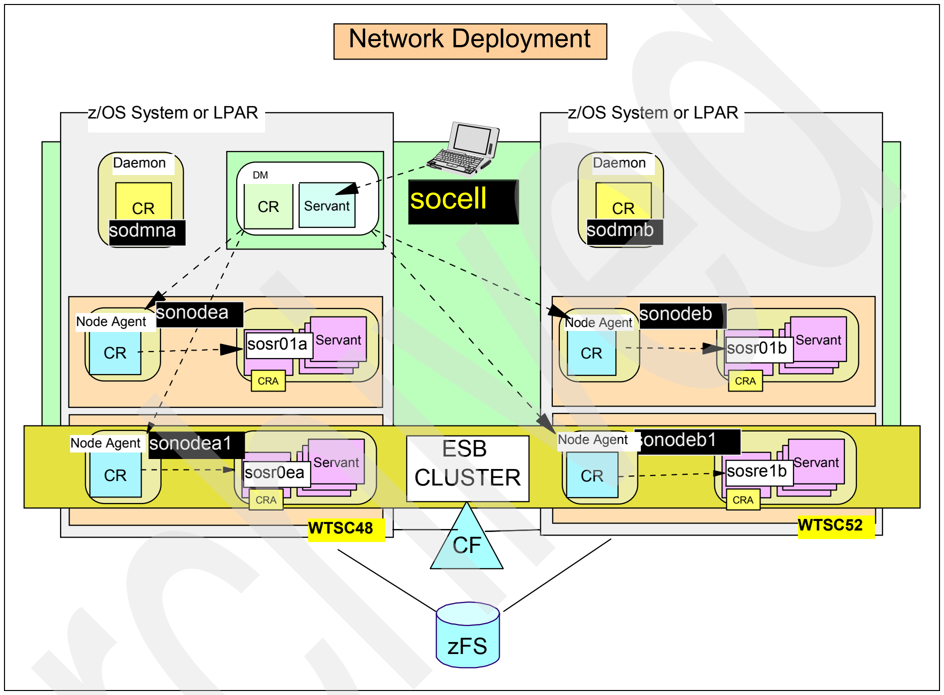


Figure 3-17 Target ND configuration with a WebSphere ESB cluster

We performed the following steps to complete this task:

1. Augment the Deployment Manager of the existing cell for WebSphere ESB.
2. Build an empty cell/node (B1) on WTSC52, with *no* federation step.
3. Augment the first empty cell/node for WebSphere ESB with the install and config scripts.
4. Federate the first (empty) cell in the existing cell.
5. Build a second empty cell/node (C1) on WTSC48 without the federation step.
6. Augment the second empty cell/node for WebSphere ESB with the install and config scripts.

7. Federate the second (empty) cell in the existing cell.
8. Create a server in the first empty node, and make it part of a cluster.
9. Clone the server, in the first node into the second empty node. As a result, you must have a cluster with two servers.

3.4.1 Installing WebSphere ESB on the existing Network Deployment

At the time of this writing, the level of WebSphere ESB (HESB601) did not support upgrading the WebSphere Application Server Application Nodes to WebSphere ESB in an existing Network Deployment (ND) server configuration. You must start out with a new ND server configuration, first augmenting the Deployment Manager node and then proceeding to federate nodes, which you previously prepared, to run WebSphere ESB.

The first element to consider in an ND configuration is the Deployment Manager. The Deployment Manager lives in its own node as a server, even though it has a special role and it is composed of a Control Region (CR) and a Servant Region (SR). In our configuration, our existing cell is called socell, as shown in Figure 3-18.

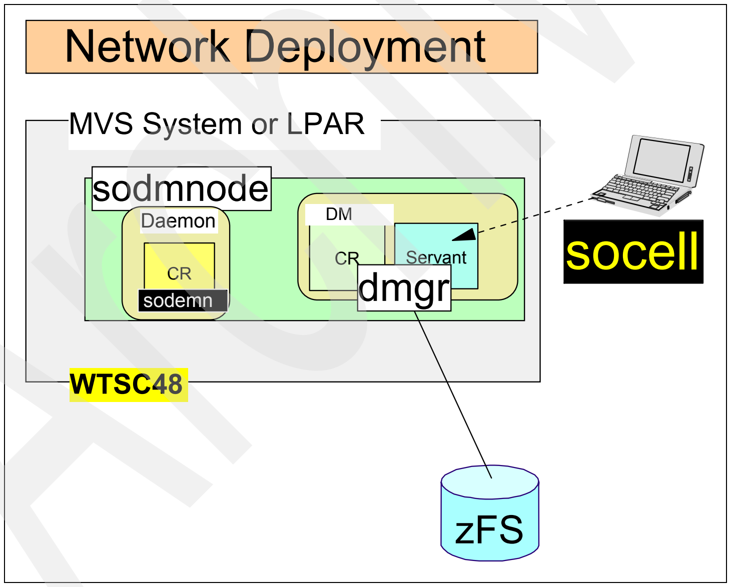


Figure 3-18 Deployment Manager node in the SOCELL

User applications are usually not deployed in this server, but it contains the "AdminConsole" application, which has to be adapted for WebSphere ESB. This has to be done before any node with WebSphere ESB capabilities can be federated into the cell.

Installing WebSphere ESB on the existing Deployment Manager (zSMPInstall)

The installation of the existing Deployment Manager must be adapted for WebSphere ESB, before any WebSphere ESB enabled nodes can be federated into the cell controlled by this Deployment Manager. It prepares the existing cell HFS file for the configuration step that follows. This step involves running the WebSphere Enterprise Service Bus for z/OS install script "zSMPInstall.sh". The commands described in this topic are contained in, and must be run from, the install HFS:

```
/usr/lpp/zWESB/V6R0/zos.config/bin
```

The shell to execute is shown in Example 3-8. Be sure to point to the installation directory of the Deployment Manager in the -runtime parameter.

Example 3-8 Shell script for zSMPInstall on Deployment Manager node

```
cd /usr/lpp/zWESBSO/V6R0/zos.config/bin
./zSMPInstall.sh -install -smproot /usr/lpp/zWESBSO/V6R0 -runtime
/wassoconfig/socell/sodmnode/DeploymentManager
```

The result of the execution creates the symbolic links on the Deployment Manager HFS and creates a post installer file, as shown in Example 3-9.

Example 3-9 Results of Deployment Manager WebSphere ESB install

```
CWPIZ0253I: parsing command arguments...
CWPIZ0254I: parsing arguments complete
CWPIZ0255I: setting up configuration...
CWPIZ0256I: set up configuration complete
CWPIZ0257I: creating the symbolic links...
CWPIZ0259I: creation of symbolic links complete
CWPIZ0260I: doing post install file updates...
CWPIZ0262I: post install updates complete
CWPIZ0263I: running Configuration Manager update...
CWPIZ0264I: Configuration Manager update complete
```

Prepare DB2 database and storage group

The database and storage group for the relational tables required by the Common Event Infrastructure have been allocated in a previous step.

Configuring WebSphere ESB on the existing Deployment Manager (zWESBConfig)

The product configuration script is run via the shell command **`zWESBConfig.sh`**. The configuration options on z/OS are driven by “response files”. The content in the response file is used to augment the WebSphere Application Server for z/OS Deployment Manager profile with WebSphere Enterprise Service Bus for z/OS configuration data. The command must be run from the /WebSphere/V6R0M0/DeploymentManager/bin directory.

This command includes the path to the response file. The property values that you set in the response file are used to create the product configuration. The response file that we use is for the “Deployment Manager”. Example 3-10 shows the shell script that we used.

Example 3-10 Deployment Manager config shell script

```
cd /wassoconfig/socell/sodmnode/DeploymentManager/bin
./zWESBConfig.sh -response /u/larryc/DmgrDB2.rsp -augment
```

During this augment, you must have the Deployment Manager running, or the step fails. The result is shown in Example 3-11.

Example 3-11 Deployment Manager config results

```
CWPIZ0253I: parsing command arguments...
CWPIZ0254I: parsing arguments complete
CWPIZ0255I: setting up configuration...
CWPIZ0256I: set up configuration complete
CWPIZ0265I: augmenting profile(s)...
INSTCONFSUCCESS: Profile augmentation succeeded.
CWPIZ0267I: augmenting profile(s) complete
```

Looking at the AdminConsole

To see all the changes in the AdminConsole, you have to recycle the Deployment Manager. The Deployment Manager is composed of the following address spaces:

- SODMGR (control region)
- SODMGRS (servant region)

After recycling the Deployment Manager, log in to the AdminConsole to verify that the Deployment Manager was correctly enabled for WebSphere ESB. The first change must be displayed in the Welcome window, where you will see the message “Welcome to WebSphere ESB”. The Deployment Manager is ready for WebSphere ESB.

In the left frame of the welcome window, under Applications, there is a new item called “SCA Modules”, as shown in Figure 3-19.

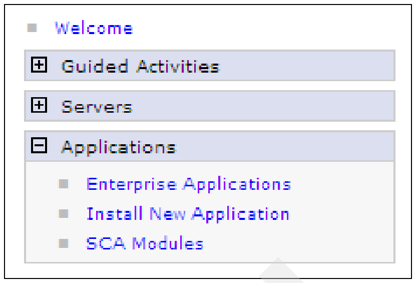


Figure 3-19 SCA option in the Welcome frame

Continue preparing two empty nodes for this cell.

Building the first empty cell/node

We are assuming that you are familiar with the building procedure for this empty node using the WebSphere Application Server V6 ISPF windows.

Note: You can use all the generated jobs with the exception of BBOWMNAN to federate the node into the Deployment Manager at this time. You federate the managed node into the Deployment Manager later, as explained in the following steps.

Figure 3-20 shows the node configuration that we built for our environment.

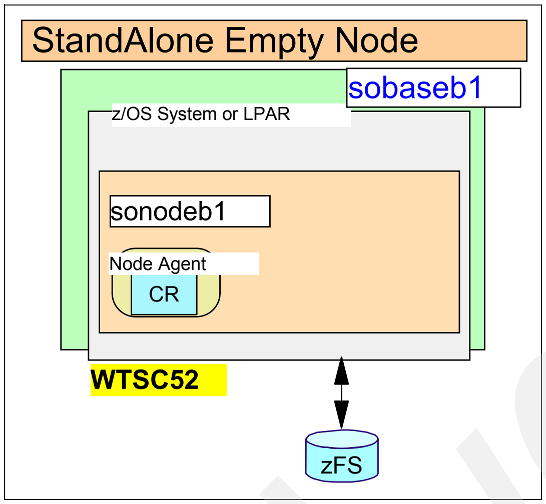


Figure 3-20 Empty node configuration

Installing WebSphere ESB on the first empty node (zSMPInstall)

First of all, this node requires access to the WebSphere ESB code. You have to execute the install step that adapts the cell/node HFS file for the configuration step. This step involves running the WebSphere Enterprise Service Bus for z/OS install script "zSMPInstall.sh", as described previously, but at the node level, as shown in Example 3-12.

Example 3-12 Shell script for zSMPInstall on the empty node

```
cd /usr/lpp/zWESBSO/V6R0/zos.config/bin
./zSMPInstall.sh -install -smproot /usr/lpp/zWESBSO/V6R0 -runtime
/wassoconfig/socell/sonodeb1/AppServer$_CBCONFIG/$_ASDIR +
```

The result of the execution is the creation of the symbolic links and the creation of the post installer file, as shown in Example 3-13.

Example 3-13 Results of the empty node WebSphere ESB install

```
CWPIZ0253I: parsing command arguments...
CWPIZ0254I: parsing arguments complete
CWPIZ0255I: setting up configuration...
CWPIZ0256I: set up configuration complete
CWPIZ0257I: creating the symbolic links...
CWPIZ0259I: creation of symbolic links complete
CWPIZ0260I: doing post install file updates...
CWPIZ0262I: post install updates complete
CWPIZ0263I: running Configuration Manager update...
CWPIZ0264I: Configuration Manager update complete
```

Configuring WebSphere ESB on the first empty node (zWESBConfig)

The configuration script is run through the shell command **`zWESBConfig.sh`**.You must use a different response file every time that you execute the script. We use the response file “ManagedDB2_b1.rsp”, as shown in Example 3-14.

Example 3-14 Shell script for zWESBConfig on the empty node

```
cd /wassoconfig/socell/sonodeb1/AppServer/bin
./zWESBConfig.sh -augment -response /u/larryc/ManagedDB2_b1.rsp
```

Verify that the result indicates a successful completion.

Building the second empty cell/node

The previous steps must be repeated for the second node, also using the WebSphere Application Server V6 ISPF panels.

Note: You can use all the generated jobs with the exception of BBOWMNAN to federate the node into the Deployment Manager at this time. You federate the managed node into the Deployment Manager later, as explained in the following steps.

Installing WebSphere ESB on the second empty node (zSMPInstall)

The node requires access to the WebSphere ESB code. This step involves running the WebSphere Enterprise Service Bus for z/OS install script “zSMPInstall.sh” targeting this node, as shown in Example 3-15.

Example 3-15 Shell script for zSMPInstall on empty node

```
cd /usr/lpp/zWESBSO/V6R0/zos.config/bin
./zSMPInstall.sh -install -smproot /usr/lpp/zWESBSO/V6R0 -runtime
/wassoconfig/socell/sonodec1/AppServer$_CBCONFIG/$_ASDIR +
```

Verify that the result indicates a successful completion.

Configuring WebSphere ESB on the second empty node (zWESBConfig)

The configuration script is run through the shell command **`zWESBConfig.sh`**. In our scenario, we use the adapted response file “ManagedDB2_c1.rsp”, as shown in Example 3-16.

Example 3-16 Shell script for zWESBConfig on empty node

```
cd /wassoconfig/socell/sonodec1/AppServer/bin
./zWESBConfig.sh -augment -response /u/larryc/ManagedDB2_c1.rsp
```

Verify that the result indicates a successful completion.

Federating the nodes into the cell

After we completed the previous steps, we can execute the steps “BBOWMNAN” for both empty nodes to federate them in the existing cell. Figure 3-21 shows the federation process. Federation has to be done under the user ID of the WebSphere Administrator, in our case, “WSADMIN”. Note that the federation from the node on wtsc52 is cross-LPAR.

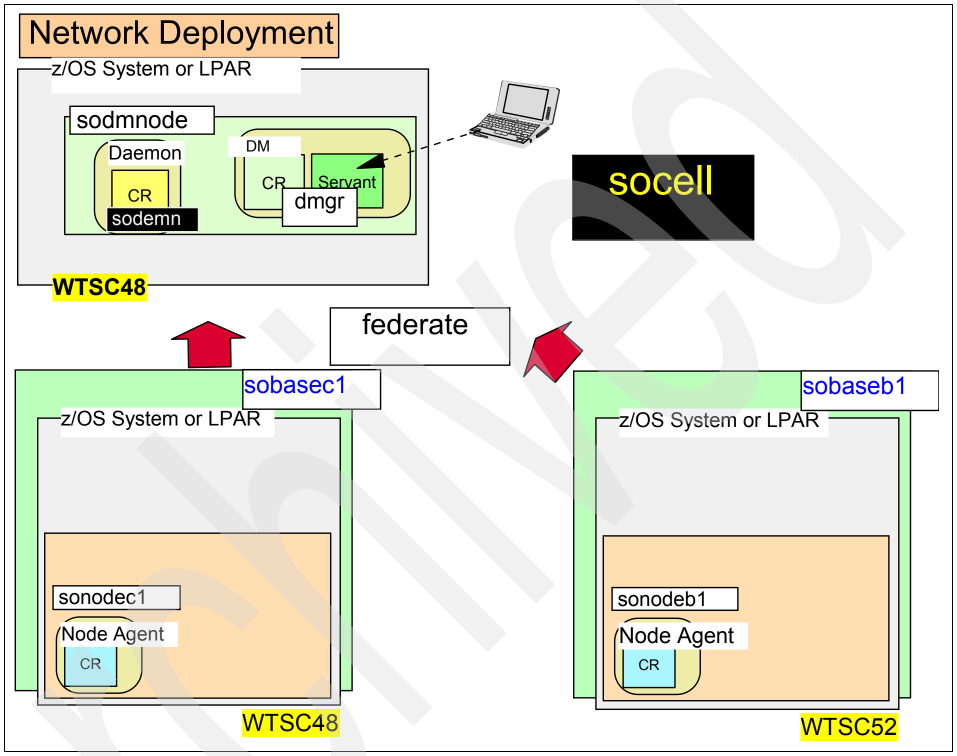


Figure 3-21 Federation of the empty nodes

During the building of the empty nodes, a temporary cell name was assigned. This name disappears at the end of the process and the cell name of the existing ND cell (socell) becomes the overall cell name.

During the federation phase of each node, the function of NodeAgent is activated and runs in an additional address space.

After the process completes, the following address spaces represent the configuration:

- SODMGR: Control region Deployment Manager
- SODMGRS: Servant region Deployment Manager
- SODEMN: One daemon on each LPAR (WTSC48 and WTSC52)
- SOAGNC1: NodeAgent for node on WTSC48
- SOAGNB1: NodeAgent for node on WTSC52

Figure 3-22 shows the socell address spaces with empty nodes.

```
SODMGR   SODMGR   BBOCTL   STC25133 ASCR1
SODEMN   SODEMN   BBODAEMN STC25134 WSDMNCR1
SODMGRS  SODMGRS  BBOSR    STC25135 ASSR1
SOAGNC1  SOAGNC1  BBOCTL   STC25147 ASCR1
SOAGNB1  SOAGNB1  BBOCTL   STC25137 ASCR1
SODEMN   SODEMN   BBODAEMN STC25138 WSDMNCR1
```

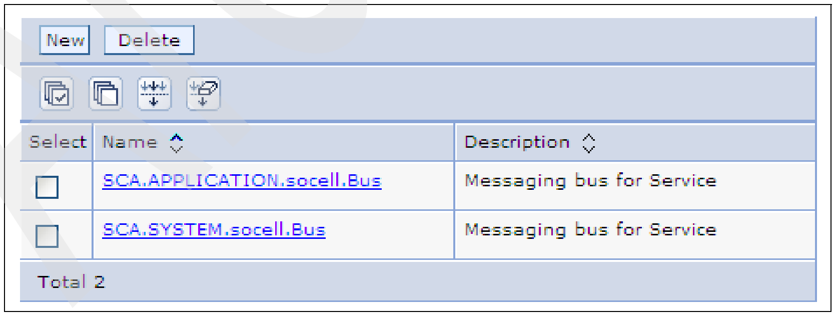
Figure 3-22 Socell address spaces with empty nodes

A daemon called “Location Services Director” must also be available on each LPAR where nodes belonging to this cell are active.

Looking at the AdminConsole again

You must STOP/START the Deployment Manager before all new changes can be seen. When you log in to the AdminConsole, you must see the message Welcome to WebSphere ESB in the Welcome window.

You must also notice that two *SIBuses* are available, currently without members because no servers are defined yet, as shown in Figure 3-23.



Select	Name	Description
☐	SCA.APPLICATION.socell.Bus	Messaging bus for Service
☐	SCA.SYSTEM.socell.Bus	Messaging bus for Service

Total 2

Figure 3-23 Default installed buses

3.4.2 Building the cluster

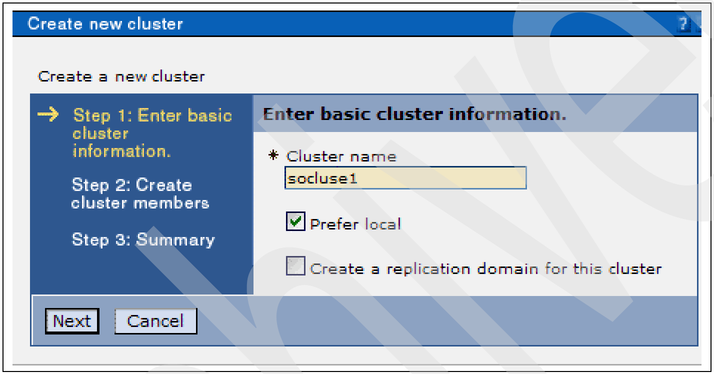
The cluster is composed of two servers, one in each node. Remember a cluster is composed of cloned servers, not by joint existing servers.

Defining the cluster and the first server

Start defining the first server, which is going to be the source for the cluster cloning.

Perform the following steps:

1. When you start the wizard on the AdminConsole, you immediately have to provide a *clusterName*, as shown in Figure 3-24.


Create new cluster

Create a new cluster

→ Step 1: Enter basic cluster information.

Step 2: Create cluster members

Step 3: Summary

Enter basic cluster information.

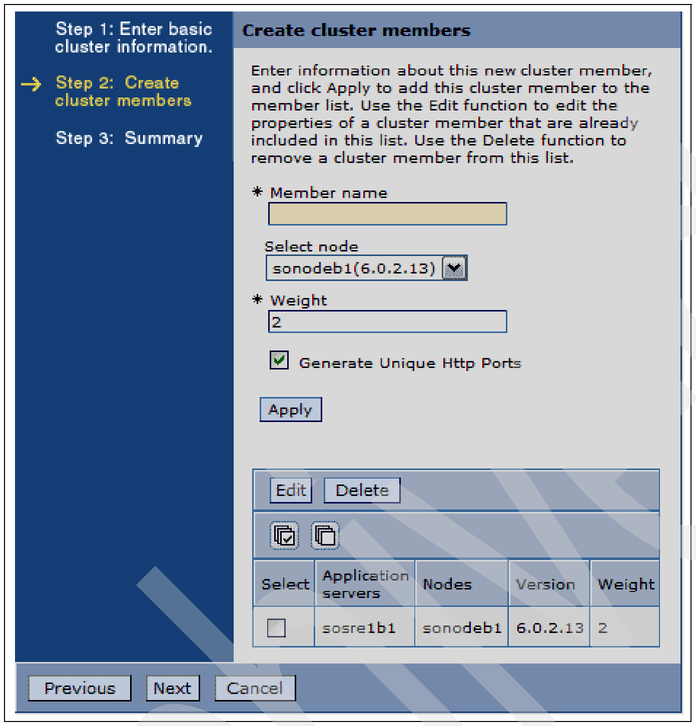
* Cluster name
socluse1

[✓] Prefer local

[ ] Create a replication domain for this cluster

Figure 3-24 Building the cluster

2. In the second step of the dialog, nominate the first server that is a member of this cluster, as shown in Figure 3-25.


Step 1: Enter basic cluster information.

→ Step 2: Create cluster members

Step 3: Summary

Create cluster members

Enter information about this new cluster member, and click Apply to add this cluster member to the member list. Use the Edit function to edit the properties of a cluster member that are already included in this list. Use the Delete function to remove a cluster member from this list.

* Member name:

Select node: sonodeb1(6.0.2.13)

* Weight: 2

✓ Generate Unique Http Ports

Apply

Edit | Delete

Select	Application servers	Nodes	Version	Weight
☐	sosre1b1	sonodeb1	6.0.2.13	2

Previous | Next | Cancel

Figure 3-25 Adding the first member to the new cluster

The first server is the origin of the cluster. All other servers in this cluster are going to be clones of this first server. The Admin Dialog is asking for additional input, as shown in Figure 3-25:

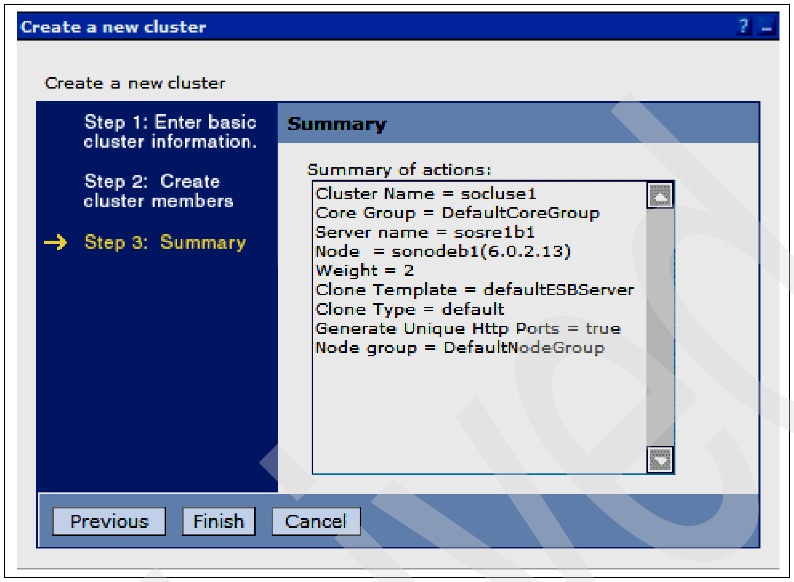
– The name of the server.

– The name of an existing node where to build.

– The type of server (normal WebSphere Application Server or WebSphere ESB). We specified WebSphere ESB type for this choice.

Click **Apply** after each selection to make sure your definition is taken.

3. At the end of the wizard, you will see a Summary that confirms the names and the options that have been taken, as shown in Figure 3-26.



Summary of actions:

```
Cluster Name = socluse1
Core Group = DefaultCoreGroup
Server name = sosre1b1
Node  = sonodeb1(6.0.2.13)
Weight = 2
Clone Template = defaultESBServer
Clone Type = default
Generate Unique Http Ports = true
Node group = DefaultNodeGroup
```

Figure 3-26 Summary for cluster creation

After this, the server is created.

We performed the following two additional actions:

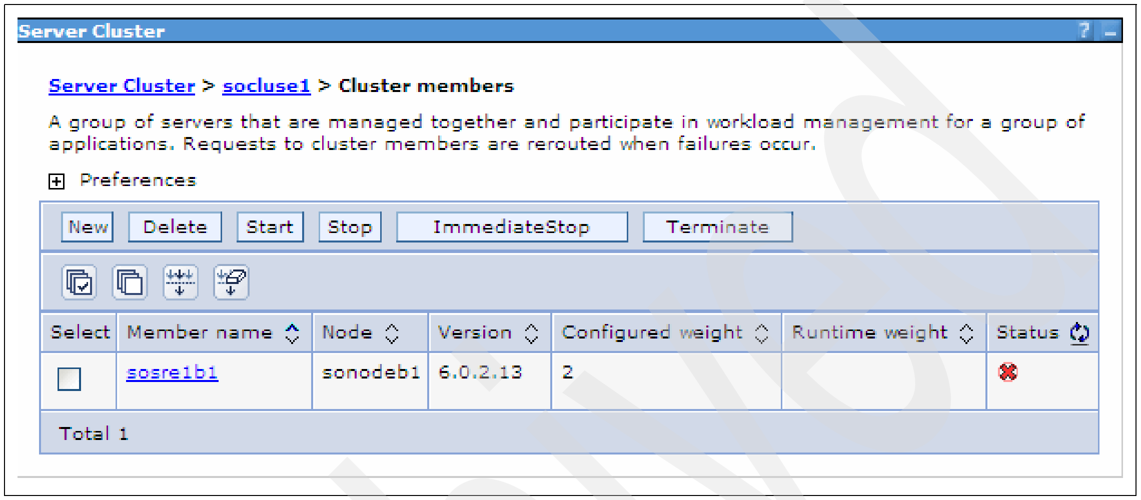
- We set the SHORTNAME from BBOS001 to our name.
- With the wsadmin function and a python script, we set the endpoints (TCP/IP) ports to our standards. For details about the script, refer to Appendix A, "WebSphere ESB related scripts" on page 303. This is also required to avoid "TCP/IP port" collisions.

Defining the second server

The next step consists of adding a second server to the cluster. This server is a clone of the first one and by definition has the same functionality. It also embraces the same "endpoint" port numbers similar to the first server.

Perform the following steps:

1. In the left frame of the AdminConsole Welcome, select **Servers** → **Clusters** and then click the option. In the right frame, click the existing cluster and under Additional Properties, click **ClusterMembers**. Figure 3-27 shows the displayed window.



A group of servers that are managed together and participate in workload management for a group of applications. Requests to cluster members are rerouted when failures occur.

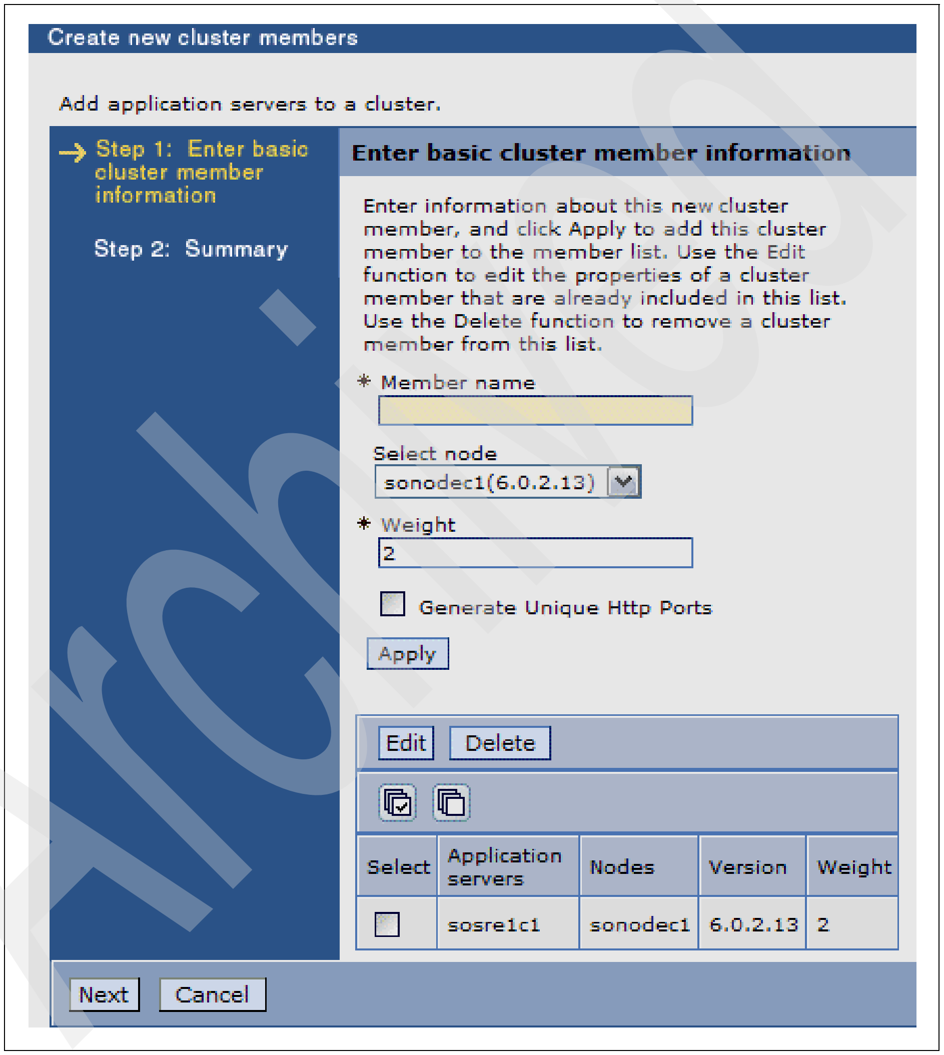
Select	Member name	Node	Version	Configured weight	Runtime weight	Status
☐	sosre1b1	sonodeb1	6.0.2.13	2		✖
Total 1						

Figure 3-27 Adding a new server to the cluster

2. In the window shown in Figure 3-27 on page 117, select the option **New**, which displays the window shown in Figure 3-28.

   In this window, specify the Member name for the new server and specify, in the Select node field, the node where you want the new cloned server to be built.

   Click **Apply** and **Next**.


Create new cluster members

Add application servers to a cluster.

Step 1: Enter basic cluster member information

Step 2: Summary

Enter basic cluster member information

Enter information about this new cluster member, and click Apply to add this cluster member to the member list. Use the Edit function to edit the properties of a cluster member that are already included in this list. Use the Delete function to remove a cluster member from this list.

* Member name

Select node
sonodec1(6.0.2.13)

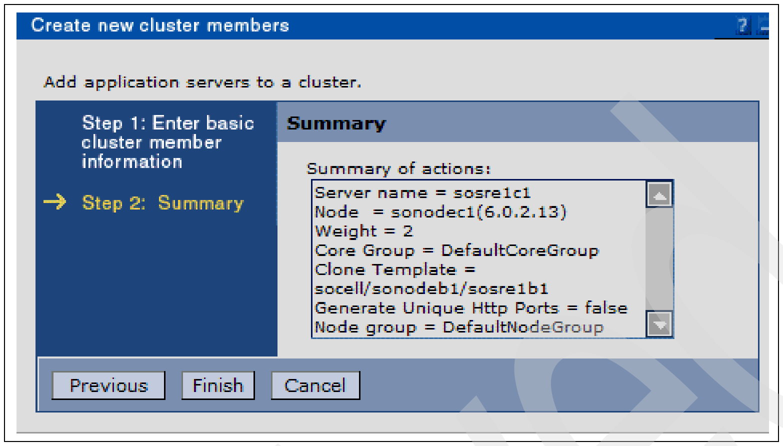
* Weight
2

Generate Unique Http Ports

Select	Application servers	Nodes	Version	Weight
☐	sosre1c1	sonodec1	6.0.2.13	2

Figure 3-28 Adding the second server

3. At the end of the wizard, the Summary confirms the names and the options that have been taken, as shown in Figure 3-29.


Create new cluster members
Add application servers to a cluster.
Step 1: Enter basic cluster member information
Step 2: Summary
Summary
Previous
Finish
Cancel

Summary of actions:

```
Server name = sosre1c1
Node  = sonodec1(6.0.2.13)
Weight = 2
Core Group = DefaultCoreGroup
Clone Template =
socell/sonodeb1/sosre1b1
Generate Unique Http Ports = false
Node group = DefaultNodeGroup
```

Figure 3-29 Summary for adding a clone

After the member is added, we changed the “SHORTNAME” to our name.

3.4.3 Preparing the Messaging Infrastructure and the SIB

The following list summarizes the list of tasks that have to be completed:

- A Service Integration Bus (SIB) has to be created. This must have already been accomplished by the installation script.
- Back ends for queues and topics have to be chosen and eventually allocated.
- Optional: Additional Message Engines (MEs) must be added to the cluster.
- Members have to be added to the SIB (Cluster and Server).
- A High Availability policy for the configuration has to be installed or activated.

Back-end considerations

Adding members to the SIB bus creates the Message Engines. In our cluster, a Message Engine is going to be created for each of the two members of the cluster. The Message Engines that contain the logic to interface with the Service Integration Bus and handle the message traffic in the bus require space to harden the queue contents.

You can choose where you would prefer this space to be allocated between the following three different options:

- DB2
- Cloudscape (This is the default option, and will not be supported in an ND configuration, especially with a cluster.)
- Flat files (only available with V6.1)

We chose DB2 because our requirements are high availability configuration and workload distribution. The partitioning of the hardened information (tables containing the queue contents) is based on the schema (table qualifier), which is indicated when we add a member to the SIB or when we create additional Message Engines.

We use three schemas as follows:

- SOME1CLA (ME1 on the Application Bus)
- SOME2CLA (ME2 on the Application Bus)
- SOMECLS (ME1 on the System Bus)

Defining the DB2 elements can be done remotely via DB2Connect or via DB2 Spufi using the Data Definition Language (DDL) statements. To obtain the DDL, you can use the following shell script:

```
sibDDLGenerator.sh
```

This script requires several parameters, some mandatory and some optional. A sample is shown in Example 3-17. We used this script to prepare the DDL statements for the Message Engine tables for all three schema's.

Example 3-17 sibDDLGenerator shell script info

```
sibDDLGenerator -system <DBMS name>
        -version <DBMS version>
        -platform <DBMS platform>
        -schema <schema>
        -user <user>¨
        -create|-drop
        -database <database> (z/OS only)
        -storagegroup <storage group> (z/OS only)
        -bufferpool <buffer pool> (z/OS only)
        -statementend <terminator>
        -noblanklines
        -nolinebreaks
        -firstline <statements>
        -lastline <statements>
        -permanent <number of permanent tables>
        -temporary <number of temporary tables>
```

Example 3-18 is a sample of the script execution. The output file is sibme1.dll.

Example 3-18 sibDDLGenerator shell script

```
cd /wassoconfig/socell/sodmnode/DeploymentManager/bin
  ./sibDDLGenerator.sh -system db2 -version 8.1 -platform zos
   -schema SOME1CLA -database ITSOESB1 -storagegroup ITSOEST1
   -statementend : -user ASSR1 -bufferpool BP0 -create
   > /u/xxxxxx/sh/ddl/sibme1.ddl
```

The parameter parsing by the shell script is quite strict. Check carefully for errors if you do achieve a successful result. The result is in EBCDIC that you can use directly as input to the DB2 Spufi.

We had to slightly adapt the DDL file to our installation and then execute the DDLs from Spufi. We repeated the script for the other two schemas and, as a result, we obtained three distinct sets of tables with indexes, that the Message Engines can use. The link between a set of tables and the Message Engines is the jndiName and schema, where:

- The jndiName points to the DB2 subsystem.
- The schema points to the tables belonging to the Message Engine.

Additional DB2 requirements

There is also an additional requirement for a TEMP database with an embedded TABLESPACE (see Example 3-19). If this database is *not* available, or if the buffer pool size is not adequate, you might get a -904 SQLCode as a result and the message engines are not going to start.

Example 3-19 TEMP database definitions

```
CREATE DATABASE TEMPD8I1 AS TEMP FOR D8I1 BUFFERPOOL BP32K;
CREATE TABLESPACE TEMP IN TEMPD8I1 BUFFERPOOL BP32K SEGSIZE 4;
CREATE DATABASE TEMPD8I2 AS TEMP FOR D8I2 BUFFERPOOL BP32K;
CREATE TABLESPACE TEMP IN TEMPD8I2 BUFFERPOOL BP32K SEGSIZE 4;
```

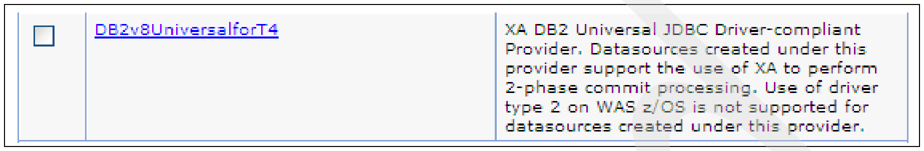
Resource definition of the Message Engine DataSource

Because we are targeting a high availability configuration, with failover capability where each cluster member is a backup for the other, the Message Engines in both members must have access to the database/tables that backs the message queues. In “Back-end considerations” on page 119, we defined the tables in the DB2 system. A resource definition means that we make the DB2 system accessible by a jndiName. This has to be done at the cluster level, or individually for each server part of the cluster. It can be done by the AdminConsole or with a Wsadmin script.

Access to DB2 from WebSphere Application Server is done through JDBC. Before you can use JDBC, you must define the provider, which gives you the resource adapter to the data and the drivers required for the access. After you have a provider with the adequate access software, you can define a supported DataSource with its specifics. We are using a DB2 V8 subsystem. The references to the DB2 libraries delivered by the provider are, in general, symbolic parameters. Access to DB2 requires some authorization, which is expressed by a *Java Authentication and Authorization System* (JAAS) alias. This alias, composed of a user ID / password, must be provided, if not available yet.

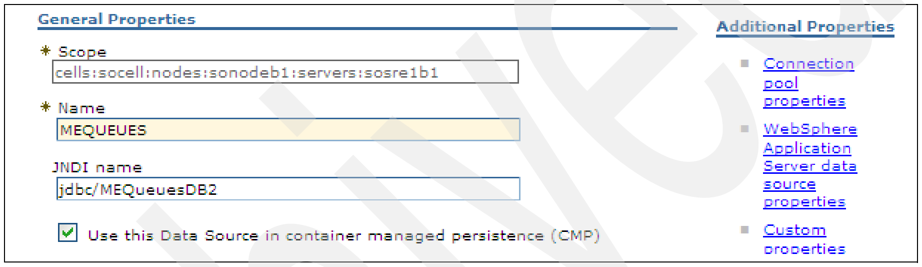
We used a wsadmin script that performs all the following steps listed:

- The creation of a JAASAlias to use with the DataSources.
- The definition of a JDBC Provider for JDBC: We selected the DB2 Universal XA driver, as shown in Figure 3-30, which has support for the Type 4 (via TCP/IP) driver, as shown in Figure 3-31 and Figure 3-32.



☐	DB2v8UniversalforT4	XA DB2 Universal JDBC Driver-compliant Provider. Datasources created under this provider support the use of XA to perform 2-phase commit processing. Use of driver type 2 on WAS z/OS is not supported for datasources created under this provider.

Figure 3-30 JDBC Provider XA



General Properties

* Scope
cells:socell:nodes:sonodeb1:servers:sosre1b1

* Name
MEQUEUES

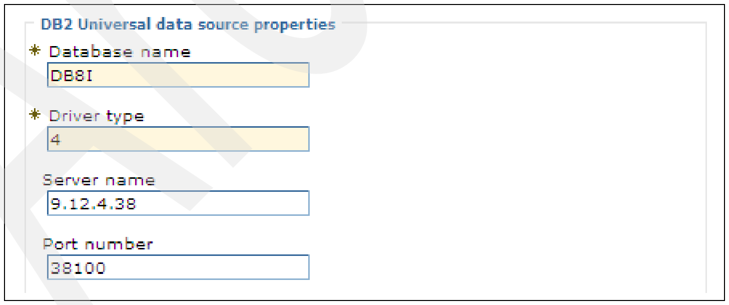
JNDI name
jdbc/MEQueuesDB2

✓ Use this Data Source in container managed persistence (CMP)

Additional Properties
- Connection pool properties
- WebSphere Application Server data source properties
- Custom properties

Figure 3-31 DataSource name and jndiName

- The setup of the environment variables, used by the provider in the paths pointing to the JDBC libraries.
- The definition of a DataSource of type 4 in both cluster members to access the MEQueue tab of both members, as shown in Figure 3-32.



DB2 Universal data source properties

* Database name
DB8I

* Driver type
4

Server name
9.12.4.38

Port number
38100

Figure 3-32 Type 4 datasource, with TCP/IP addresses

Adding members to the SIB

Currently, the following two buses are defined by the WebSphere ESB customization:

- SCA.APPLICATION.socell.Bus
- SCA.SYSTEM.socell.Bus

They have no members connected yet. You are going to connect the existing cluster to both buses using the **`AdminTask configSCAForCluster`** command. This is a new WebSphere ESB command and it is executed with Wsadmin scripting, as shown in Example 3-20.

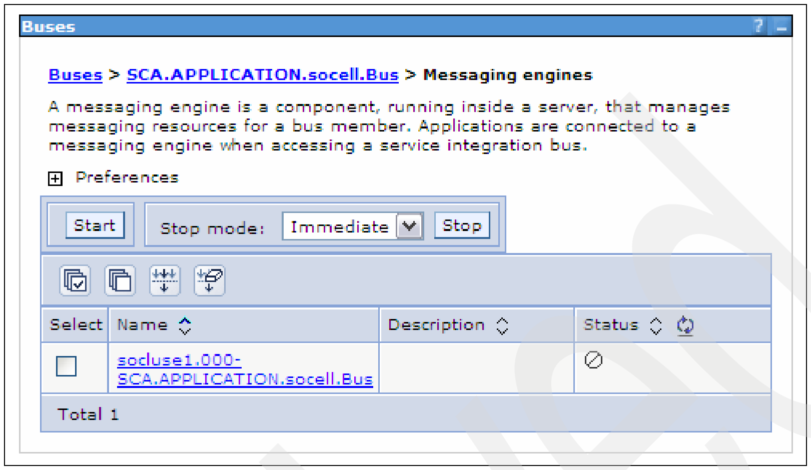
Example 3-20 configSCAForCluster command shell script

```
cd /wassoconfig/socell/soddened/DeploymentManager/bin/
./wsadmin.sh -conntype SOAP -host 127.0.0.1 -port 29510
   -c '$AdminTask configSCAForCluster
   { -clusterName socluse1  -meAuthAlias SOCELL
   -appBusDataSource jdbc/MEQueuesDB2 -appBusSchemaName  SOME1CLA
   -systemBusDataSource jdbc/MEQueuesDB2 -systemBusSchemaName SOMECLS
   -createTables false } '
```

This command, shown in Example 3-20, performs the following functions:

- Adds the Cluster to both buses.
- Sets the “jndiName” reference and the “schema” for the connection to the MEQueue tables.

These functions can be verified with the AdminConsole. Figure 3-33 shows an example for the bus SCA.APPLICATION.socell.Bus.



Buses

Buses > SCA.APPLICATION.socell.Bus > Messaging engines

A messaging engine is a component, running inside a server, that manages messaging resources for a bus member. Applications are connected to a messaging engine when accessing a service integration bus.

⊞ Preferences

Start | Stop mode: Immediate | Stop

Select	Name	Description	Status
☐	socluse1.000-SCA.APPLICATION.socell.Bus		⊘
Total 1			

Figure 3-33 Messaging Engines on bus

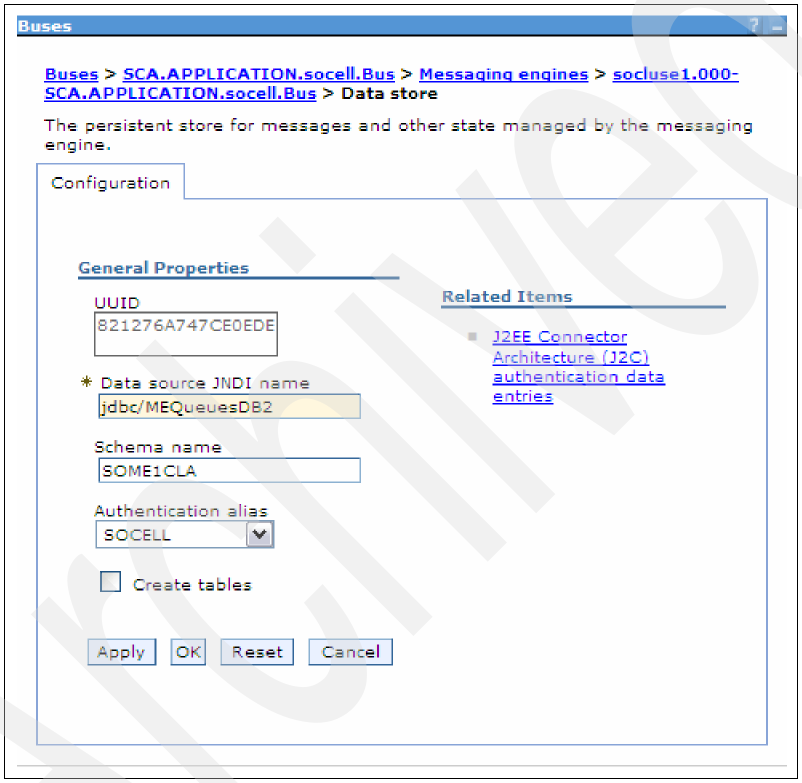
You can see that the Message Engine socluse1.000-SCA.APPLICATION.socell.Bus has been added to this bus.

If you click it, you can verify the other properties. One of the important parameter is the DataStore, as shown in Figure 3-34.

Verify the following definitions:

- DataSource jndiName
- SchemaName
- Authentication Alias

You can verify the same information for the other bus.



Buses > SCA.APPLICATION.socell.Bus > Messaging engines > socluse1.000-SCA.APPLICATION.socell.Bus > Data store

The persistent store for messages and other state managed by the messaging engine.

UUID: 821276A747CE0EDE

* Data source JNDI name: jdbc/MEQueuesDB2

Schema name: SOME1CLA

Authentication alias: SOCELL

Create tables: ☐

Apply
OK
Reset
Cancel
Related Items
J2EE Connector Architecture (J2C) authentication data entries

Figure 3-34 Datastore parameters for Message Engine of the cluster

Adding Message Engines

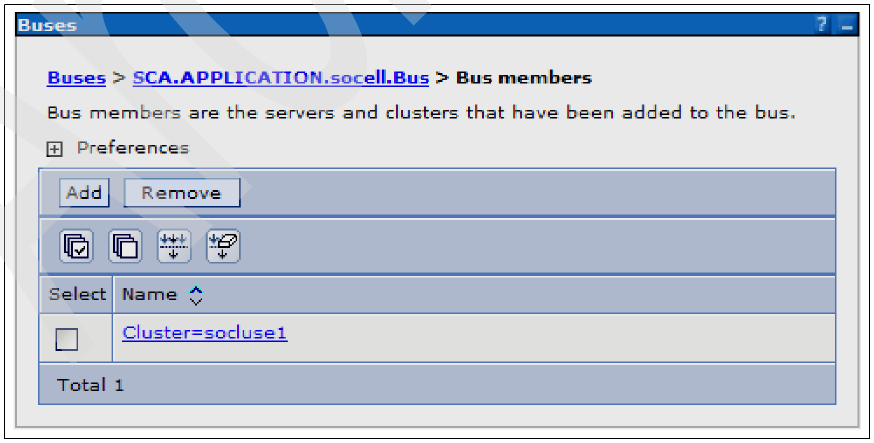
We currently have four Message Engines (two for the members of the cluster on each Bus).

The Message Engine is the point where the queues with the messages are hardened in DB2. Messages are retrieved via Message Driven Beans (MDBs), which have defined an “ActivationSpec” referencing the default Message Provider and a particular queue. In a clustered configuration, the MDBs are deployed on all members of the clusters, and MDBs could be triggered in all members. This means, in theory, the Message Engines in the cluster members are linked to the same queues. In reality, only one Message Engine can be active in a cluster; the other is in standby. From the point of view of reliability, this addresses the availability, and even though you have a message processing capacity limited to one application server, this configuration guarantees the sequence of the messages, and that everything is handled by only one application member of the cluster.

If the sequence of the messages and the affinity to a cluster member is not required, you can introduce *WorkLoad Balancing*, which means that the work must be balanced over all application servers that are members of the cluster. This can be obtained by using partitioned queues tables, which implies a second set of tables, addressed via a second active Message Engine on the bus, with a different table schema. We created this second Message Engine on the SCA.APPLICATION,socell.Bus.

To create a second Message Engine, perform the following steps:

1. Navigate to the topology for this bus, and click the **Bus members** option, as shown in Figure 3-35.



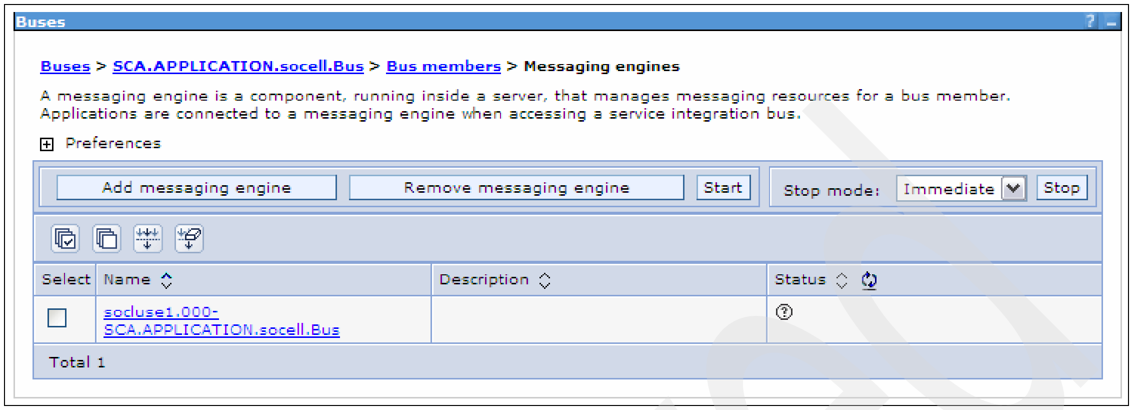
Buses > SCA.APPLICATION.socell.Bus > Bus members

Bus members are the servers and clusters that have been added to the bus.

Select	Name
☐	Cluster=socluse1
Total 1	

Figure 3-35 Members under the bus topology

2. Click the cluster where you want an additional Message Engine, as shown in Figure 3-36.


Buses

Buses > SCA.APPLICATION.socell.Bus > Bus members > Messaging engines

A messaging engine is a component, running inside a server, that manages messaging resources for a bus member. Applications are connected to a messaging engine when accessing a service integration bus.

⊞ Preferences

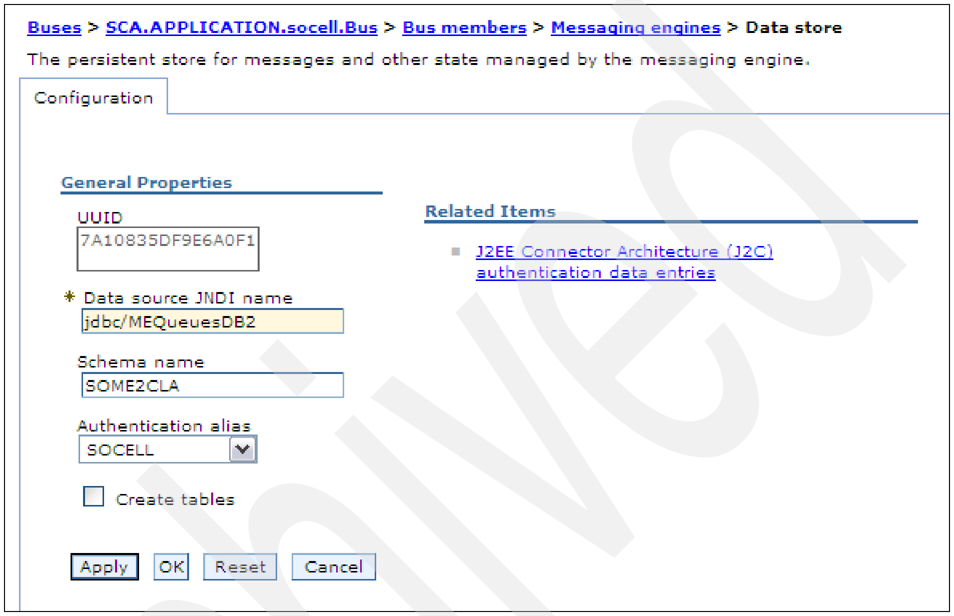
Select	Name	Description	Status
☐	socluse1.000-SCA.APPLICATION.socell.Bus		?

Total 1

Figure 3-36   Add a Message Engine

3. Click **Add messaging engine**.

4. The next window that opens is shown in Figure 3-37. Specify the following:
   – DataSource JNDIname (might be the same)
   – Schema name (has to be different from ME1 and the tables must exist)
   – Authentication Alias (might be the same)



Buses > SCA.APPLICATION.socell.Bus > Bus members > Messaging engines > Data store

The persistent store for messages and other state managed by the messaging engine.

Configuration

General Properties

UUID: 7A10835DF9E6A0F1

* Data source JNDI name: jdbc/MEQueuesDB2

Schema name: SOME2CLA

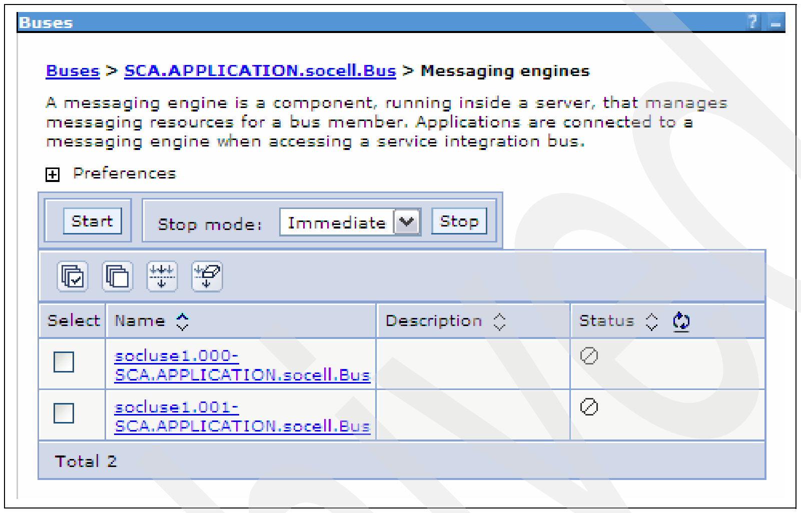
Authentication alias: SOCELL

☐ Create tables

Apply
OK
Reset
Cancel
Related Items
J2EE Connector Architecture (J2C) authentication data entries

Figure 3-37 Specifying the properties for the Message Engine

5. Finally, you obtain the following configuration:
   - SCA.APPLICATION.socell.Bus has two Message Engines on each member, as shown in Figure 3-38.
   - SCA.SYSTEM.socell.Bus has only one Message Engine on each member and can be verified with the AdminConsole.



Buses > SCA.APPLICATION.socell.Bus > Messaging engines

A messaging engine is a component, running inside a server, that manages messaging resources for a bus member. Applications are connected to a messaging engine when accessing a service integration bus.

Select	Name	Description	Status
☐	socluse1.000-SCA.APPLICATION.socell.Bus		⊘
☐	socluse1.001-SCA.APPLICATION.socell.Bus		⊘
Total 2			

Figure 3-38 Two Message Engines on the same Cluster/Bus

The current configuration is shown in Figure 3-39.

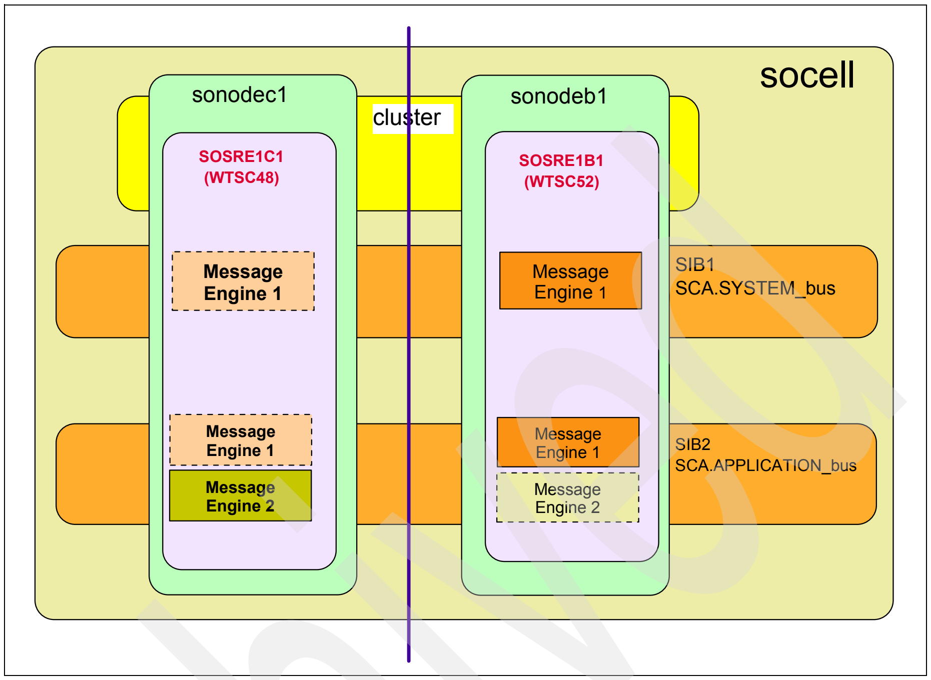


Figure 3-39 Buses with cluster and Message Engines

Specifying a High Availability policy

The WebSphere Application Server high availability framework eliminates single points of failure and provides peer to peer failover for applications and processes running within WebSphere. The configuration of service integration is very flexible. You can have individual messaging engines, or clusters containing multiple messaging engines, that can share workload, be highly available, or both.

After you have created a bus member, you can configure a policy to control the availability behavior of the messaging engine for that bus member. If you want high availability, you must use a cluster bus member and the default policy (One of N), which allows the messaging engine to fail over. You can customize your policy to specify other availability behavior, such as a preference for particular servers.

High Availability messaging engines with workload sharing configuration

The configuration that we have been building consists of two messaging engines on the SCA.APPLICATION.socell.Bus running in the cluster, with each messaging engine able to fail over to one alternative server.

This configuration has been achieved by adding a cluster to a service integration bus. This automatically created one messaging engine to which we then added, to the cluster, an additional messaging engine. The default policy, One of N, allows the messaging engines to fail over between servers in the cluster, making the messaging engines highly available. You can, optionally, give each messaging engine a preference for one or more servers, by creating a specific policy. You can further alter the policies to control the availability characteristics of each messaging engine.

Each server in the cluster contains the definition of each messaging engine that can run on it, and it creates an instance of the messaging engine so that the instance is ready to be activated if another server fails.

With the One of N policy, each messaging engine can be active in only one server at a time, with the other servers acting as standby servers for that messaging engine. The data store for the messaging engine must be accessible by all the servers in the cluster. The means of achieving this option is dependent on the data store topology used; one way to achieve it is by using type 4 connectors.

You can specify one or more preferred servers for each messaging engine, as mentioned previously. Whenever a preferred server is available, the High Availability Manager runs the messaging engine in it. When no preferred server is available, the messaging engine runs in an alternative server.

When a preferred server once again becomes available, the High Availability Manager moves the messaging engine back to it if, and only if, the failback option is set on the relevant policy.

One of N policy for service integration

This is the policy that you are going to apply to the Message Engines of the cluster. Four variants are available:

- With no preferred servers

  The messaging engine can fail over to any of the other servers in the cluster, and has no preference for any particular server.

- With preferred servers

  The messaging engine can fail over to any of the other servers in the cluster, but runs on the most preferred server that is available at the time the messaging engine is activated. You can add more than one preferred server if required. Placing a server earlier in the list indicates a stronger preference for that server. The messaging engine is not moved automatically to a server that appears earlier in the list if that server becomes available.

- With preferred servers and the Failback setting

  The messaging engine can fail over to any of the other servers in the cluster, but it always runs on the most preferred server that is available. You can add more than one preferred server if required. Placing a server earlier in the list indicates a stronger preference for that server. The messaging engine is moved automatically to a server that appears earlier in the list, if that server becomes available.

- With preferred servers and the Preferred servers only setting

  The messaging engine can fail over to any of the preferred servers in the cluster and cannot run on a server that is not in the set of preferred servers. You can add more than one preferred server if required. Placing a server earlier in the list indicates a stronger preference for that server. You can use the Failback setting in conjunction with Preferred servers only. This causes the messaging engine to be moved automatically to a server that appears earlier in the list, if that server becomes available.

Because we have only two servers in the cluster, the subpolicies *with preferred servers* and *with preferred servers and the Failback setting* has the same effect.

Configuring a policy for messaging engines

Figure 3-40 shows the steps that have to be performed to build policies for the Message Engine sets.

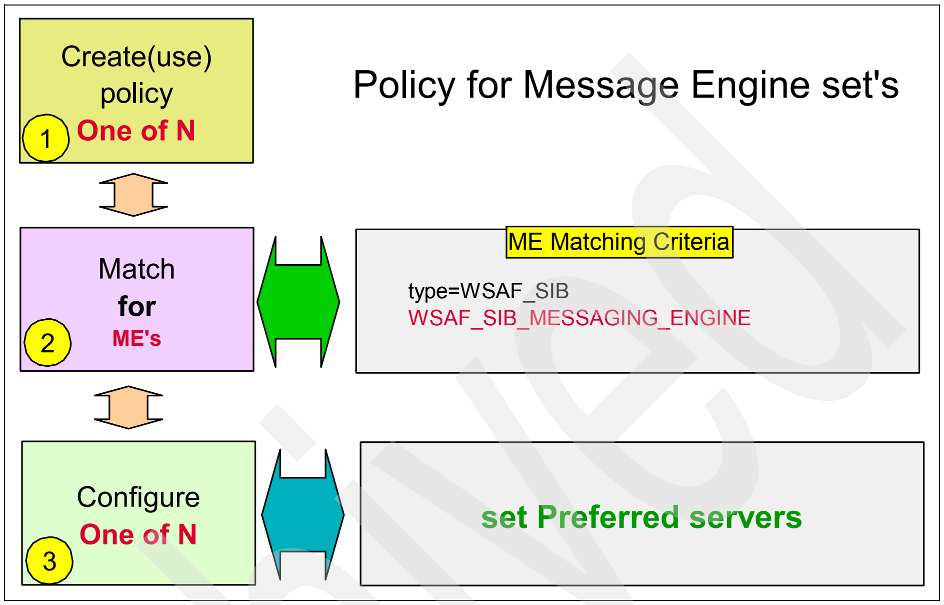


Figure 3-40 Defining the policies

We define two new policies, ITSOHA1 and ITSOHA2, both with type One of N. To define each policy, you have to perform the following three steps:

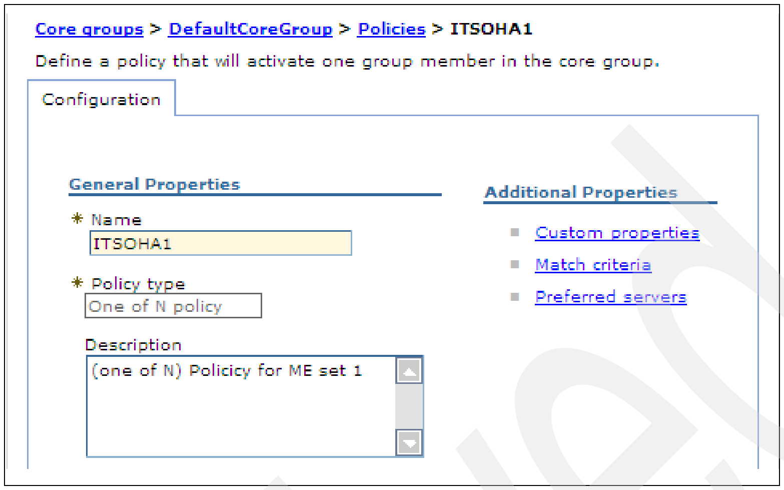
1. Create the policy.

   In the AdminConsole, select **Servers** → **Core Groups** → **DefaultCoreGroup** and click it to open.

   Under Additional Properties, select and open **Policies**.

   Enter New to create a new Policy.

On the next window that opens, enter the data, as shown in Figure 3-41.


Core groups > DefaultCoreGroup > Policies > ITSOHA1

Define a policy that will activate one group member in the core group.

Configuration

General Properties

* Name: ITSOHA1

* Policy type: One of N policy

Description: (one of N) Policicy for ME set 1

Additional Properties

- Custom properties
- Match criteria
- Preferred servers

Figure 3-41 Creation of new Policy

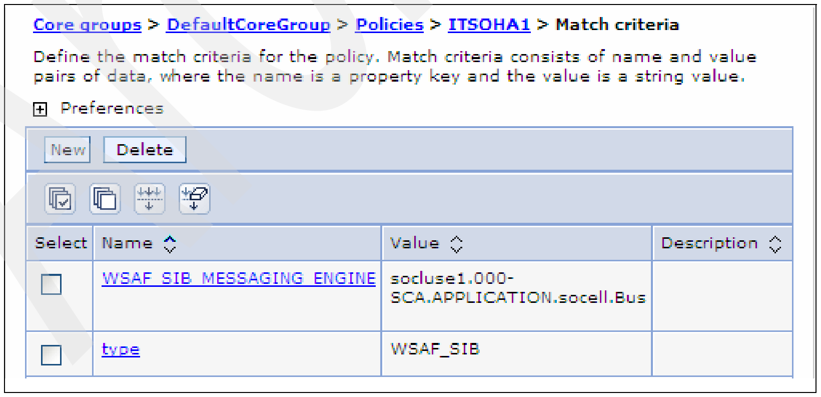
2. Set the matching criteria.

   You have to indicate the cluster or Message Engines for which this policy is applicable. We have two sets:

   – MEcluster set 000: Created when the cluster was added to the bus.

   – MEcluster set 001: Create by adding an addition Message Engine set.

   Click **Match criteria** under Additional properties and enter the data, as shown in Figure 3-42.


Core groups > DefaultCoreGroup > Policies > ITSOHA1 > Match criteria

Define the match criteria for the policy. Match criteria consists of name and value pairs of data, where the name is a property key and the value is a string value.

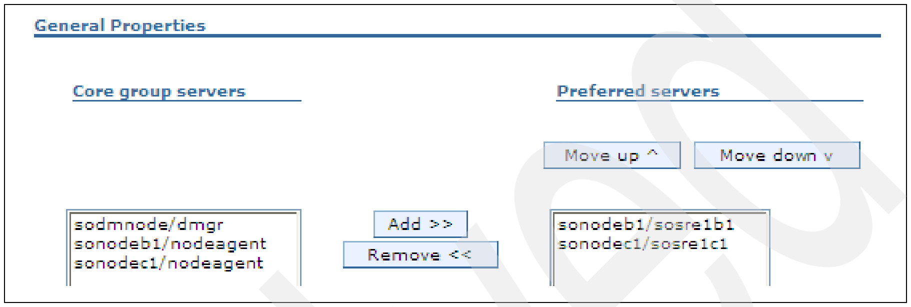
Select	Name	Value	Description
☐	WSAF_SIB_MESSAGING_ENGINE	soclusel.000-SCA.APPLICATION.socell.Bus	
☐	type	WSAF_SIB	

Figure 3-42 Matching criteria

A distinct policy is specified for each Message Engine set.

3. Specify preferred servers.

   We specify the preferred server for ITSOHA1 (b1, c1) and for ITSOHA2 (c1, b1). As a result, in a normal situation, one Message Engine belonging to set 000 must be active on server b1, while the Message Engine belonging to set 001 must be active on server c1. Remember that both have their own QueueTables, selected through a distinct schema. See an example of the selection in Figure 3-43.



Core group servers:

sodmnode/dmgr
sonodeb1/nodeagent
sonodec1/nodeagent

Preferred servers:

sonodeb1/sosre1b1
sonodec1/sosre1c1

Figure 3-43 Preferred servers

The final result is shown in Figure 3-44.

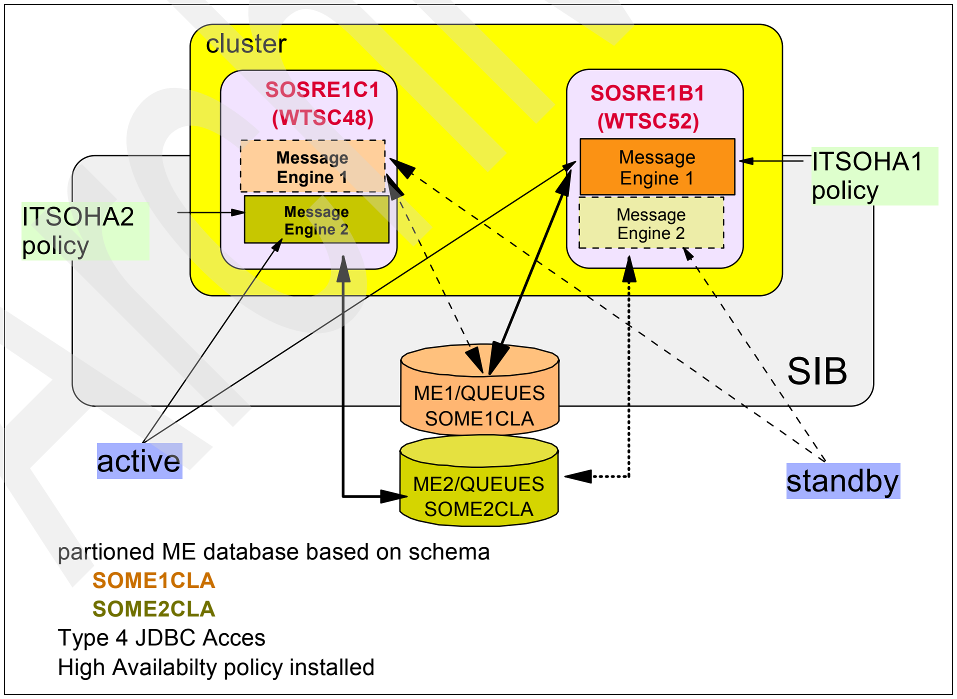


Figure 3-44 Message Engine or cluster setup with High Availability policy

Note: You have to add the CEI capability to the servers defined. You have to add a Custom Property to the Deployment Manager control region and servant region JVMs, and also to the control region, servant region, and CRA for each server. To add the property, select **dmgr** → **Process Definition** → **Servant** → **Java Virtual Machine**. The name of the property is ws.ext.dirs and the value is as follows:

```
${CEI_HOME}/lib:${CEI_HOME}/client
```

3.5 Example of Web services through WebSphere ESB to IMS

This example shows you how to build a SOAP/HTTP access through an SCA component to an existing IMS transaction. A WebSphere ESB flow provides the protocol transformation between the SOAP/HTTP and the IMS Connector protocol, and it also maps the message between the SCA interface, representing the connector, and the exported SOAP/HTTP interface. Note that it would have been equally simple to use SOAP/JMS instead of SOAP/HTTP.

You have to first build the server part that is deployed on a WebSphere Application Server with WebSphere ESB capabilities, and later you also have to make a proxy for the SOAP access.

3.5.1 Building the server

An IMS transaction is addressed through the trancode, which, in general, is part of the message. The program scheduled through the trancode takes an input message and returns an output message.

The layout of the input and output messages for this transaction is available as a COBOL copybook (extension *.cpy).

The build is done using the WebSphere Integration Developer, which immediately produces an SCA component that you can use as an import component to the mediation.

The binding for this component is SCA, but it makes use of a J2EE Connector architecture (J2C) (*Common Client Interface (CCI)*) based adapter.

In this project, the SCA component is front-extended through a WebSphere ESB mediation flow to make it available as a Web service through SOAP/HTTP.

Figure 3-45 shows the IMS transaction access from SOAP/HTTP through WebSphere ESB.

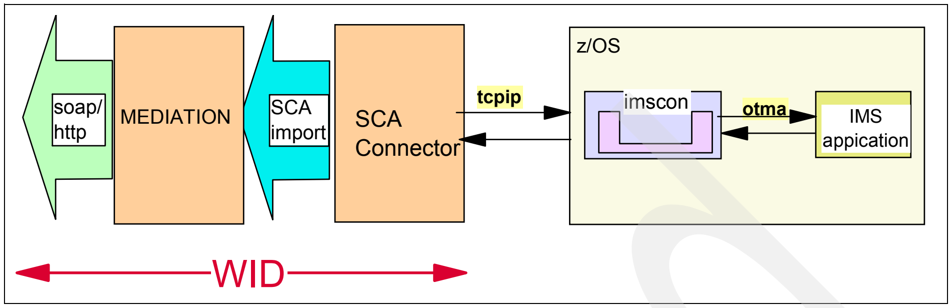


Figure 3-45 IMS transaction access from SOAP/HTTP through WebSphere ESB

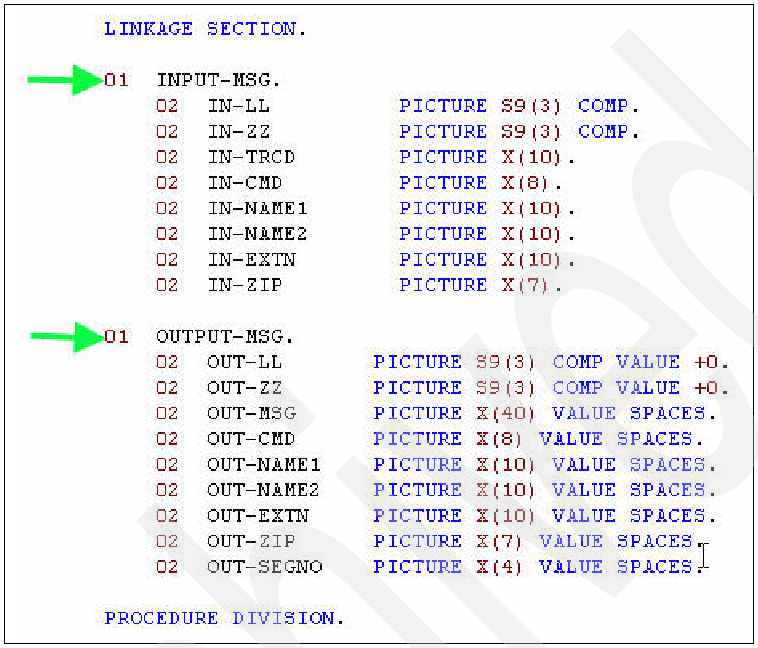
To access IMS transactions through an adapter, other elements have to be installed. An IMS adapter of type J2C always passes through a IMSCON gateway, which runs on the same LPAR as the IMS subsystem it is accessing. The connection between the adapter can be TCP/IP or XM (cross memory on z/OS), even though TCP/IP is always the recommended way when both the adapter and gateway are on the same platform. You can also use HiperSockets™ for communication.

Input and output messages for the IMS transaction

During the IMS customization, a COBOL, PLI, Java, or C program is associated with a trancode. When a message arrives with this trancode, which is part of the message, the associated program is dispatched and one of the first things performed by the program is usually reading the message. The layout of this message is dictated by the masks in the program.

After you know the layout of the *inputmsg* and the *outputmsg*, the name of the trancode, and the meaning of the fields, you are ready to build an adapter using the wizards of the WebSphere Integration Developer.

The COBOL copybook representing the input/output messages for the IMS transaction is shown in Figure 3-46. You can see the INPUT-MSG and the OUTPUT-MSG sections. These are used as input to the wizard to build the component.



```
       LINKAGE SECTION.

       01   INPUT-MSG.
            02   IN-LL            PICTURE S9(3) COMP.
            02   IN-ZZ            PICTURE S9(3) COMP.
            02   IN-TRCD          PICTURE X(10).
            02   IN-CMD           PICTURE X(8).
            02   IN-NAME1         PICTURE X(10).
            02   IN-NAME2         PICTURE X(10).
            02   IN-EXTN          PICTURE X(10).
            02   IN-ZIP           PICTURE X(7).

       01   OUTPUT-MSG.
            02   OUT-LL         PICTURE S9(3) COMP VALUE +0.
            02   OUT-ZZ         PICTURE S9(3) COMP VALUE +0.
            02   OUT-MSG        PICTURE X(40) VALUE SPACES.
            02   OUT-CMD        PICTURE X(8) VALUE SPACES.
            02   OUT-NAME1      PICTURE X(10) VALUE SPACES.
            02   OUT-NAME2      PICTURE X(10) VALUE SPACES.
            02   OUT-EXTN       PICTURE X(10) VALUE SPACES.
            02   OUT-ZIP        PICTURE X(7) VALUE SPACES.
            02   OUT-SEGNO      PICTURE X(4) VALUE SPACES.

       PROCEDURE DIVISION.
```

Figure 3-46 IMS input and output messages

The message fields IN-LL, IN-ZZ, IN-TRCD(trancode), OUT-LL, and OUT-ZZ do not have any meaning for the user, but have to be part of the message arriving in IMS and part of the response. You can deal with those fields in the mediation, and make them not visible for the SOAP/HTTP interface.

Building the EIS/SCA component for IMS

If the SCA component is used with WebSphere ESB, it has to be defined in a Mediation Module project. After starting the WebSphere Integration Developer, switch to a *Business Integration perspective* and create a new Mediation Module.

Perform the following steps:

1. From the File option in the left upper corner of your workbench window, click **New** → **Project**, as shown in Figure 3-47.

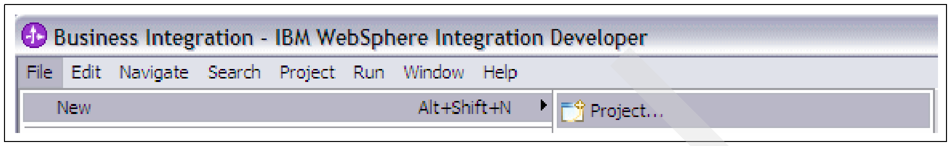

Business Integration - IBM WebSphere Integration Developer
File Edit Navigate Search Project Run Window Help
New Alt+Shift+N
Project...

Figure 3-47 New project definition

2. In the drop-down list, select **Other**. In the Select a wizard pop-up, click the **Mediation Module** project, as shown in Figure 3-48. The window in Figure 3-49 on page 141 should appear.

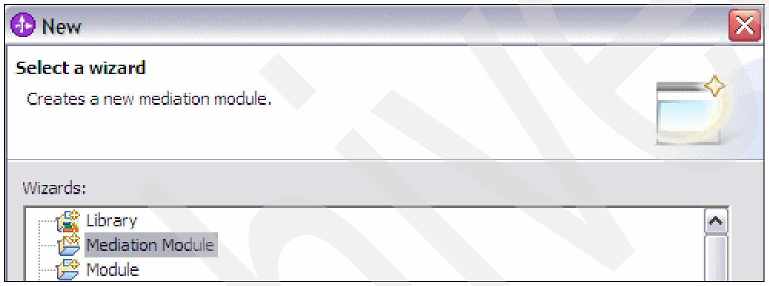
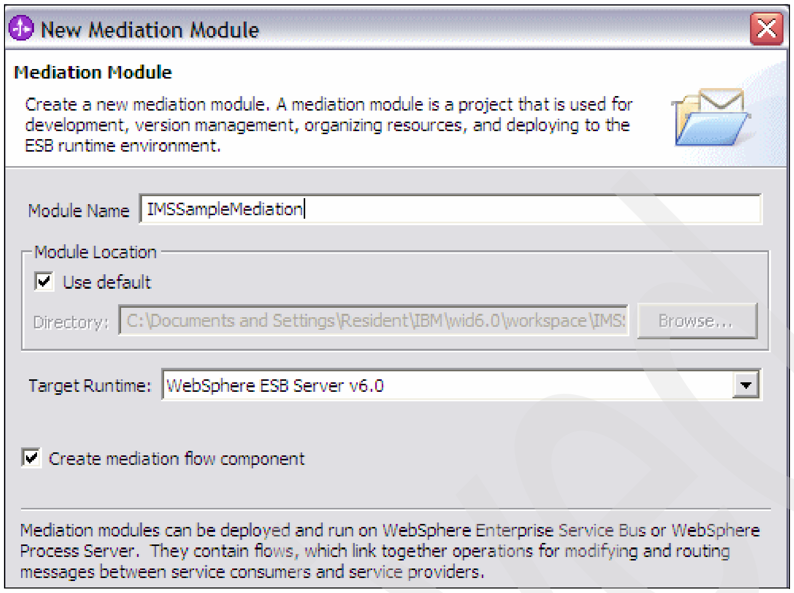


Figure 3-48 Mediation Module wizard


New Mediation Module

Mediation Module

Create a new mediation module. A mediation module is a project that is used for development, version management, organizing resources, and deploying to the ESB runtime environment.

Module Name: IMSSampleMediation

Module Location

☑ Use default

Directory: C:\Documents and Settings\Resident\IBM\wid6.0\workspace\IMS

Browse...

Target Runtime: WebSphere ESB Server v6.0

☑ Create mediation flow component

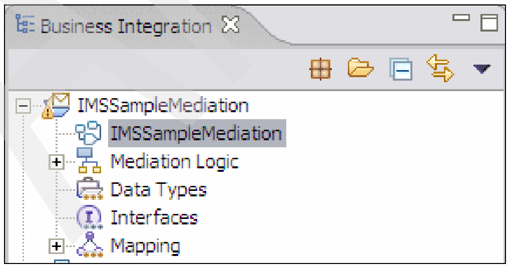
Mediation modules can be deployed and run on WebSphere Enterprise Service Bus or WebSphere Process Server. They contain flows, which link together operations for modifying and routing messages between service consumers and service providers.

Figure 3-49 Module Name and mediation flow selection

Fill in the fields as shown in the sample:

- Type the Module Name as IMSSampleMediation.
- Select the **Create mediation flow component** option. In the flow, which you compose later, you have to specify the nodes required for the message transformations.
- The Target Runtime has to be at least WebSphere ESB Server V6 (it can be WebSphere Process Server, because this includes WebSphere ESB).

3. Select **Finish** to create the module and you will see a result similar to Figure 3-50.


Business Integration

- IMSSampleMediation
  - IMSSampleMediation
  - Mediation Logic
  - Data Types
  - Interfaces
  - Mapping

Figure 3-50 IMSSampleMediation module project

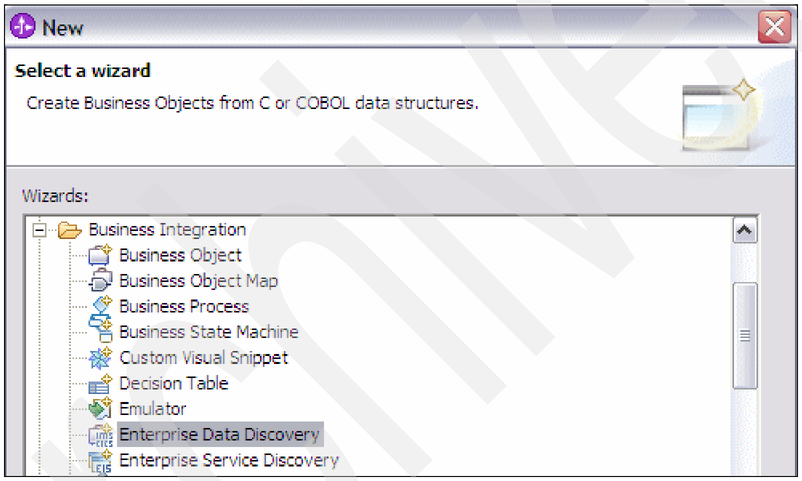
The next step is to build the SCA component to access the IMS transaction through a J2C adapter using the following steps:

- Enterprise Data Discovery
- Enterprise Service Discovery

Enterprise Data Discovery

While the input to and output from IMS are flat structures, input and output for the SCA component are in a Business Object format. We use the *Enterprise Data Discovery* wizard to build the Business Objects for the IMS input/output messages. Perform the following steps:

1. Right-click **IMSSampleMediation** → **New** → **Other**, and then in the list under Business Integration, select **Enterprise Data Discovery**, as shown in Figure 3-51.


New

Select a wizard

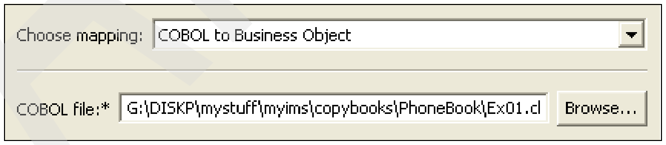
Create Business Objects from C or COBOL data structures.

Wizards:

Business Integration
- Business Object
- Business Object Map
- Business Process
- Business State Machine
- Custom Visual Snippet
- Decision Table
- Emulator
- Enterprise Data Discovery
- Enterprise Service Discovery

Figure 3-51   Select Data Discovery

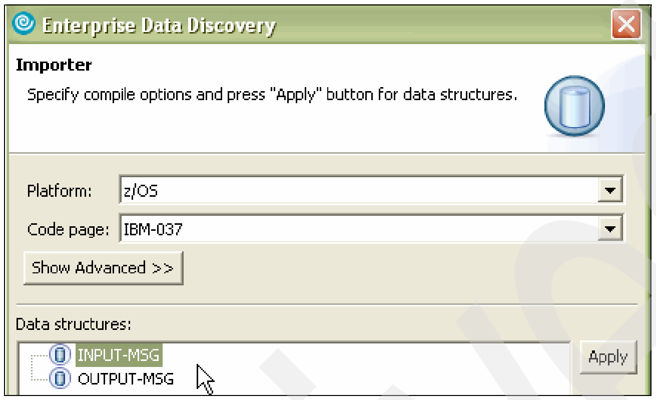
2. This wizard takes the COBOL copybook as input, as shown in Figure 3-52.


Choose mapping: COBOL to Business Object

COBOL file:* G:\DISKP\mystuff\myims\copybooks\PhoneBook\Ex01.cl

Figure 3-52   COBOL copybook as input

3. In this file, two data structures sections are available, as shown in Figure 3-53.The IMS transactional program expects data in EBCDIC. Set the platform to z/OS and the Code page to IBM-037, as indicated. Click the **Apply** button, and then select the **INPUT-MSG** data structure and go to the next step.


Enterprise Data Discovery

Importer

Specify compile options and press "Apply" button for data structures.

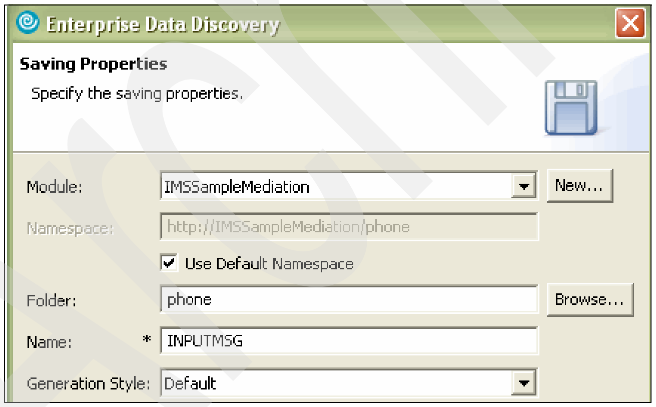
Platform: z/OS

Code page: IBM-037

Data structures:
- INPUT-MSG
- OUTPUT-MSG

Figure 3-53 Selecting input and encoding for z/OS

4. Set the Name and the Folder for the new object, as shown in Figure 3-54. Click **Finish**.


Enterprise Data Discovery

Saving Properties

Specify the saving properties.

Module: IMSSampleMediation

Namespace: http://IMSSampleMediation/phone

☑ Use Default Namespace

Folder: phone

Name: * INPUTMSG

Generation Style: Default

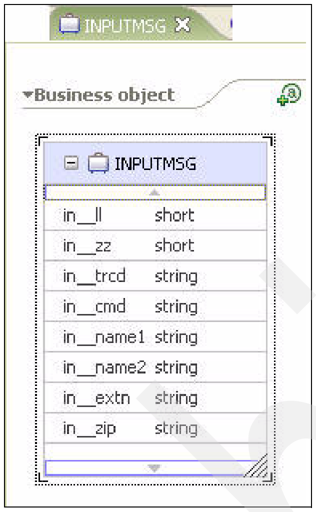
Figure 3-54 Set folder and name for the Input Business Object

In the same way, build the OUTPUTMSG Business Object. Starting with the re-invocation of Enterprise Data Recovery, repeat the steps used for the INPUT-MSG to define the OUTPUT-MSG data structure and name the Business Object “OUTPUTMSG”. The result must be two Business Objects as follows:

- ► INPUTMSG
- ► OUTPUTMSG

They are created under the Data type subdirectory of the module project. Double-click the **INPUTMSG** object to open it with the Business Object Editor, as shown in Figure 3-55.

The specific IMS fields (in_ll, in_zz, in_trcd) should appear, shown in Figure 3-55. The content of these fields is rather constant, because it is not a variable input from a user. You are going to add them in the flow of the mediation and they are going to be transparent to the SOAP/HTTP interface.



INPUTMSG

in__ll	short
in__zz	short
in__trcd	string
in__cmd	string
in__name1	string
in__name2	string
in__extn	string
in__zip	string

Figure 3-55 INPUTMSG Business Object

Enterprise Service Discovery

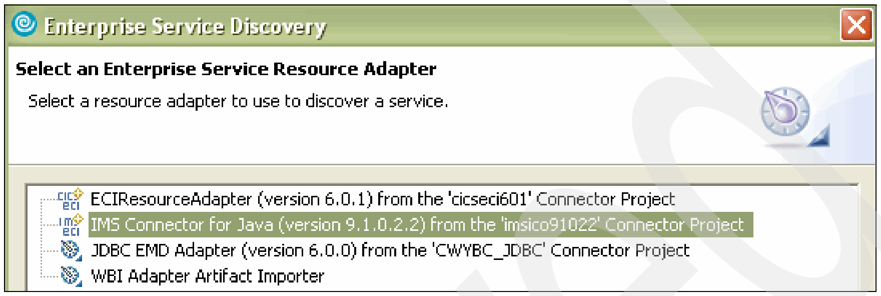
In the *Enterprise Service Discovery* phase, you can complete the *SCA module for the Enterprise Information System (EIS)* access, and you can make use of the Business Objects prepared in the previous phase.

Service discovery is not unique to IMS. You can use it for all supported Java Connector architecture (JCA1.5) adapters. Therefore, you have to indicate for which adapter (connector) you are performing the build.

Similarly, for the *Enterprise Data Discovery*, perform the following steps:

1. You must start the Enterprise Service Discovery by right-clicking **IMSSampleMediation**, and then selecting **New** or **Other** or **Enterprise Service Discovery**.

2. For IMS, select the adapter/connector **IMS Connector for Java**. You must have the appropriate *Resource Adapter (RA)* installed on the client platform. In Figure 3-56, you can see that several adapters were installed on the development platform. Select the appropriate IMS adapter, which is distributed with IMS/v9. Select the **IMS Connector** adapter and click **Next** to continue.



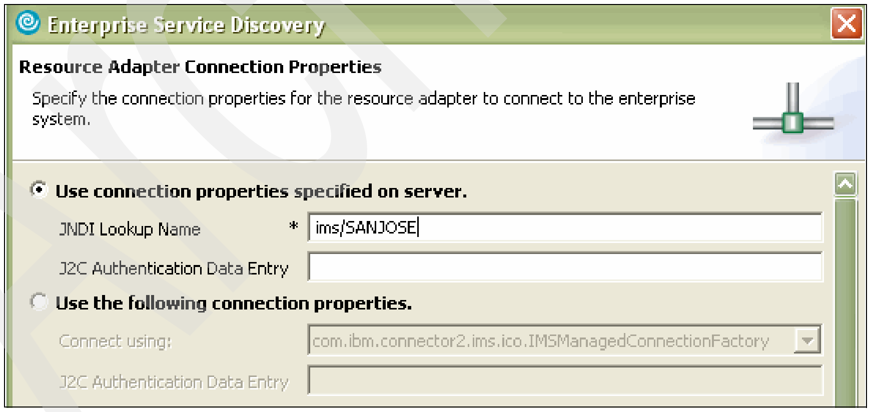
Select an Enterprise Service Resource Adapter

Select a resource adapter to use to discover a service.

- ECIResourceAdapter (version 6.0.1) from the 'cicseci601' Connector Project
- IMS Connector for Java (version 9.1.0.2.2) from the 'imsico91022' Connector Project
- JDBC EMD Adapter (version 6.0.0) from the 'CWYBC_JDBC' Connector Project
- WBI Adapter Artifact Importer

Figure 3-56 imsico91022 Connector project

3. The next step requires the data for the Connection Properties. You have to indicate how to reach the IMS Connector gateway, where the gateway is a separate address space on the same LPAR as IMS. You can specify the data for the connection here or you might specify a jndiName reference to a ConnectionFactory that has been defined in the WebSphere Application Server runtime, where you are going to deploy the Module. We selected the latter method, as shown in Figure 3-57.



Resource Adapter Connection Properties

Specify the connection properties for the resource adapter to connect to the enterprise system.

[✓ Use connection properties specified on server.]

JNDI Lookup Name *	ims/SANJOSE
J2C Authentication Data Entry	

Use the following connection properties.

Connect using:	com.ibm.connector2.ims.ico.IMSManagedConnectionFactory
J2C Authentication Data Entry	

Figure 3-57 jndiName to ConnectionFactory

The jndiName is ims/SANJOSE and is used in the lookup call for the ConnectionFactory. We assume that this factory does exist, as previously defined through the Admin Console.

Note: When accessing Enterprise Information Systems, such as IMS, CICS, and others through a J2C adapter, two specifications have to be provided:

- ConnectionSpec: This is related to the connection that is going to be used during several interactions. It specifies elements, such as ConnectionType, IPaddress, Port, and IMSSubsystem. In our specific case, all elements have been specified in the ConnectionFactory in the WebSphere Application Server. From the ConnectionFactory, a connection is obtained to the EIS, to be used with the interactions. The concept of the factory allows for reuse of connections, because how the factory builds it is transparent to the user. But the factory requires a minimum for the connection parameters.
- InteractionSpec: Within the connection, each IMS transaction type could have specific interaction characteristics. Elements to specify here relate to the communication protocol, commit mode, and unit of work.

4. Enter the name and click **Next**. After this information has been provided, specify the name of an operation (interaction) that maps to the invocation of this transaction and also the names of the input and output. These are the *Business Objects* that you build in the *data discovery* phase.

   Click **Add** to start specifying these operations. Enter Phone in the Name field and, using the **Browse** button, select **INPUTMSG** as the Input Type BO and **OUTPUTMSG** as the Output type BO, and then **Finish**, as shown in Figure 3-58.

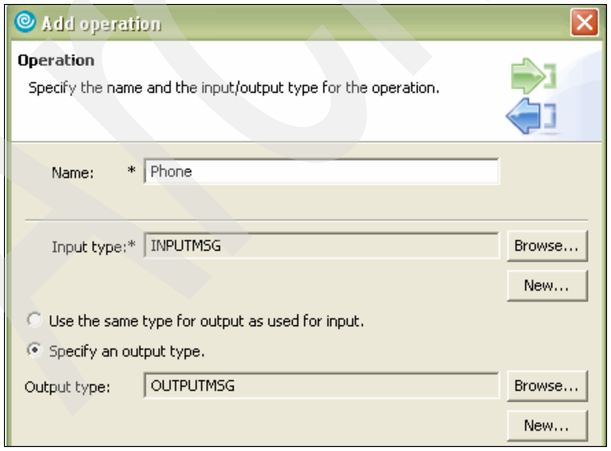
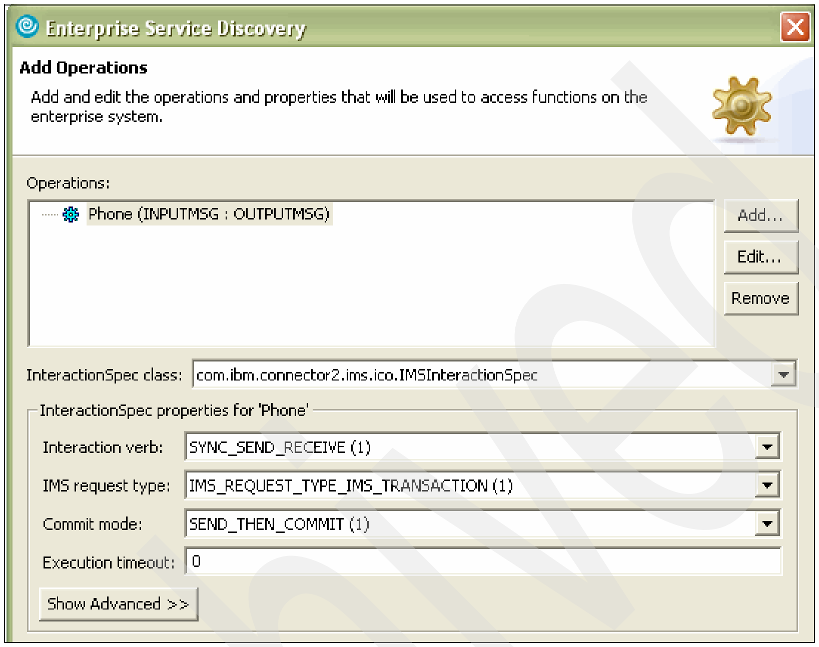


Figure 3-58 Operation name with input and output

5. The information defined here is related to synchronization, unit of work, and commit mode. For the current transaction, we leave it as indicated, as shown in Figure 3-59, and then continue with **Next**.



Add and edit the operations and properties that will be used to access functions on the enterprise system.

Operations:
Phone (INPUTMSG : OUTPUTMSG)

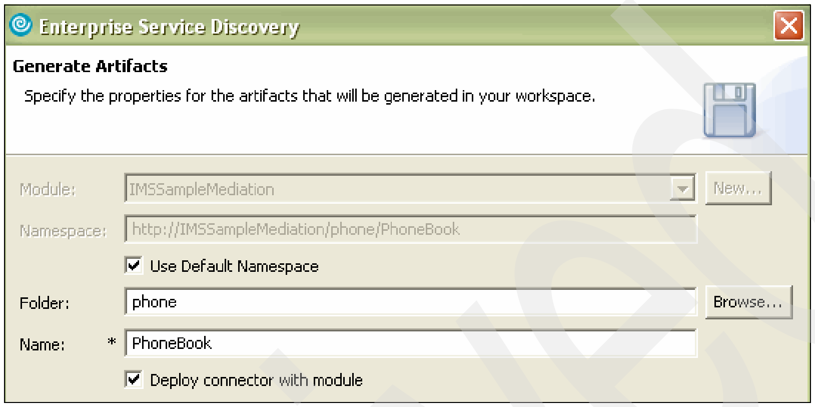
InteractionSpec class: com.ibm.connector2.ims.ico.IMSInteractionSpec

InteractionSpec properties for 'Phone'

Interaction verb: SYNC_SEND_RECEIVE (1)
IMS request type: IMS_REQUEST_TYPE_IMS_TRANSACTION (1)
Commit mode: SEND_THEN_COMMIT (1)
Execution timeout: 0

Figure 3-59 Interaction specification

6. The last step of the *Enterprise Service Discovery* activity is to specify a Name for the collection of service artifacts (in our case, we chose Phonebook) and to identify the Folder where they are saved (in our case, phone). Select the **Deploy connector with module** option, and then **Finish**, as shown in Figure 3-60.



Enterprise Service Discovery

Generate Artifacts

Specify the properties for the artifacts that will be generated in your workspace.

Module: IMSSampleMediation [New...]

Namespace: http://IMSSampleMediation/phone/PhoneBook

☑ Use Default Namespace

Folder: phone [Browse...]

Name: * PhoneBook

☑ Deploy connector with module

Figure 3-60 Saving the SCA component

You can find detailed information about Connection/InteractionSpec parameters in the IBM Redbooks publication *IMS Connectivity in an On Demand Environment: A Practical Guide to IMS Connectivity*, SG24-6794.

Building the mediation

The component that has been built is an import node and you can verify this by opening the Assembly Diagram of the Mediation Module and looking at the format of the component: the icon inside the picture shows "EIS". You can open the Assembly Diagram if you click the name **IMSSampleMediation** twice, under the module of the same name, in the left part of the WorkBench. as shown in Figure 3-61.

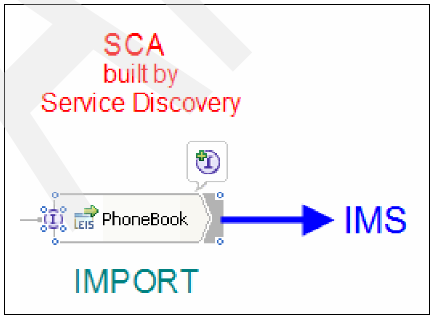


Figure 3-61 SCA Import component for EIS IMS

The interface of this component must receive input data that completely reflects the content of the input message for the IMS transaction. This includes IMS typical parameters, such as ll, zz, transcode, and so forth. Assume that you want to make this transaction accessible by SOAP/HTTP; you will require a protocol transformation. Also, you can define a simpler operational interface in which you can omit those IMS specifics. You can add the IMS required fields inside the mediation flow during Extensible Markup Language (XML) mapping and take them off on return. This could happen in an XSLT node, which is one of the primitives provided by WebSphere ESB.

Interface for the SOAP/HTTP export

With the WID wizard, you build a description of the new (SOAP) interface, which is externally exported for the users. This description is stored as a WSDL metafile. You have to define the following:

- The I/O used by this interface operation. This is defined in a BO(SDO) format.
- The interface operation.

Start with defining new *Business Objects* for the SOAP/HTTP interface. These Business Objects are going to be subsets of the ones used on the SCA(IMS) interface.

Business Objects for the SOAP/HTTP interface

While in a Business Integration perspective of WID, perform the following steps:

1. In the left frame of the workbench, right-click **Data Types** in the hierarchy under the mediation module name and select **New** → **Business Object**, as shown in Figure 3-62.

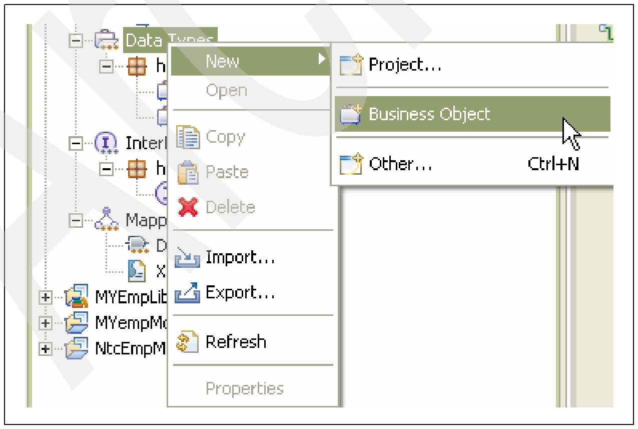
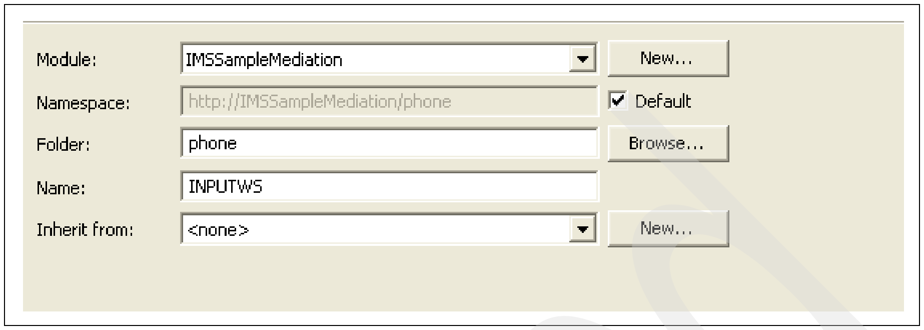


Figure 3-62 A new Business Object

2. Select the existing phone in the Folder field and specify INPUTWS as the new Business Object's Name, as indicated in Figure 3-63, and then click **Next**.

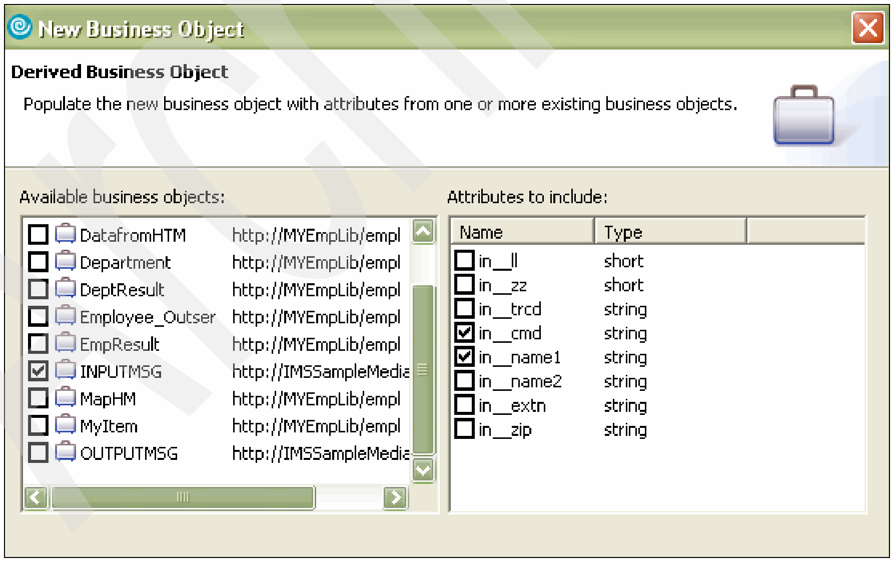


Module:	IMSSampleMediation	New...
Namespace:	http://IMSSampleMediation/phone	☑ Default
Folder:	phone	Browse...
Name:	INPUTWS	
Inherit from:	<none>	New...

Figure 3-63 Name and folder for the Web services input object

3. The next window presents a list of existing Business Objects, which might show different entries in your workbench. The new Business Object to be created is a subset of the existing INPUTMSG. You can use this as a start.

   Select, in the left column, the parent object as indicated, and in the right column, eliminate the typical IMS fields, and leave only the input fields, as shown in Figure 3-64. Click **Finish** to generate the new Business Object under the subdirectory Data Types of the module project.



New Business Object

Derived Business Object

Populate the new business object with attributes from one or more existing business objects.

Available business objects:

☐	DatafromHTM	http://MYEmpLib/empl
☐	Department	http://MYEmpLib/empl
☐	DeptResult	http://MYEmpLib/empl
☐	Employee_Outser	http://MYEmpLib/empl
☐	EmpResult	http://MYEmpLib/empl
☑	INPUTMSG	http://IMSSampleMedia
☐	MapHM	http://MYEmpLib/empl
☐	MyItem	http://MYEmpLib/empl
☐	OUTPUTMSG	http://IMSSampleMedia

Attributes to include:

	Name	Type
☐	in__ll	short
☐	in__zz	short
☐	in__trcd	string
☑	in__cmd	string
☑	in__name1	string
☐	in__name2	string
☐	in__extn	string
☐	in__zip	string

Figure 3-64 Select the fields for the new Business Object

Repeat the New Business Object creation process for the output of the Web service, also omitting the IMS fields named in_ll and in_zz. Figure 3-65 shows a visual representation of both Business Objects, obtained with the Business Object Editor.

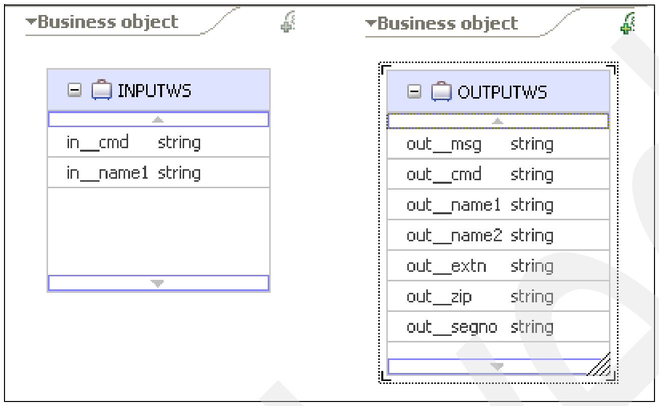


Figure 3-65 Input/output Business Object for the Web service

Interface and operations

This interface description is abstract and it does not provide any information about how you can reach it (protocol binding) and where the service is available (service point).

You have to define a new interface that uses the Business Objects you just made. With a WID wizard, you can build a WSDL metafile describing this interface. Perform the following steps:

1. From the Business Integration perspective, in the left frame of the workbench, right-click **Interfaces** in the hierarchy under the mediation module name and then select **New** → **Interface**, as shown in Figure 3-66.

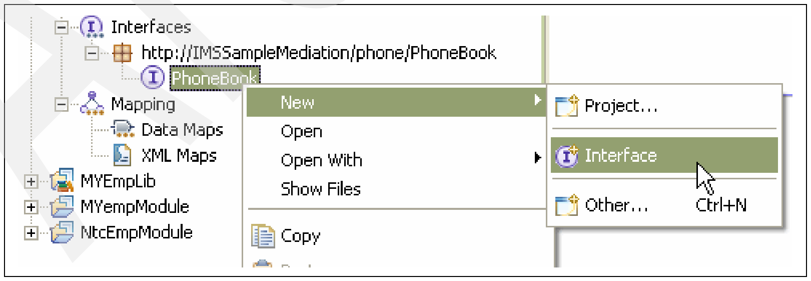
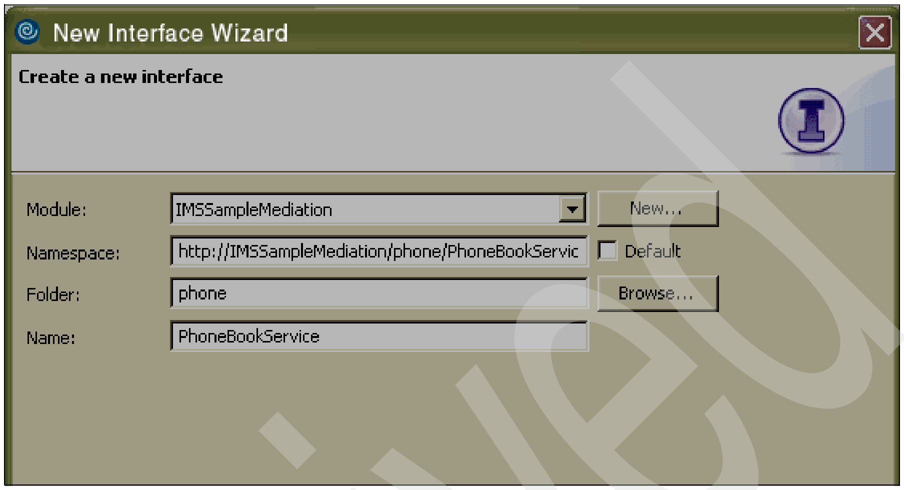


Figure 3-66 A new interface

2. In the next step, enter a Name and Folder for the new Interface, using the existing phone as the Folder and specifying PhoneBookService as the new interface's Name, as indicated in Figure 3-67. Click **Finish** to continue.



Module: IMSSampleMediation
Namespace: http://IMSSampleMediation/phone/PhoneBookServic
Folder: phone
Name: PhoneBookService

Figure 3-67 Name and folder for the WSDL described interface

Note: Although WSDL means Web Services Description Language, this XML based meta language is not limited to Web services. It can describe the interfaces for many types of bindings.

3. An interface can contain several operations, even though in this example we only define one. The operation has a name, an entry point (API), and an input and output, if it is a request/response operation. Click the appropriate icon in the upper menu for a request/response operation, as shown in Figure 3-68.

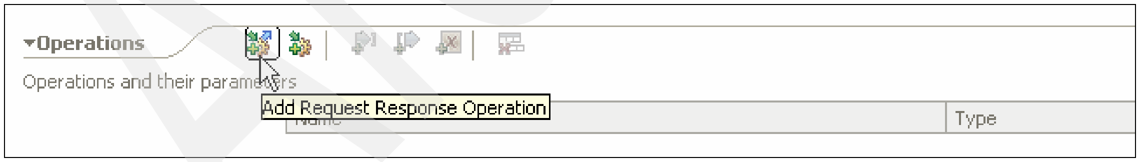

Operations
Operations and their parameters
Add Request Response Operation
Name
Type

Figure 3-68 Interface wizard and request/response selection

Change the name of the operation to getPhone. When you select the newly created operations object, other icons in the upper menu turn active.

1. Click the **Add Input** button to specify the input.

   Within the newly created row named input1, click the string entry in the Type column, and from the drop-down list, select the **INPUTWS** object, as shown in Figure 3-69.

2. Repeat this step for the output where you select the **OUTPUTWS** structure.

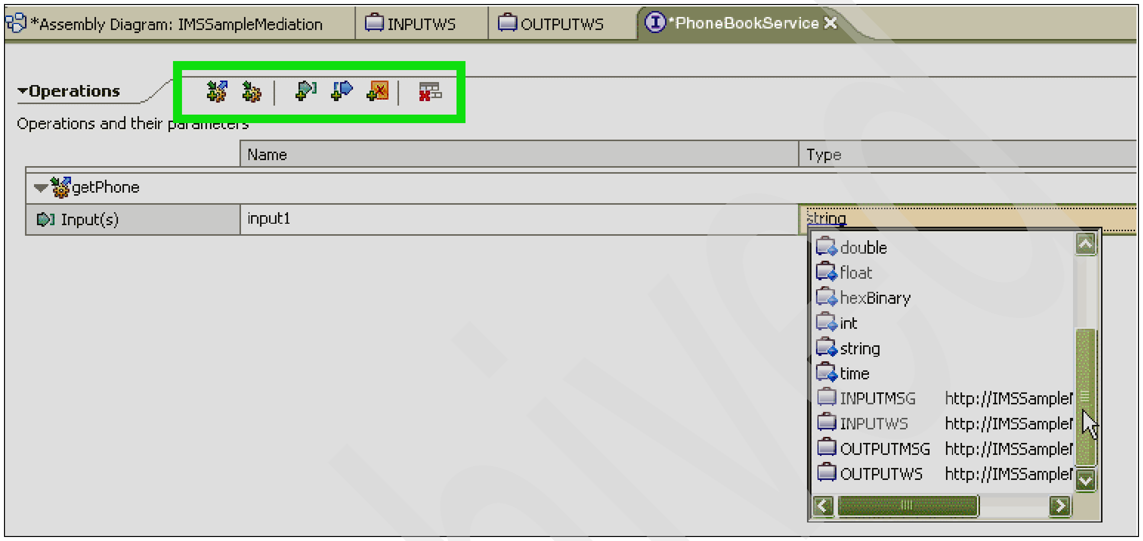
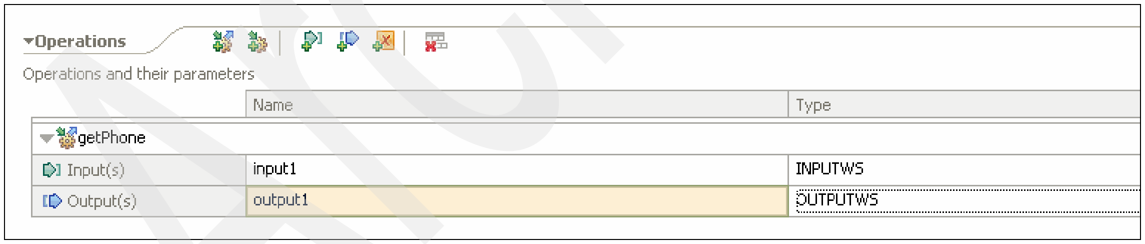

*Assembly Diagram: IMSSampleMediation
INPUTWS
OUTPUTWS
*PhoneBookService
Operations
Operations and their parameters
Name
Type
getPhone
Input(s)
input1
string
double
float
hexBinary
int
string
time
INPUTMSG http://IMSSampleI
INPUTWS http://IMSSampleI
OUTPUTMSG http://IMSSampleI
OUTPUTWS http://IMSSampleI

Figure 3-69 Input and output for the interface

3. The end result must look similar to Figure 3-70.



Operations

Operations and their parameters

	Name	Type
getPhone		
Input(s)	input1	INPUTWS
Output(s)	output1	OUTPUTWS

Figure 3-70 Complete interface

At this point, we suggest that you save all the definitions by pressing Ctrl+Shift+s.

Complete the Mediation Flow

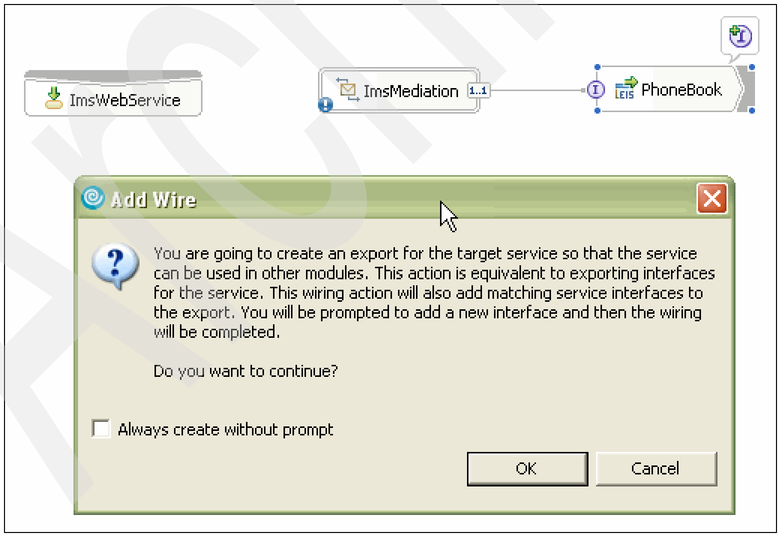
When the *Assembly Diagram* is opened, you already have two components on the board:

- Import: SCA Component PhoneBook.
- MediationFlow: Because you requested a Mediation Module with a flow. Rename the flow to ImsMediation.

Wire these two components together, as shown in the right-hand part of Figure 3-71. The mediation that you put together is entered by SOAP/HTTP.

To express this facility, you have to add an Export component with this Binding. To do this, perform the following steps:

1. You have to click the **Import** icon from the palette on the left of the Assembly Diagram.
2. Select the **Export** icon from the new palette. Drag and drop it to the Assembly Diagram canvas.
3. Change the name (right-click/Rename) of the Export component to ImsWebService.
4. Wire the tail of Export component to the front ImsMediation flow.
5. Click **OK** on the prompt shown about the use of the target service, as shown in Figure 3-71.



You are going to create an export for the target service so that the service can be used in other modules. This action is equivalent to exporting interfaces for the service. This wiring action will also add matching service interfaces to the export. You will be prompted to add a new interface and then the wiring will be completed.

Do you want to continue?

☐ Always create without prompt

Figure 3-71 Adding the “Export” component.

The next step is the customization of this component, which consists of the following tasks:

- Adding an interface and entry API to the component.
- Adding the SOAP/HTTP implementation and binding.
- Wiring it as a front end to the mediation flow.

The objective of this task is to achieve the configuration shown in Figure 3-72:

- PhoneBook: Import component of the IMS service.
- Mediationflow: This contains the XSLT transforms.
- ImsWebService: Export for SOAP/HTTP.

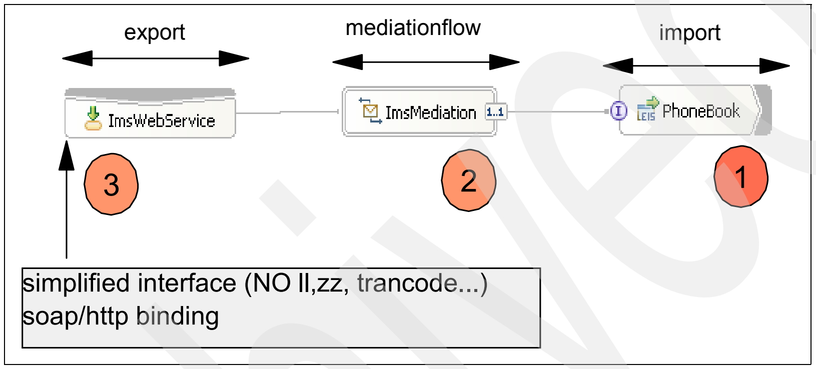


Figure 3-72 Overview of the mediation

To add an interface to the ImsWebService component, perform the following:

1. Right-click the **Add Interface** component, as shown in Figure 3-73.

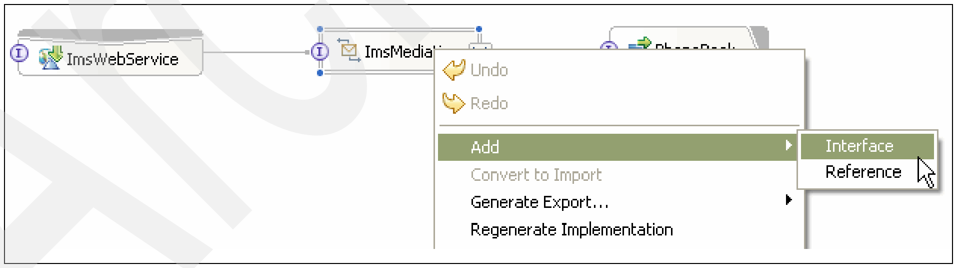
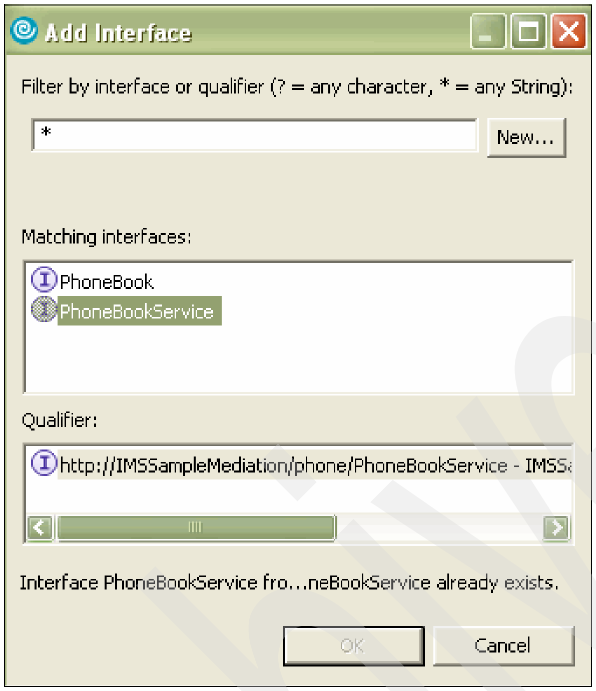


Figure 3-73 Select add interface

2. Select the **PhoneBookService** from the WSDL interface list, as shown in Figure 3-74, and click **OK**.



http://IMSSampleMediation/phone/PhoneBookService - IMSS

Interface PhoneBookService fro...neBookService already exists.

Figure 3-74 Interface select

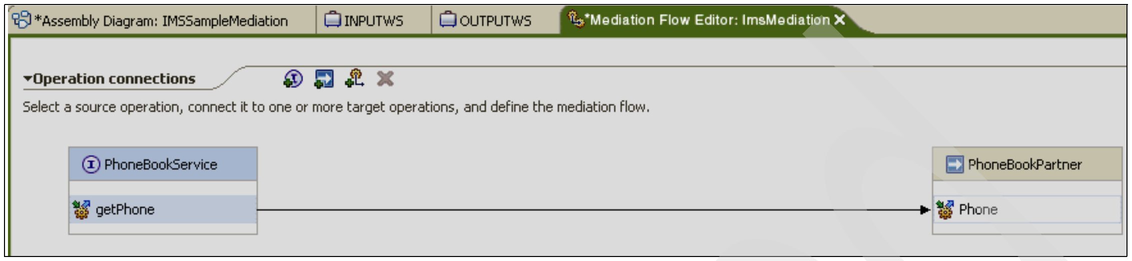
To add the implementation, perform the following tasks:

1. Right-click the component and select **Generate Binding** → **Web Service Binding**.
2. Select **Yes** in the prompt about generating a WSDL file.
3. Select **SOAP/HTTP** as the transport and click **OK**.

You still have to perform the *mapping* of the Business Objects in the ImsWebService(Export) interface to those on the PhoneBook(Import) component. This is part of the implementation of the Mediation Flow.

Perform the following steps:

1. From the Assembly Diagram, open the mediation flow by clicking the ImsMediation presentation symbol twice. Connect the getPhone operation to the Phone operation, as shown in Figure 3-75.



Select a source operation, connect it to one or more target operations, and define the mediation flow.

Figure 3-75 Mediation flow

2. To design the details of the mediation flow, click **getPhone** on the left in the diagram. A second/bottom part is added below the current Mediation Flow Editor pane. In this part, you can flip/flop between the request and response flow, clicking the appropriate fields in the left bottom corner. Remember that both are shown from left to right, even for the response.

   When it first shows up, with the request flow selected, the flow is empty, which means that the Business Objects would be passed without any mapping. In this flow, you can use the WebSphere ESB provided primitives, including custom primitives. The only primitive required here is the XSL_Transformation node for the mapping of the Business Objects.

3. From the icon palette on the left:

   a. Drag and drop an **XSL_Transformation** node (see Figure 3-76 on page 158) to the canvas.

   b. Change the name of the XSLT node to XSLWStoIMS.

   c. Wire the tail of Request-getPhone:PhoneBookService to this node.

   d. Wire this node’s out terminal to the Invoke-Phone;PhoneBookPartner.

The mapping between the Business Objects is based on XSLT techniques. You have to generate the XSLT logic, which works on a *from* and *to* XML structure, under the guidance of a metafile, on an XML file, as shown in Figure 3-76.

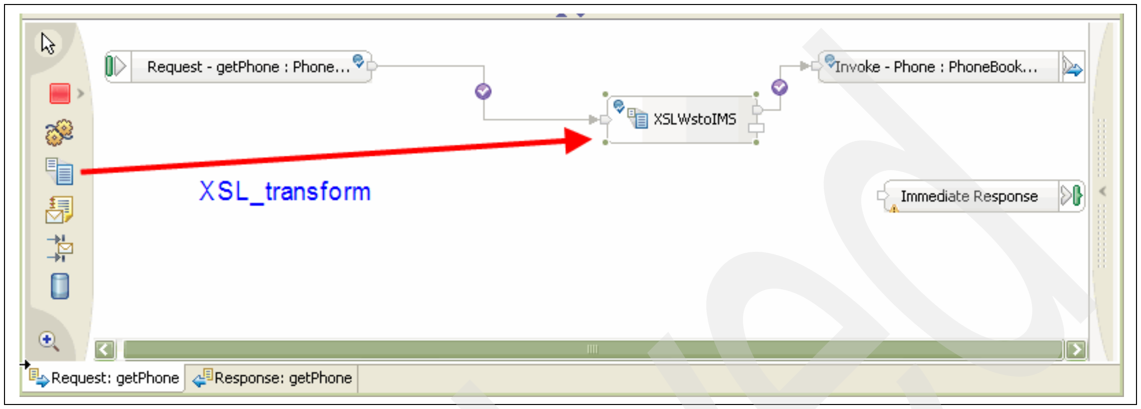
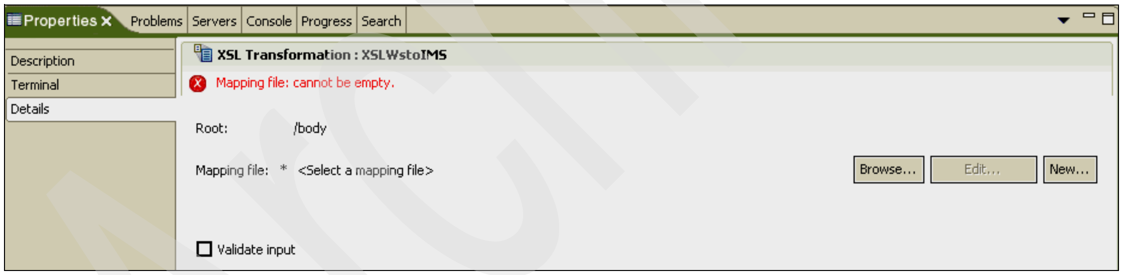


Figure 3-76 XSLT in the request flow

4. Click the XSLT node called **XSLWStoIMS** to open it. In the right bottom of the workbench window, under Properties, select the **Details**, as shown in Figure 3-77.



XSL Transformation : XSLWstoIMS

Mapping file: cannot be empty.

Root: /body

Mapping file: * <Select a mapping file>

☐ Validate input

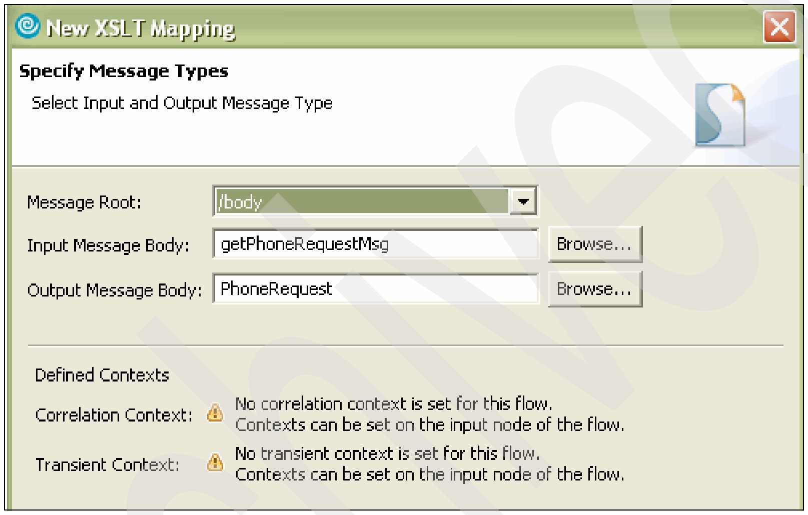
Figure 3-77 Properties of the XSLT Mapping node

5. An error message indicates that the Mapping File cannot be empty. To the right of the Mapping file, click the **New** button to build this required mapping.

6. You can accept the values shown in Figure 3-78.

   One parameter Message Root requires some more explanation: within the Enterprise Service Bus, for the transport of the data (Business Objects), a derivation of the Service Data Object (SDO) is used, which is called the Service Message Object (SMO). This is a datagraph (basically an XML structure).

   With the Message Root parameter, you can indicate at what part of the SMO structure you like to start applying the transformation. As you noticed, we selected the body starting points.



Specify Message Types

Select Input and Output Message Type

Message Root: /body

Input Message Body: getPhoneRequestMsg

Output Message Body: PhoneRequest

Defined Contexts

Correlation Context: No correlation context is set for this flow. Contexts can be set on the input node of the flow.

Transient Context: No transient context is set for this flow. Contexts can be set on the input node of the flow.

Figure 3-78 Mapping message types

7. Click **Finish** and start the mapping on the next window.

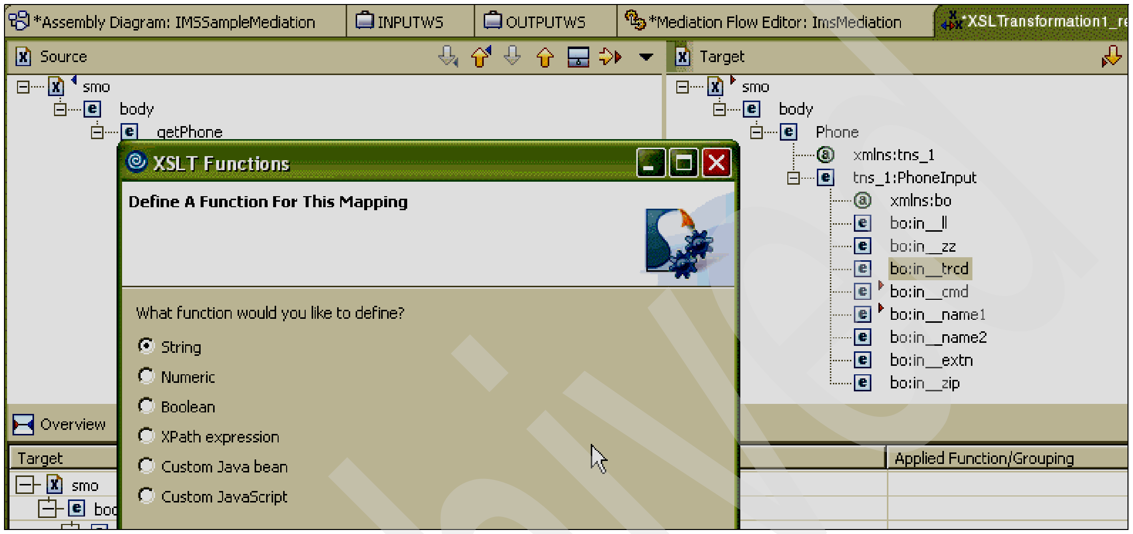
On the left of the upper panel, the source XML structure is presented and, on the right, the target mapping.

1. Expand both structures so that you see the fields.
2. Click an element in the source on the left, focus the corresponding element on the right, and right-click **Create Mapping**. This works out if the source and target element are available and without transformation. You can apply this for the following two fields:
   – in_cmd
   – in_name1

If an element is not available, such as in this case for the specific IMS fields ll, zz, and transaction code, then they can be entered here as constants.

Fields that are mapped are indicated by a small arrow.

1. Right-click a non-mapped field in the right structure and select **Define XSLT function**. This presents a group of options that you can use to build the target field, as you can see in Figure 3-79. In our case, we only use the String and Numeric options to enter the ll, zz, trcd fields. Other options are also available.



XSLT Functions

Define A Function For This Mapping

What function would you like to define?

[✓ String] Numeric Boolean XPath expression Custom Java bean Custom JavaScript

Figure 3-79 Mapping input to output

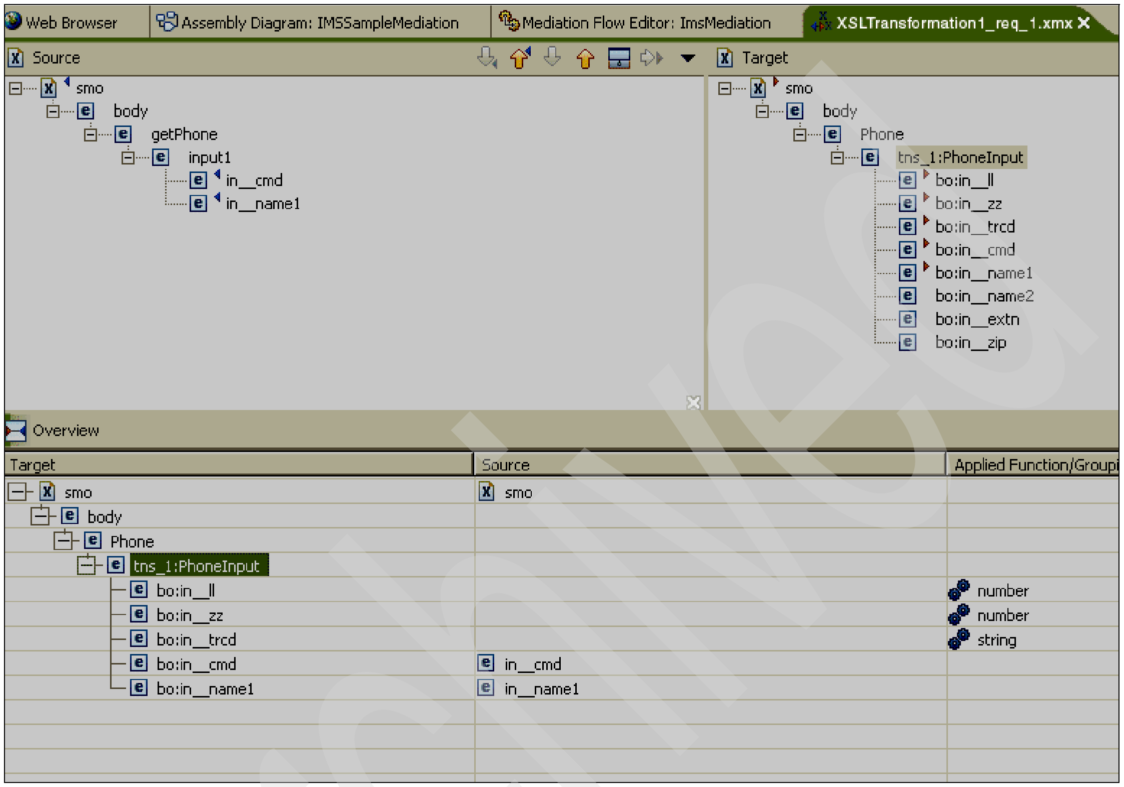
2. On the right the following fields are displayed, enter the following values:

in_ll (total length of message)
function=numeric / number / value=59

in_zz: flags (0) function= numeric / number / value=0

in_trcd:transaction code
function=string / string/ value='IVTNO' (be sure to use single quotes for the string)

The final result of the mapping can be observed in Figure 3-80. During the mapping cycle, the bottom part is actualized, so that you can always see the mapping progress.


Web Browser
Assembly Diagram: IMSSampleMediation
Mediation Flow Editor: ImsMediation
XSLTransformation1_req_1.xmx
Source
Target
Overview

Source:

```
smo
  body
    getPhone
      input1
        in__cmd
        in__name1
```

Target:

```
smo
  body
    Phone
      tns_1:PhoneInput
        bo:in__ll
        bo:in__zz
        bo:in__trcd
        bo:in__cmd
        bo:in__name1
        bo:in__name2
        bo:in__extn
        bo:in__zip
```

Target	Source	Applied Function/Groupi
smo	smo	
body		
Phone		
tns_1:PhoneInput		
bo:in__ll		number
bo:in__zz		number
bo:in__trcd		string
bo:in__cmd	in__cmd	
bo:in__name1	in__name1	

Figure 3-80 Result of request mapping

3. At the end, click **Finish** and close the mapping editor. The work in the forward/request direction is done. You have to repeat the work for the backward/response direction.
4. At the bottom of the Mediation Flow Editor, click **Response:getPhone**, as shown in Figure 3-81.

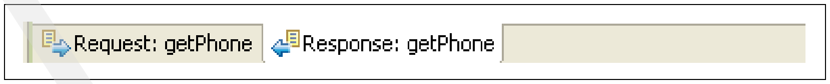


Figure 3-81 Request → Response flow selection

5. The bottom part of the editor switches to this flow, but remember that even for this flow, the traffic is depicted going from left to right. Insert an XSLT node in this flow and adapt the mapping of the output message, as shown in Figure 3-82.

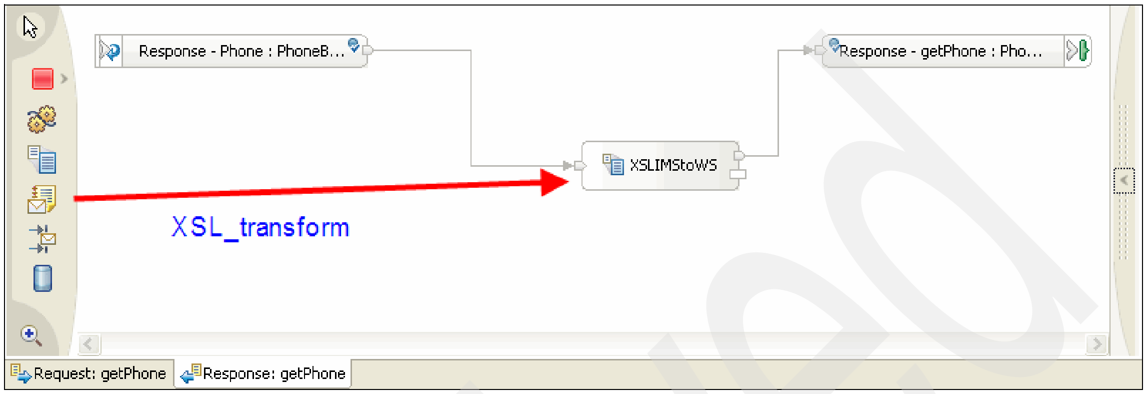
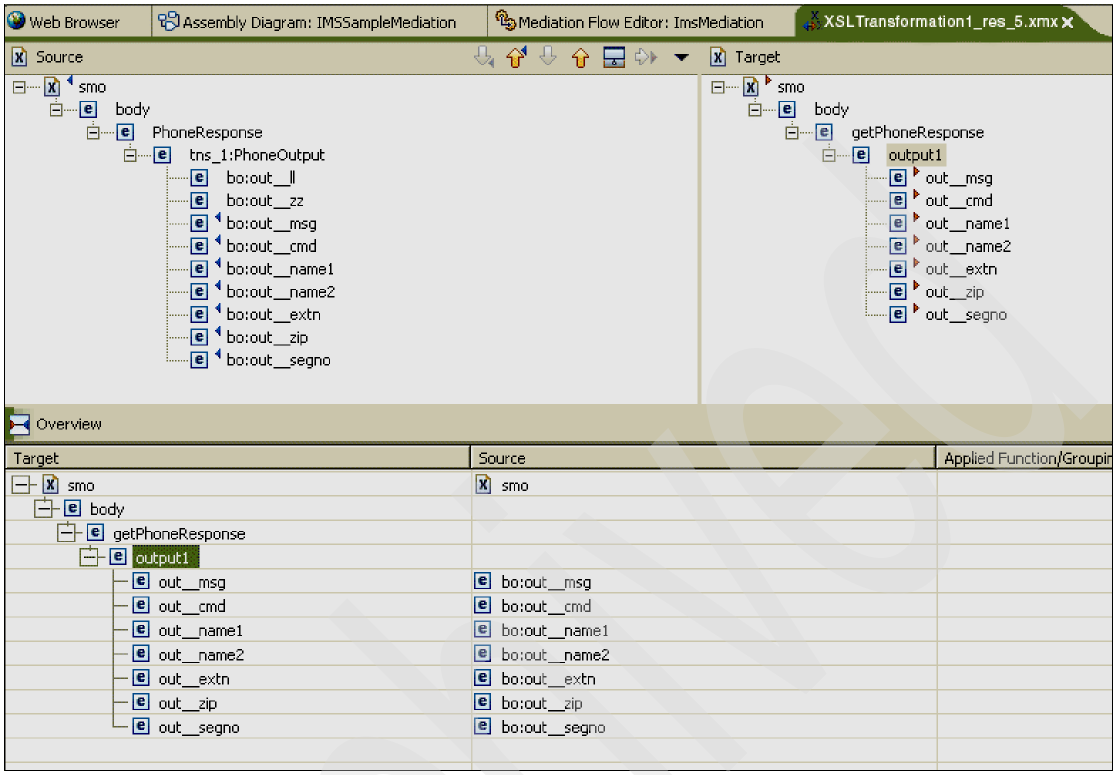


Figure 3-82 XSLT in the request flow

6. Perform the following functions:
   - Change the name of the XSLT node to XSLIMStoWS.
   - Wire the tail of Response-Phone:PhoneBookPartner to this node
   - Wire this node's *out* terminal to the Invoke-Phone:PhoneBookService.
   - Click the XSLT node to open it. In the right bottom of the workbench window, under Properties, select the details. An error message indicates that the Mapping File cannot be empty.
   - Right-click **Mapping file** and click **New** to build the required mapping. Accept the new window and build the mapping for the output messages by creating direct mappings between equally named fields.

   Remember that the output from IMS contains out-ll, out-zz fields that you are not going to carry to the SOAP/HTTP interface.

Figure 3-83 shows the result of the request mapping.


Web Browser
Assembly Diagram: IMSSampleMediation
Mediation Flow Editor: ImsMediation
XSLTransformation1_res_5.xmx
Source
Target
Overview

Source:

```
smo
  body
    PhoneResponse
      tns_1:PhoneOutput
        bo:out__ll
        bo:out__zz
        bo:out__msg
        bo:out__cmd
        bo:out__name1
        bo:out__name2
        bo:out__extn
        bo:out__zip
        bo:out__segno
```

Target:

```
smo
  body
    getPhoneResponse
      output1
        out__msg
        out__cmd
        out__name1
        out__name2
        out__extn
        out__zip
        out__segno
```

Target	Source	Applied Function/Groupir
smo	smo	
body		
getPhoneResponse		
output1		
out__msg	bo:out__msg	
out__cmd	bo:out__cmd	
out__name1	bo:out__name1	
out__name2	bo:out__name2	
out__extn	bo:out__extn	
out__zip	bo:out__zip	
out__segno	bo:out__segno	

Figure 3-83 Result of request mapping

This is the last activity for building this flow. After saving all the elements, you can generate the deployment code.

1. In a business perspective, activate the Physical Resources view by selecting **Window** → **Show View** → **Physical Resources** and open it in the left pane, switching from Business Integration to Physical Resources, as shown in Figure 3-84.

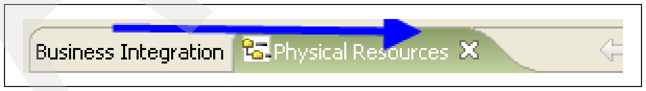


Figure 3-84 Select physical resources

2. Select the IMSSampleMediation project, and in the top menu, click **Project**, as shown in Figure 3-85.

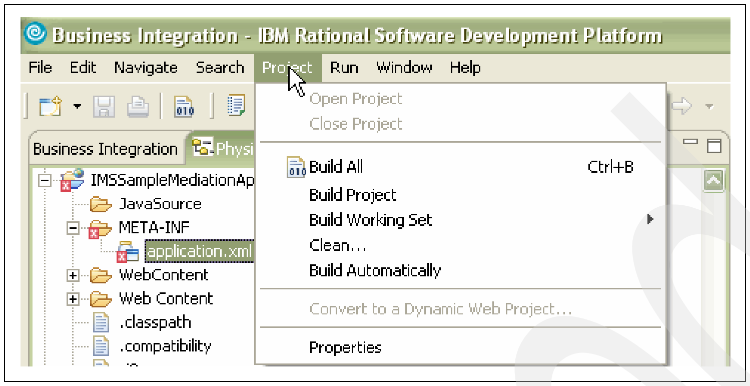
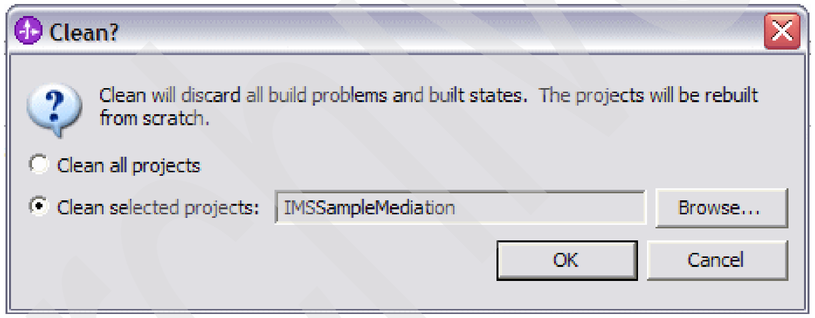


Figure 3-85 Select clean

3. In the drop-down menu, click **Clean**.



Clean will discard all build problems and built states. The projects will be rebuilt from scratch.

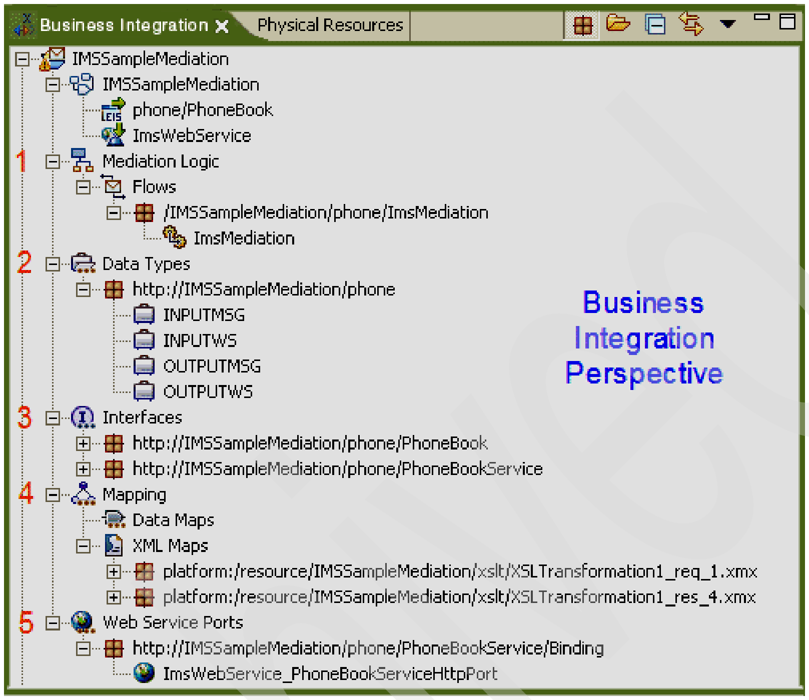
Clean all projects [✓ Clean selected projects:] IMSSampleMediation

Figure 3-86 Cleaning of the deployment code generation project

Be sure to set to **Clean selected project**, as shown in Figure 3-86. This generates the code for several new projects and brings it all together in an *Enterprise Archive* project named IMSSampleMediationApp.

It can be found in the *Project Explorer* view, which is, per default, shown on the left-hand upper side of the J2EE perspective. This is the project that has to be deployed on the WebSphere Application Server.

Take a look at the Mediation Module project, which was the origin of the generated code, as shown in Figure 3-87.



```
Business Integration ✕   Physical Resources
⊟ IMSSampleMediation
   ⊟ IMSSampleMediation
        phone/PhoneBook
        ImsWebService
1  ⊟ Mediation Logic
      ⊟ Flows
         ⊟ /IMSSampleMediation/phone/ImsMediation
              ImsMediation
2  ⊟ Data Types
      ⊟ http://IMSSampleMediation/phone
           INPUTMSG
           INPUTWS
           OUTPUTMSG
           OUTPUTWS
3  ⊟ Interfaces
      ⊞ http://IMSSampleMediation/phone/PhoneBook
      ⊞ http://IMSSampleMediation/phone/PhoneBookService
4  ⊟ Mapping
        Data Maps
      ⊟ XML Maps
         ⊞ platform:/resource/IMSSampleMediation/xslt/XSLTransformation1_req_1.xmx
         ⊞ platform:/resource/IMSSampleMediation/xslt/XSLTransformation1_res_4.xmx
5  ⊟ Web Service Ports
      ⊟ http://IMSSampleMediation/phone/PhoneBookService/Binding
           ImsWebService_PhoneBookServiceHttpPort
```

Figure 3-87 Module project

In the layout, as shown in Figure 3-87, you can see the subparts of the project:

- Mediation Logic
- Four Data Types, two for each interface
- Two Interfaces: SOAP/HTTP (export) and eis (import)
- Mapping: One for each direction of the flow
- Web service Ports

The design was translated in a set of J2EE artifacts (Servlets, EJBs (Session and Message Driven), and finally everything was bundled in a J2EE application called IMSSampleMediationApp. You can see this in the layout shown in Figure 3-88.

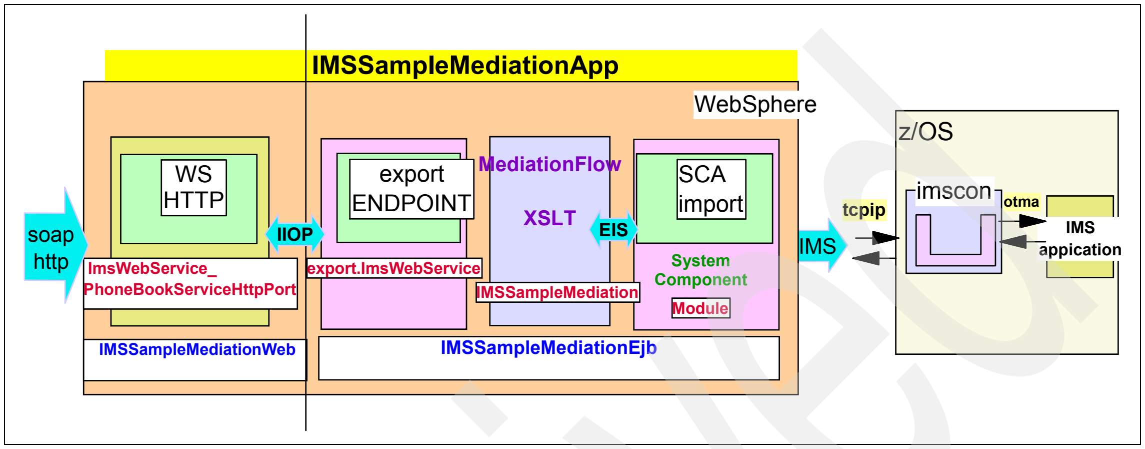


Figure 3-88 Layout of the IMSSampleMediationApp J2EE application

We have the following elements as part of the J2EE application:

- IMSSampleMediationWeb

  `ImsWebService_PhoneBookServiceHttpPort: servlet`

  This is the Web services router for HTTP. It takes the input and, using Internet Inter-ORB Protocol (IIOP), calls the EJB, which is the real logical endpoint for the SOAP service.

- IMSSampleMediationEjb
  - `exportImsWebService: stateless session EJB`

    Logical endpoint for service, called by the router.
  - `IMSSampleMediation: stateless session EJB`

    Contains the mediation logic, and calls the EIS System component.
  - Module

    SCA component with the adapter code for IMS.

Final mediation diagram

Figure 3-89 is a simple representation of the final mediation because it is deployed on the WebSphere Enterprise Service Bus server.

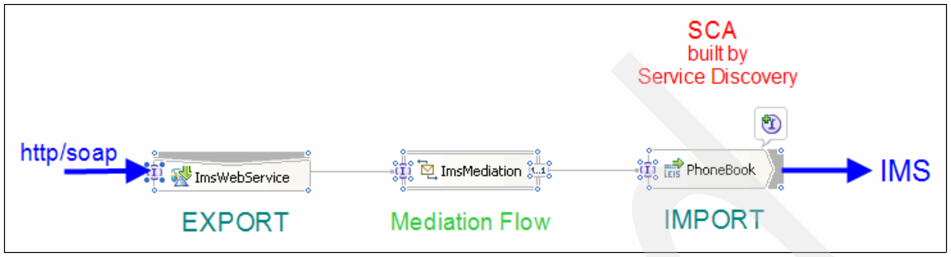


Figure 3-89 Mediation summary

The IMS transaction is accessed through an SCA component, which uses a J2C connector for Java. The interface is SCA and the Business Objects representing the input and output are a reflection of the IMS input and output messages.

The Mediation flow brings two functions:

- A bridge between a SOAP/HTTP interface and the SCA component
- Simplification of the input and output by removing the typical IMS fields

3.5.2 Building a Web services client

The client, in the meaning of Java proxy code, can be built with a WID wizard. To start the wizard, you have to switch to the J2EE perspective.

Perform the following steps:

1. In the left column of the workbench, under Other Projects, navigate to the project ImsSampleMediation, right-click **ImsWebService_PhoneBookServiceHttp_Service.wsdl**, and select **Web Services** → **Generate Client**, as shown in Figure 3-90.

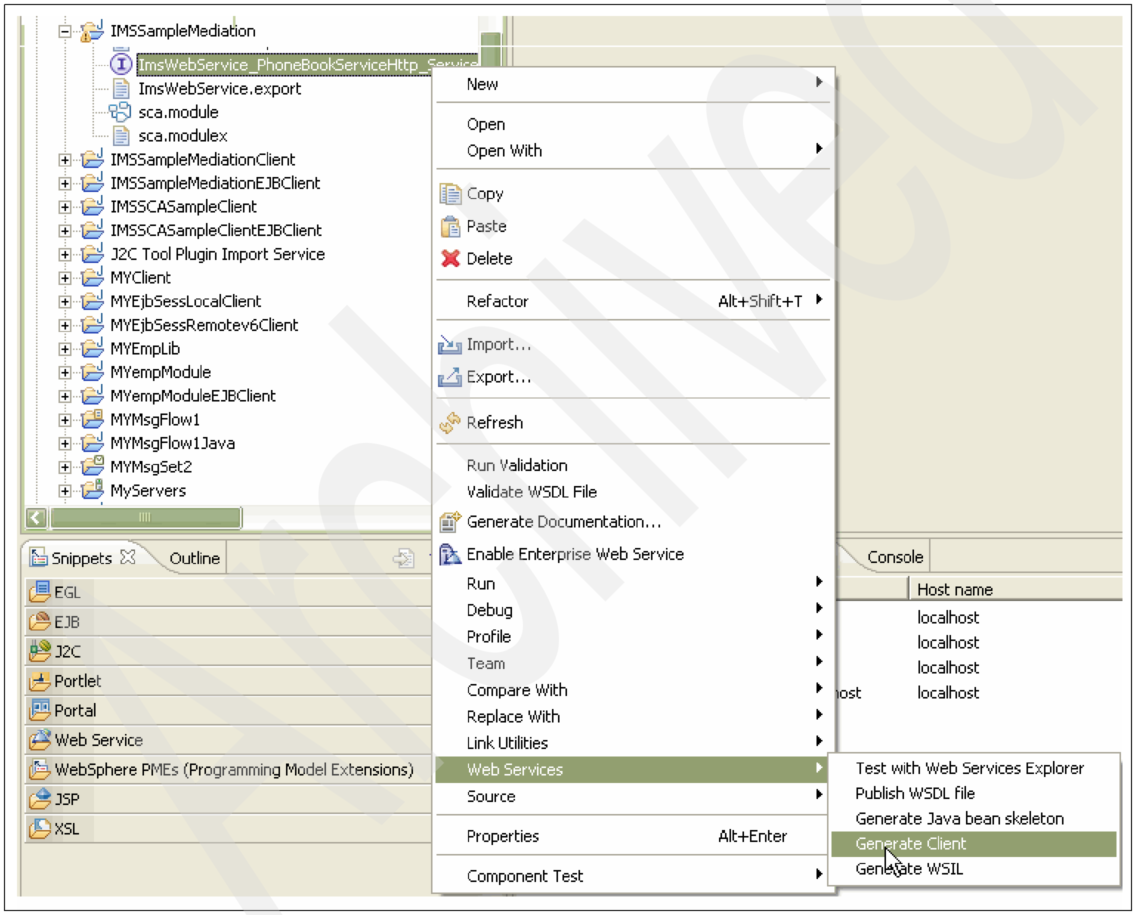
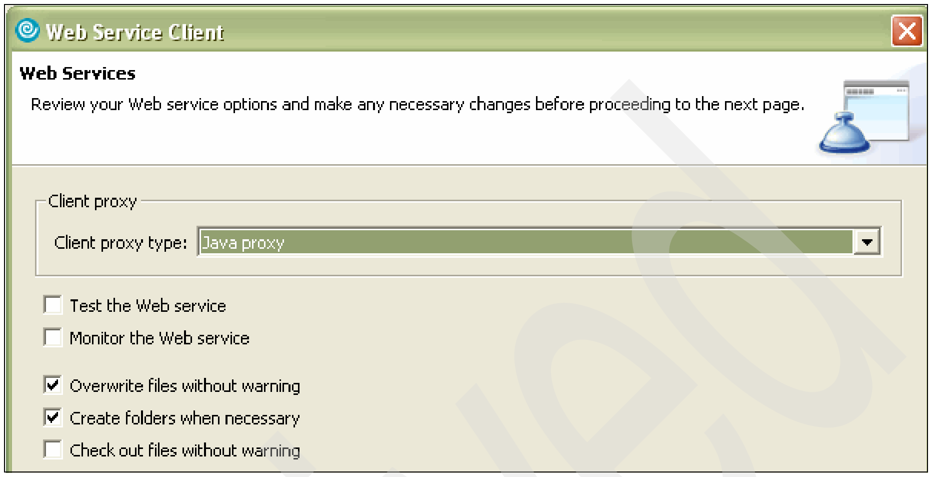


Figure 3-90 Starting the building of the SOAP client

2. You will receive the pop-up shown in Figure 3-91.

   Specify Java proxy in the Client Proxy type field and select **Next**.



Web Service Client

Web Services

Review your Web service options and make any necessary changes before proceeding to the next page.

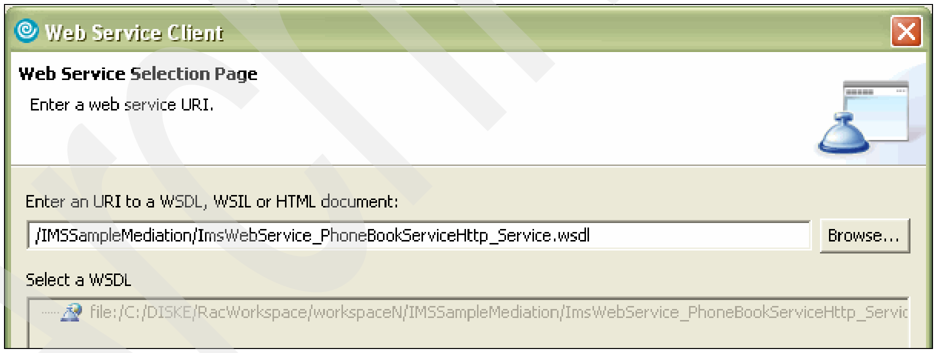
Client proxy

Client proxy type: Java proxy

- [ ] Test the Web service
- [ ] Monitor the Web service
- [x] Overwrite files without warning
- [x] Create folders when necessary
- [ ] Check out files without warning

Figure 3-91 Java proxy

3. Verify that the correct WSDL file is in the selection line and select **Next**, as shown in Figure 3-92.



Web Service Client

Web Service Selection Page

Enter a web service URI.

Enter an URI to a WSDL, WSIL or HTML document:

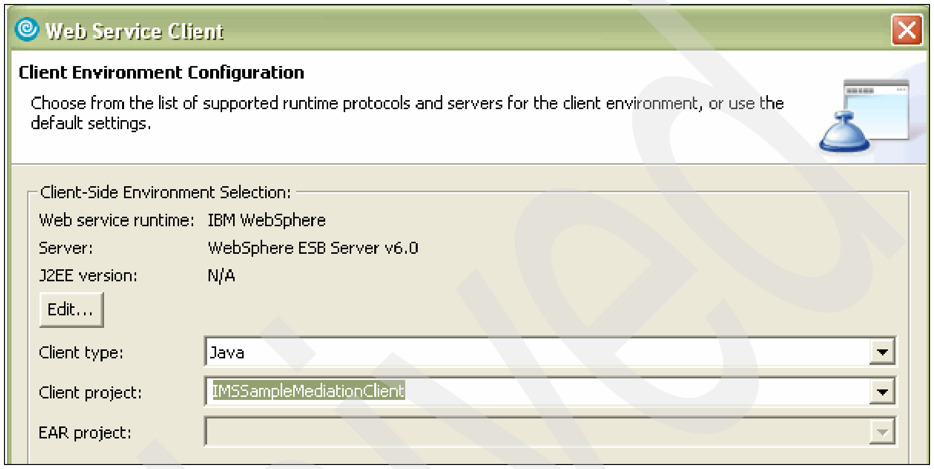
/IMSSampleMediation/ImsWebService_PhoneBookServiceHttp_Service.wsdl

Select a WSDL

file:/C:/DISKE/RacWorkspace/workspaceN/IMSSampleMediation/ImsWebService_PhoneBookServiceHttp_Servic

Figure 3-92 SOAP/HTTP WSDL metafile selection

4. For the client type, you have several options. Depending on the selection, you have to provide additional information such as ProjectName (always), the J2EE Application (EAR) project for WEB, EJB, Application Client, and the appropriate server. In our case, because we selected the Java option, the only input required would be the name of the client project, as shown in Figure 3-93.


Web Service Client

Client Environment Configuration

Choose from the list of supported runtime protocols and servers for the client environment, or use the default settings.

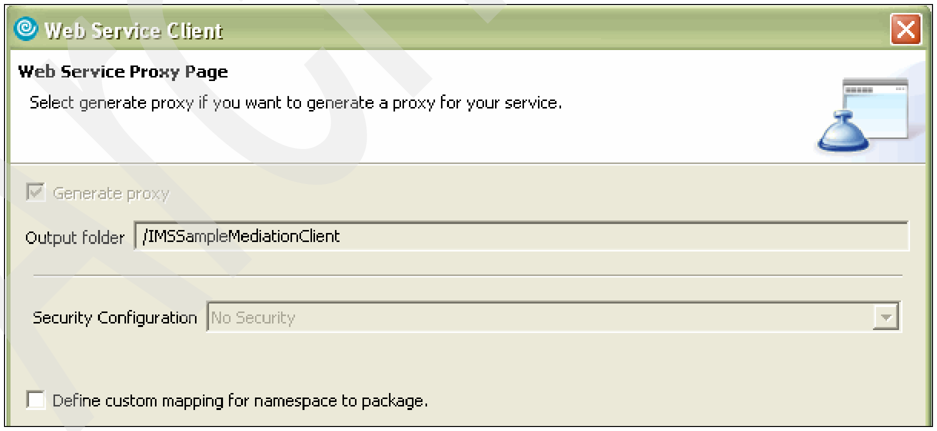
Client-Side Environment Selection:

Web service runtime:	IBM WebSphere
Server:	WebSphere ESB Server v6.0
J2EE version:	N/A

Client type:	Java
Client project:	IMSSampleMediationClient
EAR project:	

Figure 3-93 Client type, project name

5. Specify the Output Folder and then click **Finish** to complete the wizard process, as shown in Figure 3-94.


Web Service Client

Web Service Proxy Page

Select generate proxy if you want to generate a proxy for your service.

☑ Generate proxy

Output folder /IMSSampleMediationClient

Security Configuration No Security

☐ Define custom mapping for namespace to package.

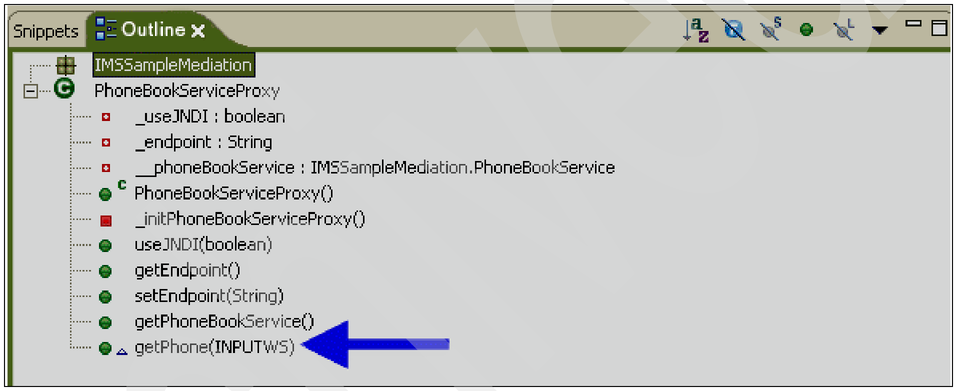
Figure 3-94 Selection of the output folder

At this point, the following Java code is generated and copied in the client project:

- Business Objects INPUTWS and OUTPUTWS.
- Helper, serializer, and deserializer code for each field in the Business Objects.
- Service locator and service accessor classes.
- Service proxy (PhoneBookServiceProxy): The service proxy offers the most simple way to access the SOAP service.

Figure 3-95 shows the outline of this class with the following methods:

- getEndpoint(): Returns the current endpoint (URL).
- setEndpoint(urlString): Can set the URL.
- getPhone(INPUTWS): Invocation of the service.
- useJNDI(boolean): To set the use of the nameSpace lookup (default On).



```
IMSSampleMediation
PhoneBookServiceProxy
    _useJNDI : boolean
    _endpoint : String
    __phoneBookService : IMSSampleMediation.PhoneBookService
    PhoneBookServiceProxy()
    _initPhoneBookServiceProxy()
    useJNDI(boolean)
    getEndpoint()
    setEndpoint(String)
    getPhoneBookService()
    getPhone(INPUTWS)
```

Figure 3-95 Outline of the proxy

You can use this proxy from any JavaCode (Java client or Web client), whether it runs in a J2EE or a non-J2EE environment, with or without the use of the NameSpace for lookup.

3.6 Connecting the WebSphere ESB/ND configuration to a WebSphere MQ queue-sharing group

This section describes how to connect the WebSphere ESB/ND configuration with the WebSphere MQ configuration described throughout Chapter 2, "WebSphere Message Broker configuration & customization" on page 23, which consisted of a queue-sharing group (QSG) named MQRG composed by two MQ queue managers (MQR1, MQR2).

In the ND configuration, we have two servers, augmented with ESB capabilities, combined in a cluster. Each cluster member is deployed on a distinct LPAR (WTSC48 and WTSC52). The cluster has been added to two Service Integration Buses (SIBs). This sample only deals with one bus, SCA.SYSTEM.socell.Bus, that is going to be extended to WebSphere MQ.

3.6.1 Connection between Application Server and MQ QSG

The connection between the two partners happens over an MQLink, which is composed of two channels, comparable to the Message Channels between two MQ queue managers. Both channels are composed of a sender and a receiver.

The MQR1 and MQR2 group listeners are both active at port number 14140, and the group listener ports from both queue managers are mapped by the Sysplex Distributor to a shown in Example 3-21 to activate the listeners.

Example 3-21 Commands to start MQ group listeners

```
-MQR1 START LISTENER INDISP(GROUP) PORT(14140)
-MQR2 START LISTENER INDISP(GROUP) PORT(14140)
```

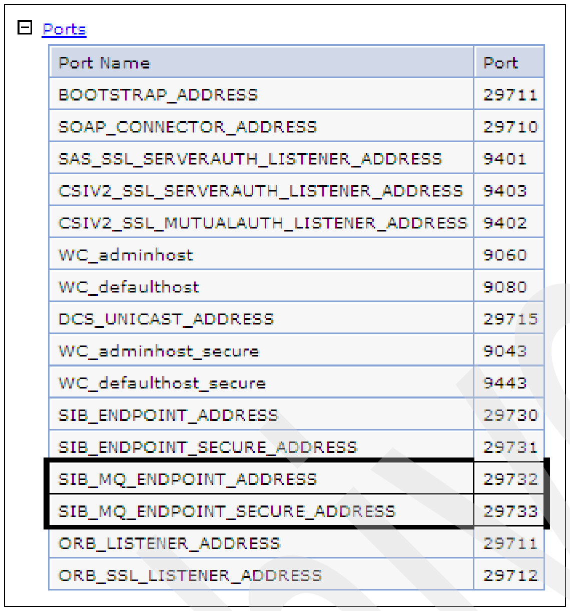
These commands are automatically executed when the Channel Initiator starts. You can use the DISPLAY CHINIT command to verify the status of the listeners, as shown in Example 3-22.

Example 3-22 Extract of the display command to verify the group active group listener

```
-MQR1 DISPLAY CHINIT
CSQM137I -MQR1 CSQMDDQM  DISPLAY CHINIT COMMAND ACCEPTED
CSQX830I -MQR1 CSQXRDQM Channel initiator active
CSQX002I -MQR1 CSQXRDQM Queue-sharing group is MQRG
....
CSQX845I -MQR1 CSQXRDQM TCP/IP system name is TCPIP
CSQX846I -MQR1 CSQXRDQM TCP/IP listener INDISP=QMGR started, 385
for port 1414 address *
CSQX846I -MQR1 CSQXRDQM TCP/IP listener INDISP=GROUP started, 386
for port 14140 address *
CSQX849I -MQR1 CSQXRDQM LU 6.2 listener INDISP=QMGR not started
CSQX849I -MQR1 CSQXRDQM LU 6.2 listener INDISP=GROUP not started
CSQ9022I -MQR1 CSQXCRPS ' DISPLAY CHINIT' NORMAL COMPLETION
```

On the WebSphere Application Server side, we have a similar situation, except that the receiver port is managed by the active Message Engine.

The active Message Engine port is reached through a DVIPA port. The value of this port number can be easily seen by looking at the ports of an Application Server, as shown in Figure 3-96.


Ports

Port Name	Port
BOOTSTRAP_ADDRESS	29711
SOAP_CONNECTOR_ADDRESS	29710
SAS_SSL_SERVERAUTH_LISTENER_ADDRESS	9401
CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS	9403
CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS	9402
WC_adminhost	9060
WC_defaulthost	9080
DCS_UNICAST_ADDRESS	29715
WC_adminhost_secure	9043
WC_defaulthost_secure	9443
SIB_ENDPOINT_ADDRESS	29730
SIB_ENDPOINT_SECURE_ADDRESS	29731
SIB_MQ_ENDPOINT_ADDRESS	29732
SIB_MQ_ENDPOINT_SECURE_ADDRESS	29733
ORB_LISTENER_ADDRESS	29711
ORB_SSL_LISTENER_ADDRESS	29712

Figure 3-96 Application Server (cluster) ports

The ports that you have to use on the WebSphere Application Server server side are 29732 and 297339(ssl). The non-secure port has been defined as a DVIPA, managed by the Sysplex Distributor at IPaddress 9.12.4.108, as shown in Example 3-23.

Example 3-23 DVIPA ports for WebSphere Application Server

```
D TCPIP,,N,VDPT
EZZ2500I NETSTAT CS V1R7 TCPIP 655
DYNAMIC VIPA DESTINATION PORT TABLE:
DEST IPADDR     DPORT DESTXCF ADDR    RDY TOTALCONN  WLM TSR FLG
9.12.4.85       14140 10.1.100.48     001 0000000004 01  100 R
9.12.4.85       14140 10.1.100.52     001 0000000005 01  100 R
9.12.4.108      29732 10.1.100.48     000 0000000000 01  100 R
9.12.4.108      29732 10.1.100.52     001 0000000000 01  100 R
........
9.12.4.108      38100 10.1.100.48     001 0000000000 01  100 R
9.12.4.108      38100 10.1.100.52     001 0000000000 01  100 R
```

In the same display, you can also see that port 14140 is defined for the shared queue group and it is addressable via IP address 9.12.4.85. Port 38100, which is is addressable through IP address 9.12.4.108, is the address for the DB2 DDF address space.

3.6.2 Definitions in WebSphere Application Server

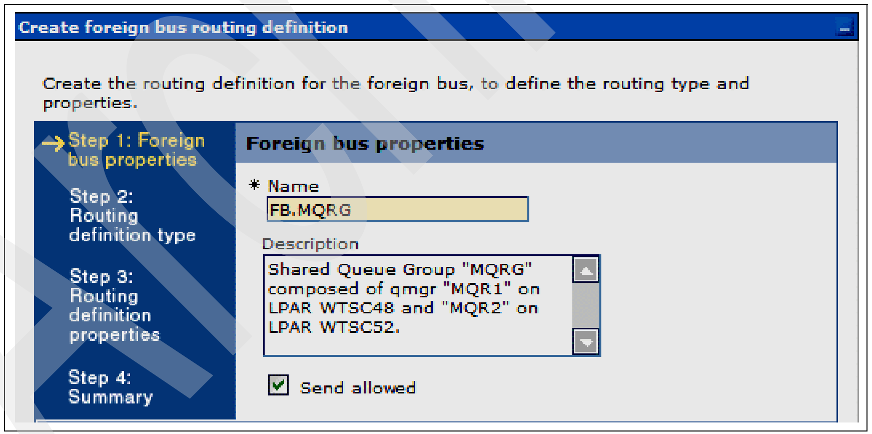
To define the MQLink representing the Foreign Bus, the following two steps have to be performed:

1. Definition of the Foreign Bus
2. Definition of the Link and the channels

Foreign Bus definition

The Foreign Bus to be defined here is the representation of the queue-sharing group MQRG. Perform the following steps:

1. Select **Service Integration** → **Buses** → **SCA.SYSTEM.socell.Bus**. This is the bus where we decided to make the link. Click the busName to open it and under Topology, click **Foreign Buses**.
2. Click **New** for the foreign bus, and follow the definition steps.
3. In Step 1, enter the Name and a description and select **Next**, as shown in Figure 3-97.


Create foreign bus routing definition
Step 1: Foreign bus properties
Step 2: Routing definition type
Step 3: Routing definition properties
Step 4: Summary
Foreign bus properties

Create the routing definition for the foreign bus, to define the routing type and properties.

* Name: FB.MQRG

Description: Shared Queue Group "MQRG" composed of qmgr "MQR1" on LPAR WTSC48 and "MQR2" on LPAR WTSC52.

[✓] Send allowed

Figure 3-97 Step 1 foreign bus

The Name you specify here has an effect on how messages are built that are targeted to queues owned by the queue manager (or, in our case, by the queue-sharing group) that is represented by the Foreign bus defined here.

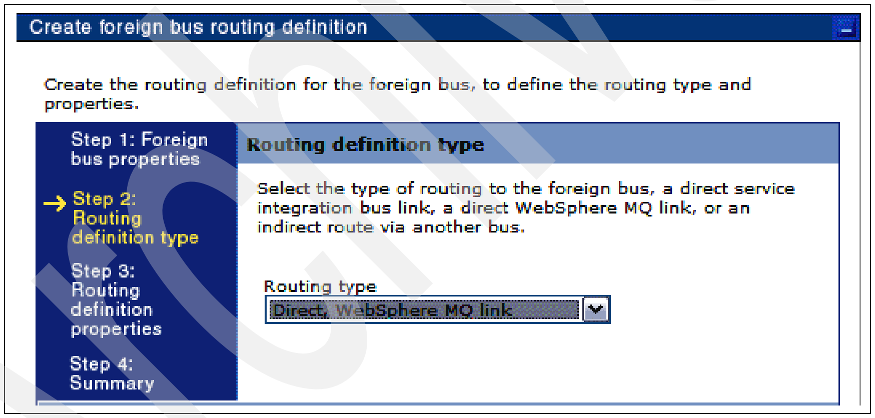
The Name specified here is treated as the target queue manager name for queue destinations and does not explicitly specify a queue manager name.

This is displayed within the messages' transmission header and has to be resolved by the receiving MQ queue manager.

Make note of the following:

- Things are simple if the Foreign Bus name is the same as the MQ queue manager (or QSG) name; no further efforts have to be taken.
- If the Foreign Bus name is different from the MQ queue manager (or QSG) name (as in our case), then on the MQ side, the name has to be resolved by a queue manager alias, which is a Remote Queue definition object with the name chosen for the Foreign Bus. Refer to "Queue Manager Alias" on page 191 for more details.

4. In Step 2, define the routing type. Make sure to specify Direct, WebSphere MQ link in the Routing type field. Select **Next**, as shown in Figure 3-98.


Create foreign bus routing definition

Create the routing definition for the foreign bus, to define the routing type and properties.

Step 1: Foreign bus properties
→ Step 2: Routing definition type
Step 3: Routing definition properties
Step 4: Summary

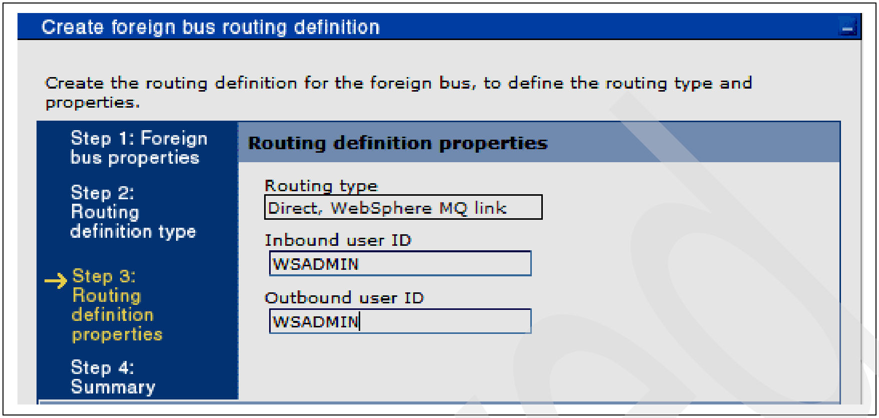
Routing definition type

Select the type of routing to the foreign bus, a direct service integration bus link, a direct WebSphere MQ link, or an indirect route via another bus.

Routing type: Direct, WebSphere MQ link

Figure 3-98 Step 2 foreign bus

5. In Step 3, enter the user IDs that are going to be used for authorization and then click **Next**, as shown in Figure 3-99.


Create foreign bus routing definition

Create the routing definition for the foreign bus, to define the routing type and properties.

Step 1: Foreign bus properties
Step 2: Routing definition type
Step 3: Routing definition properties
Step 4: Summary

Routing definition properties

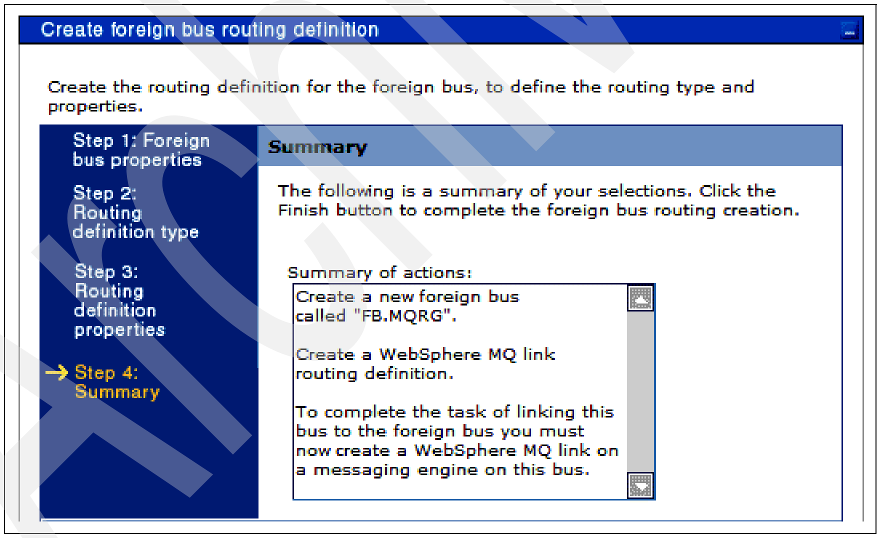
Routing type: Direct, WebSphere MQ link

Inbound user ID: WSADMIN

Outbound user ID: WSADMIN

Figure 3-99 Step 3 foreign bus

6. In the last window, you get a summary of the definitions. Click **Finish** to complete the process, as shown in Figure 3-100.


Create foreign bus routing definition

Create the routing definition for the foreign bus, to define the routing type and properties.

Step 1: Foreign bus properties
Step 2: Routing definition type
Step 3: Routing definition properties
Step 4: Summary

Summary

The following is a summary of your selections. Click the Finish button to complete the foreign bus routing creation.

Summary of actions:

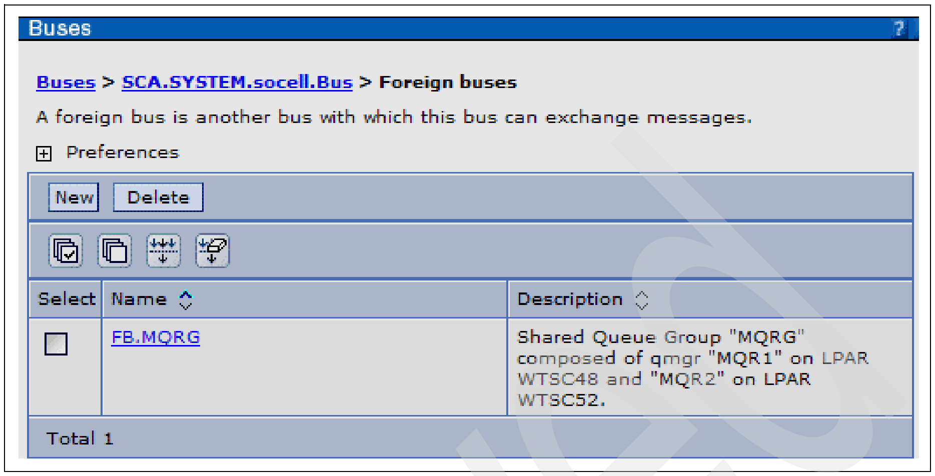
Create a new foreign bus called "FB.MQRG".

Create a WebSphere MQ link routing definition.

To complete the task of linking this bus to the foreign bus you must now create a WebSphere MQ link on a messaging engine on this bus.

Figure 3-100 Summary foreign bus definition

Figure 3-101 shows the foreign bus overview.


Buses

Buses > SCA.SYSTEM.socell.Bus > Foreign buses

A foreign bus is another bus with which this bus can exchange messages.

⊞ Preferences

Select	Name	Description
☐	FB.MQRG	Shared Queue Group "MQRG" composed of qmgr "MQR1" on LPAR WTSC48 and "MQR2" on LPAR WTSC52.

Total 1

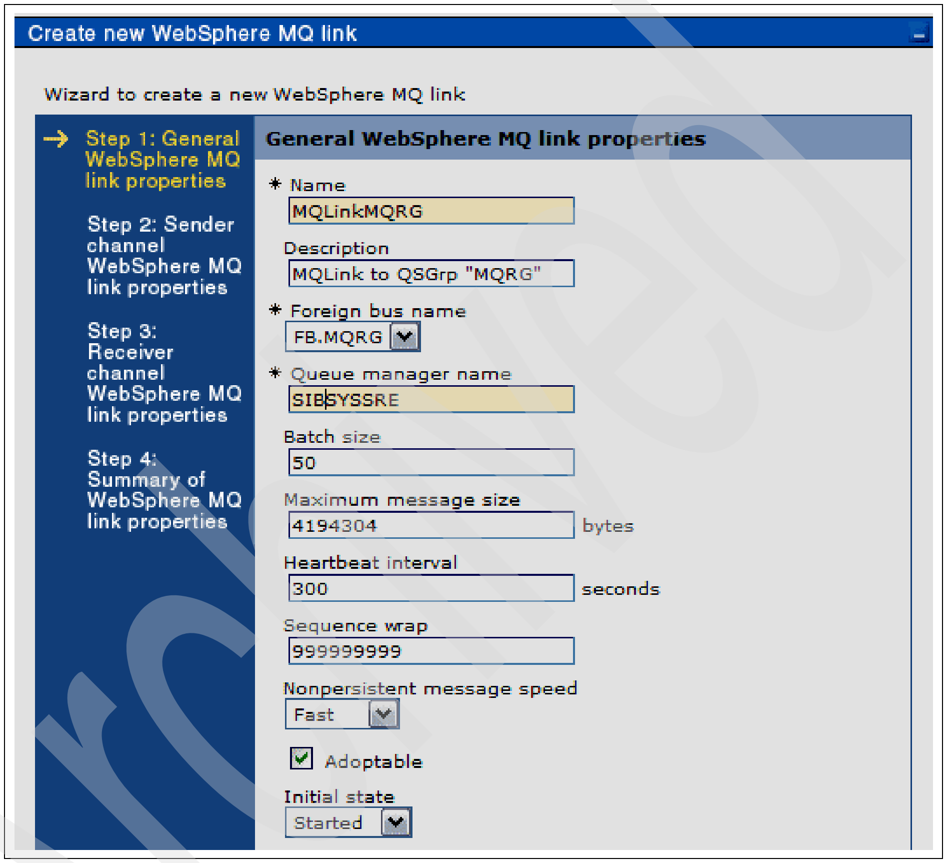
Figure 3-101 Foreign bus overview

Configuring the MQLink

To set this definition, perform the following steps:

1. Select **Service Integration** → **Buses** → **SCA.SYSTEM.socell.Bus**. Click the **busName** to open it and under Topology, click **Message engines**.
2. This bus has only one set of Message Engines defined. Click **socluse1.000-SCA.SYSTEM.socell.Bus** to open it and under Additional properties, click **WebSphere MQ Links**.
3. Click **New** for the MQLink, and follow the definition steps.

4. In Step 1, select a name for the MQLink and reference a foreign bus. Make sure that the value of the field Queue manager name is correctly filled (this bus is known by the Qmgr name to other external qmgrs). Figure 3-102 shows the relevant window.
5. Go to **Next**.


Create new WebSphere MQ link

Wizard to create a new WebSphere MQ link

→ Step 1: General WebSphere MQ link properties

Step 2: Sender channel WebSphere MQ link properties

Step 3: Receiver channel WebSphere MQ link properties

Step 4: Summary of WebSphere MQ link properties

General WebSphere MQ link properties

* Name	MQLinkMQRG
Description	MQLink to QSGrp "MQRG"
* Foreign bus name	FB.MQRG
* Queue manager name	SIBSYSSRE
Batch size	50
Maximum message size	4194304 bytes
Heartbeat interval	300 seconds
Sequence wrap	999999999
Nonpersistent message speed	Fast
Adoptable	✓
Initial state	Started

Figure 3-102 Step 1 MQLink definition

6. Step 2 requires the following definitions:
   - Channel name for the sender: We chose SIBSYSSRE.TO.MQRG as the channel name, following the common MQ practice to name the two channel partners and the direction of the message flow.

     The same name has to be specified on the MQ side as the name of the receiver channel.

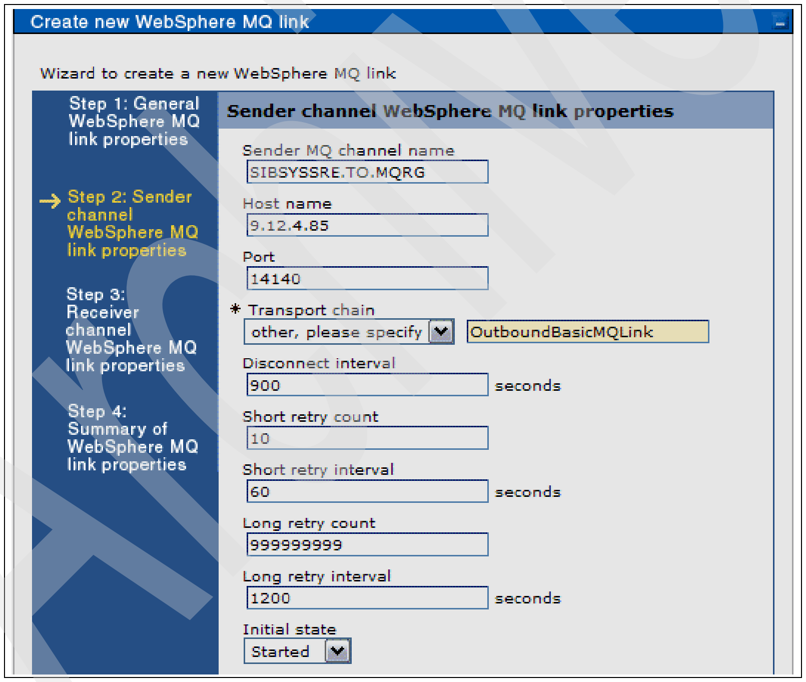
Note: The MQ partner is not a single MQ queue manager, but the queue-sharing group is. This is supported by the use of shared channels on the MQ QSG side.

– The distributed DVIPA and port to generically address the QSG group listeners, which is 9.12.4.85(14140), as the settings for Hostname and Port.

– The *Transport chain*. Refer to APAR PK10576: “Setup MQLink on clustered Messaging Engine (ME) using admin console found no transport chains listed” for an explanation.

– The *Disconnect interval*.

The sender channel is activated, when a message is ready to be sent. After an inactivity period of length, specified here, it reverts to an “standby” state, as shown in Figure 3-103.


Create new WebSphere MQ link
Wizard to create a new WebSphere MQ link
Step 1: General WebSphere MQ link properties
Step 2: Sender channel WebSphere MQ link properties
Step 3: Receiver channel WebSphere MQ link properties
Step 4: Summary of WebSphere MQ link properties

Sender channel WebSphere MQ link properties

Sender MQ channel name	SIBSYSSRE.TO.MQRG
Host name	9.12.4.85
Port	14140
* Transport chain	other, please specify · OutboundBasicMQLink
Disconnect interval	900 seconds
Short retry count	10
Short retry interval	60 seconds
Long retry count	999999999
Long retry interval	1200 seconds
Initial state	Started

Figure 3-103 Step 2 MQLink definition

Note: If you try to perform the steps listed in the following Web site to set up an MQLink using the admin console on a Message Engine, which is part of a cluster and at the cluster scope, then you will not find transport chains listed in the drop-down:

`http://publib.boulder.ibm.com/infocenter/ws60help/index.jsp?topic=/com.ibm.websphere.pmc.nd.doc/tasks/tjc0001_.html`

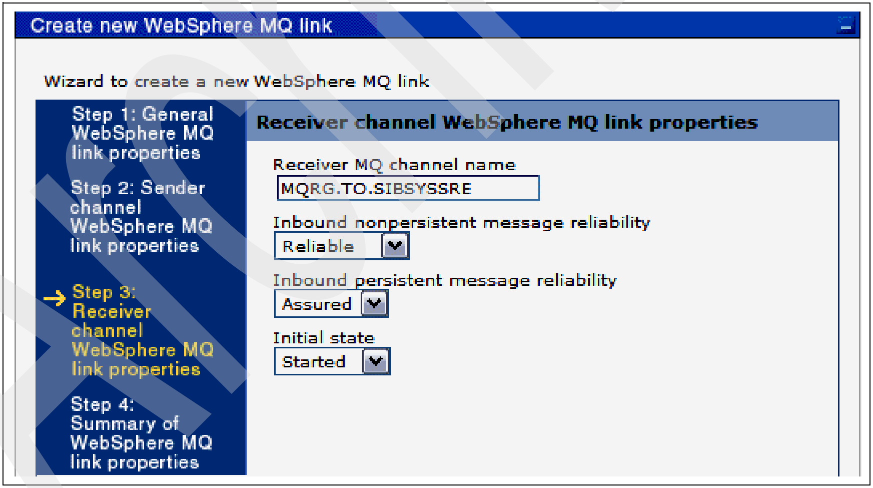
The admin console drop-down box shows only one entry.

Local Fix: Selecting the **other, please specify** entry brings up an edit box where the user can type in a value. This allows the user to enter either one of the following default transport chains:

- OutboundBasicMQLink
- OutboundSecureMQLink

7. Step 3 requires the description for the Receiver channel, for which you have to specify a name. In line with the naming rule applied to the sender, we name it MQRG.TO.SIBSYSSRE.

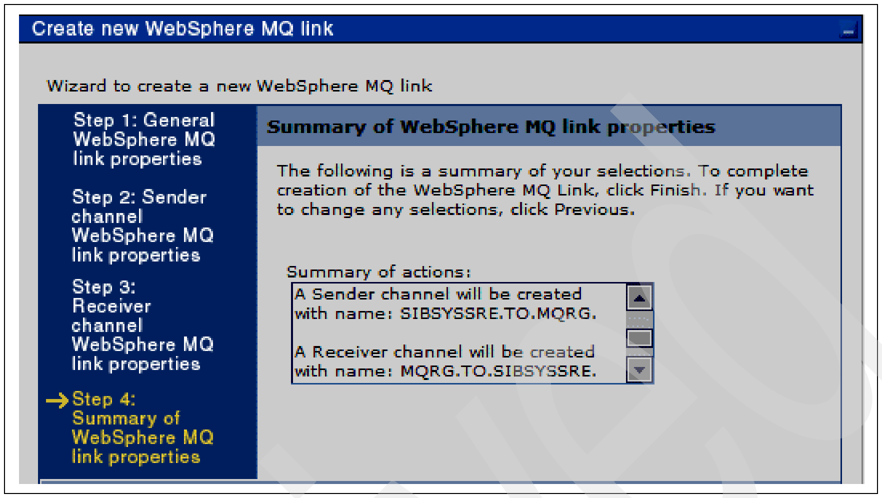
   On the MQ side, this name has to be used as the name of the sender channel, as shown in Figure 3-104.


Create new WebSphere MQ link
Wizard to create a new WebSphere MQ link
Step 1: General WebSphere MQ link properties
Step 2: Sender channel WebSphere MQ link properties
Step 3: Receiver channel WebSphere MQ link properties
Step 4: Summary of WebSphere MQ link properties
Receiver channel WebSphere MQ link properties

Receiver MQ channel name: MQRG.TO.SIBSYSSRE
Inbound nonpersistent message reliability: Reliable
Inbound persistent message reliability: Assured
Initial state: Started

Figure 3-104 Step 3 MQLink definition

8. Step 4 is a Summary of the definition. Click **Finish** to complete the process, as shown in Figure 3-105.



Create new WebSphere MQ link

Wizard to create a new WebSphere MQ link

Step 1: General WebSphere MQ link properties
Step 2: Sender channel WebSphere MQ link properties
Step 3: Receiver channel WebSphere MQ link properties
Step 4: Summary of WebSphere MQ link properties

Summary of WebSphere MQ link properties

The following is a summary of your selections. To complete creation of the WebSphere MQ Link, click Finish. If you want to change any selections, click Previous.

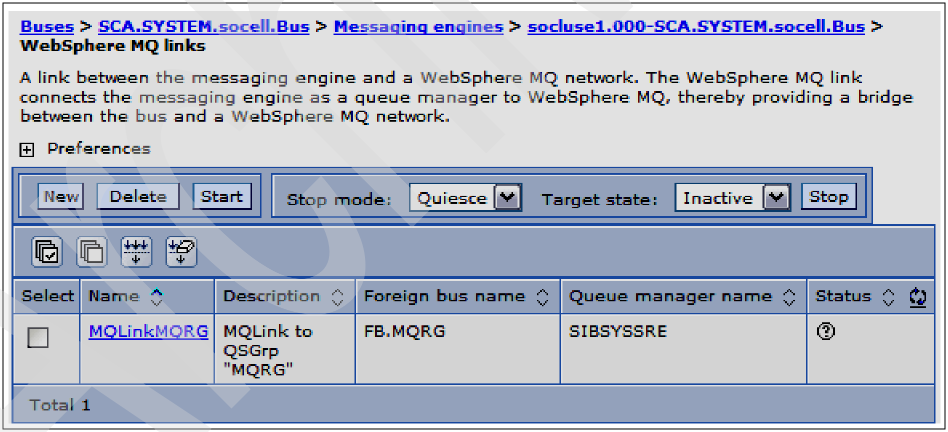
Summary of actions:

A Sender channel will be created with name: SIBSYSSRE.TO.MQRG.

A Receiver channel will be created with name: MQRG.TO.SIBSYSSRE.

Figure 3-105 Summary of MQLink definition

Figure 3-106 shows the MQLink definition status.



Buses > SCA.SYSTEM.socell.Bus > Messaging engines > socluse1.000-SCA.SYSTEM.socell.Bus > **WebSphere MQ links**

A link between the messaging engine and a WebSphere MQ network. The WebSphere MQ link connects the messaging engine as a queue manager to WebSphere MQ, thereby providing a bridge between the bus and a WebSphere MQ network.

Select	Name	Description	Foreign bus name	Queue manager name	Status
☐	MQLinkMQRG	MQLink to QSGrp "MQRG"	FB.MQRG	SIBSYSSRE	?

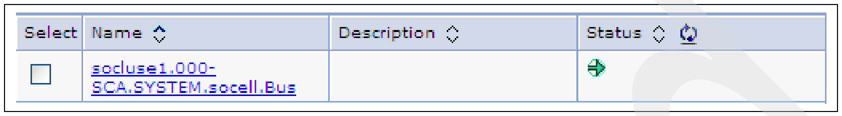
Total 1

Figure 3-106 MQLink definition status

To have all elements in a correct *started* status, the cluster has to be stopped and started to pick up the new definition.

You can check the following to verify that everything started:

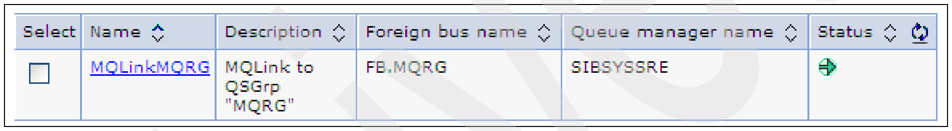
- After a restart of the cluster, verify that both servers and cluster members came up correctly.
- Verify that the status of the message engine socluse1.000-SCA.SYSTEM.socell.Bus on the bus SCA.SYSTEM.socell.Bus is green, as shown in Figure 3-107.



Select	Name	Description	Status
☐	socluse1.000-SCA.SYSTEM.socell.Bus		➜

Figure 3-107 Status of Message Engine

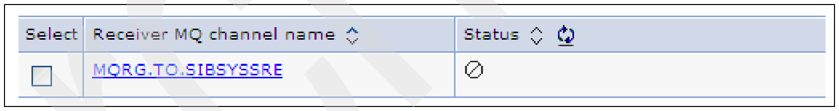
- Verify that the status of the WebSphere MQ link called MQLinkMQRG is green, as shown in Figure 3-108.



Select	Name	Description	Foreign bus name	Queue manager name	Status
☐	MQLinkMQRG	MQLink to QSGrp "MQRG"	FB.MQRG	SIBSYSSRE	➜

Figure 3-108 Status of MQLink

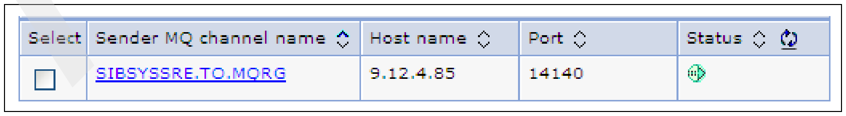
- Verify the status of the Receiver MQRG.TO.SIBSYSSRE. This must be started, but the final status depends on the partner in MQ, which takes the initiative on the sender side, as shown in Figure 3-109.



Select	Receiver MQ channel name	Status
☐	MQRG.TO.SIBSYSSRE	⊘

Figure 3-109 Status of Receiver channel

- Verify the status of the Sender channel SIBSYSSRE.TO.MQRG. You must start this, but the final status depends on the activity. This side takes the initiative, although after an inactivity period, for which the duration was specified during definition, the channel reverts to a status standby on the sender side, as shown in Figure 3-110.



Select	Sender MQ channel name	Host name	Port	Status
☐	SIBSYSSRE.TO.MQRG	9.12.4.85	14140	➜

Figure 3-110 Status of Sender channel

3.6.3 Defining an MQ queue to WebSphere ESB

This section is added to complete the documentation of how to connect WebSphere ESB's Message Engine to the MQ queue-sharing group. The actual definitions used by our samples are documented in Chapter 4, "Sample scenarios" on page 195.

This section describes, in general, how an existing MQ queue has to be defined from a WebSphere ESB configuration object.

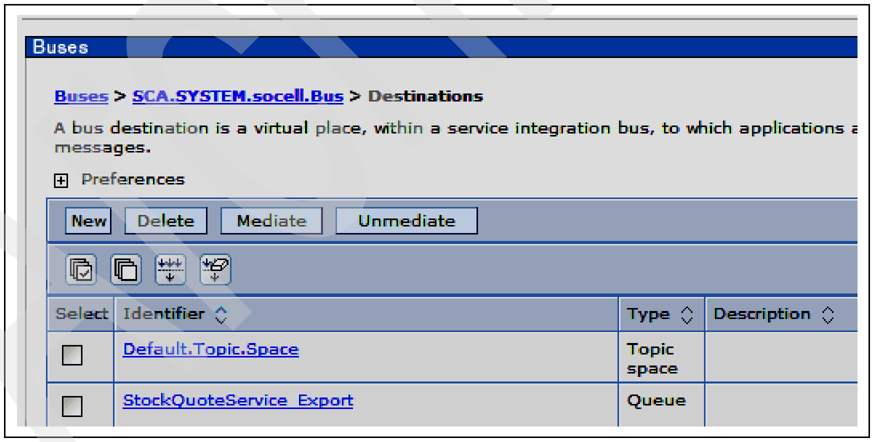
Our configuration sample refers to an MQ queue named Sample.FlowInputQ, which is a shared queue owned by the queue-sharing group MQRG.

Defining a foreign destination to WebSphere ESB

The MQ queue has to be defined to WebSphere ESB as a foreign bus destination. A bus destination is a virtual place, within a service integration bus, to which applications attach as producers, or consumers, or both, to exchange messages. In our case, it is a foreign bus destination, because it denotes a destination owned by a foreign bus, which is our MQ queue-sharing group.

Perform the following steps:

1. In the WebSphere ESB admin console, start with selecting **Service Integration** → **Buses** → **SCA.SYSTEM.socell.Bus**, as shown in Figure 3-111, and click **New** to define a new destination object.

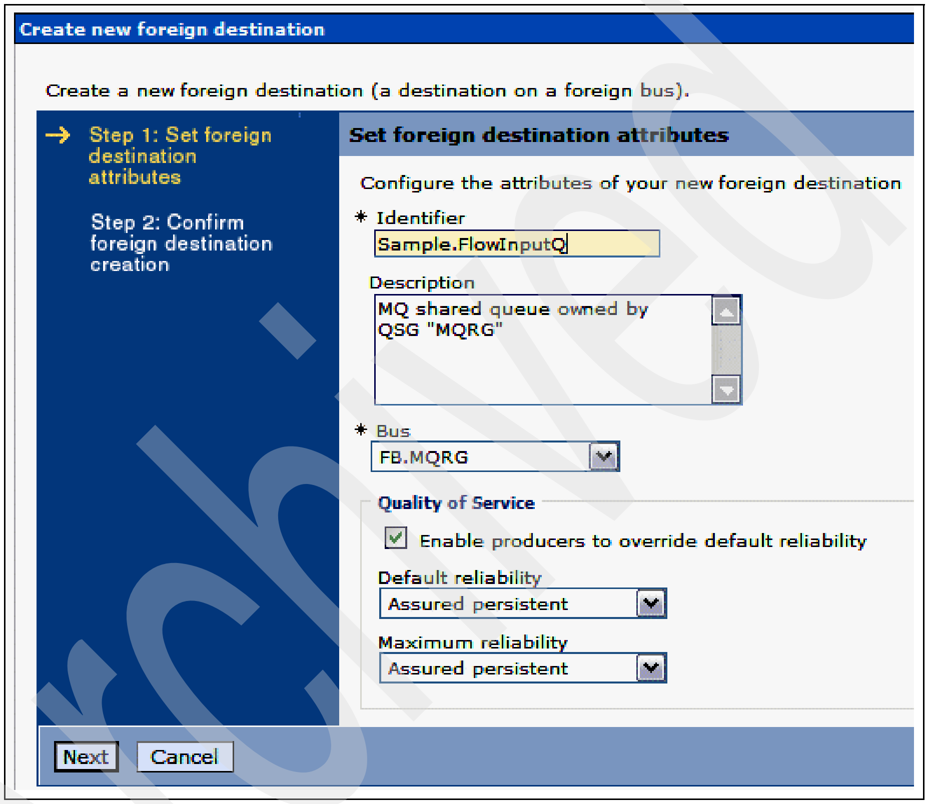


A bus destination is a virtual place, within a service integration bus, to which applications a
messages.

Select	Identifier	Type	Description
☐	Default.Topic.Space	Topic space	
☐	StockQuoteService_Export	Queue	

Figure 3-111 WebSphere ESB console: bus destinations definition point

2. Select the **Foreign** destination type in the next window (not shown here) and click **Next**, which takes you to the Set foreign destination attributes window.
3. In the Set foreign destination attributes window, specify the name of the destination (which is the name of the queue in our case), and the foreign bus that owns the resource, which in our case is FB.MQRG, as shown in Figure 3-112. Accept the defaults for the other options and select **Next**.


Create new foreign destination

Create a new foreign destination (a destination on a foreign bus).

Step 1: Set foreign destination attributes

Step 2: Confirm foreign destination creation

Set foreign destination attributes

Configure the attributes of your new foreign destination

* Identifier: Sample.FlowInputQ

Description: MQ shared queue owned by QSG "MQRG"

* Bus: FB.MQRG

Quality of Service

[✓] Enable producers to override default reliability

Default reliability: Assured persistent

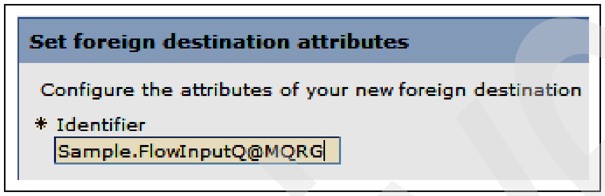
Maximum reliability: Assured persistent

Figure 3-112 Foreign destination attribute definition

4. A confirmation window opens that summarizes the options chosen. Click **Finish** to complete the definitions.

Note: If the MQ queue is defined this way, messages sent to this destination carry the (foreign) Bus name specified as the target queue manager name within their transmission header. If this is *not* the name of the queue manager represented by the foreign bus or the name of a transmission queue owned by that queue manager, then this name has to be resolved by a queue manager alias definition object, as described in "Queue Manager Alias" on page 191.

5. Alternatively, the MQ queue can be named by a fully qualified name, as shown in Figure 3-113.


Set foreign destination attributes
Configure the attributes of your new foreign destination

* Identifier

Sample.FlowInputQ@MQRG

Figure 3-113 Fully qualified destination (queue) name

In this case, MQRG is used as the target queue manager name in the messages' transmission header, and, because this is a valid queue manager name accepted by any member of the MQRG queue-sharing group, it does not require additional resolution. In this case, no queue manager alias object is required.

Note: However, this queue name might be mapped by a regular QREMOTE definition object in the receiving queue managers to a different queue name, thus making the WebSphere ESB *Identifier* name some sort of an alias name.

6. Make sure the newly defined queue (destination) opens in the list of known destinations.

3.6.4 Definitions in WebSphere MQ

The WebSphere Application Server message engine appears to MQ as another queue manager. Thus, connectivity setup within MQ is just the same as for any other remote queue manager, which means that we have to create the following:

- A transmission queue
- A sender channel, triggered from this transmission queue
- A receiver channel
- A queue manager alias (QRemote) object, because the WebSphere ESB name of the foreign bus is not equal to the queue-sharing group name

To exploit sysplex and queue-sharing group high availability functions, all these items are created as shared objects, as described in 2.2.2, "An outline of WebSphere MQ queue sharing groups" on page 33.

In our case, we created:

- A *shared transmission queue*, which we called the same name as the queue manager name assigned to the MQLink, which is SIBSYSSRE, with trigger settings to automatically start the sender channel named next.
- A *shared sender channel* as a QSG object named MQRG.TO.SIBSYSSRE, which references the transmission queue mentioned previously. The sender name has to match one of the WebSphere Application Server MQLink's receiver channel definitions, as shown in Figure 3-103 on page 179.
- A *shared receiver channel* as a QSG object named SIBSYSSRE.TO.MQRG. The sender name has to match one of the WebSphere Application Server MQLink's sender channel definition, as shown in Figure 3-104 on page 180.

Example 3-24 shows the definition commands for MQ objects to support MQLink connection.

Example 3-24 Definition commands for MQ objects to support MQLink connection

```
DEFINE QLOCAL('SIBSYSSRE')                      -
 QSGDISP(SHARED) USAGE(XMITQ)                   -
 DESCR('SIB/MQLink transmission queue')         -
 CFSTRUCT('APPLICATION1')                       -
 MAXMSGL(64512)                                 -
 TRIGGER                                        -
   INITQ('SYSTEM.CHANNEL.INITQ')                -
   TRIGTYPE(FIRST) TRIGDPTH(1)                  -
   TRIGDATA('MQRG.TO.SIBSYSSRE')                -

DEFINE CHANNEL('MQRG.TO.SIBSYSSRE')                   -
 CHLTYPE(SDR) QSGDISP(COPY)                           -
 DESCR('Shared SDR channel to WAS Msg Engine')        -
 XMITQ('SIBSYSSRE')                                   -
 TRPTYPE(TCP)                                         -
 CONNAME('9.12.4.108(29732)')                         -

DEFINE CHANNEL('SIBSYSSRE.TO.MQRG')                        -
 CHLTYPE(RCVR) QSGDISP(COPY)                               -
 DESCR('Shared RCVR chl for msgs from WAS to MQLink')      -
 TRPTYPE(TCP)                                              -
```

Verification of the objects

It is good practice to verify the objects created to make sure they exist with the correct names and attributes:

1. Invoke the MQ ISPF main menu (by using the TSO CSQOREXX command or an appropriate ISPF menu option, as configured by your installation), and verify the basic settings, as shown in Example 3-25.

Example 3-25 MQ ISPF panels basic display settings

```
IBM WebSphere MQ for z/OS - Main Menu

 Complete fields. Then press Enter.

 Action  . . . . . . . . . . 1     0. List with filter   4. Manage
                                   1. List or Display    5. Perform
                                   2. Define like        6. Start
                                   3. Alter              7. Stop
 Object type . . . . . . . .              +
 Name  . . . . . . . . . . .
 Disposition . . . . . . . . A  Q=Qmgr, C=Copy, P=Private, G=Group,
                                S=Shared, A=All

 Connect name  . . . . . . . MQR1  - local queue manager or group
 Target queue manager  . . . MQR1
           - connected or remote queue manager for command input
 Action queue manager  . . . *     - command scope in group
 Response wait time  . . . . 15    5 - 999 seconds
```

2. Crucial settings are as follows:
   - Disposition=A for *all* objects.
   - Action queue manager=* to include any active QSG queue manager.
3. Display the newly defined transmission queue, specifying, in the named fields of the MQ ISPF Main Menu:
   - Object type .... QL
   - Name .... SIB*

Example 3-26 shows the result.

Example 3-26 Shared transmission queue result display

```
Get Usage
     Name                 Disposition    Put  Trig
 <>  SIB*                 ALL     MQRG
     SIBSYSSRE            SHARED        YY  X YF
           ******** End of list ********
```

4. Display the newly defined sender channel, specifying, in the named fields of the MQ ISPF Main Menu:

   – Object type .... CHL

   – Name .... MQRG.TO.SIB*

   Example 3-27 shows the results.

Example 3-27 QSG-to-MQLink sender result display

```
Name                Type       Disposition   Status
<>  MQRG.TO.SIB*        CHANNEL    ALL     MQRG
    MQRG.TO.SIBSYSSRE   SENDER     COPY    MQR1  INACTIVE
    MQRG.TO.SIBSYSSRE   SENDER     COPY    MQR2  INACTIVE
    MQRG.TO.SIBSYSSRE   SENDER     GROUP
           ******** End of list ********
```

Note that there are three objects with the same name, one being marked as the group object, and the other two representing the real channels as defined to the two QSG queue managers.

5. Display the newly defined receiver channel, specifying the following, in the named fields of the MQ ISPF Main Menu:

   – Object type .... CHL

   – Name .... SIB*

   Example 3-28 shows the results.

Example 3-28 MQLink-to-MQ receiver channel

```
Name                Type        Disposition   Status
<>  SIB*                CHANNEL     ALL     MQRG
    SIBSYSSRE.TO.MQRG   RECEIVER    COPY    MQR1  INACTIVE
    SIBSYSSRE.TO.MQRG   RECEIVER    COPY    MQR2  INACTIVE
    SIBSYSSRE.TO.MQRG   RECEIVER    GROUP
                ******** End of list ********
```

Verify the connectivity

After the channels are set up on both the WebSphere Application Server/MQLink side and the MQ side, you must be able to start them. With a sender-receiver configuration, a channel can only be started from the sender side. From MQ, you can check the channel that serves outbound messages, for example, messages that are sent from MQ to the MQLink, by starting the shared sender channel manually.

The following is an example about how you can accomplish this task by using the MQ ISPF windows:

1. Display the sender channel by specifying its name in the named fields of the MQ ISPF Main Menu:
   - Object type .... CHL
   - Name .... MQRG.TO.SIB*
2. In the screen shown in Example 3-27 on page 189, type a `6` (which means start) in the left-most column of the MQR1's sender channel instance:

   ```
   6   MQRG.TO.SIBSYSSRE   SENDER     COPY   MQR1  INACTIVE
   ```

3. Press the Enter key, and in the following screen, enter an `A` in the Disposition field, and then press Enter again.

   If everything works fine and the channel is able to start, a pop-up screen opens that looks similar to Example 3-29.

   Example 3-29 Messages on successful start of sender channel

   ```
                   Display messages                         Row 1 of 3

   CSQN138I -MQR1 'START CHANNEL' command generated for CMDSCOPE(MQR1),

   sent to 1

   CSQ9022I -MQR1 CSQXCRPS ' START CHANNEL' NORMAL COMPLETION
   ```

   At the same time, a system log message is issued that reads:

   ```
   +CSQX500I -MQR1 CSQXRCTL Channel MQRG.TO.SIBSYSSRE started
   ```

   When you go back (using F12) to the Display sender screen and refresh its contents by pressing F5, you will see the MQR1 channel instance status as RUN.

4. A running channel can be manually stopped with option 7 applied to the channel in RUN state within the display result screen. Again, the Disposition field has to be set to `A` for the shared option.

5. If ISPF MQ panels were not available or native command input is preferred, then you can use the following command to start and stop shared channels, where you can issue the commands to either of the QSG queue managers:
   - Start Channel:
     ```
     START CHL(MQRG.TO.SIBSYSSRE) CHLDISP(SHARED)
     ```
   - Stop Channel:
     ```
     STOP CHL(MQRG.TO.SIBSYSSRE) CHLDISP(SHARED) STATUS(INACTIVE)
     ```

   When you use MQSC commands:
   - The channel names have to be fully spelled and they are case sensitive.
   - The CHLDISP keyword must be used, because the default is CHLDISP(PRIVATE), which is not valid for shared channels.

Queue Manager Alias

The Queue Manager Alias definition shown in Example 3-30 is required to resolve the foreign bus name, from the specification in the WebSphere ESB message engine (FB.MQRG) to the name of the queue-sharing group (MQRG).

Messages targeted for an MQ queue owned by the MQRG queue-sharing group, where the real name of the QSG has not been explicitly defined with the queue type destination object within the WebSphere ESB, is going to have FB.MQRG as the target queue manager name in the MQ transmission header structure.

Example 3-30 Queue Manager Alias definition for Foreign Bus name

```
Define a Remote Queue - 1

 Queue name  . . . . . . . . . FB.MQRG
 Disposition . . . . . . . . . G  G=Group, Q=Qmgr on MQR1
 Description . . . . . . . . . Queue Mgr Alias for QSG name whe
                               n addressed by SIB
......
 Remote name . . . . . . . . .
 Remote queue manager  . . . . MQRG
 Transmission queue  . . . . .
```

Keep the following in mind:

- The QR object must be named with the name specified for the foreign bus within WebSphere ESB.
- The Remote queue manager name must be set to the QSG name.
- The best way to handle this object is to define it as a group object taking, in effect, all (in our case: both) queue managers that are in the queue-sharing group.
- If this object is missing, messages sent from the WebSphere ESB ends as follows:
  - Either to the receiving queue manager's Default Transmission Queue.
  - If such a queue is not there, then to its Dead Letter Queue.

Remote queue definitions for WebSphere ESB bus queue destinations

There are no peculiarities with MQ remote queue definitions that describe a WebSphere ESB integration bus queue destination:

- The destination's identifier must be coded as the remote queue name.
- The name assigned to the MQLink, in our case SIBSYSSRE, has to be coded as the remote queue manager name.

A WebSphere ESB queue destination named sca/StockQuote is defined to our QSG queue managers by the QRemote definition, as shown in Example 3-31.

Example 3-31 MQ QRemote definition for a WebSphere ESB queue destination

```
Define a Remote Queue - 1

 Complete fields, then press F8 for further fields, or Ent


 Queue name  . . . . . . . . . WESB.STOCKQUOTE
 Disposition . . . . . . . . . G  G=Group, Q=Qmgr on MQR1
 Description . . . . . . . . . WESB bus queue destination

 .....
 Remote name . . . . . . . . . sca/StockQuote
 Remote queue manager  . . . . SIBSYSSRE
 Transmission queue  . . . . . SIBSYSSRE
```

Keep the following in mind:

- The WESB.STOCKQUOTE QR object name is just a sample.
- The slash character ('/') is allowed as part of an MQ object name.
- All names are case sensitive.

3.6.5 MQLink communication

Figure 3-114 resumes the connection configuration between a WebSphere Application Server cluster and an MQ queue-sharing group.

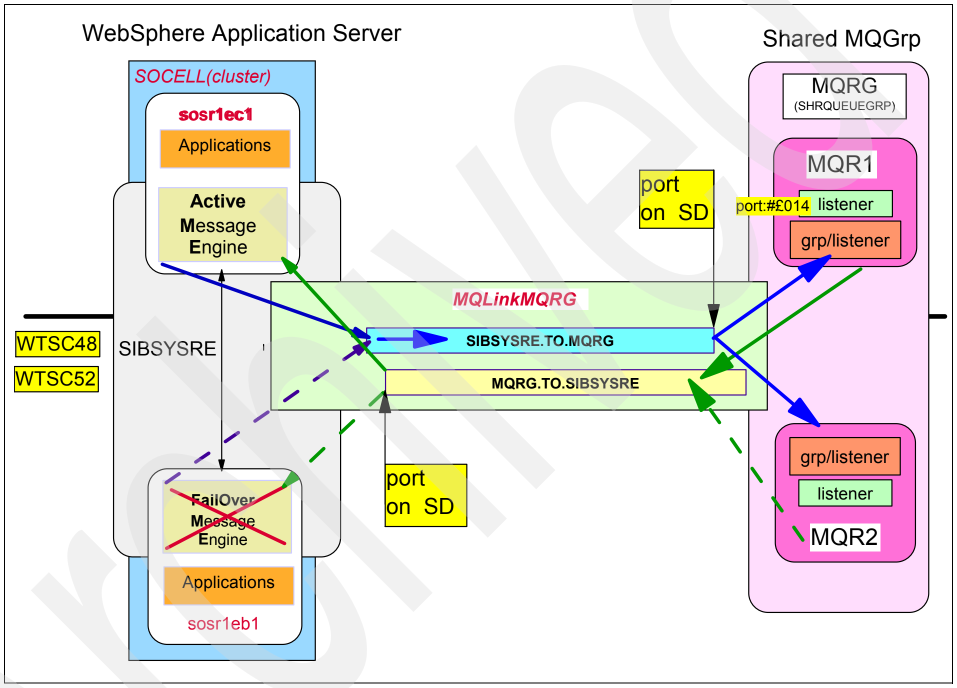


Figure 3-114 SIB to MQ QSG connection

Note the following about Figure 3-114:

- On both receiving sides we are using port numbers managed by the Sysplex Distributor. This guarantees the availability for inbound communication from both components' point of view.
- On the WebSphere Application Server Cluster sender side, the initiative always comes from within the active *Message Engine*. The port is dynamically selected when the connection with the server becomes active, in this case, the group listener of one or the other MQ QSG queue managers. The One of N policy controls the availability of the message engines.

- On the MQ queue-sharing group side, shared sender channel rules determine the start-up of the sender, as described in “Queue sharing group communication” on page 35. The sender channel, started by one of the QSG queue managers, points to the MQLinks distributed DVIPA and is connected through the Sysplex Distributor to the Message Engine, on the Service Integration Bus in a system where it is active.

4

Sample scenarios

This chapter describes multiple scenarios that demonstrate how to integrate a client and server application over a mix of protocols. You are going to use both the Advanced Enterprise Service Bus (ESB), also known as WebSphere Message Broker, and WebSphere ESB. They both support mediation among services.

We provide our sample scenario's as Project Interchange files, along with the resources required to run the CICS target application program used by the samples, in the SG247335.zip file. You can find the instructions on how to download the SG247335.zip file in Appendix B, "Additional material" on page 327.

4.1 Introduction

In our environment, the target CICS client application is invocable as a Distributed Program Link (DPL) program. The program invocation is supported either through the MQ CICS Bridge, or over a Web Service call. We also develop a combined mediation, exploiting WebSphere Message Broker and WebSphere ESB concurrently, which involves additional CICS applications. These are referred to as “traditional” applications and all of them are reachable over MQ messaging.

The target CICS application implements a basic trader environment that allows users to buy or sell shares belonging to one of four companies. The user provides input regarding their decision to buy or sell, the company short name, their user name, and the number of shares to trade.

The target application returns confirmation of their decision to buy or sell, the operation return code, the user name, the company long name (truncated), the number of shares the user now owns, and the value of the shares bought or sold in this transaction.

We chose this application as the base for our sample scenarios, because it seemed quite suitable from the scope of activities performed within CICS and the structure of the COMMAREA based interface. This means that it was not too simple or too complicated. For this reason, it has been used by previous IBM Redbooks projects. A complete documentation of the application is contained in the IBM Redbooks publication *A Performance Study of Web Access to CICS*, SG24-5748.

The original CICS 3270 application is a multistep application. The selection of one out of four companies and an (arbitrary) user name are prompted by the first steps and the buy/sell, which uses this information, is the final step within the pseudo-conversational dialog. We only run this final invocation of the CICS program and, therefore, have to provide all the information required within one step.

To invoke the back-end program through a Web service request, we used the DFHLS2WS CICS utility to generate a Web Services Description Language (WSDL) from the COBOL commarea structure and exposed the program as a Web service with CICS as the service provider.

The MQ client application can be any standard MQ program using either the MQ application programming interface (API) or Java Message Service (JMS) or MQ. In our environment, we used the RFHUtilc.exe utility, which is provided as part of the IBM IH03 MQ SupportPac™.

The Web service client application can be any service-oriented architecture (SOA) application invoking Web services. In our environment, we used the Web Services Explorer provided in the Message Broker Toolkit and we developed a SOAP client using WebSphere Integration Developer wizards.

4.2 WebSphere Message Broker implementation and deployment

Four WebSphere Message Broker scenarios are developed as follows:

- MQ to MQ

  An MQ client sends an Extensible Markup Language (XML) message over MQ to the ESB. The ESB transforms the request into a older format and forwards it to the target application using MQ. The target application returns a older format message that the ESB converts to an XML message before returning the message to the client using MQ.

- SOAP to SOAP

  A client supporting Web services sends a SOAP Hypertext Transfer Protocol (HTTP) request to the ESB. The ESB transforms the request according to the target service interface and delivers a SOAP/HTTP request to the target application. The service interface conforms to the request/response model. The Web service provider environment in this case is the WS-enabled CICS application.

- MQ to SOAP

  An MQ client sends an XML message over MQ to the ESB. The ESB transforms the request according to the target service interface and delivers a SOAP/HTTP request to the target application. The service interface conforms to the request/response model. The ESB transforms the SOAP/HTTP response to an XML message before returning the message to the client using MQ.

- SOAP to MQ

  A client supporting Web services sends a SOAP/HTTP request to the ESB. The ESB transforms the request according to the target service interface and delivers an older format message to the target application using MQ. The target application returns an older format message to the ESB, which converts it to a SOAP/HTTP response and returns this message to the client.

Figure 4-1 shows the sample scenarios described previously.

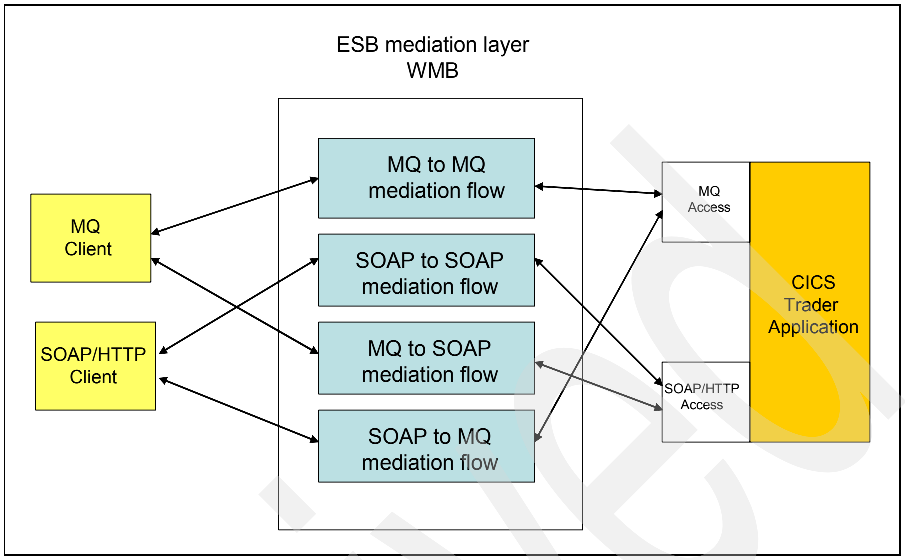


Figure 4-1 ESB mediation layer for WebSphere Message Broker

4.2.1 Message sets

Four message sets are used in the implementation of the following sample scenarios:

- TraderMQClient

  This represents the XML input and output messages used by the MQ client to invoke the WebSphere Message Broker message flow.

- TraderMQTarget

  This represents the COBOL input and output messages used by the target CICS application. Because the CICS application is going to be invoked over the MQ CICS Bridge, the message must contain the CICS program name followed by the CICS COMMAREA.

- TraderSoapClient

  This represents the SOAP request and response messages used by the Web services client to invoke the WebSphere Message Broker message flow.

- TraderSoapTarget

  This represents the SOAP input and output messages used by the target CICS application.

TraderMQClient message set

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderMQClient_Project and, within this project, create a new Message Set called MS_TraderMQClient. Select the **Use namespaces** option and in the next window, select the **XML Wire Format Name**. This sets the field to XML1.
3. Import the MQClient.xsd schema file into the MS_TraderMQClient_Project project. You can find instructions to download the MQClient.xsd schema file in Appendix B, "Additional material" on page 327.
4. Right-click the MQClient.xsd schema file and create a new Message Definition File within the MS_TraderMQClient message set. Check for the creation of the new message definition file based on the root element, MQClient. This message definition file is the WebSphere Message Broker representation of the input and output XML message exchanged with the client.

TraderMQTarget message set

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderMQTarget_Project and within this, a new Message Set called MS_TraderMQTarget. Select the **Custom Wire Format Name**. This sets the field to CWF1.
3. Import the COBOL copy book structure, commarea.cpy, into the MS_TraderMQClient_Project project. You can find the instructions to download the commarea.cpy file in Appendix B, "Additional material" on page 327.
4. Right-click the commarea.cpy file and create a new Message Definition File within the MS_TraderMQTarget message set. Check for the creation of the new message definition file based on the root element COMMAREA-BUFFER.
5. The message, `msg_COMMAREABUFFER`, which was created by the previous step, does not represent the full message exchanged with the MQ CICS Bridge. A new message must be manually created that contains the program name and the COMMAREA. Create a new message called MQCics and add elements, ProgramName followed by Commarea. Ensure that the ProgramName is of type string, eight characters long, and the Default value is set to TRADERBL. The Commarea element must be of type COMMAREABUFFER. The MQCics

message definition file is the WebSphere Message Broker representation of the input and output COBOL message exchanged with the MQ CICS Bridge target.

TraderSoapClient message set

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderSoapClient_Project and within this, a new Message Set called MS_TraderSoapClient. Select the **Use namespaces** option and in the next window, select the **XML Wire Format Name**. This sets the field to XML1.
3. Import the TraderHTTPSimpleClient.wsdl file into the MS_TraderSoapClient_Project project. You can find instructions to download the TraderHTTPSimpleClient.wsdl file in Appendix B, “Additional material” on page 327.
4. Right-click the TraderHTTPSimpleClient.wsdl file and accept all the defaults to create a new Message Definition File within the MS_TraderSoapClient message set.
5. Four different folders are created by the previous step. The folder names are based on the namespaces in the TraderHTTPSimpleClient.wsdl file. The root element, Envelope, is contained within the org.xmlsoap.schemas.soap.envelope folder. The request message is contained within the com.request.traderca.traderbl.www folder and the response message is contained within the com.response.traderca.traderbl.www folder. The message definitions within this message set represent the WebSphere Message Broker request exchange and response exchange with the Web services client.

TraderSoapTarget message set

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderSoapTarget_Project and within this, a new Message Set called MS_TraderSoapTarget. Select the **Use namespaces** option and in the next window, select the **XML Wire Format Name**. This sets the field to XML1.
3. Import the TraderHttp.wsdl file into the MS_TraderSoapTarget_Project project. You can find the instructions to download the TraderHttp.wsdl file in Appendix B, “Additional material” on page 327.

4. Right-click the TraderHttp.wsdl file and accept all the defaults to create a new Message Definition File within the MS_TraderSoapTarget message set.
5. Four different folders are created by the previous step. The folder names are based on the namespaces in the TraderHttp.wsdl file. The root element, Envelope, is contained within the org.xmlsoap.schemas.soap.envelope folder. The request message is contained within the com.request.traderca.traderbl.www folder and the response message is contained within the com.response.traderca.traderbl.www folder. The message definitions within this message set represent the WebSphere Message Broker request and response exchange with the target CICS application that has been exposed as a Web service.

4.2.2 Message flows

The four message flows that this section describes are going to be implemented on WebSphere Message Broker.

MF_TraderMQtoMQ

The TraderMQtoMQ sample flow demonstrates how to mediate between an MQ client application issuing requests via XML MQ messages and a target CICS application. The target application is invoked through the MQ CICS Bridge through a message built corresponding to a COBOL copy book structure. This flow implements a typical MQ Request-Reply model.

The client application is responsible for specifying the queue (MQReplyToQ and MQReplyToQMgr) regarding where the client wants to receive the final reply. The client eventually gets a response message with a CorrelId matching the MsgId of the original request message.

The MQ CICS Bridge implements a similar Request-Reply model with the MsgId in the original request behaving as a unique request ID. The WebSphere Message Broker forwards its own ReplyTo information to the MQ CICS Bridge. For this reason, the message flow logic has to save the original ReplyTo information provided by the client application before sending the subsequent request message to the MQ CICS Bridge. When the response is received from the MQ CICS Bridge, WebSphere Message Broker retrieves the client's original ReplyTo information using the unique request ID and recreates the ReplyTo information to match the one provided by the client application.

The TraderMQtoMQ message flow uses an intermediate queue to store the MQMD of the original request message and an MQGET node to retrieve this information. This information could alternatively be stored in a database, a file, or even an environment variable, but a queue is a quick, safe, and efficient holding area.

The message flow for the TraderMQtoMQ sample performs the following tasks:

1. Request Branch:
   a. The TRADER.RequestFromClient MQInput node reads a message from the request queue.
   b. The SaveHeaders Compute node copies the message headers and sets the CorrelId equal to the MsgId.
   c. The Trader.SaveData MQOutput node copies this information to the intermediate queue.
   d. The SetReplyQueue Compute node forwards the entire message and sets the following values:
      - ReplyToQ equal to the queue name where the Response branch of the message flow wishes to receive the response from CICS.
      - ReplyToQMgr equal to blank in order to allow this value to default to the local queue manager.
      - CorrelId equal to a value indicating the requirement for a new MQ CICS Bridge session. Note that the original MsgId received from the client application is simply forwarded to the CICS Bridge.
   e. The XMLtoCobol Mapping node converts the XML input message to a COBOL-like message to fit the format expected by the MQ CICS Bridge.
   f. The TRADER.RequestToCics MQOutput node delivers the message to CICS via the MQ CICS Bridge.
2. Response Branch:
   a. The TRADER.ReplyFromCics MQInput node retrieves the response from the MQ CICS Bridge.
   b. The CobolToXML Mapping node converts the COBOL-like message format to XML.
   c. The MQGet node retrieves the MQMD of the original request message and replaces the MQMD in the current message with this.
   d. The MQReply node sends the response message to the client application.

The TraderMQtoMQ sample message flow is shown in Figure 4-2. Note that both the flow between the client and the CICS Bridge and also the return flow between the CICS Bridge and the original client application are combined into this one single flow. As you can see, they share a common failure queue named TRADER.FAILURE.

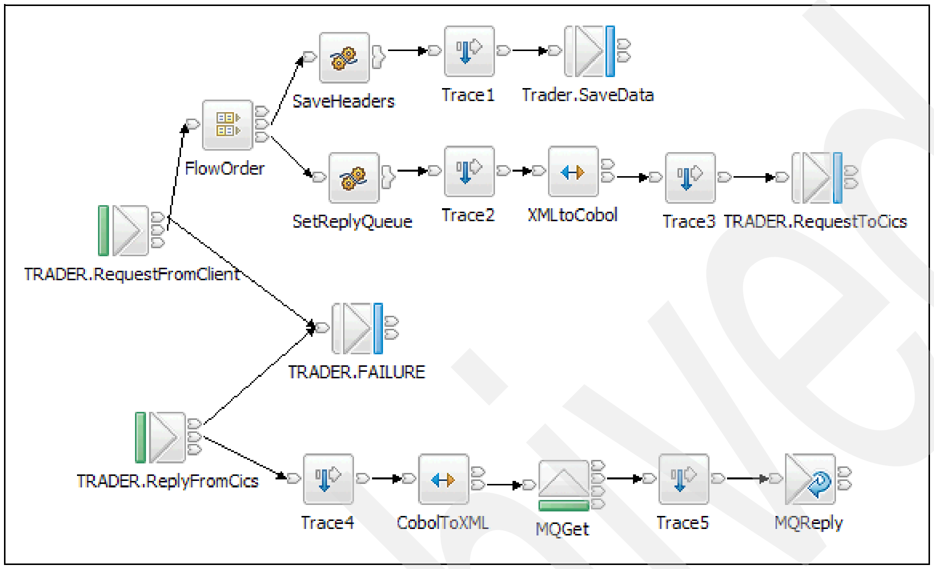


Figure 4-2 TraderMQtoMQ sample message flow

Node name: TRADER.RequestFromClient

This node reads the request message from the client request queue. Table 4-1 shows a description of the property values changed from the defaults.

Table 4-1 Description of changed property values

Property	Property name: value	Comment
Basic	Queue Name:TRADER.MQTOMQ.INPUT	Defines the queue the input node is reading from.
Default	Message Domain: MRM	The domain the input message belongs to.
	MessageSet: MS_TraderMQClient	The message set describing the incoming message.
	Message Type: MQClient	The message type describing the incoming message.
	Message Format: XML1	The physical format of the incoming message.

Node name: Flow Order

This node governs the order of the flow branches that leave from the output terminals, which are First and Second.

Node name: Save Headers

Default properties can be accepted in this Compute node. The relevant ESQL computed logic is detailed in Example 4-1. The node forwards the headers removing the message payload and copies the MsgId to the CorrelId of the current MQMD.

Example 4-1 ESQL computed logic for Save Headers Node

```
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      CALL CopyMessageHeaders();
      -- CALL CopyEntireMessage();

      -- Copy MsgId to CorrelId as the Back-end reply application would 
do
      -- This enables the MQGet node in the Reply flow to retrieve the
      -- corresponding store message by CorrelId even if the requestor 
did
      -- not send the message as a MQSeries request message
      SET OutputRoot.MQMD.CorrelId = InputRoot.MQMD.MsgId;
      RETURN TRUE;

   END;
```

Node name: Trader.SaveData

See Table 4-2 for a description of the property values changed from the defaults.

Table 4-2 Properties values

Property	Property name: value	Comment
Basic	Queue Name: TRADER.SAVEDATA	Defines the queue where the current MQMD is saved.

Node name: SetReplyQueue

Default properties can be accepted in this Compute node. The relevant ESQL computed logic is detailed in Example 4-2. The node sets the ReplyToQ to the queue expected by the response message flow and starts a new session with the MQ CICS Bridge.

Example 4-2 ESQL computed logic for SetReplyQueue Node

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
   -- CALL CopyMessageHeaders();
   CALL CopyEntireMessage();
   SET OutputRoot.MQMD.ReplyToQ = 'TRADER.CICSREPQ';
   SET OutputRoot.MQMD.ReplyToQMgr = ' ';
   SET OutputRoot.MQMD.CorrelId = MQCI_NEW_SESSION;
   RETURN TRUE;
END;
```

Node name: XMLtoCobol

The map is responsible for mapping the property values so that they match the values expected by the target application.

The following three property values have to be set manually:

- MessageSet

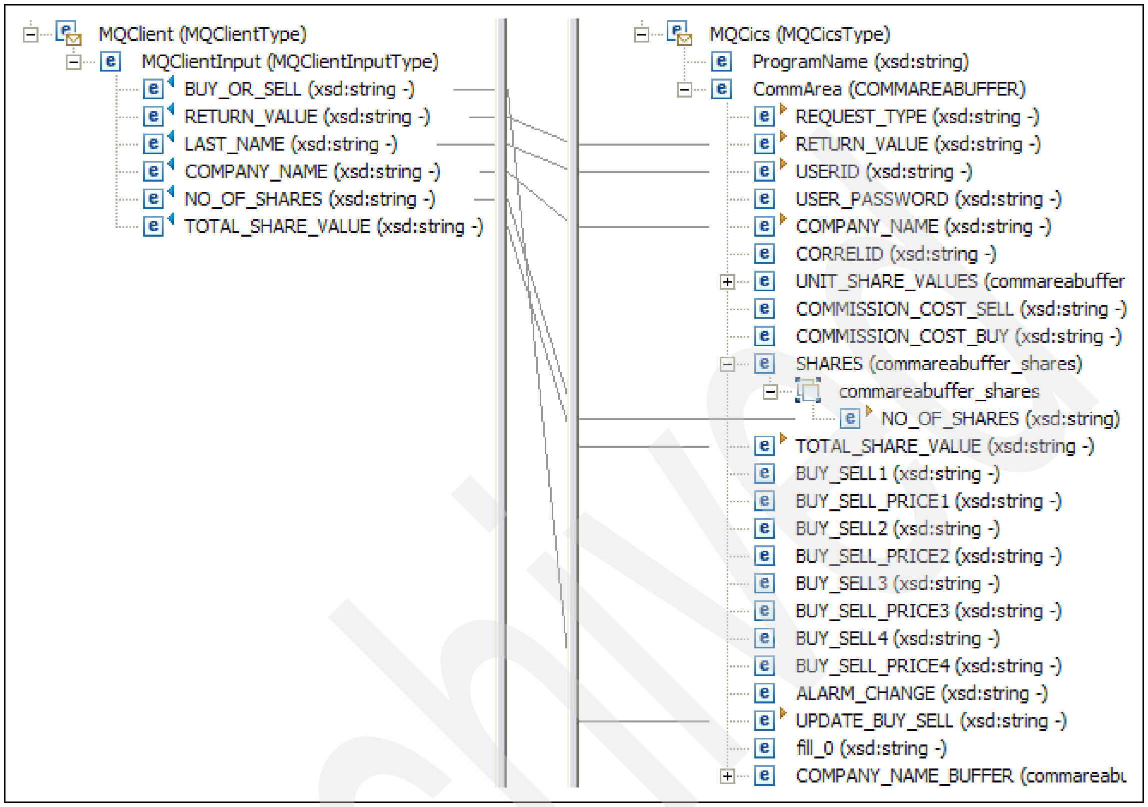
  The unique message set identifier of the target message set MS_TraderMQTarget.
- MessageType

  The message type name of the target message, MQCics.
- MessageFormat

  The physical format of the target message, CWF1.

The remaining fields in the Properties tree can be copied unchanged.

Figure 4-3 shows how to map the message payload.


MQClient (MQClientType)
MQClientInput (MQClientInputType)
BUY_OR_SELL (xsd:string -)
RETURN_VALUE (xsd:string -)
LAST_NAME (xsd:string -)
COMPANY_NAME (xsd:string -)
NO_OF_SHARES (xsd:string -)
TOTAL_SHARE_VALUE (xsd:string -)

MQCics (MQCicsType)
ProgramName (xsd:string)
CommArea (COMMAREABUFFER)
REQUEST_TYPE (xsd:string -)
RETURN_VALUE (xsd:string -)
USERID (xsd:string -)
USER_PASSWORD (xsd:string -)
COMPANY_NAME (xsd:string -)
CORRELID (xsd:string -)
UNIT_SHARE_VALUES (commareabuffer
COMMISSION_COST_SELL (xsd:string -)
COMMISSION_COST_BUY (xsd:string -)
SHARES (commareabuffer_shares)
commareabuffer_shares
NO_OF_SHARES (xsd:string)
TOTAL_SHARE_VALUE (xsd:string -)
BUY_SELL1 (xsd:string -)
BUY_SELL_PRICE1 (xsd:string -)
BUY_SELL2 (xsd:string -)
BUY_SELL_PRICE2 (xsd:string -)
BUY_SELL3 (xsd:string -)
BUY_SELL_PRICE3 (xsd:string -)
BUY_SELL4 (xsd:string -)
BUY_SELL_PRICE4 (xsd:string -)
ALARM_CHANGE (xsd:string -)
UPDATE_BUY_SELL (xsd:string -)
fill_0 (xsd:string -)
COMPANY_NAME_BUFFER (commareabu

Figure 4-3 XMLtoCobol message map

Furthermore, the map also builds the following fields:

- REQUEST_TYPE is set to the fixed value ‘Buy_Sell’.
- COMPANY_NAME is set through the user ESQL function, Map_Company_Name, which receives the source field COMPANY_NAME as its input. Example 4-3 describes the ESQL code of this user function.

Example 4-3 ESQL function for XMLtoCobol Node

```
CREATE FUNCTION Map_Company_Name (ShortName CHAR) RETURNS CHAR
   BEGIN
      DECLARE LongName CHAR;
      IF ShortName='CAS' THEN
         SET LongName='Casey_Import_Export';
      ELSEIF ShortName='GLP' THEN
         SET LongName='Glass_and_Luget_Plc';
      ELSEIF ShortName='HWE' THEN
         SET LongName='Headworth_Electrical';
      ELSE
         SET LongName='IBM';
      END IF;
   RETURN LongName;
   END;
```

Node name: TRADER.RequestToCics

Table 4-3 shows a description of the property values changed from the defaults.

Table 4-3 Properties values

Property	Property name: value	Comment
Basic	Queue Name: SYSTEM.CICS.BRIDGE.QUEUE	Defines the queue from where the MQ CICS Bridge reads the message.

Node name: TRADER.ReplyFromCics

This is the node from which the response from CICS is read. See Table 4-4 for a description of the property values changed from the defaults.

Table 4-4 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADER.CICSREPQ	Defines the queue from which the input node is reading.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderMQTarget	The message set describing the incoming message.
	Message Type: MQCics	The message type describing the incoming message.
	Message Format: CWF1	The physical format of the incoming message.

Node name: CobolToXML

The map is responsible for mapping the property values so that they match the values expected by the client application.

You have to set the following three property values manually:

- MessageSet

  The unique message set identifier of the client message set MS_TraderMQClient.

- MessageType

  The message type name of the target message MQClient.

- MessageFormat

  The physical format of the target message XML1.

The remaining fields in the Properties tree can be copied unchanged.

Figure 4-4 shows how to map the message payload.

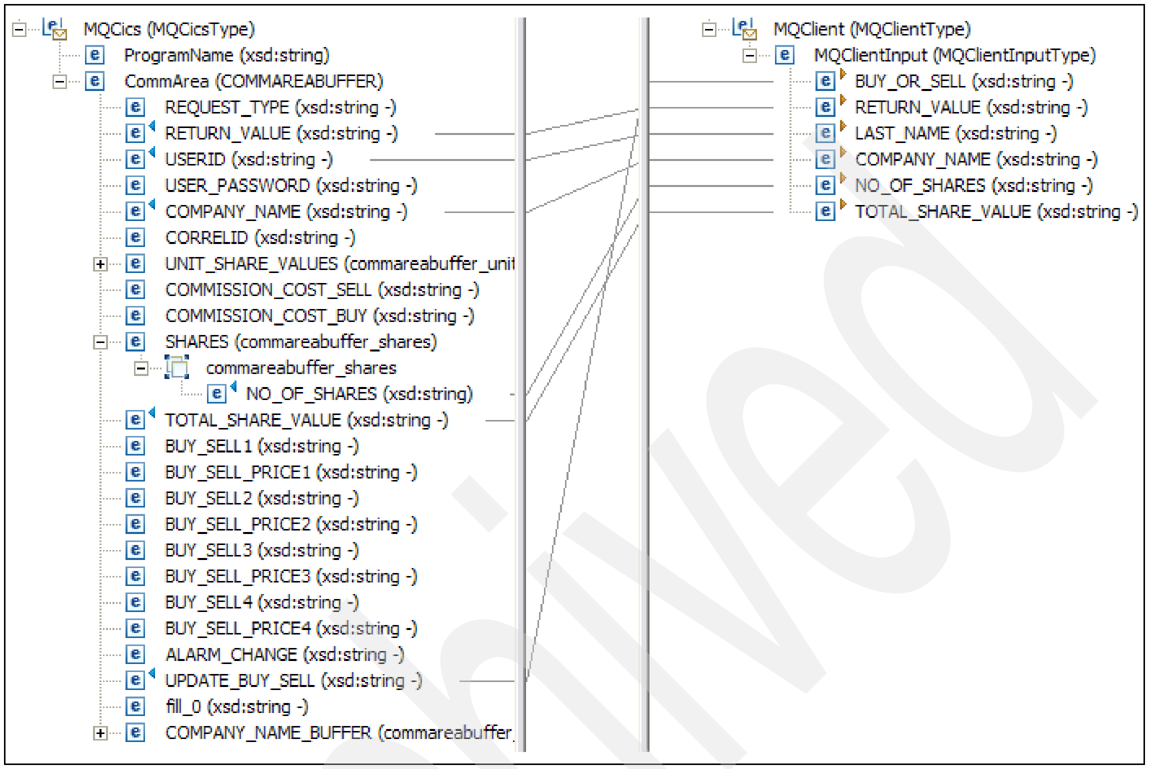

MQCics (MQCicsType)
ProgramName (xsd:string)
CommArea (COMMAREABUFFER)
REQUEST_TYPE (xsd:string -)
RETURN_VALUE (xsd:string -)
USERID (xsd:string -)
USER_PASSWORD (xsd:string -)
COMPANY_NAME (xsd:string -)
CORRELID (xsd:string -)
UNIT_SHARE_VALUES (commareabuffer_unit
COMMISSION_COST_SELL (xsd:string -)
COMMISSION_COST_BUY (xsd:string -)
SHARES (commareabuffer_shares)
commareabuffer_shares
NO_OF_SHARES (xsd:string)
TOTAL_SHARE_VALUE (xsd:string -)
BUY_SELL1 (xsd:string -)
BUY_SELL_PRICE1 (xsd:string -)
BUY_SELL2 (xsd:string -)
BUY_SELL_PRICE2 (xsd:string -)
BUY_SELL3 (xsd:string -)
BUY_SELL_PRICE3 (xsd:string -)
BUY_SELL4 (xsd:string -)
BUY_SELL_PRICE4 (xsd:string -)
ALARM_CHANGE (xsd:string -)
UPDATE_BUY_SELL (xsd:string -)
fill_0 (xsd:string -)
COMPANY_NAME_BUFFER (commareabuffer
MQClient (MQClientType)
MQClientInput (MQClientInputType)
BUY_OR_SELL (xsd:string -)
RETURN_VALUE (xsd:string -)
LAST_NAME (xsd:string -)
COMPANY_NAME (xsd:string -)
NO_OF_SHARES (xsd:string -)
TOTAL_SHARE_VALUE (xsd:string -)

Figure 4-4 CobolToXML message map

The map also trims the field, COMPANY_NAME, down to three characters using the esql function left(). The code is as follows:

```
esql:left($source/MQCics/CommArea/COMPANY_NAME,3)
```

Node name: MQGet

This node is responsible for retrieving the previously saved MQMD and uses its content to replace the MQMD of the current response message so that the message is sent to the correct client. The MsgId of the current message is used to retrieve the correct MQMD from the TRADER.SAVEDATA queue. See Table 4-5 for a description of the property values changed from the defaults.

Table 4-5 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADER.SAVEDATA	Defines the queue from which the MQGet node is reading.
Default	We are not setting any values in the Default Property Folder because we only read the MQMD of the saved message.	
Advanced	Copy Message: Copy Entire Message	The entire incoming message is going to be forwarded by this node.
	The remaining properties in the Advanced Property Folder are left as the default values.	
Request	Get by Message ID: checked	This option allows us to retrieve the saved MQMD by using the MsgId that matches the MsgId of the incoming message.
Result	Output Data Location: OutputRoot.MQMD	The MQMD retrieved from the TRADER.SAVEDATA queue replaces this portion of the output message tree.
	Result Data Location: ResultRoot.MQMD	The message retrieved from the TRADER.SAVEDATA queue is stored in this message tree.

Node name: MQReply

This node delivers the message to the queue specified in the ReplyToQ fields. The properties were left as the default values.

Testing the message flow

In order to test the message flow, you must create a BAR file containing the following elements:

- MF_TraderMQtoMQ.msgflow
- MS_TraderMQClient
- MS_TraderMQTarget

Deploy this BAR file to the runtime broker.

The following MQ queues are also required:

- TRADER.MQTOMQ.INPUT
- TRADER.FAILURE
- TRADER.SAVEDATA
- SYSTEM.CICS.BRIDGE.QUEUE (default for the MQ CICS Bridge support)
- TRADER.CICSREPQ

The MQ reply is delivered to the client as specified in the ReplyTo queue information of the request message.

MF_TraderSoapToSoap

The TraderSoapToSoap sample flow demonstrates how to mediate between a Web service application issuing a SOAP/HTTP message and a target CICS application exposed as a Web service that can also be accessed through SOAP/HTTP, but with the requirement for a data structure different than the one used by the client.

The message flow for the TraderSoapToSoap sample does the following tasks:

1. The HTTP Input node receives a SOAP/HTTP request from the client application.
2. The ClientToTarget Mapping node converts the client SOAP request to the target SOAP message format.
3. The HTTP Request node delivers the request to the specified target endpoint and receives a response message.
4. The TargetToClient Mapping node converts the response from the target application to a SOAP/HTTP response, as expected by the client application.
5. The HTTP Reply node delivers the response message to the client application.

The TraderSoapToSoap sample message flow is shown in Figure 4-5.

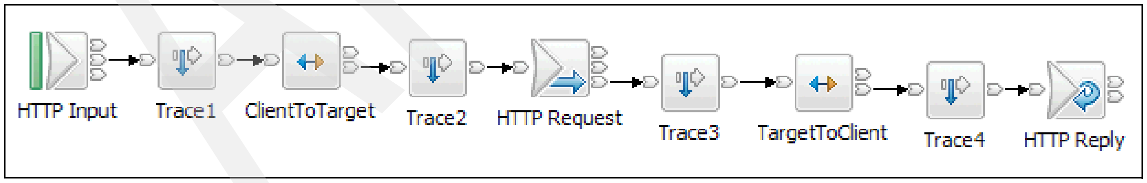


Figure 4-5 TraderSoapToSoap sample message flow

Node name: HTTP Input

See Table 4-6 for a description of the property values changed from the defaults.

Table 4-6 Properties values

Property	Property name: value	Comment
Basic	URL Selector: /soaptosoap	Defines the context portion of the URL from which this node receives Web service requests.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderSoapClient	The message set describing the incoming message.
	Message Type: Envelope	The message type describing the incoming message. All the messages received by this node conform the structure of an Envelope message
	Message Format: XML1	The physical format of the incoming message.

Node name: ClientToTarget

This map transforms the client application request message to the target service request message. The map is split into a main map and a submap The submap is invoked by the main map. The main map is responsible for mapping the values in the Properties tree and establishing the link between the body of the incoming message and the corresponding body of the outgoing message.

The following three Property values have to be set manually in the main map:

- MessageSet

  The unique message set identifier of the target message set MS_TraderSoapTarget.

- MessageType

  The message type name of the target message, Envelope.

- MessageFormat

  The physical format of the target message, XML1.

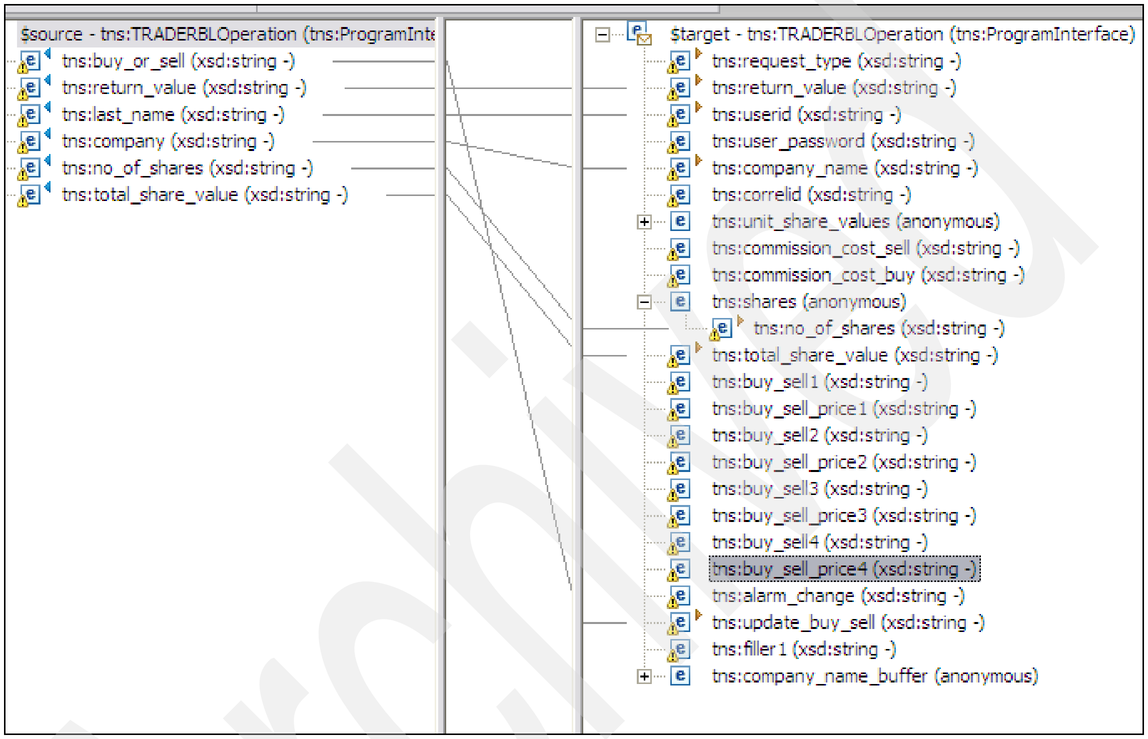
The remaining fields in the Properties tree can be copied unchanged.

Perform the following steps:

1. In the main map, expand both the source panel and the target panel by selecting **Envelope** → **Body** → **choice** → **sequence** → **message** → **Wildcard Message**.

2. Click **Wildcard Message** in both the source and target panels, then select **Map** → **Create New Submap** from the menu bar located at the top of the Application Development Perspective. A new submap is created.

Figure 4-6 shows how you can use the submap to map the message payload of the request message.



```
$source - tns:TRADERBLOperation (tns:ProgramInte
    tns:buy_or_sell (xsd:string -)
    tns:return_value (xsd:string -)
    tns:last_name (xsd:string -)
    tns:company (xsd:string -)
    tns:no_of_shares (xsd:string -)
    tns:total_share_value (xsd:string -)
```

```
$target - tns:TRADERBLOperation (tns:ProgramInterface)
    tns:request_type (xsd:string -)
    tns:return_value (xsd:string -)
    tns:userid (xsd:string -)
    tns:user_password (xsd:string -)
    tns:company_name (xsd:string -)
    tns:correlid (xsd:string -)
    tns:unit_share_values (anonymous)
    tns:commission_cost_sell (xsd:string -)
    tns:commission_cost_buy (xsd:string -)
    tns:shares (anonymous)
        tns:no_of_shares (xsd:string -)
    tns:total_share_value (xsd:string -)
    tns:buy_sell1 (xsd:string -)
    tns:buy_sell_price1 (xsd:string -)
    tns:buy_sell2 (xsd:string -)
    tns:buy_sell_price2 (xsd:string -)
    tns:buy_sell3 (xsd:string -)
    tns:buy_sell_price3 (xsd:string -)
    tns:buy_sell4 (xsd:string -)
    tns:buy_sell_price4 (xsd:string -)
    tns:alarm_change (xsd:string -)
    tns:update_buy_sell (xsd:string -)
    tns:filler1 (xsd:string -)
    tns:company_name_buffer (anonymous)
```

Figure 4-6 ClientToTarget message map

Furthermore, the submap also builds these fields:

- request_type is set to the fixed value 'Buy_Sell'.
- company_name is set through the user ESQL function, Map_Company_Name, which receives the source field company as its input.

Example 4-4 describes the ESQL code of this user function.

Example 4-4 ESQL code for ClientToTarget Node

```
CREATE FUNCTION Map_Company_Name (ShortName CHAR) RETURNS CHAR
   BEGIN
      DECLARE LongName CHAR;
      IF ShortName='CAS' THEN
         SET LongName='Casey_Import_Export';
      ELSEIF ShortName='GLP' THEN
         SET LongName='Glass_and_Luget_Plc';
      ELSEIF ShortName='HWE' THEN
         SET LongName='Headworth_Electrical';
      ELSE
         SET LongName='IBM';
      END IF;
   RETURN LongName;
   END;
```

Node name: HTTP Request

This node invokes the target Web service. See Table 4-7 for a description of the property values changed from the defaults.

Table 4-7 Properties values

Property	Property name: value	Comment
Basic	Web service URL*: http://9.12.4.38:03601/traderApp/TraderHttp	Defines the endpoint of the target Web service to invoke.
	The remaining properties in the Basic Property Folder are left as the default values.	
Default	Message Domain: MRM	The domain to which the message returned by the target Web service belongs.
	MessageSet: MS_TraderSoapTarget	The message set to which the message returned by the target Web service belongs.
	Message Type: Envelope	The message type of the message returned by the target Web service.
	Message Format: XML1	The physical format of the message returned by the target Web service.

Node name: TargetToClient

This map transforms the target service response message to the client application response message.

The map is split into a main map and a submap. The submap is invoked by the main map.

The main map is responsible for mapping the values in the Properties tree and establishing the link between the body of the incoming message and the corresponding body of the outgoing message.

The following three Property values have to be set manually in the main map:

- MessageSet

  The unique message set identifier of the client message set MS_TraderSoapClient.
- MessageType

  The message type name of the target message, Envelope.
- MessageFormat

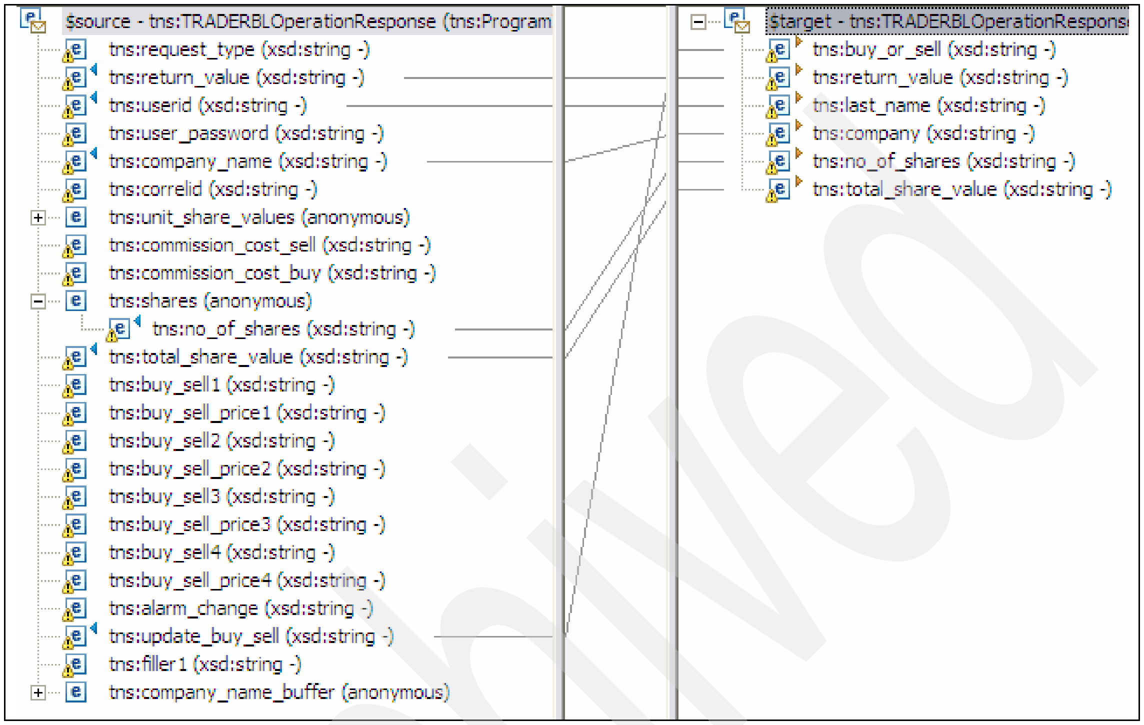
  The physical format of the target message, XML1.

The remaining fields in the Properties tree can be copied unchanged.

Perform the following steps:

1. In the main map, expand both the source panel and the target panel by selecting **Envelope** → **Body** → **choice** → **sequence** → **message** → **Wildcard Message**.
2. Click **Wildcard Message** in both the source and target panels, and then select **Map** → **Create New Submap** from the menu bar located at the top of the Application Development Perspective. A new submap will be created.

Figure 4-7 shows how you can use the submap to map the message payload of the response message.


$source - tns:TRADERBLOperationResponse (tns:Program
tns:request_type (xsd:string -)
tns:return_value (xsd:string -)
tns:userid (xsd:string -)
tns:user_password (xsd:string -)
tns:company_name (xsd:string -)
tns:correlid (xsd:string -)
tns:unit_share_values (anonymous)
tns:commission_cost_sell (xsd:string -)
tns:commission_cost_buy (xsd:string -)
tns:shares (anonymous)
tns:no_of_shares (xsd:string -)
tns:total_share_value (xsd:string -)
tns:buy_sell1 (xsd:string -)
tns:buy_sell_price1 (xsd:string -)
tns:buy_sell2 (xsd:string -)
tns:buy_sell_price2 (xsd:string -)
tns:buy_sell3 (xsd:string -)
tns:buy_sell_price3 (xsd:string -)
tns:buy_sell4 (xsd:string -)
tns:buy_sell_price4 (xsd:string -)
tns:alarm_change (xsd:string -)
tns:update_buy_sell (xsd:string -)
tns:filler1 (xsd:string -)
tns:company_name_buffer (anonymous)

$target - tns:TRADERBLOperationRespons
tns:buy_or_sell (xsd:string -)
tns:return_value (xsd:string -)
tns:last_name (xsd:string -)
tns:company (xsd:string -)
tns:no_of_shares (xsd:string -)
tns:total_share_value (xsd:string -)

Figure 4-7 TargetToClient message map

The submap also trims the field, which is called company, down to three characters using the esql function left(). The code is as follows:

```
esql:left($source/tns:company_name,3)
```

Node name: HTTPReply

This node delivers the HTTP reply to the client application. The properties were left as the default values.

Testing the message flow

In order to test this message flow, you must create a BAR file containing the following elements:

- MF_TraderSoapToSoap.msgflow
- MS_TraderSoapClient
- MS_TraderSoapTarget

Deploy the file to the run time broker.

MF_TraderMQToSoap

The TraderMQtoSoap sample flow demonstrates how to mediate between an MQ client application issuing requests through XML MQ messages and a target CICS application exposed as a Web service that can be accessed through SOAP/HTTP.

It is the responsibility of the WebSphere Message Broker message flow to ensure that the MQ headers are correctly rebuilt after the Web service invocation.

The message flow for the TraderMQtoSoap sample does the following tasks:

1. The TRADER.RequestFromClient MQInput node reads a message from the request queue.
2. The SaveMQMD Compute node copies the entire message and saves the current MQMD in the Environment tree so that it is available to the nodes downstream in the message flow. Note that unlike the MQtoMQ scenario, with this MQtoSoap scenario, the temporary data can now best be saved in a memory environment variable because the call to the CICS target service is synchronous.
3. The ClientToTarget Mapping node converts the XML input message to the target SOAP message format.
4. The HTTP Request node delivers the request to the specified target endpoint and receives a response message.
5. The TargetToClient Mapping node converts the SOAP message to a XML response as expected by the client application.
6. The RestoreMQMD Compute node rebuilds a standard MQ message using the information previously stored in the Environment tree.
7. The MQReply node delivers the response message to the client application.

The TraderMQtoSoap sample message flow is shown in Figure 4-8.

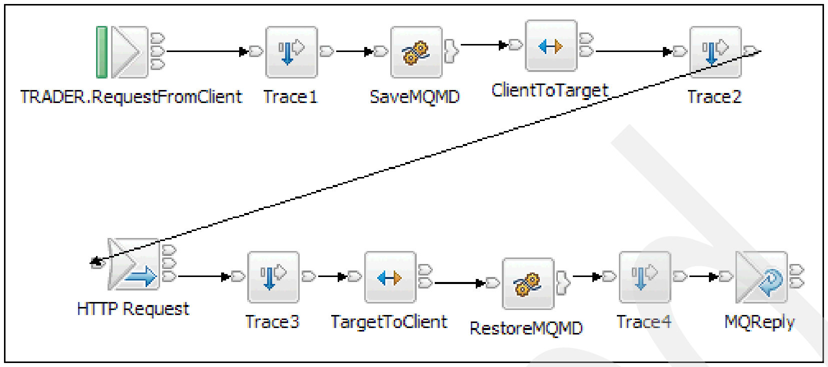


Figure 4-8 TraderMQtoSoap sample message flow

Node name: TRADER.RequestFromClient

This node reads the request message from the client request queue. See Table 4-8 for a description of the property values changed from the defaults.

Table 4-8 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADER.MQTOSOAP.INPUT	Defines the queue from which the input node is reading.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderMQClient	The message set describing the incoming message.
	Message Type: MQClient	The message type describing the incoming message.
	Message Format: XML1	The physical format of the incoming message.

Node name: SaveMQMD

Default properties can be accepted in this Compute node. The relevant ESQL computed logic is detailed in Example 4-5. The node copies the entire message and saves the current MQMD in the Environment tree so that it is available to the nodes downstream in the message flow.

Example 4-5 ESQL code for SaveMQMD Node

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
   -- CALL CopyMessageHeaders();
   CALL CopyEntireMessage();
   SET Environment.Temp.MQMD = OutputRoot.MQMD;
   RETURN TRUE;
END;
```

Note that in this MQtoSoap scenario, unlike the earlier MQtoMQ scenario, the MQMD data can be most efficiently stored temporarily in a memory environment variable. This is because the back-end call to CICS is now going to be synchronous through a Web service. Although the blocking call is not as scalable, it has the advantage of simplifying the code and there is no longer any requirement to persist data in a temporary queue.

Node name: ClientToTarget

This map transforms the MQ client application request message to the target SOAP request message.

The map is split into a main map and a submap The submap is invoked by the main map.

The main map is responsible for mapping the values in the Properties tree and establishing the link between the incoming message and the body within the Envelope of the outgoing message.

The following three Property values have to be set manually in the main map:

- MessageSet

  The unique message set identifier of the target message set MS_TraderSoapTarget.

- MessageType

  The message type name of the target message, Envelope.

- MessageFormat

  The physical format of the target message, XML1.

The remaining fields in the Properties tree can be copied unchanged.

Perform the following steps:

1. In the main map, expand the source panel to the MQClient and the target panel to **Envelope** → **Body** → **choice** → **sequence** → **message** → **Wildcard Message**.
2. Click **MQClient** in the source panel and **Wildcard Message** in the target panel. Select **Map** → **Create New Submap** from the menu bar located at the top of the Application Development Perspective. A new submap is created.

Figure 4-9 shows how to map the message payload.

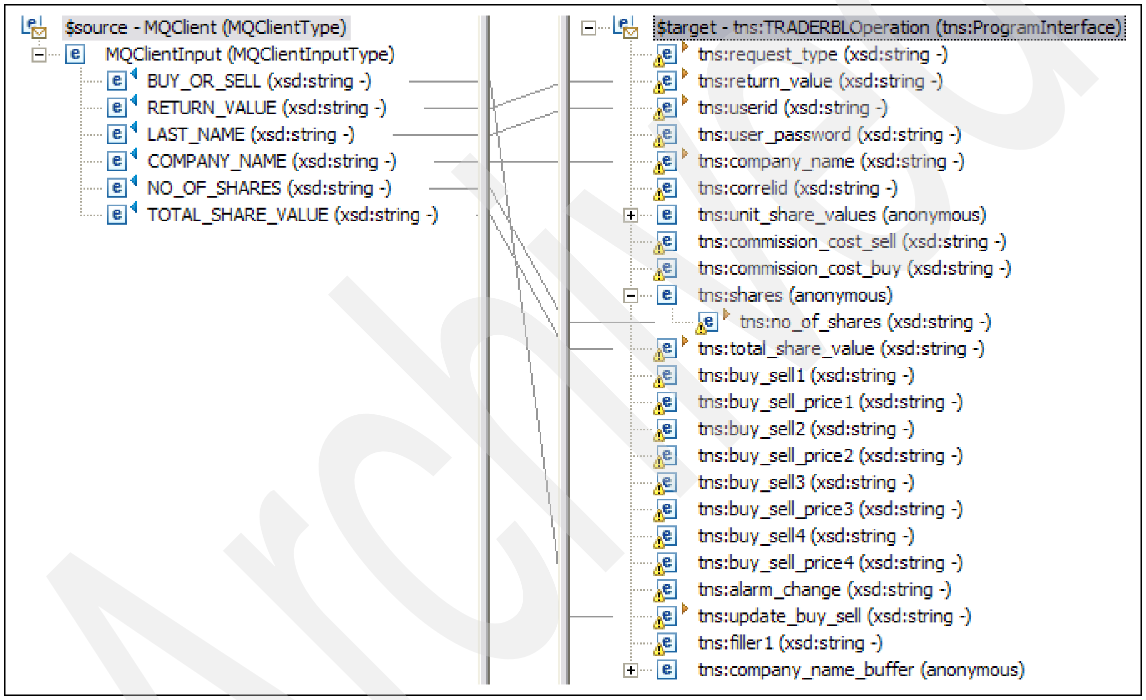

$source - MQClient (MQClientType)
MQClientInput (MQClientInputType)
BUY_OR_SELL (xsd:string -)
RETURN_VALUE (xsd:string -)
LAST_NAME (xsd:string -)
COMPANY_NAME (xsd:string -)
NO_OF_SHARES (xsd:string -)
TOTAL_SHARE_VALUE (xsd:string -)
$target - tns:TRADERBLOperation (tns:ProgramInterface)
tns:request_type (xsd:string -)
tns:return_value (xsd:string -)
tns:userid (xsd:string -)
tns:user_password (xsd:string -)
tns:company_name (xsd:string -)
tns:correlid (xsd:string -)
tns:unit_share_values (anonymous)
tns:commission_cost_sell (xsd:string -)
tns:commission_cost_buy (xsd:string -)
tns:shares (anonymous)
tns:no_of_shares (xsd:string -)
tns:total_share_value (xsd:string -)
tns:buy_sell1 (xsd:string -)
tns:buy_sell_price1 (xsd:string -)
tns:buy_sell2 (xsd:string -)
tns:buy_sell_price2 (xsd:string -)
tns:buy_sell3 (xsd:string -)
tns:buy_sell_price3 (xsd:string -)
tns:buy_sell4 (xsd:string -)
tns:buy_sell_price4 (xsd:string -)
tns:alarm_change (xsd:string -)
tns:update_buy_sell (xsd:string -)
tns:filler1 (xsd:string -)
tns:company_name_buffer (anonymous)

Figure 4-9 ClientToTarget message map

Furthermore, the submap also builds the following fields:

- REQUEST_TYPE is set to the fixed value 'Buy_Sell'.
- COMPANY_NAME is set through the user ESQL function, Map_Company_Name, which receives the source field COMPANY_NAME as its input. The sample shown in Example 4-6 describes the ESQL code of this user function.

Example 4-6 ESQL code for ClientToTarget Node

```
CREATE FUNCTION Map_Company_Name (ShortName CHAR) RETURNS CHAR
   BEGIN
      DECLARE LongName CHAR;
      IF ShortName='CAS' THEN
         SET LongName='Casey_Import_Export';
      ELSEIF ShortName='GLP' THEN
         SET LongName='Glass_and_Luget_Plc';
      ELSEIF ShortName='HWE' THEN
         SET LongName='Headworth_Electrical';
      ELSE
         SET LongName='IBM';
      END IF;
   RETURN LongName;
   END;
```

Node name: HTTP Request

This node invokes the target Web service. See Table 4-9 for a description of the property values changed from the defaults.

Table 4-9 Properties values

Property	Property name: value	Comment
Basic	Web Service URL*: http://9.12.4.38:03601/traderApp/TraderHttp	Defines the endpoint of the target Web service to invoke.
	The remaining properties in the Basic Property Folder are left as the default values.	
Default	Message Domain: MRM	The domain to which the message returned by the target Web service belongs.
	MessageSet: MS_TraderSoapTarget	The message set to which the message returned by the target Web service belongs.
	Message Type: Envelope	The message type of the message returned by the target Web service.
	Message Format: XML1	The physical format of the message returned by the target Web service.

Node name: TargetToClient

This map transforms the target SOAP application response message to the client MQ response message. The map is split into a main map and a submap The submap is invoked by the main map. The main map is responsible for mapping the values in the Properties tree and establishing the link between the body within the Envelope of the incoming message and the outgoing message.

The following three Property values have to be set manually in the main map:

- MessageSet

  The unique message set identifier of the target message set MS_TraderMQClient.

- MessageType

  The message type name of the target message, MQClient.

- MessageFormat

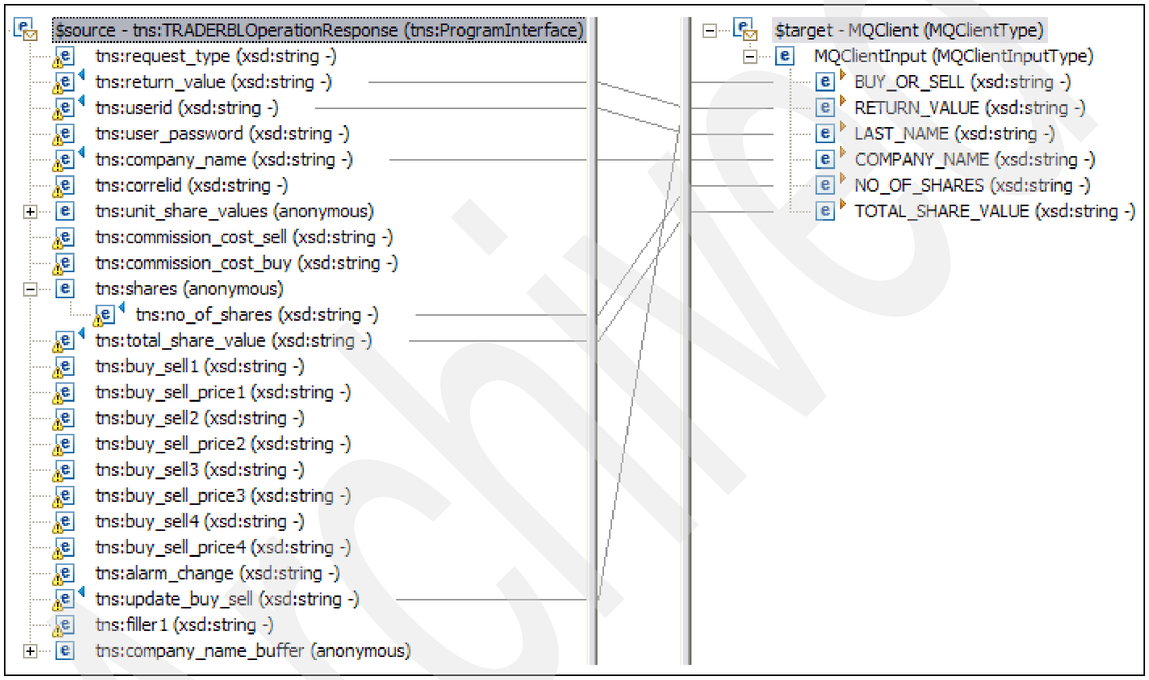
  The physical format of the target message, XML1.

The remaining fields in the Properties tree can be copied unchanged.

Do the following steps:

1. In the main map, expand the source panel by selecting **Envelope** → **Body** → **choice** → **sequence** → **message** → **Wildcard Message** and the target panel down to MQClient.
2. Click **Wildcard Message** in the source panel and **MQClient** in the target panel. Select **Map** → **Create New Submap** from the menu bar located at the top of the Application Development Perspective. A new submap is created.

Figure 4-10 shows how to map the message payload.



```
$source - tns:TRADERBLOperationResponse (tns:ProgramInterface)
    tns:request_type (xsd:string -)
    tns:return_value (xsd:string -)
    tns:userid (xsd:string -)
    tns:user_password (xsd:string -)
    tns:company_name (xsd:string -)
    tns:correlid (xsd:string -)
    tns:unit_share_values (anonymous)
    tns:commission_cost_sell (xsd:string -)
    tns:commission_cost_buy (xsd:string -)
    tns:shares (anonymous)
        tns:no_of_shares (xsd:string -)
    tns:total_share_value (xsd:string -)
    tns:buy_sell1 (xsd:string -)
    tns:buy_sell_price1 (xsd:string -)
    tns:buy_sell2 (xsd:string -)
    tns:buy_sell_price2 (xsd:string -)
    tns:buy_sell3 (xsd:string -)
    tns:buy_sell_price3 (xsd:string -)
    tns:buy_sell4 (xsd:string -)
    tns:buy_sell_price4 (xsd:string -)
    tns:alarm_change (xsd:string -)
    tns:update_buy_sell (xsd:string -)
    tns:filler1 (xsd:string -)
    tns:company_name_buffer (anonymous)

$target - MQClient (MQClientType)
    MQClientInput (MQClientInputType)
        BUY_OR_SELL (xsd:string -)
        RETURN_VALUE (xsd:string -)
        LAST_NAME (xsd:string -)
        COMPANY_NAME (xsd:string -)
        NO_OF_SHARES (xsd:string -)
        TOTAL_SHARE_VALUE (xsd:string -)
```

Figure 4-10 TargetToClient message map

The submap also trims the field, called company, down to three characters using the esql function left(). The code is as follows:

```
esql:left($source/tns:company_name,3)
```

Node name: RestoreMQMD

Default properties can be accepted in this Compute node. The relevant ESQL computed logic is detailed in Example 4-7. The node rebuilds a standard MQ reply message using the information previously stored in the Environment tree.

Example 4-7 ESQL code for RestoreMQMD Node

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
   -- CALL CopyMessageHeaders();
   -- CALL CopyEntireMessage();
   SET OutputRoot.Properties = InputRoot.Properties;
   SET OutputRoot.MQMD = Environment.Temp.MQMD;
   SET OutputRoot.MQMD.MsgType = MQMT_REPLY;
   SET OutputRoot.MRM = InputRoot.MRM;

   RETURN TRUE;
END;
```

Node name: MQReply

This node delivers the message to the queue specified in the ReplyToQ fields. The properties were left as the default values.

Testing the message flow

In order to test this message flow, you must create a BAR file containing the following elements:

- MF_TraderMQToSoap.msgflow
- MS_TraderMQClient
- MS_TraderSoapTarget

Deploy the file to the run time broker.

The MQ queue called TRADER.MQTOSOAP.INPUT is also required.

The MQ reply is delivered to the client, as specified in the ReplyTo queue information of the request message.

MF_TraderSoapToMQ

The TraderSoapToMQ sample flow demonstrates how to mediate between a Web service application issuing a SOAP/HTTP message and a target CICS application that is invoked through the MQ CICS Bridge through a message built according to a COBOL copy book structure. This flow implements a typical Request-Reply model.

It is the responsibility of the WebSphere Message Broker message flow to build a standard MQ message and to specify which MQReplyToQ from which it gets the response message.

It is also the responsibility of the WebSphere Message Broker message flow to collect the RequestIdentifier from the LocalEnvironment when the request is received by the HTTPInput node, and ensure that the same value is forwarded to the corresponding HTTPReply node.

The RequestIdentifier is copied to the MQMD.MsgId of the message that is sent to the MQ CICS Bridge. The MQ CICS Bridge returns this value in the MQMD.CorrelId of the response message.

The message flow for the TraderSoapToMQ sample does the following tasks:

1. Request Branch:
   a. The HTTP Input node receives a SOAP/HTTP request from the client application.
   b. The ClientToTarget Mapping node converts the client SOAP request to a COBOL-like message to fit the format expected by the MQ CICS Bridge.
   c. The BuildMQmessage Compute node builds a standard MQ message and sets the required values in the Properties tree and the MQMD structure.
   d. The TRADER.RequestToCics MQOutput node delivers the message to CICS through the MQ CICS Bridge.
2. Response Branch:
   a. The TRADER.ReplyFromCics MQInput node retrieves the response from the MQ CICS Bridge.
   b. The SetHTTPid Compute node rebuilds the RequestIdentifier in the LocalEnvironment so that the response message is sent to the correct SOAP/HTTP client. It also removes the MQMD.
   c. The TargetToClient Mapping node converts the COBOL-like message format to a SOAP/HTTP response, as expected by the client application.
   d. The HTTP Reply node delivers the response message to the client application Web.

The TraderSoapToMQ sample message flow is shown in Figure 4-11.

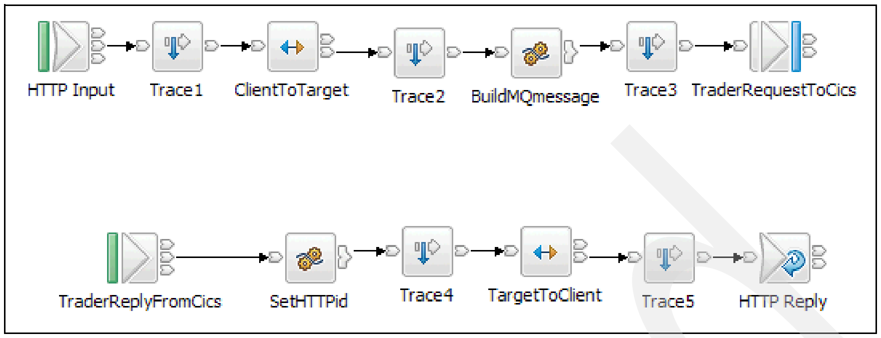


Figure 4-11 TraderSoapToMQ sample message flow

Node name: HTTP Input

See Table 4-10 for a description of the property values changed from the defaults.

Table 4-10 Properties values

Property	Property name: value	Comment
Basic	URL Selector: /soaptomq	Defines the context portion of the URL from which this node receives Web Service requests.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderSoapClient	The message set describing the incoming message.
	Message Type: Envelope	The message type describing the incoming message. All the messages received by this node will conform the structure of an Envelope message
	Message Format: XML1	The physical format of the incoming message.

Node name: ClientToTarget

This map transforms the client SOAP application request message to the target MQ request message. The map is split into a main map and a submap. The submap is invoked by the main map. The main map is responsible for mapping the values in the Properties tree and establishing the link between the incoming message body within the Envelope and the outgoing message.

The following three Property values have to be set manually in the main map:

- MessageSet

  The unique message set identifier of the target message set MS_TraderMQTarget.

- MessageType

  The message type name of the target message, MQCics.

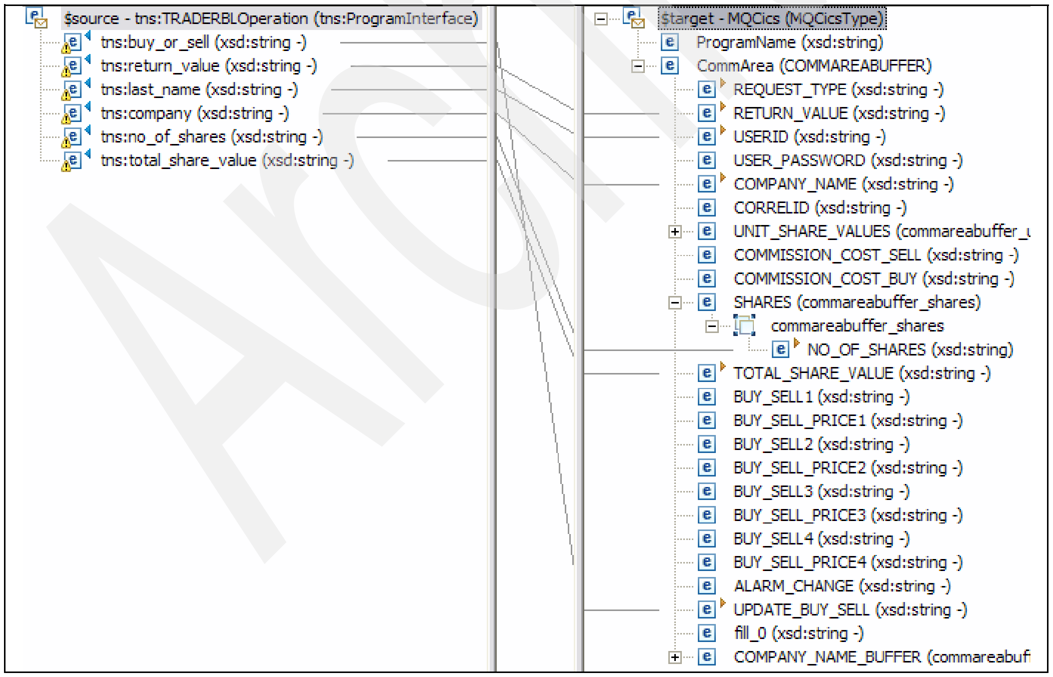
- MessageFormat

  The physical format of the target message, CWF1.

The remaining fields in the Properties tree can be copied unchanged. Perform the following steps:

1. In the main map, expand the source panel by selecting **Envelope** → **Body** → **choice** → **sequence** → **message** → **Wildcard Message** and the target panel down to MQCics.
2. Click **Wildcard Message** in the source panel and **MQCics** in the target panel. Select **Map** → **Create New Submap** from the menu bar located at the top of the Application Development Perspective. A new submap is created.

Figure 4-12 shows how to map the message payload.



```
$source - tns:TRADERBLOperation (tns:ProgramInterface)
    tns:buy_or_sell (xsd:string -)
    tns:return_value (xsd:string -)
    tns:last_name (xsd:string -)
    tns:company (xsd:string -)
    tns:no_of_shares (xsd:string -)
    tns:total_share_value (xsd:string -)
```

```
$target - MQCics (MQCicsType)
    ProgramName (xsd:string)
    CommArea (COMMAREABUFFER)
        REQUEST_TYPE (xsd:string -)
        RETURN_VALUE (xsd:string -)
        USERID (xsd:string -)
        USER_PASSWORD (xsd:string -)
        COMPANY_NAME (xsd:string -)
        CORRELID (xsd:string -)
        UNIT_SHARE_VALUES (commareabuffer_u
        COMMISSION_COST_SELL (xsd:string -)
        COMMISSION_COST_BUY (xsd:string -)
        SHARES (commareabuffer_shares)
            commareabuffer_shares
                NO_OF_SHARES (xsd:string)
        TOTAL_SHARE_VALUE (xsd:string -)
        BUY_SELL1 (xsd:string -)
        BUY_SELL_PRICE1 (xsd:string -)
        BUY_SELL2 (xsd:string -)
        BUY_SELL_PRICE2 (xsd:string -)
        BUY_SELL3 (xsd:string -)
        BUY_SELL_PRICE3 (xsd:string -)
        BUY_SELL4 (xsd:string -)
        BUY_SELL_PRICE4 (xsd:string -)
        ALARM_CHANGE (xsd:string -)
        UPDATE_BUY_SELL (xsd:string -)
        fill_0 (xsd:string -)
        COMPANY_NAME_BUFFER (commareabuf
```

Figure 4-12 ClientToTarget message map

Furthermore, the submap also builds the following fields:

- REQUEST_TYPE is set to the fixed value 'Buy_Sell'.
- COMPANY_NAME is set through the user ESQL function, Map_Company_Name, which receives the source field company as its input. The sample shown in Example 4-8 describes the ESQL code of this user function.

Example 4-8 ESQL code for ClientToTarget Node

```
CREATE FUNCTION Map_Company_Name (ShortName CHAR) RETURNS CHAR
   BEGIN
      DECLARE LongName CHAR;
      IF ShortName='CAS' THEN
         SET LongName='Casey_Import_Export';
      ELSEIF ShortName='GLP' THEN
         SET LongName='Glass_and_Luget_Plc';
      ELSEIF ShortName='HWE' THEN
         SET LongName='Headworth_Electrical';
      ELSE
         SET LongName='IBM';
      END IF;
   RETURN LongName;
   END;
```

Node name: BuildMQmessage

Default properties can be accepted in this Compute node. As the incoming message type is HTTP, this node is required so that a valid MQ message is built. The node also sets the field, ReplyIdentifier in the Property tree, to the value `MQCI_NEW_SESSION`. This value is automatically copied to the CorrelId so that a new session with the MQ CICS Bridge is initiated. The relevant ESQL computed logic is detailed in Example 4-9.

Example 4-9 ESQL computed logic for BuildMQmessage Node

```
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      -- CALL CopyMessageHeaders();
      -- CALL CopyEntireMessage();
      SET OutputRoot.Properties = InputRoot.Properties;
      SET OutputRoot.Properties.ReplyIdentifier = MQCI_NEW_SESSION;
      SET OutputRoot.Properties.ReplyProtocol = 'MQ';
      SET OutputRoot.MQMD.Version = 2;
      SET OutputRoot.MQMD.MsgType = MQMT_REQUEST;
      SET OutputRoot.MQMD.MsgId =
InputLocalEnvironment.Destination.HTTP.RequestIdentifier;
      SET OutputRoot.MQMD.CorrelId = MQCI_NEW_SESSION;
      SET OutputRoot.MQMD.ReplyToQ = 'TRADER.CICSREPQ.SOAPMQ';
      SET OutputRoot.MQMD.Format = MQFMT_STRING;
      SET OutputRoot.MRM = InputRoot.MRM;
      RETURN TRUE;
   END;
```

Node name: TRADER.RequestToCics

See Table 4-11 for a description of the property values changed from the defaults.

Table 4-11 Properties values

Property	Property name: value	Comment
Basic	Queue Name: SYSTEM.CICS.BRIDGE.QUEUE	Defines the queue from which the MQ CICS Bridge reads the message.

Node name: TRADER.ReplyFromCics

This is the node from which the response from CICS is read. See Table 4-12 for a description of the property values changed from the defaults.

Table 4-12 Properties values

Property	Property name: value	Comment
Basic	Queue Name: TRADER.CICSREPQ.SOAPMQ	Defines the queue from which the input node is reading.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderMQTarget	The message set describing the incoming message.
	Message Type: MQCics	The message type describing the incoming message.
	Message Format: CWF1	The physical format of the incoming message.

Node name: SetHTTPid

The node rebuilds the RequestIdentifier in the LocalEnvironment so that the response message is sent to the correct SOAP/HTTP client. It also removes the MQMD, because this is not required by the HTTP protocol.

The relevant ESQL computed logic is detailed in Example 4-10.

Example 4-10 ESQL computed logic for SetHTTPid Node

```
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      -- CALL CopyMessageHeaders();
      CALL CopyEntireMessage();

      SET OutputLocalEnvironment.Destination.HTTP.RequestIdentifier =
InputRoot.MQMD.CorrelId;
      SET OutputRoot.Properties.ReplyProtocol = 'SOAP-HTTP';
      SET OutputRoot.MQMD = NULL;
      RETURN TRUE;
   END;
```

Table 4-13 details the property changes for this node.

Table 4-13 Properties values

Property	Property name: value	Comment
Basic	Compute Mode: LocalEnvironment and Message	Because you are making a change to the LocalEnvironment, you have to propagate both the LocalEnvironment and the Message.

Node name: TargetToClient

This map transforms the target MQ application response message to the client SOAP response message. The map is split into a main map and a submap. The submap is invoked by the main map. The main map is responsible for mapping the values in the Properties tree and establishing the link between the incoming message and the body within the Envelope of the outgoing message.

The following three Property values have to be set manually in the main map:

- MessageSet

  The unique message set identifier of the target message set MS_TraderSoapClient.

- MessageType

  The message type name of the target message, Envelope.

- MessageFormat

  The physical format of the target message, XML1.

The remaining fields in the Properties tree can be copied unchanged.

Perform the following steps:

1. In the main map, expand the source panel down to **MQCics** and the target panel down to **Envelope** → **Body** → **choice** → **sequence** → **message** → **Wildcard Message**.
2. Click **MQCics** in the source panel and **Wildcard Message** in the target panel. Select **Map** → **Create New Submap** from the menu bar located at the top of the Application Development Perspective. A new submap is created.

Figure 4-13 shows how to map the message payload.

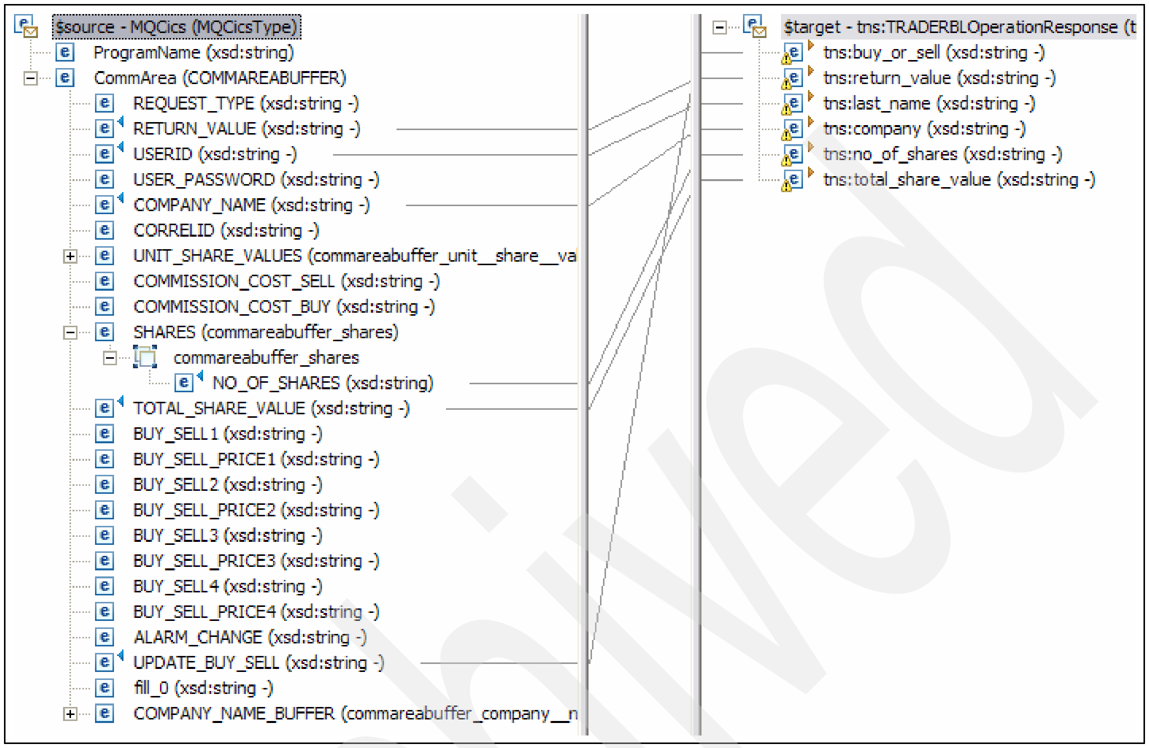

```
$source - MQCics (MQCicsType)
  ProgramName (xsd:string)
  CommArea (COMMAREABUFFER)
    REQUEST_TYPE (xsd:string -)
    RETURN_VALUE (xsd:string -)
    USERID (xsd:string -)
    USER_PASSWORD (xsd:string -)
    COMPANY_NAME (xsd:string -)
    CORRELID (xsd:string -)
    UNIT_SHARE_VALUES (commareabuffer_unit__share__val
    COMMISSION_COST_SELL (xsd:string -)
    COMMISSION_COST_BUY (xsd:string -)
    SHARES (commareabuffer_shares)
      commareabuffer_shares
        NO_OF_SHARES (xsd:string)
    TOTAL_SHARE_VALUE (xsd:string -)
    BUY_SELL1 (xsd:string -)
    BUY_SELL_PRICE1 (xsd:string -)
    BUY_SELL2 (xsd:string -)
    BUY_SELL_PRICE2 (xsd:string -)
    BUY_SELL3 (xsd:string -)
    BUY_SELL_PRICE3 (xsd:string -)
    BUY_SELL4 (xsd:string -)
    BUY_SELL_PRICE4 (xsd:string -)
    ALARM_CHANGE (xsd:string -)
    UPDATE_BUY_SELL (xsd:string -)
    fill_0 (xsd:string -)
    COMPANY_NAME_BUFFER (commareabuffer_company__n

$target - tns:TRADERBLOperationResponse (t
  tns:buy_or_sell (xsd:string -)
  tns:return_value (xsd:string -)
  tns:last_name (xsd:string -)
  tns:company (xsd:string -)
  tns:no_of_shares (xsd:string -)
  tns:total_share_value (xsd:string -)
```

Figure 4-13 TargetToClient message map

The submap also trims the field, called company, down to three characters using the esql function left(). The code is as follows:

```
esql:left($source/CommArea/COMPANY_NAME,3)
```

Node name: HTTPReply

This node delivers the HTTP reply to the client application. The properties were left as the default values.

Testing the message flow

In order to test this message flow, you must create a BAR file containing the following elements:

- ► MF_TraderSoapToMQ.msgflow
- ► MS_TraderSoapQClient
- ► MS_TraderMQTarget

Deploy the file to the run time broker.

The following MQ queues are also required:

- SYSTEM.CICS.BRIDGE.QUEUE (the default for the MQ CICS Bridge support)
- TRADER.CICSREPQ.SOAPMQ

4.2.3 MQ monitoring message flows

When we ran the flows, we used MQ queue monitoring to determine which queue manager has served in different parts of the flow. This also proved to be a good way to demonstrate high availability capabilities, as discussed in "High availability provided by shared queues" on page 56. Where appropriate, these are commented on within the screen captures that follow.

The information of interest is the time stamp information recorded for the last GET and the last PUT operation run within a particular queue and the length of time a message has spent on the queue.

You get this information out of MQ by setting the queue monitoring option to its highest level, using the following command on a per queue basis:

```
ALTER QLOCAL(<qname>) MONQ(HIGH)
```

Display the queue manager MONQ option using the following command:

```
DISPLAY QMGR MONQ
```

Ensure this is *not* set to "OFF". All other settings are fine; the default is "NONE".

Queue monitoring of the TraderMQtoMQ message flow

This section applies to the TraderMQtoMQ message flow described in detail in "MF_TraderMQtoMQ" on page 201.

We have summarized the queue names and what they are used for.

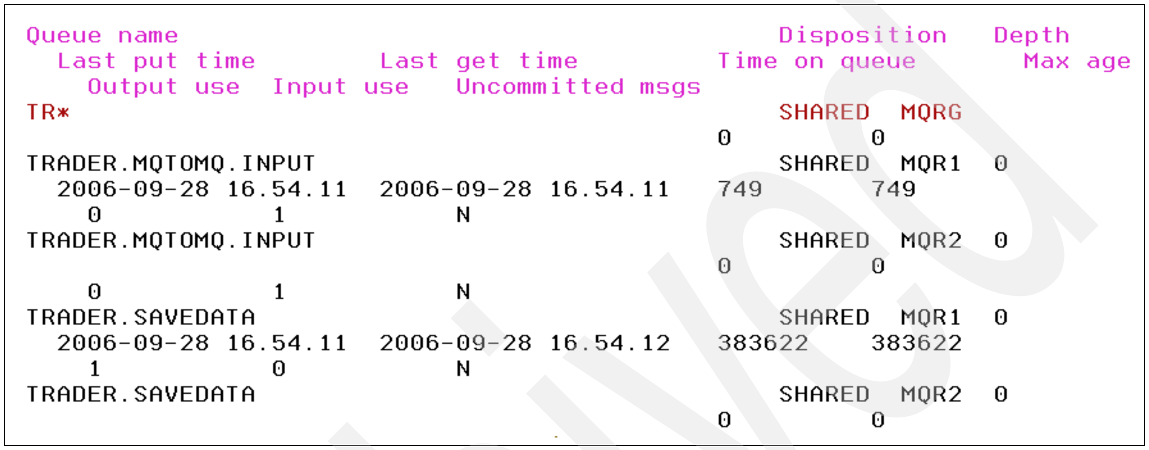
TRADER.MQTOMQ.INPUTQ	The queue where the MQ client application puts its input (request) message and from where the WebSphere Message Broker message flow starts.
TRADER.SAVEDATA	To store the name of the client's reply-to queue.
SYSTEM.CICS.BRIDGE.QUEUE	The request flow output queue, which is also the CICS input queue.
TRADER.CICSREPQ	Where the CICS Bridge puts its output, which is the input queue for the reply message flow.
TRADER.CLIENTREPQ	To deliver the final reply message to the client, which is the reply flow output queue.

Capture one - part 1: input and request flow execution

We ran the flow by issuing a single input message. We then displayed the queue status of the shared trader flow queues using the following command:

```
(DISPLAY QSTATUS(TRADER.*)).
```

The results for the first two queues involved in the flow are shown in Figure 4-14.



```
Queue name                                       Disposition   Depth
  Last put time        Last get time         Time on queue       Max age
    Output use  Input use   Uncommitted msgs
TR*                                              SHARED  MQRG
                                             0         0
TRADER.MQTOMQ.INPUT                              SHARED  MQR1  0
  2006-09-28 16.54.11  2006-09-28 16.54.11   749       749
    0           1           N
TRADER.MQTOMQ.INPUT                              SHARED  MQR2  0
                                             0         0

    0           1           N
TRADER.SAVEDATA                                  SHARED  MQR1  0
  2006-09-28 16.54.11  2006-09-28 16.54.12   383622    383622
    1           0           N
TRADER.SAVEDATA                                  SHARED  MQR2  0
                                             0         0
```

Figure 4-14 Queue monitoring status of request message flow queues

Interpretation of the queue monitoring data:

- There is only one time stamp, 16.54.11, which opens as the “Last put time” and the “Last get time” of TRADER.MQTOMQ.INPUTQ, both reported by queue manager MQR1.
- The TRADER.SAVEDATA queue also shows PUT and GET timestamps, with one second between them, both originating from queue manager MQR1.

This tells us:

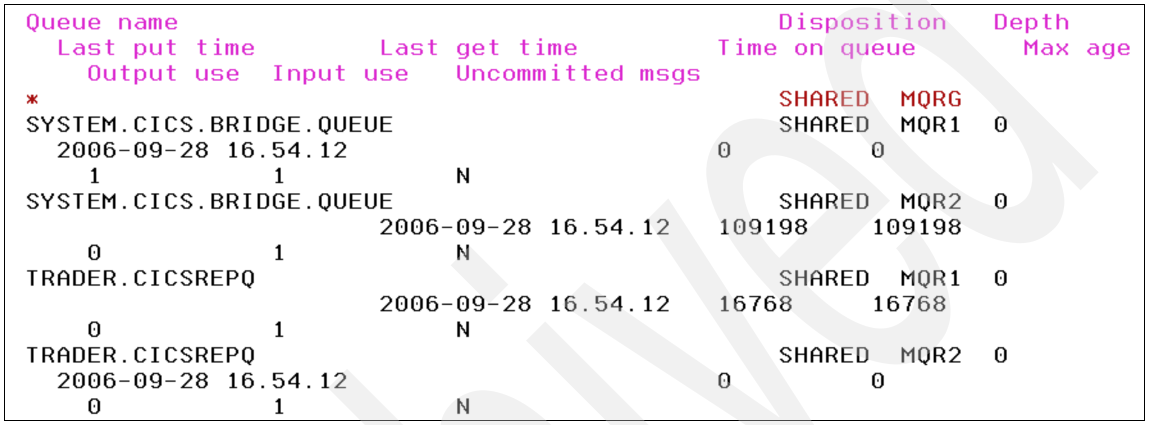
- MQR1 served the client application from where the request originated. In our case, it was the rfhutil.exe utility program connected to MQR1 via an MQ client connection.
- The MQGET from TRADER.MQTOMQ.INPUTQ was also done by MQR1. This indicated that the message flow was executed on LPAR1.

  The message spent 749 microseconds on the queue.
- Consequently, the message to TRADER.SAVEDATA queue is also written (PUT) through MQR1.
- The MQR1 GET time stamp displayed for TRADER.SAVEDATA already tells us that the reply message flow also ran on LPAR1.

The 383 milliseconds the message spent on this queue gives us an idea of the time the CICS related processing took.

Capture one - part 2: Flow output to and reply from CICS

Figure 4-15 shows the equivalent monitoring data for the final (output) queue of the request flow, which is the MQ CICS Bridge queue. This is also the queue where the CICS Bridge puts its reply (TRADER.CICSREPQ), which is the input queue to the reply flow.



```
Queue name                                       Disposition   Depth
  Last put time        Last get time        Time on queue        Max age
    Output use  Input use   Uncommitted msgs
*                                                SHARED  MQRG
SYSTEM.CICS.BRIDGE.QUEUE                         SHARED  MQR1  0
  2006-09-28 16.54.12                       0         0
    1           1           N
SYSTEM.CICS.BRIDGE.QUEUE                         SHARED  MQR2  0
                       2006-09-28 16.54.12   109198    109198
    0           1           N
TRADER.CICSREPQ                                  SHARED  MQR1  0
                       2006-09-28 16.54.12   16768     16768
    0           1           N
TRADER.CICSREPQ                                  SHARED  MQR2  0
  2006-09-28 16.54.12                       0         0
    0           1           N
```

Figure 4-15 Queue monitoring data of server application (CICS) input and output queue.

Here is our interpretation of the queue monitoring data:

- As expected, the PUT to the CICS Bridge queue was executed by MQR1, where the request flow ran.
- The GET, however, is shown to be done by MQR2. This means that the CICS Bridge on LPAR2 took the message and, therefore, the server program (CICS) also ran on LPAR2.
- According to the data, the reply message produced by CICS was placed to TRADER.CICSREPQ, also through MQR2 (last queue entry in the figure).
- We already knew from the GET access to the TRADER.SAVEDATA in Figure 4-14 on page 234 that the reply message flow was executed on LPAR1.

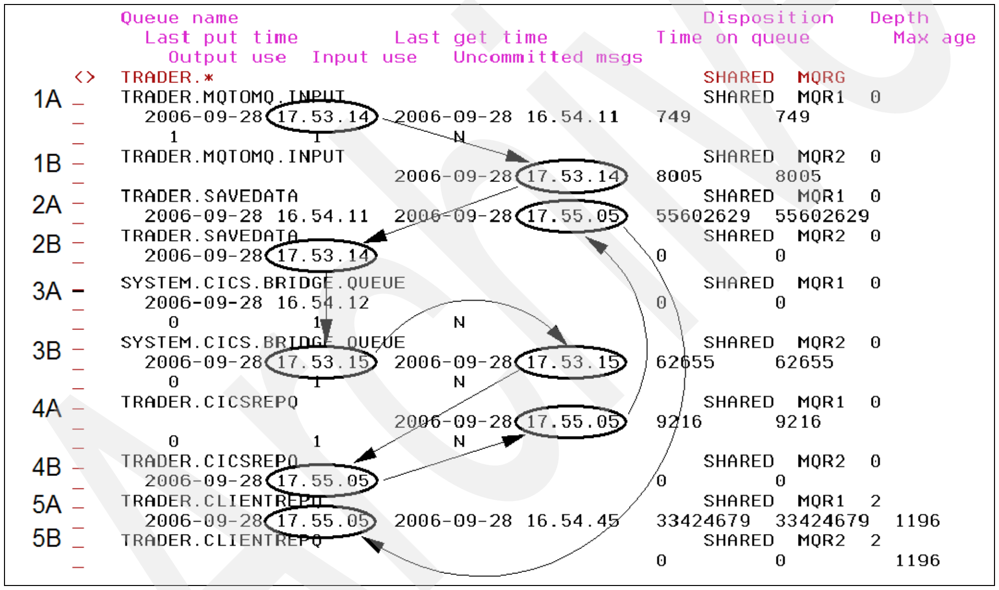
  This is confirmed by the GET for TRADER.CICSREPQ, shown as being done by MQR1.

It is worth pointing out that the ESB target server program (CICS) ran in an LPAR different from both the request and the reply message flow. In our case, this happened by chance, but it shows the capabilities of this configuration in terms of the high availability:

- If there was no target system (CICS) available on the LPAR where the request flow ran, the target application could pick up the message and run.
- If there was no message flow or broker available on the LPAR where the CICS application ran, the reply message would be processed by the reply flow and put to the application reply-to queue. (This is not shown in the figures.)

Capture two: With resources specifically disabled

For the second monitoring data capture, we applied some manipulations that simulated non-availability of resources at particular times. We used the same flow as before, with both the request and the reply message flow shown by the (composited) screen capture shown in Figure 4-16.



```
        Queue name                                        Disposition    Depth
          Last put time           Last get time       Time on queue       Max age
            Output use   Input use   Uncommitted msgs
    <>  TRADER.*                                          SHARED   MQRG
1A  _   TRADER.MQTOMQ.INPUT                               SHARED   MQR1   0
    _     2006-09-28 17.53.14   2006-09-28 16.54.11   749         749
    _       1          1           N
1B  _   TRADER.MQTOMQ.INPUT                               SHARED   MQR2   0
    _                           2006-09-28 17.53.14   8005        8005
2A  _   TRADER.SAVEDATA                                   SHARED   MQR1   0
    _     2006-09-28 16.54.11   2006-09-28 17.55.05   55602629   55602629
2B  _   TRADER.SAVEDATA                                   SHARED   MQR2   0
    _     2006-09-28 17.53.14                         0           0
3A  -   SYSTEM.CICS.BRIDGE.QUEUE                          SHARED   MQR1   0
    _     2006-09-28 16.54.12                         0           0
    _       0          1           N
3B  _   SYSTEM.CICS.BRIDGE.QUEUE                          SHARED   MQR2   0
    _     2006-09-28 17.53.15   2006-09-28 17.53.15   62655       62655
    _       0          1           N
4A  _   TRADER.CICSREPQ                                   SHARED   MQR1   0
    _                           2006-09-28 17.55.05   9216        9216
    _       0          1           N
4B  _   TRADER.CICSREPQ                                   SHARED   MQR2   0
    _     2006-09-28 17.55.05                         0           0
5A  _   TRADER.CLIENTREPQ                                 SHARED   MQR1   2
    _     2006-09-28 17.55.05   2006-09-28 16.54.45   33424679   33424679   1196
5B  _   TRADER.CLIENTREPQ                                 SHARED   MQR2   2
    _                                                 0           0          1196
```

Figure 4-16 Full MQtoMQ flow, both request and reply, as reflected by queue monitoring

Interpretation of the queue monitoring data is as follows:

- For simplification of the explanation, queue entries have been numbered 1...5. Where the numbers are appended with an “A”, LPAR1 was involved. Where the numbers are appended with a “B”, LPAR2 was involved.

  ** Start of request message flow**

Before we put the request message, we stopped the TraderMQtoMQ message flow on the LPAR1 broker, so that the request flow could only execute on LPAR2.

- (1A/PUT) The message has been put to the flow input queue by MQR1. We used the same client connection as before to write the message.
- (1B/GET) Delivery of the 17.53.14 message to the message flow is, as expected, served by queue manager MQR2, proving that the flow started on LPAR2.
- (2B/PUT) Consequently, the TRADER.SAVEDATA queue is also written too by MQR2. The same is true for the CICS Bridge queue, which is the request flow output message (3B/PUT).

End of request message flow

- ► We did not influence the choice of the CICS region, but we trapped the CICS Bridge task (CKBP) by starting the CEDX execution diagnostic facility against that transaction code on both regions ("CEDX CKBP") so that we could discover which CICS region had been selected.
- ► In this run, the Bridge task was started in the CICS region on LPAR2.
- ► Before we let the CICS task run, we stopped the TraderMQtoMQ message flow within the MQR2BRK (LPAR2) execution group, but activated the same flow in MQR1BRK (LPAR1). We wanted to prove that the reply message flow could be executed by broker MQR1BRK (LPAR1).

(3B/GET) This entry represents the CICS Bridge task on LPAR2, which takes the message from the Bridge queue, as expected. With (4B/PUT), the reply message written by the CICS task is inevitably written through MQR2 as well. The long time between these two entries (1:50 min.) comes from our manual intervention, as described previously.

** Start of reply message flow**

- (4A/GET) The reply message flow starts immediately on LPAR1 and gets the message from TRADER.CICSREPQ using MQR1, which gets the message within 9,216 microseconds of the CICS Bridge having placed it there.
- (2A/GET) The reply message flow gets the message through MQR1 from the TRADER.SAVEDATA queue in order to identify the external reply-to queue.
- (5A/PUT) The reply message flow puts the final reply message to the final reply-to queue TRADER.CLIENTREPQ, served by MQR1, as expected.
- (5A/5B) We have not taken the message from the TRADER.CLIENTREPQ queue for this, or the previous run, hence the message count of 2.

End of reply message flow

Survey and conclusions:

- Input message was written on LPAR1
- Request message flow executed on LPAR2
- Back-end CICS server task ran in LPAR2

All processes ran cleanly and the performance was high, demonstrating the strengths of this configuration in terms of both throughput and high availability.

4.3 WebSphere ESB implementation and deployment

Two WebSphere ESB scenarios are developed as follows:

- SOAP to SOAP

  A client supporting Web services sends a SOAP/HTTP request to the ESB. The ESB transforms the request according to the target service interface and delivers a SOAP/HTTP request to the target application. The service interface conforms to the request/response model. The Web service provider environment, in this case, is the WS-enabled CICS application.

- JMS to SOAP

  A JMS client sends an XML message to the ESB. The ESB transforms the request according to the target service interface and delivers a SOAP/HTTP request to the target application. The service interface conforms to the request/response model. The ESB transforms the SOAP/HTTP response to an XML message before returning the message to the client using JMS.

Figure 4-17 shows the sample scenarios described.

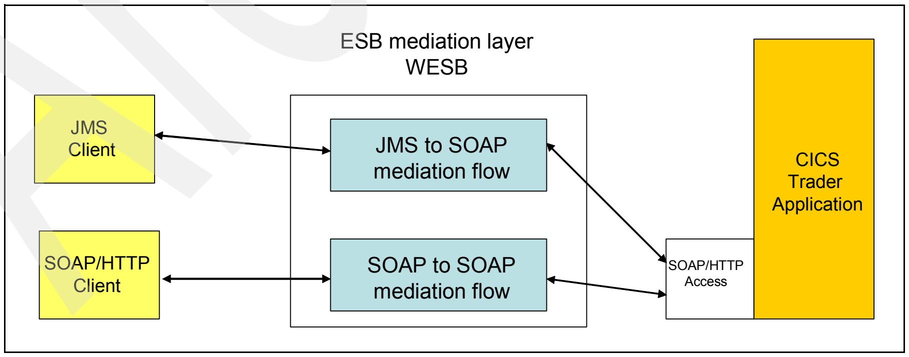


Figure 4-17 Mediation layer for WebSphere ESB

The files describing the interfaces and messages for each of the client and target applications are detailed in the files that you can download. Refer to Appendix B, “Additional material” on page 327 for further information.

This section details how the sample scenario’s previously described have been implemented. We provide instructions on how to create the libraries and mediation flows required by the implementation.

4.3.1 Mediation libraries

Two mediation libraries are used in the implementation of the sample scenarios:

- TraderClientLibrary

  This contains the Business Object definition and interface definition to support the incoming client request.

- TraderTargetLibrary

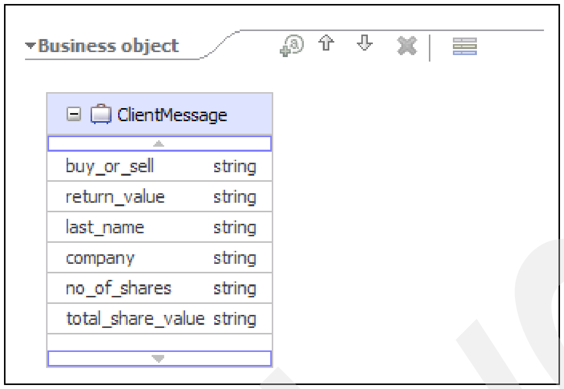
  This contains the Business Object definition and interface definition to support the target CICS application invocation.

TraderClientLibrary

Perform the following steps:

1. Start WebSphere Integration Developer and switch to the Business Integration perspective (this is usually the default perspective).
2. Select **File** → **New** → **Library** and create a new library named TraderClientLibrary.
3. Right-click **TraderClientLibary** and manually create a new Business Object. Select **New** → **Business Object**. Call it ClientMessage.

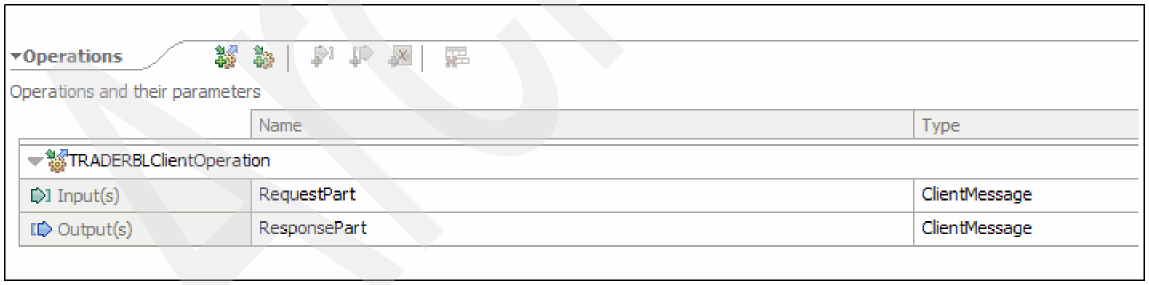
4. Add attributes to this object according to Figure 4-18. All the attributes are defined as strings.
5. You can find the instructions to download the ClientMessage.xsd schema file in Appendix B, “Additional material” on page 327.



ClientMessage	
buy_or_sell	string
return_value	string
last_name	string
company	string
no_of_shares	string
total_share_value	string

Figure 4-18 TraderClientLibary: ClientMessage Business Object

6. Create a new Interface by right-clicking **TraderClientLibary** and selecting **New** → **Interface**. Name this interface ClientInterface. Create a two-way operation named TRADERBLClientOperation. Add an input message RequestPart and an output message ResponsePart. Make sure both RequestPart and ResponsePart are of type ClientMessage. The Interface must look similar to Figure 4-19. You can find instructions to download the ClientInterface.wsdl file in Appendix B, “Additional material” on page 327.



	Name	Type
TRADERBLClientOperation		
Input(s)	RequestPart	ClientMessage
Output(s)	ResponsePart	ClientMessage

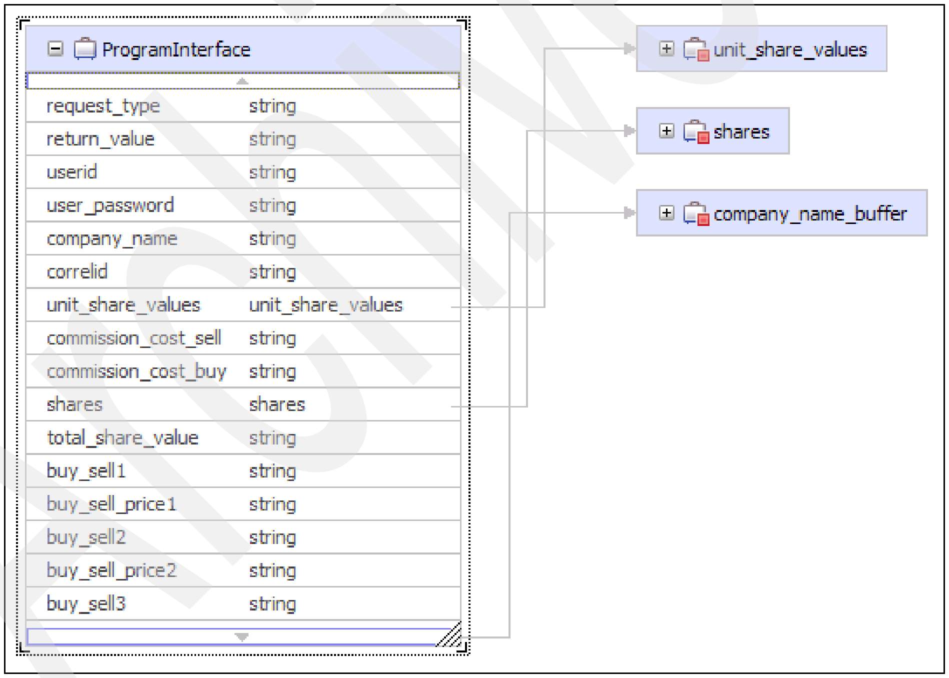
Figure 4-19 TraderClientLibary: ClientInterface Interface

TraderTargetLibary

Perform the following steps:

1. Start WebSphere Integration Developer and switch to the Business Integration perspective (this is usually the default perspective).
2. Select **File** → **New** → **Library** and create a new library named TraderTargetLibrary.
3. Import the CICS Interface TraderHttp.wsdl into the project. You can find instructions to download the TraderHttp.wsdl file in Appendix B, “Additional material” on page 327. Ensure that both the ProgramInterface Business Object and the TRADERBLPort Interface have been created.

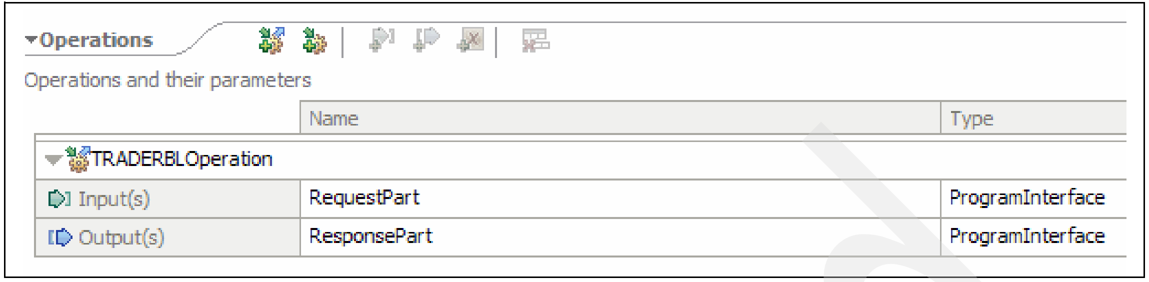
   You notice that there are two separate copies of the Business Object ProgramInterface. Each of these has the same fields, but different namespaces. You can check the namespaces by double-clicking each of the Business Objects and selecting the **Properties** tag. The ProgramInterface Business Object must look similar to Figure 4-20.



request_type	string
return_value	string
userid	string
user_password	string
company_name	string
correlid	string
unit_share_values	unit_share_values
commission_cost_sell	string
commission_cost_buy	string
shares	shares
total_share_value	string
buy_sell1	string
buy_sell_price1	string
buy_sell2	string
buy_sell_price2	string
buy_sell3	string

Figure 4-20 TraderTargetLibary: ProgramInterface Business Object

The TRADERBLPort Interface must look similar to Figure 4-21.



	Name	Type
TRADERBLOperation		
Input(s)	RequestPart	ProgramInterface
Output(s)	ResponsePart	ProgramInterface

Figure 4-21 TraderTargetLibary: TRADERBLPort Interface

4.3.2 Mediation modules

This section consists of the descriptions of the mediation modules.

TraderSoapToSoap mediation module

The TraderSoapToSoap sample mediation demonstrates how to use WebSphere ESB to mediate between a Web service client application issuing a SOAP/HTTP message and a target CICS application exposed as a Web service that you can access through SOAP/HTTP.

The mediation module for the TraderSoapToSoap sample performs the following tasks:

1. The mediation is exposed as a Web service through the Export node so that the client can invoke it.
2. The ClientToTarget XSL Transformation node, within the Request flow, converts the client SOAP request to the target SOAP message format.
3. The Import node ToCICS invokes the target CICS application.
4. The TargetToClient XSL Transformation node, within the Response flow, converts the target SOAP response to the client SOAP message format.

Creating the TraderSoapToSoap Mediation Module

Perform the following steps:

1. Start WebSphere Integration Developer and switch to the Business Integration perspective (this is usually the default perspective).
2. Select **File** → **New** → **Mediation Module** and create a new mediation module named TraderSoapToSoap. Ensure the Target Runtime option is consistent with your target run time environment.

In the Select Required Libraries window, select the **TraderClientLibrary** and the **TraderTargetLibrary**. These are the libraries containing the client and target Business Object's and Interfaces.

3. Double-click the Assembly Diagram **TraderSoapToSoap**. Follow the steps to build the TraderSoapToSoap mediation module assembly diagram, as shown in Figure 4-22.

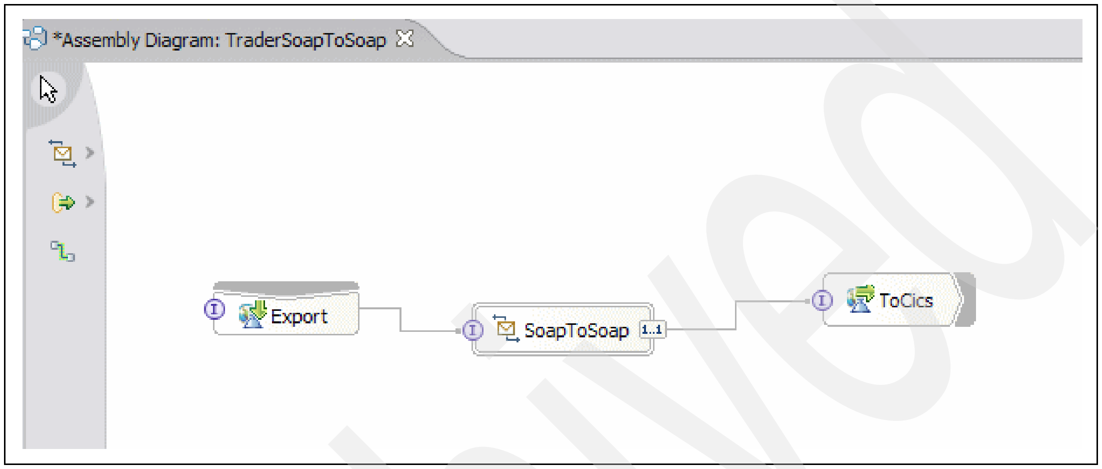


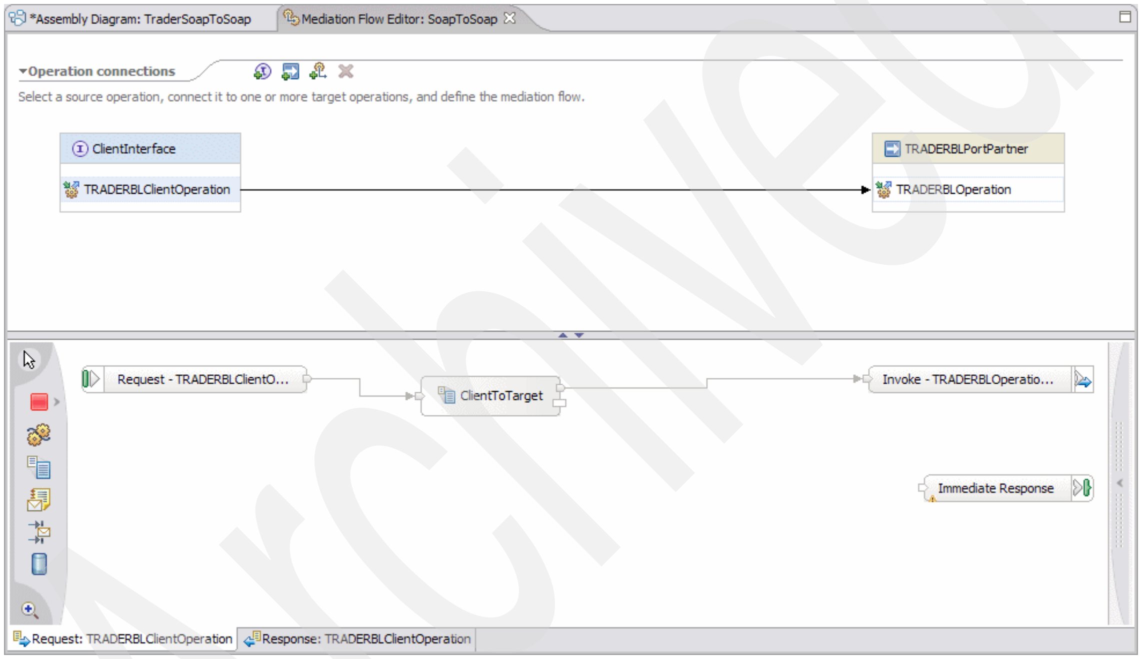
Figure 4-22 TraderSoapToSoap mediation module assembly diagram

4. Rename Mediation1 to SoapToSoap.
5. Using the menu above the Mediation component, select **Add Interface**. Choose **ClientInterface**. This is the input interface to the mediation.
6. Using the menu above the Mediation component, select **Add Reference**. Choose **TRADERBLPort**. This is the reference to the Web services that the mediation invokes.
7. From the TraderClientLibrary, drag the **ClientInterface** and drop it into the canvas. Select **Export with Web Service Binding** and click **OK**. You will be asked if a WSDL file must be automatically generated; select **Yes**. Select the SOAP/HTTP transport type. Rename Export1 to Export. This Export node allows Web services invocation of the mediation from an external Web services client.
8. From the TraderTargetLibrary, drag the **TRADERBLPort** interface and drop it into the canvas. Select **Import with Web Service Binding** and click **OK**. Rename Import1 **to `ToCics`. This** import node supports Web services binding to the CICS Web services application. The relevant Web services details, such as the target endpoint, are automatically generated according to the interface content.

9. Connect the Export node to the SoapToSoap mediation component and the mediation component to the ToCics node.
10. Right-click the SoapToSoap mediation component and select **Generate Implementation**. The mediation flow editor opens automatically.

Perform the following steps to build the request and response mediation flows:

1. Connect the TRADERBLClientOperation within the ClientInterface interface to the TRADERBLOperation within the TRADERBLPortPartner reference.
2. Ensure that the Request: **TRADERBLClientOperation** tag is selected. We create the request flow, as shown in Figure 4-23.



*Assembly Diagram: TraderSoapToSoap
Mediation Flow Editor: SoapToSoap
Operation connections
Select a source operation, connect it to one or more target operations, and define the mediation flow.
ClientInterface
TRADERBLClientOperation
TRADERBLPortPartner
TRADERBLOperation
Request - TRADERBLClientO...
ClientToTarget
Invoke - TRADERBLOperatio...
Immediate Response
Request: TRADERBLClientOperation
Response: TRADERBLClientOperation

Figure 4-23 Request: TRADERBLClientOperation mediation flow

3. From the node palette, select the **XSL Transformation** node and drop it into the canvas. Rename the node to ClientToTarget. Connect the Request node to the Transformation node and the Transformation node to the Invoke node.
4. Click the **ClientToTarget** node and select the **Details** section of the Properties tag. Click **New** to create a new map and accept the defaults.
5. Expand the **Source** and **Target** details.

6. Drag and drop the following fields from the Source to the Target:

buy_or_sell	update_buy_sell
return_value	return_value
last_name	user ID
no_of_shares	no_of_shares

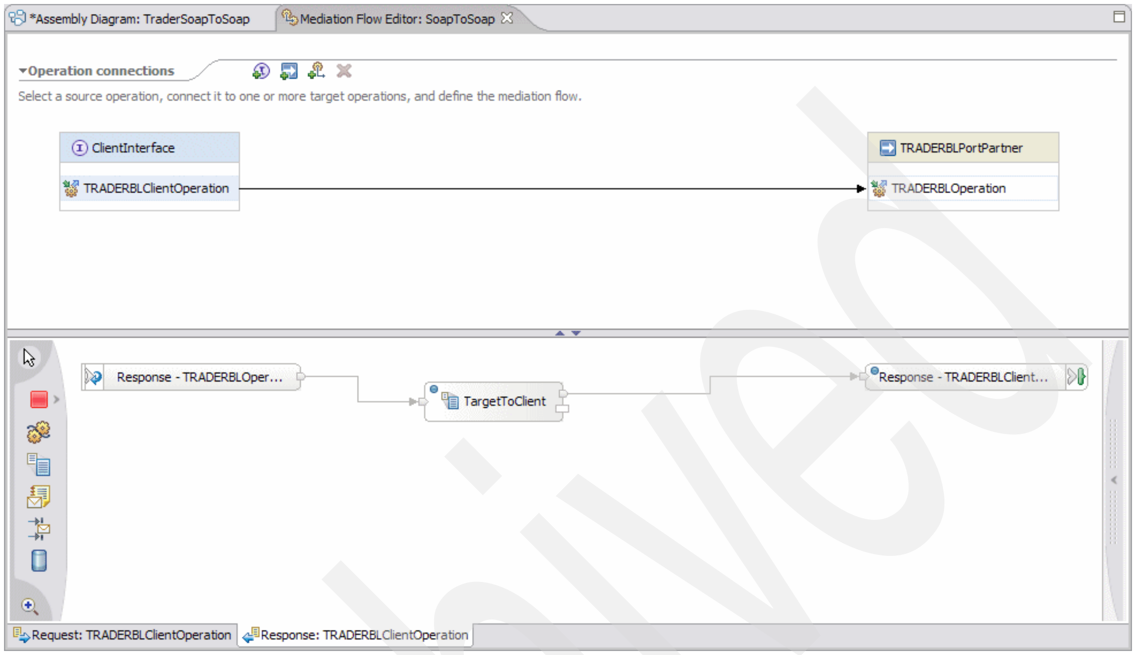
7. Right-click the target field, **request_type**, and select **Define XSLT Function**. Select the function type, **String**, and choose the Function name: **string**. Add an Input parameter: ‘Buy_Sell’.
8. Right-click **company_name** and select **XSL Choose**. Add the required xsl:choose options until you have built up the code to resemble the function shown in Example 4-11.

Example 4-11 XSL for ClientToTarget node

```
<xsl:choose>
  <xsl:when
test="body/TRADERBLClientOperation/RequestPart/company=string('CAS')
">
    <xsl:value-of select="string('Casey_Import_Export')"/>
  </xsl:when>
  <xsl:when
test="body/TRADERBLClientOperation/RequestPart/company=string('GLP')
">
    <xsl:value-of select="string('Glass_and_Luget_Plc')"/>
  </xsl:when>
  <xsl:when
test="body/TRADERBLClientOperation/RequestPart/company=string('HWE')
">
    <xsl:value-of select="string('Headworth_Electrical')"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="string('IBM')"/>
  </xsl:otherwise>
</xsl:choose>
```

9. After saving the mapping, go back to the Mediation Flow Editor and select **Regenerate XSL**.
10. The Request flow is now complete.

11. Select the Response: **TRADERBLClientOperation** tag. You are going to create the response flow, as shown in Figure 4-24.



*Assembly Diagram: TraderSoapToSoap
Mediation Flow Editor: SoapToSoap
Operation connections
Select a source operation, connect it to one or more target operations, and define the mediation flow.
ClientInterface
TRADERBLClientOperation
TRADERBLPortPartner
TRADERBLOperation
Response - TRADERBLOper...
TargetToClient
Response - TRADERBLClient...
Request: TRADERBLClientOperation
Response: TRADERBLClientOperation

Figure 4-24 Response: TRADERBLClientOperation mediation flow

12. From the node palette, select the **XSL Transformation** node and drop it into the canvas. Rename the node to TargetToClient. Connect the Response: TRADERBLOperation node to the Transformation node and the Transformation node to the Response: TRADERBLClientOperation node.

13. Click the **TargetToClient** node and select the **Details** section of the Properties tag. Click **New** to create a new map and accept the defaults.

14. Expand the **Source** and **Target** details.

15. Drag and drop the following fields from the Source to the Target:

update_buy_or_sell	buy_sell
return_value	return_value
userid	last_name
no_of_shares	no_of_shares
total_share_value	total_share_value

16. Right-click the target field, **company_name**, and select **Define XSLT Function**. Select the function type, **String**, and choose the Function name: **substring**.

Add the following input parameters:
   - /body/TRADERBLOperationResponse/tns:company_name/text()
   - 1
   - 3
17. After saving the mapping, go back to the Mediation Flow Editor and select **Regenerate XSL**.
18. The Response flow is now complete.

In order to invoke this mediation, you can develop a simple Web client, as described in “Creating a Web client to invoke the TraderSoapToSoap mediation” on page 254.

Testing the mediation

In order to run this mediation out of the WebSphere Integration Developer Integrated Test Environment, you can create an EAR file containing the involved J2EE elements. Select **File** → **Export** → **EAR file** and choose the Project **TraderSoapToSoapApp**. Also select a valid destination, for example as follows:

c:\temp\TraderSoapToSoapApp.ear

Deploy the EAR file to a running WebSphere ESB server.

TraderJmsToSoap mediation module

The TraderJmsToSoap sample mediation demonstrates how to use WebSphere ESB to mediate between a JMS client application using a JMS request/reply model and a target CICS application exposed as a Web service that can be accessed through SOAP/HTTP. Note that the client interface definition is the same one used in the previous TraderSoapToSoap scenario. The physical implementation of the Export node is different, in order to support JMS binding instead of SOAP binding.

The mediation module for the TraderJmsToSoap sample performs the following tasks:

1. The mediation exposes a JMS interface through the Export node so that it can be invoked by the client.
2. The ClientToTarget XSL Transformation node, within the Request flow, converts the client JMS request to the target SOAP message format.
3. The Import node ToCICS invokes the target CICS application.
4. The TargetToClient XSL Transformation node, within the Response flow, converts the target SOAP response to the client JMS message format.

Creating the TraderJmsToSoap Mediation Module

Perform the following steps:

1. Start WebSphere Integration Developer and switch to the Business Integration perspective (this is usually the default perspective).
2. Select **File** → **New** → **Mediation Module** and create a new mediation module named TraderJmsToSoap. Ensure the Target Runtime option is consistent with your target run time environment. In the Select Required Libraries panel, select the **TraderClientLibrary** and the **TraderTargetLibrary**. These are the libraries containing the Client and Target Business Objects and Interfaces.
3. Double-click the Assembly Diagram: **TraderJmsToSoap**. Follow the steps to build the TraderJmsToSoap mediation module assembly diagram, as shown in Figure 4-25.

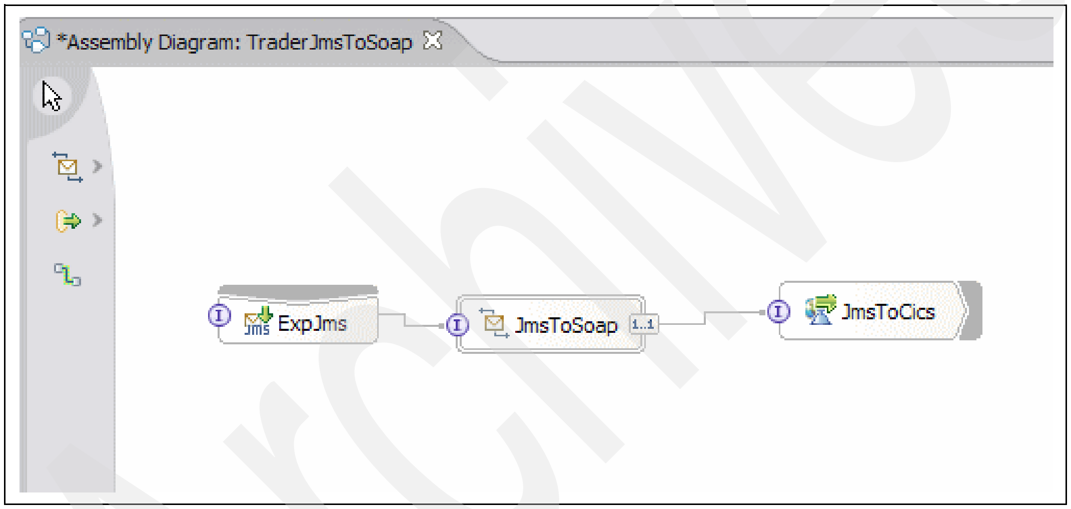


Figure 4-25 Assembly Diagram: TraderJmsToSoap

4. Rename Mediation1 to JmsToSoap.
5. Using the menu above the Mediation component, select **Add Interface**. Choose **ClientInterface**. This is the input interface to the mediation.
6. Using the menu above the Mediation component, select **Add Reference**. Choose **TRADERBLPort**. This is the reference to the Web services that the mediation invokes.
7. From the TraderClientLibrary, drag the **ClientInterface** and drop it into the canvas. Select **Export with no binding** and click **OK**. Rename Export1 to ExpJms.

8. Right-click **ExpJms** and select **Generate Binding** → **Jms Binding**. Although the system can automatically create all the JMS resources required, you want a better control on what is defined, so select **Use pre-configured messaging provider resources**.

   Ensure that the Serialization Type is chosen as **Business Object XML using JMS TextMessage**. This means that you can send requests to the mediation in the form of text XML messages.

   Enter the following Java Naming and Directory Interface™ (JNDI) values, as shown in Table 4-14, to point to the requested messaging resources.

Table 4-14 Properties values

Property name	**Property content**
JNDI name for Activation Specification	jms/TraderAS
JNDI name for Receive Destination	jms/TraderReceive
JNDI name for Send Destination	jms/TraderSend

9. There are still resources that have to be defined in the ExpJms node. Click this node and select **Properties** → **Binding** → **Endpoint** configuration. Select **JMS Destination** and **Callback Destination Properties**. Choose **Specify JNDI name for pre-configured messaging provider resource** and enter the value jms/TraderCallback.

   Select **Response Connection**. Choose **Specify JNDI name for pre-configured messaging provider resource** and in the property Managed Connection Factory JNDI Lookup Name, enter the value jms/TraderQCF.

10. From the TraderTargetLibrary, drag the TRADERBLPort interface and drop it into the canvas. Select **Import with Web Service Binding** and click **OK**. Rename Import1 to JmsToCics. This import node supports Web services binding to the CICS Web services application. The relevant Web services details, such as the target endpoint, are automatically generated according to the interface content.

11. Connect the ExpJms node to the JmsToSoap mediation component and the mediation component to the JmsToCics node.

12. Right-click the **JmsToSoap** mediation component and select **Generate Implementation**. The mediation flow editor opens automatically.

Perform the following steps to build the request and response mediation flows:

1. Connect the TRADERBLClientOperation within the ClientInterface interface to the TRADERBLOperation within the TRADERBLPortPartner reference.
2. Ensure that the Request: **TRADERBLClientOperation** tag is selected. Create the request flow, as shown in Figure 4-26.

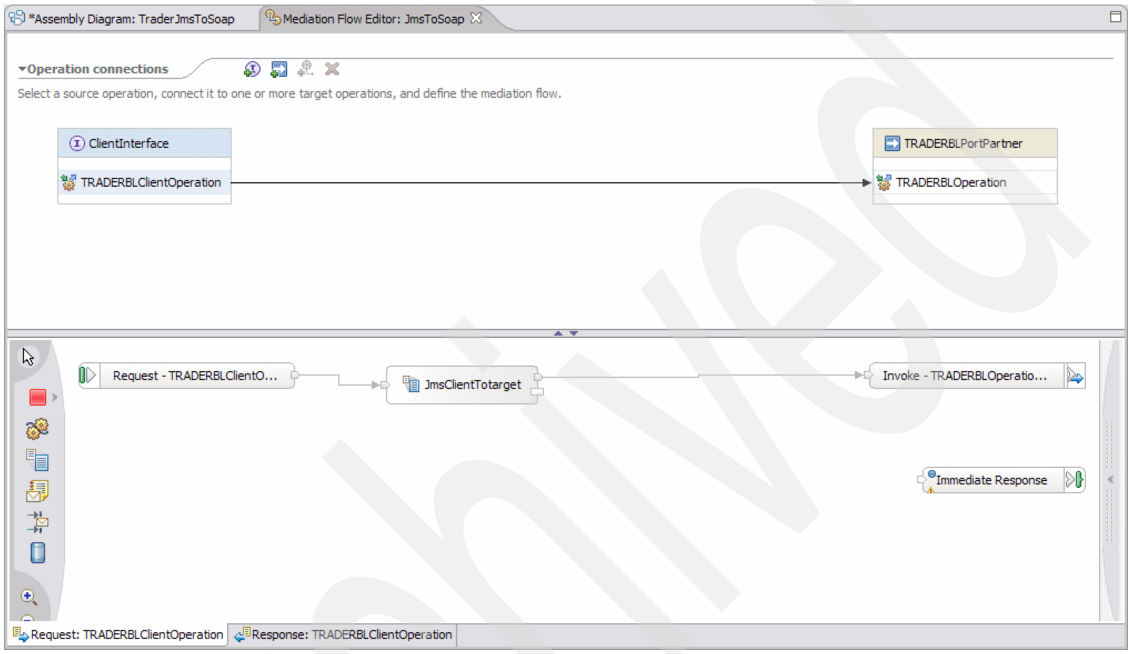

*Assembly Diagram: TraderJmsToSoap
Mediation Flow Editor: JmsToSoap
Operation connections
Select a source operation, connect it to one or more target operations, and define the mediation flow.
ClientInterface
TRADERBLClientOperation
TRADERBLPortPartner
TRADERBLOperation
Request - TRADERBLClientO...
JmsClientTotarget
Invoke - TRADERBLOperatio...
Immediate Response
Request: TRADERBLClientOperation
Response: TRADERBLClientOperation

Figure 4-26 Request: TRADERBLClientOperation mediation flow

3. From the node palette, select the **XSL Transformation** node and drop it into the canvas. Rename the node to JmsClientToTarget. Connect the Request node to the Transformation node and the Transformation node to the Invoke node.
4. Click the **JmsClientToTarget** node and select the **Details** section of the Properties tag. Click **New** to create a new map and accept the defaults.
5. Expand the **Source** and **Target** details.
6. Drag and drop the following fields from the Source to the Target:

buy_or_sell	update_buy_sell
return_value	return_value
last_name	userid
no_of_shares	no_of_shares

7. Right-click the target field, **request_type**, and select **Define XSLT Function**. Select the function type, **String**, and choose the Function name **string**. Add the Input parameter ‘Buy_Sell’.
8. Right-click **company_name** and select **XSL Choose**. Add the required xsl:choose options until you have built the code so that it resembles the function shown in Example 4-12.

Example 4-12 XSL for JmsClientToTarget node

```
<xsl:choose>
  <xsl:when
test="body/TRADERBLClientOperation/RequestPart/company=string('CAS')
">
    <xsl:value-of select="string('Casey_Import_Export')"/>
  </xsl:when>
  <xsl:when
test="body/TRADERBLClientOperation/RequestPart/company=string('GLP')
">
    <xsl:value-of select="string('Glass_and_Luget_Plc')"/>
  </xsl:when>
  <xsl:when
test="body/TRADERBLClientOperation/RequestPart/company=string('HWE')
">
    <xsl:value-of select="string('Headworth_Electrical')"/>
  </xsl:when>
  <xsl:otherwise>
    <xsl:value-of select="string('IBM')"/>
  </xsl:otherwise>
</xsl:choose>
```

9. After saving the mapping, go back to the Mediation Flow Editor and select **Regenerate XSL**.
10. The Request flow is now complete.

11. Select the Response: **TRADERBLClientOperation** tag. You will create the response flow, as shown in Figure 4-27.

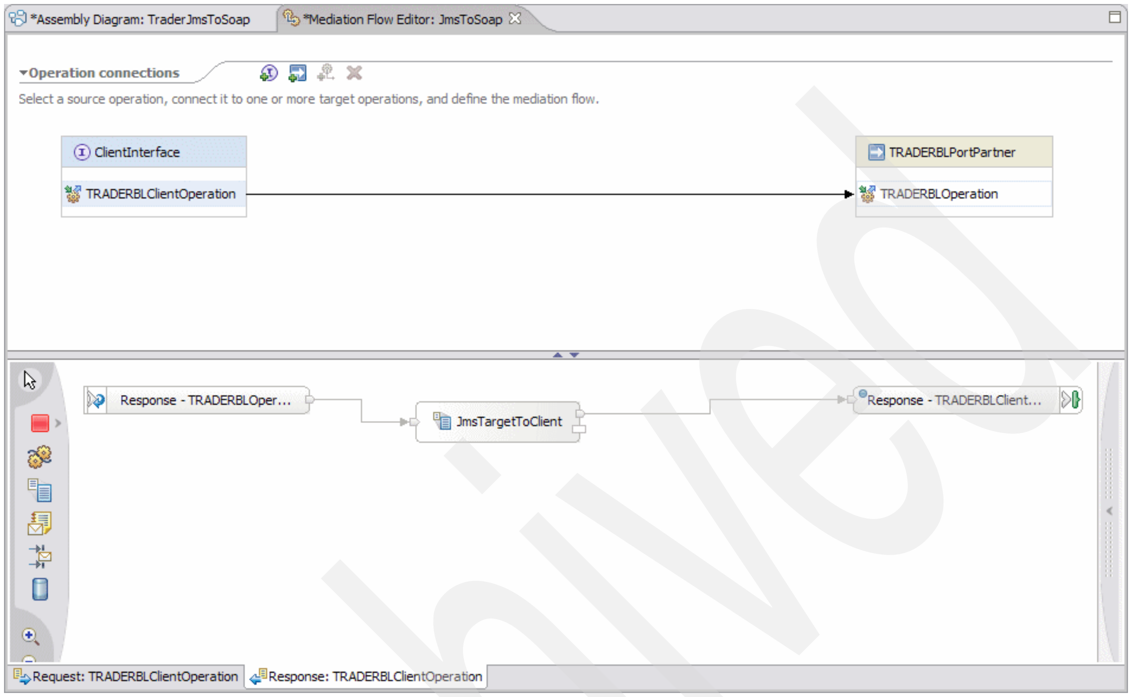

*Assembly Diagram: TraderJmsToSoap
*Mediation Flow Editor: JmsToSoap
Operation connections
Select a source operation, connect it to one or more target operations, and define the mediation flow.
ClientInterface
TRADERBLClientOperation
TRADERBLPortPartner
TRADERBLOperation
Response - TRADERBLOper...
JmsTargetToClient
Response - TRADERBLClient...
Request: TRADERBLClientOperation
Response: TRADERBLClientOperation

Figure 4-27 Response: TRADERBLClientOperation mediation flow

12. From the node palette, select the **XSL Transformation** node and drop it into the canvas. Rename the node to JmsTargetToClient. Connect the Response: TRADERBLOperation node to the Transformation node and the Transformation node to the Response: TRADERBLClientOperation node.

13. Click the **TargetToClient** node and select the **Details** section of the Properties tag. Click **New** to create a new map and accept the defaults.

14. Expand the **Source** and **Target** details.

15. Drag and drop the following fields from the Source to the Target:

update_buy_or_sell	buy_sell
return_value	return_value
userid	last_name
no_of_shares	no_of_shares
total_share_value	total_share_value

16. Right-click the target field, **company_name**, and select **Define XSLT Function**. Select the function type, **String**, and choose the Function name: **substring**. Add the following Input parameters:

    – /body/TRADERBLOperationResponse/tns:company_name/text()

    – 1

    – 3

17. After saving the mapping, go back to the Mediation Flow Editor and select **Regenerate XSL**.

18. The Response flow is now complete.

Resources that have to be defined in the application server

Several JMS resources have to be defined in the WebSphere Application Server that run the mediation. All the JMS resources must be defined on a WebSphere messaging bus. In our samples, we chose to use the Bus SCA.SYSTEM. Using the Administrative console, you must define the following queues:

TraderReceive	This queue is used by the mediation to receive the request message.
TraderSend	This queue is used by the mediation to send the response message. If the client, which invokes the mediation, provides a ReplyToQ in its request, the mediation uses that queue instead.
TraderCallback	This queue is used by the mediation to forward asynchronous messages to the client. It must be defined although it must not be used in this example.

The following JNDI resources must also be specified from the Administrative console by selecting **Resources** → **JMS Providers** → **Default messaging**:

jms/TraderQCF	This is the JMS queue connection factory. It must point to the Bus containing the definitions.
jms/TraderReceive	This is the JNDI name for the queue TraderReceive.
jms/TraderSend	This is the JNDI name for the queue TraderSend.
jms/TraderCallback	This is the JNDI name for the queue TraderCallback.
eis/TraderAS	This is the activation specification responsible for starting the mediation. It must be linked to the Destination JNDI name jms/TraderReceive, so that the mediation is automatically started as soon as a message arrives in the queue, TraderReceive.

Tip: If you plan for your messaging engine to be eventually accessed by remote client applications, which means an application living outside of the application server, then you must also fill in the extra QCF property, “Provider endpoints”. The default behavior for a QCF is to point to a bootstrap server that is local to the client application issuing the lookup operation. If the bootstrap server must be remote, the QCF property, “Provider endpoints” must be configured. This property must look as follows:

```
hostaddress:7276:BootstrapBasicMessaging
```

Where:

- `hostaddress` is the address where the bootstrap server resides.
- `7276` must be changed to the value of the port SIB_ENDPOINT_ADDRESS of the server hosting the bootstrap server.
- `BootstrapBasicMessaging` is the name of the protocol chain.

In order to invoke this mediation, you can develop a simple Java client, as described in “Creating a Java client to invoke the TraderJMStoSoap mediation” on page 257.

Testing the mediation

In order to run this mediation out of the WebSphere Integration Developer Integrated Test Environment, you can create an EAR file containing the involved J2EE elements. Select **File** → **Export** → **EAR file** and choose the Project **TraderJmsToSoapApp**. Also, select a valid destination, for example, c:\temp\TraderJmsToSoapApp.ear.

Deploy the EAR file to a running WebSphere ESB server.

4.3.3 Developing clients to invoke the mediations

This section describes the phases used to create and use the Web client.

Creating a Web client to invoke the TraderSoapToSoap mediation

This section briefly describes how to create a Web client to invoke the SoapToSoap mediation. The client is automatically built starting from the WSDL file, which describes the SOAP mediation client interface. The Export1_ClientInterfaceHttp_Service.wsdl file in the TraderSoapToSoap project was created when you generated the Web Service Binding for the Export node.

The ClientInterface.wsdl and the ClientMessage.xsd in the TraderClientLibrary project were created when you manually defined the client interface and the Business Object used by this interface.

In order to successfully create a Web service client application, the three files mentioned previously must exist in the same project. Therefore, copy the Export1_ClientInterfaceHttp_Service.wsdl file and paste it into the TraderClientLibrary project.

Perform the following steps:

1. Switch to the J2EE perspective.
2. In order to generate the Web services client, you must enable some additional WebSphere Integration Developer capabilities. Select **Window** → **Preferences** from the Menu Toolbar. Click **Workbench** → **Capabilities** and select **Web service developer** completely.
3. Right-click the **Export1_ClientInterfaceHttp_Service.wsdl** file. Then select **Web Services** → **Generate Client**. Select the **Test the Web service** option. This automatically creates a basic Web application able to invoke the Web service.
4. Click **Next** twice, which brings you to the Client Environment Configuration window. Ensure the Client Side Environment Selection fits your current test environment. If not, you can change it with the Edit button. Select Client Type: **Web** and enter a Client Project Name (for example, WebClientProject) and a EAR Project Name (for example, WebClientEAR).
5. In the Web Service Proxy Page window, click **Next**.
6. In the Web Service Client Test window, ensure all the methods are checked and click **Finish**.

Using a Web Client to invoke the mediation

The Web client can be invoked from the WebSphere Integration Developer J2EE perspective. Perform the following steps:

1. Open the **Dynamic Web Projects** folder and select **WebClient** → **WebContent** → **sampleClientInterfaceProxy**.
2. Right-click **TestClient.jsp** and select **Run** → **Run on Server**.
3. Ensure that the server you want to work with is selected and click **Finish**.
4. Use the getEndpoint method to check the current Web service endpoint at:

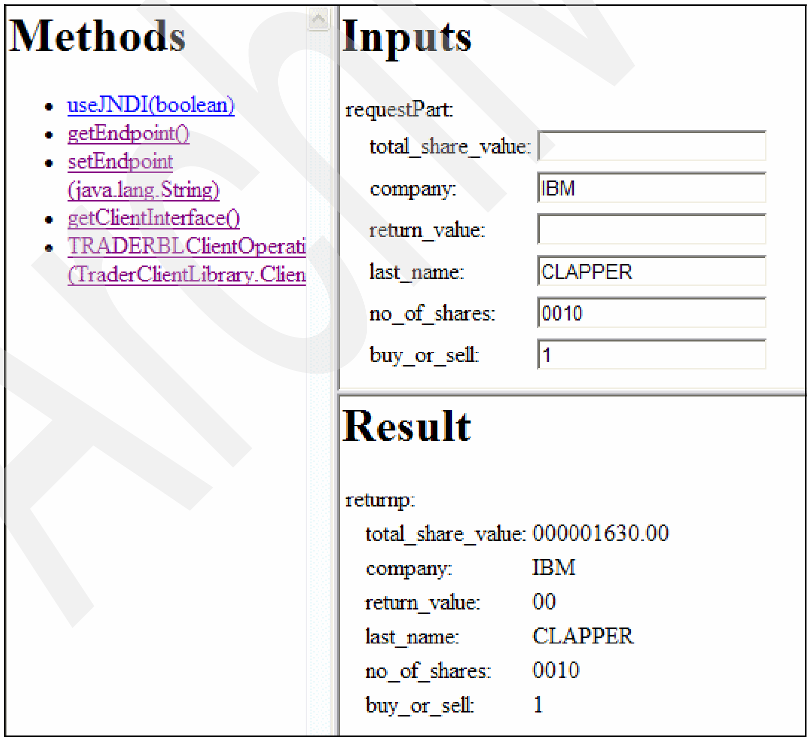
   ```
   http://localhost:9080/TraderSoapToSoapWeb/sca/Export
   ```

   You can use the setEndpoint method to change it, if the endpoint is not correct.

Note: In order to check the current values of your Web service endpoint, perform the following steps:

1. Under the Enterprise Application folder, open the **Deployment Descriptor** of the TraderSoapToSoapApp folder.
2. Switch to the Module tag and click the **TraderSoapToSoapWeb.war** module. Take a note of the Context root in the top right corner of the page. This is the first part of the endpoint address (for example, TraderSoapToSoapWeb).
3. Under the Dynamic Web Projects folder, open the **Deployment Descriptor** of the TraderSoapToSoapWeb folder.
4. Switch to the Source tag. Look for the <url-pattern> in the <servlet-mapping>. Make a note of this value (for example, sca/Export). This is the second part of the endpoint address.

5. Select the TRADERBLClientOperation (TraderClientLibrary.ClientMessage) operation and fill in the required fields. Click the **Invoke** button to invoke the mediation flow. See the example displayed in Figure 4-28.


Methods
useJNDI(boolean)
getEndpoint()
setEndpoint (java.lang.String)
getClientInterface()
TRADERBLClientOperati (TraderClientLibrary.Clien
Inputs
Result

requestPart:

total_share_value:	
company:	IBM
return_value:	
last_name:	CLAPPER
no_of_shares:	0010
buy_or_sell:	1

returnp:

total_share_value:	000001630.00
company:	IBM
return_value:	00
last_name:	CLAPPER
no_of_shares:	0010
buy_or_sell:	1

Figure 4-28 Web client page

Creating a Java client to invoke the TraderJMStoSoap mediation

We now provide some hints about the creation of a Java client that you can use to invoke the mediation. The client must be able to invoke JMS services and is responsible for creating a valid XML representation of the business object that is sent to the mediation request queue TraderReceive.

You must first understand how a valid XML message must look in order to represent the Business Object. Refer to Example 4-13.

Example 4-13 Valid XML message

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<ns1:ClientMessage xmlns:ns1=\"http://TraderClientLibrary\">
   <buy_or_sell>1</buy_or_sell>
   <return_value></return_value>
   <last_name>Giusta</last_name>
   <company>HWE</company>
   <no_of_shares>0020</no_of_shares>
   <total_share_value></total_share_value>
</ns1:ClientMessage>
```

This XML message must be consistent with the corresponding business object ClientMessage definition in the TraderClientLibrary project. The XML root element, ClientMessage, corresponds to the business object name. The namespace `http://TraderClientLibrary` reported in the XML document is the same value as the namespace defined within the business object. Finally, all the XML child elements correspond to the business object attributes.

The Java client program is also responsible for providing the name of the interface operation that we want to invoke. This name is TRADERBLClientOperation and can be inspected in the ClientInterface definition within the TraderClientLibrary. The Java client program must use the JMS user property, TargetFunctionName, to set the operation name.

Finally, the Java client program can use queue JNDI definitions to invoke the mediation and receive replies from it. For this reason, it must issue lookup calls to the URL pointing to the JNDI server that contains those definitions. In this case, we are using WebSphere ESB as a JNDI server, so the URL must refer to the address of the WebSphere Application Server that hosts WebSphere ESB.

Now you can create the Java client program that invokes the mediation. This program runs in a WebSphere Application Server client container, in order to have access to the WebSphere Application Server JNDI nameserver and issue JNDA lookup calls against it. You can now create the client project to host it. Perform the following steps:

1. Switch to the J2EE perspective. To create a new J2EE client project, right-click **Application Client Projects** and select **New** → **Application Client Project**. Call the project TraderClientJMSProject and click **Finish**.
2. Right-click **TraderClientJMSProject** and from the context menu choose **New** → **Class**. In the package name, enter com.ibm.client.jms. In the class Name, enter ServiceJMSClient. Click **Finish**.
3. The Java editor opens. Remove all the auto generated code and replace it with the code provided in the file ServiceJMSClient.java. You can find instructions to download the ServiceJMSClient.java file in Appendix B, “Additional material” on page 327.
4. In the deployment descriptor of the client application, you must point to its main class. In the J2EE perspective, select **Application Client Projects** → **TraderClientJMSProject** and double-click the **TraderClientJMSproject** deployment descriptor.
5. In the Overview window of the deployment descriptor, click **Edit** next to Main-Class.
6. In the Dependencies window, click **Browse** next to the Main-Class.
7. In the Type selection, begin typing ServiceJMSClient. After it is displayed, select it. Click **OK** and save the deployment descriptor. The client application is now able to invoke the correct main class.

Before launching the Java client project, you must check the port number of the JNDI server. In the Administrative console, select **Communications** and expand the list of **Ports** of your system. Take a note of the BOOTSTRAP_ADDRESS. You must use this port in the JNDI URL reference. Adjust the Java client program code so that the URL address and port are those that you expect.

The code fragment is shown in Example 4-14.

Example 4-14 Java client program code fragment for JNDI lookup

```
//XML string representing the input business object
      String xmlMessage = new String
         ("<?xml version=\"1.0\" encoding=\"UTF-8\"?>"
         + "<ns1:ClientMessage 
xmlns:ns1=\"http://TraderClientLibrary\">"
              + "<buy_or_sell>1</buy_or_sell>"
              + "<return_value></return_value>"
              + "<last_name>Giusta</last_name>"
              + "<company>HWE</company>"
              + "<no_of_shares>0020</no_of_shares>"
              + "<total_share_value></total_share_value>"
         + "</ns1:ClientMessage>");

      //interface operation to call
      String operation = "TRADERBLClientOperation";

      // jms resources
      String jmsQCF = new String("jms/TraderQCF");// qcf name
      String jmsReceive = new String("jms/TraderReceive");// request 
queue name
      String jmsSend = new String("jms/TraderSend");// reply queue name

      // URL pointing to the JNDI server
      String providerURL = "iiop://localhost:2809";
```

The description of this code is as follows:

- Creates the XML message that is going to be delivered to the mediation.
- Defines the name of the operation within the ClientInterface that is to be invoked.
- Defines the JMS Queue Connection Factory name that is used to invoke JMS services over the proper WebSphere messaging Bus.
- Defines the JMS queue names of the receive and send mediation queues.
- Defines the URL to point to the JNDI server. This name is used in the lookup operation.

Look at the following code fragment:

```
outMessage.setStringProperty("TargetFunctionName", operation);// <-
interface method to call
```

The setStringProperty method sets the user property TargetFunctionName to the name of the operation you want to invoke. The mediation JMS export binding uses this property in the incoming JMS request to locate the right operation in the ClientInterface interface.

You still have one final consideration regarding the Java client program. This client works in a typical JMS request/response mode. The default behavior of the JMS mediation binding is to copy the message ID of the incoming JMS message to the correlation ID of the outgoing reply. This allows the client to look for its proper reply using the standard request/reply approach.

The fragment code shown in Example 4-15 shows how the client can get its reply through a JMS selector.

Example 4-15 Code fragment to get reply through JMS selector

```
TextMessage outMessage = session.createTextMessage();
      outMessage.setText(request);
      outMessage.setStringProperty("TargetFunctionName", operation);//
<- interface method to call
      outMessage.setJMSReplyTo(iQueue);
      System.out.println("send message created");
      sender.send(outMessage);
      String messageID = outMessage.getJMSMessageID();

      System.out.println("message has been queued with messageId " +
messageID);

      String selector = "JMSCorrelationID = '" + messageID + "'";
      receiver = session.createReceiver(iQueue, selector);


      Message inMessage = receiver.receive(30000);
```

It is quite simple to launch the Java client using the WebSphere Integration Developer. Perform the following steps:

1. Ensure that you are in the J2EE perspective. Right-click **ServiceJMSClient** and choose **Run** → **Run...**.
2. Within the Run wizard, you can create a launch profile for the client. Select **WebSphere V6.0 Application Client** as your configuration and click **New** to create a new run profile.

3. Name the profile ServiceJMSClient, select the **Enable application client to connect to a server**, and explicitly specify the JNDI provider URL pointing to the correct JNDI server, for example:

   ```
   ioop://localhost:2809/
   ```

4. Click **Run**. The client must trace its activity, invoke the mediation, and receive its reply.

4.4 WebSphere ESB/WebSphere Message Broker implementation and deployment

The WebSphere ESB/WebSphere Message Broker combined scenario extends the TraderSoapToSoap WebSphere ESB mediation and, in addition to invoking the target CICS application over SOAP, it also sends a JMS message to WebSphere Message Broker for further processing, as shown in Figure 4-29.

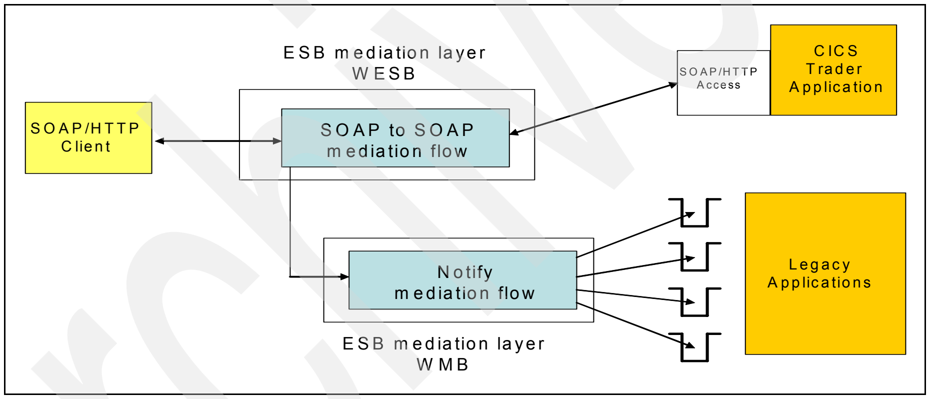


Figure 4-29 Mediation layer for WebSphere ESB/WebSphere Message Broker

4.4.1 WebSphere ESB mediation libraries

Three mediation libraries are used in the implementation of the sample scenario:

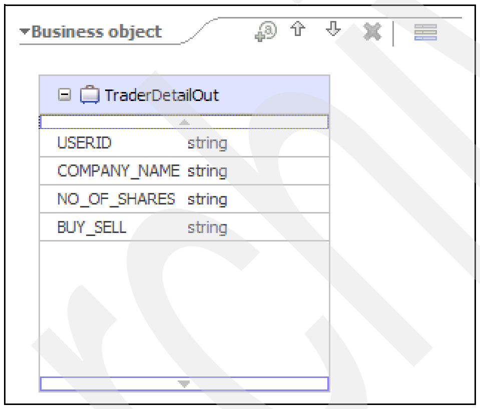
TraderClientLibrary	This contains the Business Object definition and interface definition to support the incoming client request.
TraderTargetLibrary	This contains the Business Object definition and interface definition to support the target CICS application invocation.
TraderBrokerLibary	This contains the Business Object definition and interface definition to support the WebSphere Message Broker invocation.

TraderClientLibrary and TraderTargetLibrary already exist, as they were created for the WebSphere ESB samples.

TraderBrokerLibary

Perform the following steps:

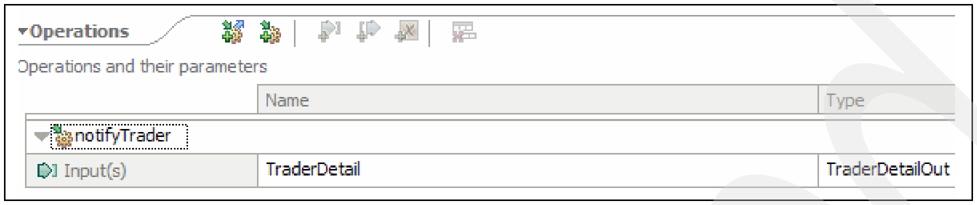
1. Start WebSphere Integration Developer and switch to the Business Integration perspective (this is usually the default perspective).
2. Select **File** → **New** → **Library** and create a new library named TraderBrokerLibrary.
3. Right-click **TraderBrokerLibary** and manually create a new Business Object. Select **New** → **Business Object**. Call it TraderDetailOut.
4. Add attributes to this object according to Figure 4-30. All the attributes are defined as strings.
5. You can find instructions to download the TraderDetailOut.xsd schema file in Appendix B, “Additional material” on page 327.



USERID	string
COMPANY_NAME	string
NO_OF_SHARES	string
BUY_SELL	string

Figure 4-30 TraderBrokerLibary: TraderDetailOut Business Object

6. Create a new Interface by right-clicking **TraderBrokerLibary** and selecting **New** → **Interface**. Name this interface TraderBrokerInterface. Create a one way operation named notifyTrader. Add an input message TraderDetail of type TraderDetailOut. The interface must look similar to Figure 4-31.
7. You can find instructions to download the TraderBrokerInterface.wsdl file in Appendix B, “Additional material” on page 327.



	Name	Type
notifyTrader		
Input(s)	TraderDetail	TraderDetailOut

Figure 4-31 TraderBrokerLibary: TraderBrokerInterface Interface

4.4.2 WebSphere ESB mediation module

This section discusses the various WebSphere ESB mediation modules we developed.

TraderSoapToSoap mediation module with WebSphere Message Broker invocation

Extend the TraderSoapToSoap Mediation Module to send a JMS message to WebSphere Message Broker for further processing. For each incoming SOAP/HTTP request, the TraderSoapToSoap mediation delivers a notification message to the WebSphere Message Broker and also continues to invoke the target CICS application. The WebSphere Message Broker message flow gets the notification message, transforms it according to a COBOL copybook structure, and sends it to the appropriate application.

You can use the TraderBrokerInterface interface previously created to invoke the WebSphere Message Broker message flow.

The mediation module for the extended TraderSoapToSoap sample performs the following tasks:

1. The mediation is exposed as a Web service through the Export node so that the client can invoke it.
2. The ClientToTarget XSL Transformation node, within the Request flow, converts the client SOAP request to the target SOAP message format.
3. The ClientToBroker XSL Transformation node, within the Request flow, converts the client SOAP request to the XML message format as expected by the WebSphere Message Broker message flow.

4. The Import node, ToCICS, invokes the target CICS application.
5. The Import node, ToBroker, sends an XML message to the WebSphere Message Broker message flow input queue.
6. The TargetToClient XSL Transformation node, within the Response flow, converts the target SOAP response to the client SOAP message format.

Note that there is no response expected from WebSphere Message Broker.

Creating the extended TraderSoapToSoap Mediation Module

Perform the following steps:

1. Start WebSphere Integration Developer and switch to the Business Integration perspective (this is usually the default perspective).
2. Right-click the **TraderSoapToSoap** Project folder and select **Open Dependency Editor**. Add the **TraderBrokerLibrary**.
3. Double-click the Assembly Diagram: **TraderSoapToSoap**.
4. Using the menu above the Mediation component, select **Add Reference**. Choose **TraderBrokerInterface**. This is the reference to the JMS service that sends the notification message to the Broker.
5. From the TraderBrokerLibrary, drag the **TraderBrokerInterface** interface and drop it into the canvas. Select **Import with no Binding** and click **OK**. Rename Import1 to ToBroker. This import node contains the JMS details to invoke the WebSphere Message Broker service.
6. Right-click the **ToBroker** node and select **Generate Binding → JMS Binding**. Although the system can automatically create all the JMS resources required, you must want a better control on what is defined, so select **Use pre-configured messaging provider resources**.

   Ensure that the JMS messaging domain field is set to **Point-to-Point** and that the Serialization Type is chosen as **Business Object XML using JMS TextMessage**, which means that you can send requests in the form of text XML messages.

   Enter the JNDI values shown in Table 4-15 to point to the required messaging resources.

Table 4-15 Properties values

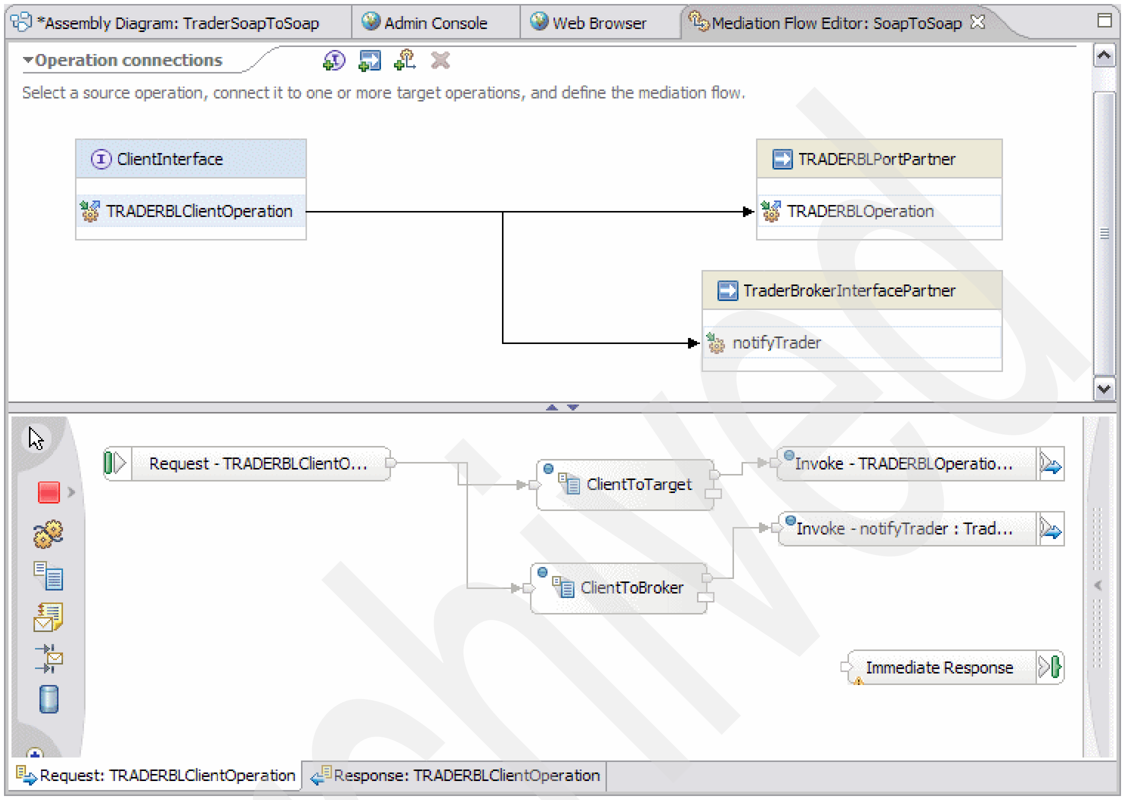
Property name	Property content
JNDI name for Connection Factory:*	jms/TraderQCF
JNDI name for Send Destination	jms/TraderNotify

7. Connect the ToBroker node to the SoapToSoap mediation component.
8. Double-click the SoapToSoap mediation component to open the mediation flow editor.

Perform the following steps to build the add connection to the new target endpoint:

1. Click the **TRADERBLClientOperation** in the ClientInterface node and ensure that the **Request flow** tag is selected.
2. You have to add a reference to the new target service. Click the **Add Reference** icon (represented as a white arrow in a blue box at the top of the Mediation Flow Editor window). Select **TargetBrokerInterface** in the Matching interfaces list and click **OK**.
3. Connect the TRADERBLClientOperation within the ClientInterface interface to the notifyTrader operation within the TraderBrokerInterfacePartner reference.

4. Ensure that the Request: **TRADERBLClientOperation** tag is selected. You can extend the request flow, as shown in Figure 4-32.



*Assembly Diagram: TraderSoapToSoap
Admin Console
Web Browser
Mediation Flow Editor: SoapToSoap
Operation connections
Select a source operation, connect it to one or more target operations, and define the mediation flow.
ClientInterface
TRADERBLClientOperation
TRADERBLPortPartner
TRADERBLOperation
TraderBrokerInterfacePartner
notifyTrader
Request - TRADERBLClientO...
ClientToTarget
Invoke - TRADERBLOperatio...
Invoke - notifyTrader : Trad...
ClientToBroker
Immediate Response
Request: TRADERBLClientOperation
Response: TRADERBLClientOperation

Figure 4-32 Request: TRADERBLClientOperation mediation flow

5. From the node palette, select the **XSL Transformation** node and drop it into the canvas. Rename the node to ClientToBroker. Connect the Request node to the new Transformation node and the Transformation node to the Invoke-notifyTrader node.
6. Click the **ClientToBroker** node and select the **Details** section of the Properties tag. Click **New** to create a new map and accept the defaults.
7. Expand the **Source** and **Target** details.
8. Drag and drop the following fields from the Source to the Target:

buy_or_sell	BUY_SELL
last_name	USERID
company	COMPANY_NAME
no_of_shares	NO_OF_SHARES

9. After saving the mapping, go back to the Mediation Flow Editor and select **Regenerate XSL**.
10. The Request flow is now complete.

4.4.3 The extended TraderSoapToSoap WebSphere Message Broker message sets

This section discusses the WebSphere Message Broker message sets.

TraderNotifyXML message set

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderNotifyXML_Project and within this, a new Message Set called MS_TraderNotifyXML. Select the **XML Wire Format Name**. This sets the field to XML1.
3. Copy the TraderDetailOut.xsd schema file from the TraderBrokerLibrary project into the MS_TraderNotifyXML_Project project.
4. Right-click the **TraderDetailOut.xsd** schema file and create a new Message Definition File within the MS_TraderNotifyXML message set. As the .xsd file does not contain a root element, this procedure only creates the type definition. You can create the message definition in the next step.
5. Expand the **MS_TraderNotifyXML_Project** folder down to the Messages folder and double-click it. Right-click **Messages** in the spreadsheet panel and select **Add Message**. Name the message TraderDetailOut and select its type as **TraderDetailOut**. This message definition file is the WebSphere Message Broker representation of the input XML message delivered by the mediation.

TraderNotifyCobol message set

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderNotifyCobol_Project and within this, a new Message Set called MS_TraderNotifyCobol. Select the **Custom Wire Format Name**. This sets the field to CWF1.
3. Import the COBOL copy book structure, TraderNotify.cpy, into the MS_TraderNotifyCobol_Project project.

   You can find instructions to download the TraderNotify.cpy file in Appendix B, “Additional material” on page 327.

4. Right-click the **TraderNotify.cpy** file and create a new Message Definition File within the MS_TraderNotifyCobol message set.

   Check for the creation of the new message definition file based on the root element, NOTIFY-DATA.

   This message file is the WebSphere Message Broker representation of the COBOL message delivered to the applications.

4.4.4 WebSphere Message Broker TraderNotify message flow

The TraderNotify sample flow receives the XML message from the WebSphere ESB mediation, transforms the XML structure to the COBOL format as expected by the applications, and delivers the message to a specified queue.

The queue name is based on the input message content.

The message flow for the TraderNotify sample performs the following tasks:

1. The TraderNotify MQInput node reads a message from the input queue.
2. The XMLtoCobol Mapping node converts the XML input message to a COBOL-like message to fit the format expected by the applications.

   Furthermore, this mapping node fills in a label destination based on the field COMPANY_NAME, which forms part of the incoming message.
3. The RouteToLabel node jumps to one of four different label nodes as determined by the previous mapping node.
4. The MQOutput node, downstream from the selected label node delivers the message to back-end application.

The TraderNotify sample message flow is shown in Figure 4-33.

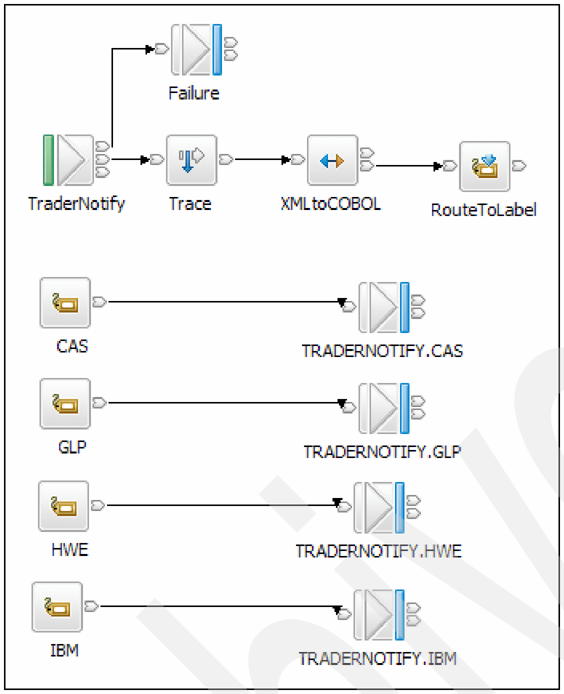


Figure 4-33 TraderNotify message flow

Node name: TraderNotify

See Table 4-16 for a description of the property values changed from the defaults.

Table 4-16 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADERNOTIFY	Defines the queue from which the input node is reading.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderNotifyXML	The message set describing the incoming message.
	Message Type: TraderDetailOut	The message type describing the incoming message.
	Message Format: XML1	The physical format of the incoming message.

Node name: XMLtoCobol

Table 4-17 details the property changes for this node.

Table 4-17 Properties values

Property	Property name: value	Comment
Basic	Compute Mode: LocalEnvironment and Message	Because you are making a change to the LocalEnvironment, you have to propagate both the LocalEnvironment and the Message.

The map is responsible for mapping the property values so that they match the values expected by the target application. The following three property values have to be set manually:

- MessageSet

  The unique message set identifier of the target message set is MS_TraderNotifyCobol.

- MessageType

  The message type name of the target message is msg_NOTIFYDATA.

- MessageFormat

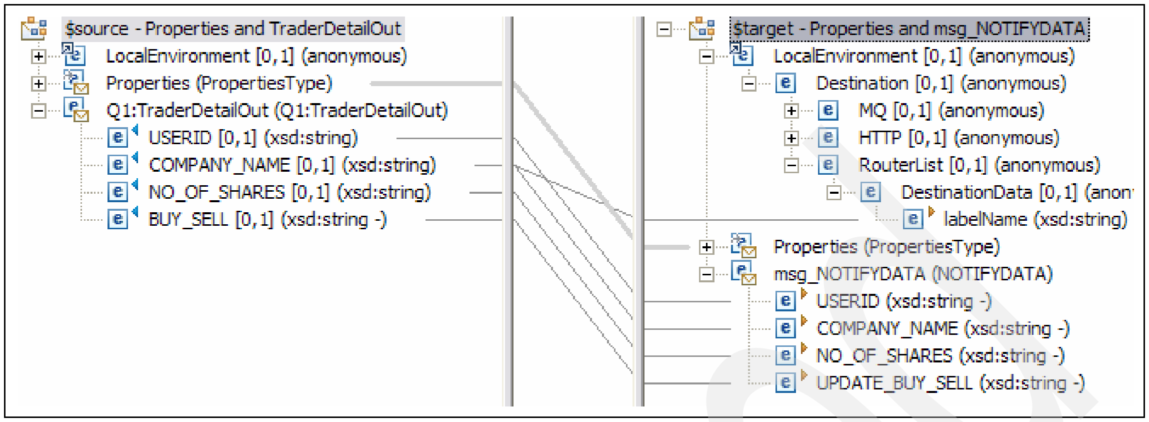
  The physical format of the target message is CWF1.

The remaining fields in the Properties tree can be copied unchanged.

The map is also responsible for filling the field, **LocalEnvironment** → **Destination** → **RouterList** → **DestinationData** → **labelName**, with the value stored in COMPANY_NAME.

The flow subsequently jumps to the label node that contains this value. In order to map the COMPANY_NAME field to the labelName field, you must populate the LocalEnvironment tree.

Figure 4-34 shows you how to map the message payload.



$source - Properties and TraderDetailOut
LocalEnvironment [0,1] (anonymous)
Properties (PropertiesType)
Q1:TraderDetailOut (Q1:TraderDetailOut)
USERID [0,1] (xsd:string)
COMPANY_NAME [0,1] (xsd:string)
NO_OF_SHARES [0,1] (xsd:string)
BUY_SELL [0,1] (xsd:string -)
$target - Properties and msg_NOTIFYDATA
LocalEnvironment [0,1] (anonymous)
Destination [0,1] (anonymous)
MQ [0,1] (anonymous)
HTTP [0,1] (anonymous)
RouterList [0,1] (anonymous)
DestinationData [0,1] (anon
labelName (xsd:string)
Properties (PropertiesType)
msg_NOTIFYDATA (NOTIFYDATA)
USERID (xsd:string -)
COMPANY_NAME (xsd:string -)
NO_OF_SHARES (xsd:string -)
UPDATE_BUY_SELL (xsd:string -)

Figure 4-34 XMLtoCobol message map

Node name: RouteToLabel

This node jumps to one of four different label nodes as determined by the previous mapping node.

Node name: CAS

See Table 4-18 for a description of the property values changed from the defaults.

Table 4-18 Properties values

Property	Property name: value	Comment
Basic	Label Name: CAS	Defines the label related to this node.

Node name: TRADERNOTIFY.CAS

This node inputs the message to the queue targeted to the application. See Table 4-19 for a description of the property values changed from the defaults.

Table 4-19 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADERNOTIFY.CAS	Defines the output queue.

Node name: GLP

See Table 4-20 for a description of the property values changed from the defaults.

Table 4-20 Properties values

Property	Property name: value	Comment
Basic	Label Name: GLP	Defines the label related to this node.

Node name: TRADERNOTIFY.GLP

This node puts the message to the queue targeted to the application. See Table 4-21 for a description of the property values changed from the defaults.

Table 4-21 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADERNOTIFY.GLP	Defines the output queue.

Node name: HWE

See Table 4-22 for a description of the property values changed from the defaults.

Table 4-22 Properties values

Property	Property name: value	Comment
Basic	Label Name: HWE	Defines the label related to this node.

Node name: TRADERNOTIFY.HWE

This node puts the message to the queue targeted to the application. See Table 4-23 for a description of the property values changed from the defaults.

Table 4-23 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADERNOTIFY.HWE	Defines the output queue.

Node name: IBM

See Table 4-24 for a description of the property values changed from the defaults.

Table 4-24 Properties values

Property	Property name: value	Comment
Basic	Label Name: IBM	Defines the label related to this node.

Node name: TRADERNOTIFY.IBM

This node puts the message to the queue targeted to the application. See Table 4-25 for a description of the property values changed from the defaults.

Table 4-25 Properties values

Property	Property name: value	Comment
Basic	Queue Name:TRADERNOTIFY.IBM	Defines the output queue.

4.4.5 How to connect WebSphere MQ and WebSphere messaging

The previous sample requires a connection between WebSphere MQ and the WebSphere messaging services.

This connection is supported by a definition in the WebSphere Messaging Engine known as an MQ Link and by standard WebSphere MQ sender and receiver channels. Furthermore, the target queue manager must be defined as a “foreign bus”. WebSphere messaging looks at the WebSphere MQ queue manager in a similar way to another bus.

WebSphere messaging supports a listener service listening over a WebSphere MQ communication protocol. This is known as SIB_MQ_ENDPOINT_ADDRESS.

There is also a corresponding SIB_MQ_ENDPOINT_SECURE_ADDRESS for SSL support. Inspect the port numbers in your environment to select the correct port. The default value for SIB_MQ_ENDPOINT_ADDRESS is 5558 and the default value for SIB_MQ_ENDPOINT_SECURE_ADDRESS is 5578.

Define a foreign bus

Perform the following steps:

1. From the Administrative console, select **Service integration** → **Buses** and select the bus that supports your messaging engine.
2. Click **Foreign buses**. Now define the target WebSphere MQ queue manager as a foreign bus.
3. Create a new foreign bus. Give it a meaningful name, for example, MQRG, if MQRG is your target queue manager.
4. Select the Routing type as **Direct, service integration bus link** and fill in valid Inbound and Outbound user IDs. Click **Finish**.

Define an MQLink

Perform the following steps:

1. From the Administrative console, select **Service integration** → **Buses** and select the bus that supports your Messaging Engine.
2. In the Topology session, click **Messaging engines**.
3. Select your Messaging Engine and then click **WebSphere MQ links**.
4. Create a new link. Give it a name, for example, MQLinkMQRG, if MQRG is the name of the WebSphere MQ queue manager with which you are going to interact.

   Select the foreign bus that corresponds to your target queue manager. In the property “Queue manager name”, enter the name that is going to be used by the current Messaging Engine when communicating with your target queue manager, for example, SIB. This is a virtual queue manager name. The target real queue manager thinks that it is receiving from a queue manager with this name.
5. In the Receiver channel window, enter the name and definition of the receiving MQ channel, for example, MQRG.TO.SIB. This channel has a corresponding sender channel in the WebSphere MQ definitions.
6. In the Sender channel window, enter the name and definition of the sender channel, for example, SIB.TO.MQRG. This channel has a corresponding receiver channel in WebSphere MQ definitions.

   Ensure that the Transport chain property is set to **OutboundBasicMQLink** (MQ communication protocol with no SSL support) or **OutboundSecureMQLink** (MQ communication protocol with SSL support).

Define JMS resources in the application server

Some JMS resources have to be defined in the WebSphere Application Server that runs the mediation. All the resources required can be defined in the WebSphere Messaging Bus that supports your mediation. Using the Administrative console, you must define the following queues within the appropriate bus:

TraderNotify	This must be defined as an Alias queue and must belong to the bus that also contains all the other definitions. In the target identifier field, enter the name of this queue as defined in the target WebSphere MQ queue manager, for example, TRADERNOTIFY. In the target bus field, enter the name of the foreign bus to which this queue refers, for example, MQRG.

The following JNDI resources must also be specified from the Administrative console by selecting **Resources** → **JMS Providers** → **Default messaging**:

jms/TraderQCF	This is the JMS queue connection factory. It must point to the bus where all the definitions have been defined.
jms/TraderNotify	This is the JNDI name for the queue alias TraderNotify.

Define WebSphere MQ resources

You must define sender and receiver channels with names matching those used in the WebSphere messaging definitions.

The WebSphere MQ sender channel must point to a listening port number as defined by the property SIB_MQ_ENDPOINT_ADDRESS.

You must also define the following queues:

TRADERNOTIFY	This is the queue from where the message flow reads.
TRADERNOTIFY.CAS	This is the output queue for the CAS back-end application.
TRADERNOTIFY.GLP	This is the output queue for the GLP back-end application.
TRADERNOTIFY.HWE	This is the output queue for the HWE back-end application.
TRADERNOTIFY.IBM	This is the output queue for the IBM back-end application.

In order to invoke this sample, you can reuse the simple Web client that is described in “Using a Web Client to invoke the mediation” on page 255.

Testing the combined message flow and mediation sample

In order to run the mediation, which is part of this sample, out of the WebSphere Integration Developer Integrated Test Environment, create an EAR file containing the involved J2EE elements.

Select **File** → **Export** → **EAR file** and choose the Project **TraderSoapToSoapApp**. Also select a valid destination, for example, c:\temp\TraderSoapToSoapAppExt.ear.

Deploy the EAR file to a running WebSphere ESB server.

In order to run the message flow, which is part of this sample, you must create a BAR file containing the following elements:

- MF_TraderNotify.msgflow
- MS_TraderNotifyCobolt
- MS_TraderNotifyXML

Deploy this file to the runtime broker.

The following MQ queues are also required, as mentioned in "Define WebSphere MQ resources" on page 275:

- TRADERNOTIFY
- TRADERNOTIFY.CAS
- TRADERNOTIFY.GLP
- TRADERNOTIFY.HWE
- TRADERNOTIFY.IBM

5

z/OS specific WebSphere Message Broker nodes

This chapter provides a general overview of WebSphere Message Broker functions that are exclusive to z/OS.

WebSphere Message Broker for z/OS offers extra nodes that are not available in distributed platforms and supports z/OS specific environments, such as:

- Access to QSAM files
- Access to VSAM files
- CICS programs invocation

These functions are available as category 3 SupportPacs and you can download them from the following site:

`http://www.ibm.com/support/docview.wss?rs=977&uid=swg27007205`

Category 3 SupportPacs are provided as product extensions and are fully supported.

Detailed information regarding these functions is given in the SupportPacs themselves. Although it is not the intent of this book to replicate that information, we are going to summarize some of the functions they provide. We are also going to describe how to modify the sample flows described in Chapter 4, “Sample scenarios” on page 195, in order to make use of some of the nodes.

5.1 (IA11): File Adapter for z/OS sequential files

File Adapter for z/OS sequential files allows the WebSphere Message Broker to interact with sequential (QSAM) files (simple data sets or PDS). This can, therefore, provide an additional way for a batch application to pass data to a WebSphere Message Broker flow. Although this interaction can be supported over MQ to MQ communication, using files can be simpler and more appropriate in certain circumstances. For example, a batch program could create a file containing invoice details. When the file is closed, a WebSphere Message Broker message flow can be started to access the file and perform a complex routing and transformation logic.

File Adapter nodes are available in SupportPac IA11 and you can download them from the following site:

http://www.ibm.com/support/docview.wss?rs=977&uid=swg27007205

The file adapter only supports QSAM files.

There are four different nodes in the file adapter:

- FileRead node
- FileWrite node
- FileRename node
- FileDelete node

We now provide a brief description of these node capabilities.

5.1.1 FileRead node

The FileRead node is also called the FileRead plug-in.

The FileRead node reads from a sequential file (QSAM data set in z/OS) and propagates each record as a message through the “out” terminal.

This node does not start reading the file immediately after the message flow is deployed to the broker. Instead, it must be activated by an *action message* that has to be sent to the node action terminal, requesting the file to be opened.

When the end of the file is reached, it is automatically closed. The node has a Status terminal where it sends *status messages* notifying different events. For example, a “close input file” status message is produced when the file being read has reached the end of file condition.

Action and status messages give full control over the node behavior.

A typical batch scenario where you can use the FileRead node is summarized as follows:

- A batch program creates a file. When the file is closed, it sends an action message to kick off the FileRead node in a WebSphere Message Broker message flow.
- The message flow begins reading the records in the file and each of them is transformed and routed to target applications.
- When the file has been completely read by the message flow, a status message can start a second message flow or trigger an MQ batch program.

5.1.2 FileWrite node

The FileWrite node is also called the FileWrite plug-in.

The FileWrite node usually appears in the middle of a message flow. It receives an input message and writes it to a sequential file (QSAM data set in z/OS). It also creates *status messages* to notify open and close file events.

The criteria the node uses to open and close files are based on two different options:

- Automatic mode

  In this case, the file is opened when the first message arrives. It remains open for a configurable amount of time, or until a certain number of records have been written, or until a specified time of the day is reached. Each time a file is opened, a new file name is created. The *Filename* property of the node specifies the name prefix and the node automatically appends a suffix to make the name unique.
- Action message control mode

  With this option, you must send open and close action messages to the node in order to control the file open/close life cycle.

You can use the FileRead and FileWrite nodes in conjunction with the new WebSphere Message Broker V6 nodes, TimeoutControl, and TimeoutNotification. Timeout nodes are designed to produce events at particular times. You can configure these Timeout nodes to create the action message, which starts a message flow that reads records via a FileRead node at a specific time of the day. You can also use them to generate the open and close file events of a FileWrite node configured to run in message control node.

5.1.3 FileRename node

The FileRename node is also called the FileRename plug-in.

It is entirely controlled by the action message, which has the responsibility to provide both old and new file names. It produces status messages to indicate the result of the delete operation.

5.1.4 FileDelete node

The FileDelete node is also called the FileDelete plug-in.

It is entirely controlled by the action message, which defines which file has to be deleted. It produces status messages to indicate the result of the delete operation.

5.2 (IA12): CICSRequest node

CICSRequest node supports synchronous CICS programs invocation from WebSphere Message Broker message flows. The invoked CICS program can be any program that is callable through a Distributed Program Link (DPL) request. CICSRequest nodes use the CICS External Call Interface (EXCI) to invoke the CICS program.

CICSRequest node support is available as SupportPac IA12 and you can download it from the following site:

http://www.ibm.com/support/docview.wss?rs=977&uid=swg27007205

Using the CICSRequest node is straight forward. The message flow must build a valid message containing the CICS COMMAREA and then send that message to the CICSRequest input terminal.

CICSRequest properties allow you to configure the target CICS APPLID, the program name that is to be invoked, some security options, and where to look for the COMMAREA content within the incoming message tree.

Furthermore, you can also specify whether or not the EXCI request is to be part of a unit of work. This can be important in z/OS environments. A message flow can invoke CICS update requests with transactional integrity. A message flow can contain one or more CICSRequest nodes. All the CICS programs, which are invoked from the same message flow, can be part of a single unit of work.

The following are the main properties that have to be configured:

- CICS Network Applid

  The target CICS application ID.

- Program name

  The CICS program to invoke.

- CICS UserId

  The user ID that the CICS transaction runs with, dependent on CICS security settings.

- Message location of request COMMAREA

  Where to look within the message tree for the COMMAREA that is to be sent to CICS. It can be any valid ESQL reference, for example, InputRoot.MRM.MyRequestCommarea.

- Message location of returned COMMAREA

  Where to look within the message tree for the COMMAREA that is going to be returned from CICS. It can be any valid ESQL reference, for example, OutputRoot.MRM.MyResponseCommarea.

- Include CICS request in message flow transaction

  It can be either *Automatic* if the EXCI is part of the current unit of work or *Commit* if CICS commits the unit of work immediately after sending the response to WebSphere Message Broker.

There are a couple of caveats when using the CICSRequest node:

- The node is responsible for sending a valid COMMAREA to CICS. Although the CICS program could potentially be developed to accept a generic format, in most cases, a CICS COMMAREA has to be built according to a COBOL copybook. This can be easily managed by the broker using MRM and the Custom Wire Format (CWF) physical layer.
  - One method of achieving this is to define a message in a Message Set project that describes the COMMAREA structure. You must also consider using default values for fields that could be empty. When the CICSRequest node is configured, you can point the “Message location of request COMMAREA” property to the full message containing the COMMAREA. WebSphere Message Broker uses the message definition file information to marshall all the fields and build a valid COMMAREA.
  - You might also choose to configure the CICSRequest properties so that the COMMAREA you are going to send to CICS is built from any valid ESQL reference. This is different from a full message definition, for example, InputRoot.MRM.MyData.MyDataForCics.

In this case, WebSphere Message Broker does not invoke the MRM parser and the data sent to CICS is only based on the content of the current WebSphere Message Broker message tree. Default values are not applied to non-existing fields, and padding characters are not used. This can potentially cause a problem in the target CICS program if it receives a COMMAREA with a structure different from the one that is expected.

  – If you use an MRM message to define the CICS COMMAREA, you usually do not encounter this problem.

- The second recommendation has to do with some level of control in the CICS EXCI invocation:
  – A *specific* EXCI connection has to be defined to CICS to support the connection of the WebSphere Message Broker CICSRequest node with the NETNAME set to the name by which the WebSphere Message Broker flow identifies itself to CICS.
  – By default, this name is the user ID that runs the WebSphere Message Broker execution group, in our case, MQR1BRK for the first broker, and MQR2BRK for the second.

    This means that for our configuration, we have to define two CONNECTION PROTOCOL(EXCI) objects with the appropriate NETNAMEs to CICS to support the use of the CICSRequest node from both broker environments.
  – This name (unfortunately denoted as *USER_NAME* within EXCI interface documentation) can be specified explicitly by use of the environment variable named NODE_CICSREQUEST_IRC_USER_NAME.

    You set this in the broker's ENVFILE.

    When set to a common value for multiple broker instances, then only one EXCI Connection resource is required in CICS, which has that value coded on its NETNAME parameter.

> **Note:** Information related to EXCI CONNECTION resource objects required in CICS and the WebSphere Message Broker environment variable that determines the Connection's NETNAME setting are contained in the readme.txt and readme.zos file of the IA12 SupportPac package, *not* in the IA12.pdf main documentation file.
>
> The CICS and VSAM nodes can only run in a logical partition where VSAM and CICS are running because there is remote support for these infrastructures.

5.3 (IA13): VSAM nodes

There are five VSAM nodes that you can use for input, or to read, write, update, and delete records in a VSAM file. The VSAM nodes are available in Support Pac IA13 and you can download them from the following site:

http://www.ibm.com/support/docview.wss?rs=977&uid=swg27007205

KSDS, ESDS, and RRDS VSAM files are supported.

Two different VSAM FileRead scenarios are supported:

- A batch of VSAM records are read and each record is propagated to the message flow to be processed one at a time. Each VSAM record is managed as a standard input message to the flow.
- A single VSAM record can be read in the middle of a message flow and its content can be managed by subsequent nodes. The current message content can be augmented with data read from the VSAM record.

Note: The CICS and VSAM nodes can only run in a logical partition where VSAM and CICS are running because there is remote support for these infrastructures.

5.3.1 VSAMInput node

You can use this node for reading a certain number of VSAM records from a file (a batch of records). The number of records read can be controlled in several ways. You can read the entire file or you can specify a range of keys (KSDS), a range of RBA (ESDS), or a range of RRN (RRDS) values.

A VSAMInput node can appear anywhere in a flow where file reading is necessary. It is not the first node in a message flow, because it requires an incoming VSAM Request message, received by the Control terminal. The VSAM Request message starts the VSAMInput node read process. For this reason, this node is usually preceded by other nodes, (at least by an MQInput node), which creates the VSAM Request message. Beside starting the VSAMInput node activity, the VSAM Request message can also override the node properties, for example, the file name or the read criteria.

The VSAMInput node has parsing capabilities. You must provide the Message Set, Message Type, and Message Format describing the structure of the record read from the VSAM file.

This node can append a Report Message as part of the message propagated through the out terminal. This message reports information regarding the read operation just concluded, for example, the current key.

It also creates a Status Message that is propagated through the VSAM batch when the entire read operation is over. This message details overall information, such as the number of VSAM records read.

5.3.2 VSAMRead node

You can use this node for reading a VSAM record from a file and forwarding its contents to the downstream nodes in the message flow.

Unlike a VSAMInput node, a VSAMRead node usually appears in the middle of a running message flow, which has been started by some event, such as an MQ message read by an MQInput node or a VSAM record read by a VSAMInput node. Before entering this node, the message flow is responsible for creating the Request message somewhere in the message tree. The Request messages dictate which record in the VSAM file must be read. You can use the VSAMRead node property Request Location to point to the ESQL section of the incoming message that contains the Request message. The Request message usually details which key or RBA or RRN to use for the VSAM read request.

You can use the node property Result to enter the ESQL reference that decides where, in the outgoing message, to store data read from the VSAM record. In this way, the message that was entered in the node and data coming from VSAM can coexist in the outgoing message.

The VSAMRead node has parsing capabilities. You must provide the Message Set, Message Type, and Message Format describing the record read from the VSAM file.

A report message can also be included in the outgoing message to detail the result of the read operation.

5.3.3 VSAMWrite node

You can use this node for writing a VSAM record to a file.

This node is driven by the node properties, but, as with the other nodes, the properties can be overwritten by the incoming Request message.

You can use the Request property of this node to specify the ESQL reference pointing to the Request message, and also the ESQL reference of the data that has to be written to the VSAM file.

The VSAM Write node has parsing capabilities. You must provide the Message Set, Message Type, and Message Format describing the record that has to be written to the VSAM file.

A Report message can be included in the outgoing message to detail the write VSAM results.

5.3.4 VSAMUpdate node

You can use this node for updating a VSAM record to a file.

Updates can happen either for a key, for an RBA, or for an RRN.

You can use the Request property of this node to specify where to look for the new updated data in the incoming message tree. This data is used to replace the VSAM record.

A report message can be optionally part of the outgoing message to describe the VSAM update results.

The VSAMUpdate node has parsing capabilities. You must provide the Message Set, Message Type, and Message Format describing the record that has to be updated in the VSAM file.

5.3.5 VSAMDelete node

You can use this node to delete a record from a VSAM file.

You can use the Request property of this node to specify the ESQL reference pointing to the Request message in the incoming message tree. You can use the Request message to determine which key, RBA, or RRN in the VSAM file is to be deleted, and also override other properties in the node configuration.

A report message can optionally be part of the outgoing message to describe the VSAM delete results.

This node does not have parsing capabilities. It is worth mentioning that upon leaving the node, you no longer have the data of the record just deleted.

5.4 A working example

We have extended the MQ Client to the MQ Target message flow and its related message sets in order to incorporate some of the z/OS specific nodes.

5.4.1 Extended message sets

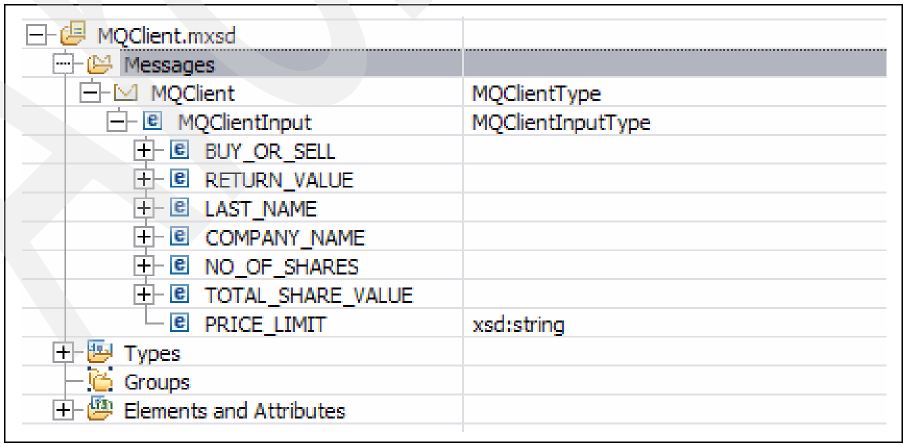
This section describes the various message sets we used.

TraderMQClient_Ext message set

We extended the TraderMQClient message set by adding an additional field to the actual message. The field, PRICE_LIMIT, is intended to provide the maximum price the “buyer” is prepared to accept as the share value. Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderMQClient_Ext_Project and within this, a new Message Set called MS_TraderMQClient_Ext. Base on the existing message set MS_TraderMQCient_Project/MS_TraderMQCient. In the next window, select the **XML Wire Format Name**. This sets the field to XML1.
3. Open the MQClient message by double-clicking it. In the message spreadsheet, right-click **MQClient** and select **Add Local Element**. Name this new field, PRICE_LIMIT and set its type to string.

The new message definition will look similar to Figure 5-1.



MQClient.mxsd	
Messages	
MQClient	MQClientType
MQClientInput	MQClientInputType
BUY_OR_SELL	
RETURN_VALUE	
LAST_NAME	
COMPANY_NAME	
NO_OF_SHARES	
TOTAL_SHARE_VALUE	
PRICE_LIMIT	xsd:string
Types	
Groups	
Elements and Attributes	

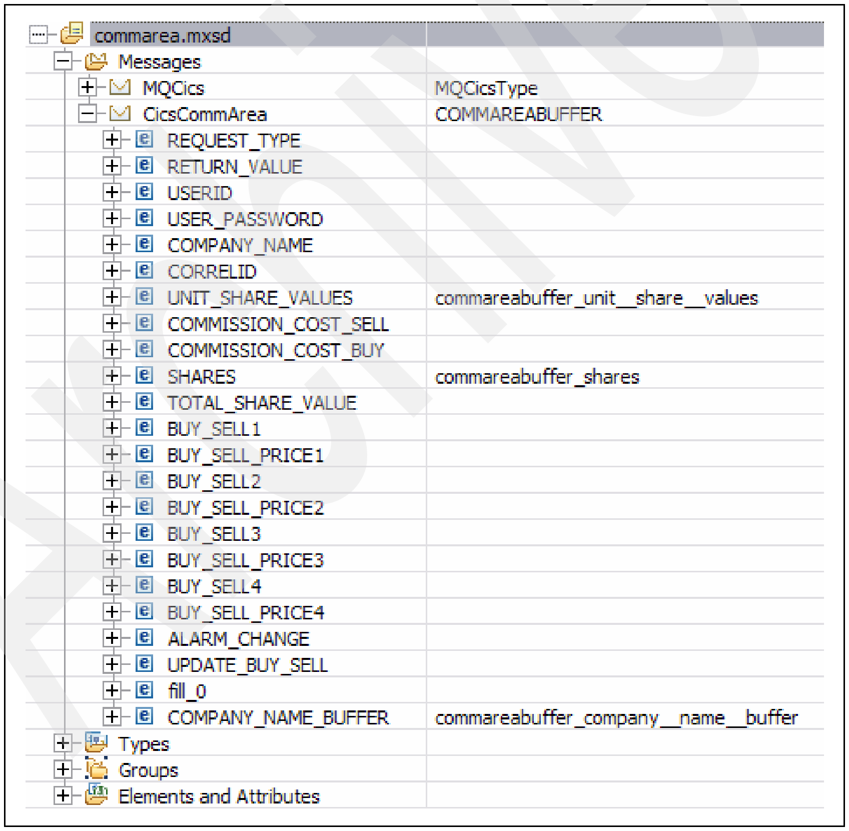
Figure 5-1 Sample message flow definition

TraderMQTarget message set

The message MQCics, currently defined within MS_TraderMQTarget_Project, is designed to interact with the MQ CICS Bridge. For this reason, this message contains both the CICS program name and the CICS COMMAREA. When using the CICSRequest node, the CICS program name is provided in the node properties. Furthermore, the node has to send a message containing only the COMMAREA to CICS. For this reason, create a new message called CicsCommArea that contains only the COMMAREA data:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Open the **MS_TraderMQTarget_Project** and start the Messages spreadsheet. Right-click **Messages** and select **Add Message**. Name the new message CicsCommArea and select its type as **COMMAREABUFFER**.

The new message definition must look similar to Figure 5-2.



commarea.mxsd	
Messages	
MQCics	MQCicsType
CicsCommArea	COMMAREABUFFER
REQUEST_TYPE	
RETURN_VALUE	
USERID	
USER_PASSWORD	
COMPANY_NAME	
CORRELID	
UNIT_SHARE_VALUES	commareabuffer_unit__share__values
COMMISSION_COST_SELL	
COMMISSION_COST_BUY	
SHARES	commareabuffer_shares
TOTAL_SHARE_VALUE	
BUY_SELL1	
BUY_SELL_PRICE1	
BUY_SELL2	
BUY_SELL_PRICE2	
BUY_SELL3	
BUY_SELL_PRICE3	
BUY_SELL4	
BUY_SELL_PRICE4	
ALARM_CHANGE	
UPDATE_BUY_SELL	
fill_0	
COMPANY_NAME_BUFFER	commareabuffer_company__name__buffer
Types	
Groups	
Elements and Attributes	

Figure 5-2 Sample message flow definition

TraderCompanyFile message set

We created a new message set to define the format of the company records read from the CICS VSAM file. The COBOL copybook, detailed in Example 5-1, describes the structure, and you can use it to create a new message definition file.

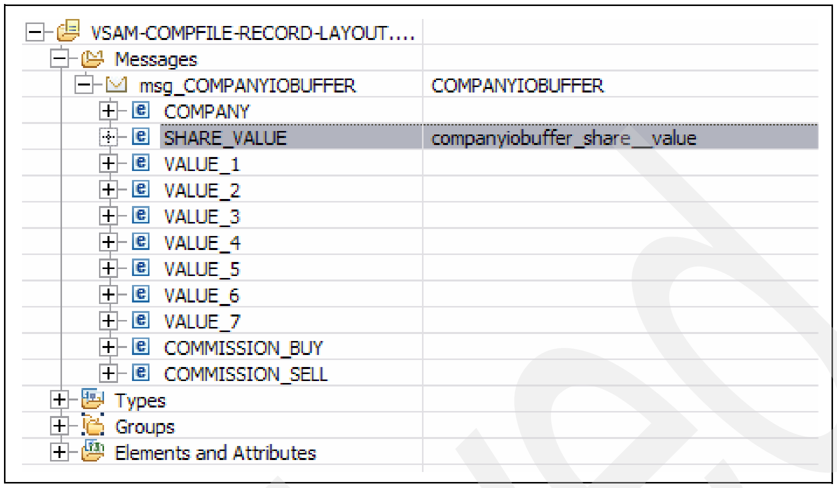
Example 5-1 COBOL copybook for company records

```
01 COMPANY-IO-BUFFER.
   03 COMPANY                 PIC X(20).
   03 SHARE-VALUE.
      05 SHARE-VALUE          PIC X(08).
   03 VALUE-1                 PIC X(08).
   03 VALUE-2                 PIC X(08).
   03 VALUE-3                 PIC X(08).
   03 VALUE-4                 PIC X(08).
   03 VALUE-5                 PIC X(08).
   03 VALUE-6                 PIC X(08).
   03 VALUE-7                 PIC X(08).
   03 COMMISSION-BUY          PIC X(03).
   03 COMMISSION-SELL         PIC X(03).
```

Perform the following steps:

1. Start the WebSphere Message Broker Toolkit and open the Broker Application Development perspective.
2. Create a new Message Set Project called MS_TraderCompanyFile_Project and within this, a new Message Set called MS_TraderCompanyFile. In the next window, select the **Custom Wire Format Name**. This sets the field to CWF1.
3. Import the COBOL copybook structure, VSAM-COMPFILE-RECORD-LAYOUT.cpy, into the MS_TraderCompanyFile_Project project.
4. Right-click the **VSAM-COMPFILE-RECORD-LAYOUT.cpy** file and create a new Message Definition File within the MS_TraderCompanyFile message set. Check for the creation of the new message definition file based on the root element, COMPANYIOBUFFER.

The new message definition will look similar to Figure 5-3.



VSAM-COMPFILE-RECORD-LAYOUT....	
Messages	
msg_COMPANYIOBUFFER	COMPANYIOBUFFER
COMPANY	
SHARE_VALUE	companyiobuffer_share__value
VALUE_1	
VALUE_2	
VALUE_3	
VALUE_4	
VALUE_5	
VALUE_6	
VALUE_7	
COMMISSION_BUY	
COMMISSION_SELL	
Types	
Groups	
Elements and Attributes	

Figure 5-3 Sample message flow definition

5.4.2 TraderMQtoMQ extended message flow

The TraderMQtoMQ_Ext sample flow is an extended version of the TraderMQtoMQ message flow previously described in Chapter 4, “Sample scenarios” on page 195. This extended message flow demonstrates the use of a few of the z/OS specific nodes.

The Trader MQtoMQ_Ext message flow shows the mediation between an MQ Client application issuing requests through XML MQ messages and a target CICS application. For demonstrative purposes, we have used a FileWrite node to log Request messages to a QSAM file and a VSAMWrite node to log Response messages to a ESDS VSAM file.

A CICSRequest node is used to invoke the target application through a message built according to the CommArea structure of the CICS target application.

The CICS target application stores a record for each of the Companies whose stock in trades is CICSUSER.CICS220.CICSERW1.COMPFILE, a KSDS VSAM file, with COMPANY defined as the Key, as shown in Example 5-2.

Example 5-2 VSAM file sample

```
CATALOG
 RBA 0            Key 'Casey_Import_Export '           Col 1        Format CHAR
 VOLSER TOTCII   Type KSDS   DSNAME CICSUSER.CICS220.CICSERW1.COMPFILE

 RBA        Len     <===5===10====5====>....5...30....5...40....5...50....5...60
 0            90    Casey_Import_Export 00079.0000077.0000078.0000072.0000070.00
 90           90    Glass_and_Luget_Plc 00019.0000022.0000025.0000020.0000016.00
 180          90    Headworth_Electrical00124.0000131.0000133.0000133.0000133.00
 270          90    IBM                 00163.0000163.0000162.0000160.0000161.00
```

The TraderMQtoMQ_Ext message flow implements a VSAMRead node that reads a company record from the CICS VSAM file. The flow compares the share value in the record with a value passed in by the request message, to determine whether or not to buy shares.

The message flow for the TraderMQtoMQ_Ext sample performs the following tasks:

1. The ClientRequest MQInput node reads a message from the client request queue.
2. The SetVSAMRequest Compute node copies the entire message, and converts the short version of the company name, passed in by the request message, to the long version. It then sets the VSAMRead Request property, called Key, to this value.
3. The GetCompanyRecord VSAMRead node accesses the CICS application VSAM file, CICSUSER.CICS220.CICSERW1.COMPFILE, and searches the records based on the Key named COMPANY. If it finds a company name that matches the value of the Key set in the SetVSAMRequest node, it reads that record and propagates it to the rest of the message flow.
4. The LimitExceeded Filter node returns true if the SHARE-VALUE in the record is greater than the PRICE_LIMIT passed in by the request message.

For a True Branch, the following steps are performed:

1. Flow Order - first:
   a. The ClientToTarget Mapping node converts the XML input message to a message containing the CICS COMMAREA, as expected by the CICS target application.
   b. The CICSRequest node synchronously executes the CICS target application and the response COMMAREA is returned in the response message. Control is then passed to the next node in the message flow.
   c. The TargetToClient mapping node converts the response message containing the CICS COMMAREA to an XML response message, as expected by the client application.
   d. The LogResponse VSAMWrite node logs the positive response messages to the specified ESDS VSAM file.
   e. The PositiveResponse MQReply node sends the positive response message to the client application.
2. Flow Order - second:
   a. The ConvertCCSID Compute node changes the request message from Unicode to EBCDIC (CCSID 500), so that the record in the QSAM file is readable on the z/OS system.
   b. The LogRequest FileWrite node logs the request messages to the specified QSAM file and propagates a status message to the QSAMstatus MQOutput node.

For a False Branch, the following steps are performed:

1. The NegativeReply Mapping node updates the RETURN_VALUE field of the negative response message to ‘99’.
2. The LogResponse1 VSAMWrite node logs the negative response messages to the specified ESDS VSAM file.
3. The NegativeResponse MQReply node sends the negative response message to the client application.

The TraderMQtoMQ_Ext sample message flow is shown in Figure 5-4.

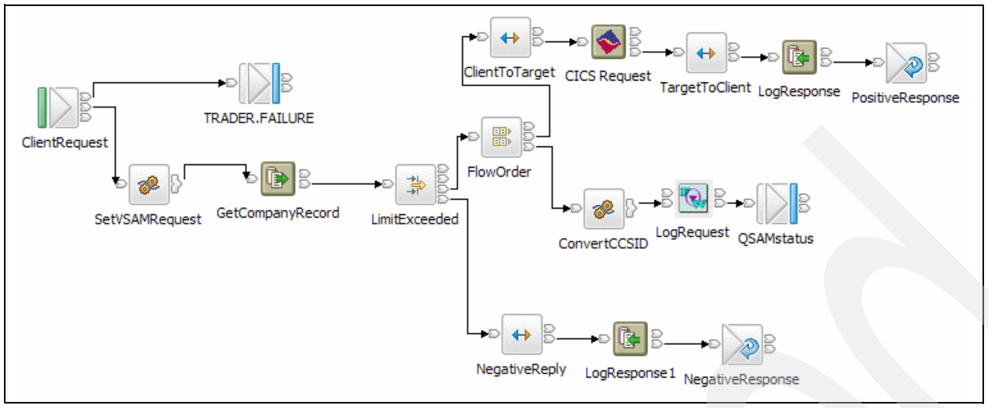


Figure 5-4 TraderMQtoMQ_Ext sample message flow

Message flow node details

This section details the properties of each significant node:

Node name: ClientRequest

This node reads a the request message from the client request queue. See Table 5-1 for a description of the property values changed from the defaults.

Table 5-1 Property values description

Property	Property name: value	Comment
Basic	Queue Name:TRADER.MQTOMQ.INPUT	Defines the queue from which the input node is reading.
Default	Message Domain: MRM	The domain to which the input message belongs.
	MessageSet: MS_TraderMQClient_Ext	The message set describing the incoming message.
	Message Type: MQClient	The message type describing the incoming message.
	Message Format: XML1	The physical format of the incoming message.

Node name: SetVSAMRequest

The relevant ESQL computed logic is detailed in Example 5-3. The node copies the entire message and uses the Map_Company_Name user defined function to convert the short version of the company name, passed in by the request message, to the long version. It then sets the VSAMRead request property, Key, to this value.

Example 5-3 ESQL for SetVSAMRequest Node

```
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      CALL CopyEntireMessage();
      DECLARE CompanyName CHAR;
      -- Convert the company short name to the company long name
SET CompanyName =
Map_Company_Name(InputRoot.MRM.MQClientInput.COMPANY_NAME);
      -- We are going to set the VSAM request information
      SET OutputLocalEnvironment.VSAM.Request.Position.Key =
CompanyName;
      RETURN TRUE;
   END;
```

Table 5-2 details the property changes for this node.

Table 5-2 Property changes description

Property	Property name: value	Comment
Basic	Compute Mode: LocalEnvironment and Message	Because you are making a change to the LocalEnvironment, you have to propagate both the LocalEnvironment and the Message.

Node name: GetCompanyRecord

This node searches the CICS application VSAM file that contains the company records. If it matches a value in the COMPANY field with the value stored in Key, the record is read and propagated to the rest of the message flow.

Table 5-3 shows a description of the property values changed from the defaults.

Table 5-3 Property values for GetCompanyRecord

Property	Property name: value	Comment
Basic	Default File Name: //'CICSUSER.CICS220.CICSERW1.COMPFILE'	Defines the VSAM data set from which the node is reading.
Advanced	Generate Mode: Message And LocalEnvironment	Specifies what is propagated in the output message assembly.
	Copy Message: Copy Entire Message	Specifies how much of the input message is copied into the output message.
	Copy Local Environment: Copy Entire LocalEnvironment	Specifies whether the input local environment is copied to the output local environment.
	Retain File: Until Idle	Indicates whether the file remains open.
Default	Message Domain: MRM	The domain to which the message describing the VSAM record belongs.
	MessageSet: MS_TraderCompanyFile	The message set to which the message describing the VSAM record belongs.
	Message Type: msg_COMPANYIOBUFFER	The message type to which the message describing the VSAM record belongs.
	Message Format: CWF1	The physical format of the message describing the VSAM record.
Request	Request Location: InputLocalEnvironment.VSAM.Request	Specifies the location of the request message in the form of an ESQL field reference.
	Position Mode: KEY_EQ - Key equal	Specifies the method of positioning to the record that is read.
	Key: ‘ ‘	Key to be used. This is left blank, as we have set this value in the SetVSAMRequest node.
	Key Type: Parser String	Describes the format of the key.
Result	Output Data Location: OutputLocalEnvironment.VSAM.Result	The location in the message assembly where the VSAM record data is written.

Node name: LimitExceeded

The default properties can be accepted in this Filter node. The relevant ESQL filtered logic is detailed in Example 5-4. The node returns true if the SHARE-VALUE in the record is greater than the PRICE_LIMIT passed in by the request message.

Example 5-4 Relevant ESQL for LimitExceeded Node

```
   CREATE FUNCTION Main() RETURNS BOOLEAN
   BEGIN
      RETURN CAST(LocalEnvironment.VSAM.Result.SHARE_VALUE.SHARE_VALUE
AS DECIMAL) > CAST(Root.MRM.MQClientInput.PRICE_LIMIT AS DECIMAL);
   END;
```

Node name: Flow Order

This node governs the order of the flow branches that leave from the output terminals, which are First and Second.

Node name: ClientToTarget

The map is responsible for mapping the property values so that they match the values expected by the target application. The following three property values have to be set manually:

MessageSet	The unique message set identifier of the target message set MS_TraderMQTarget.
MessageType	The message type name of the target message, CicsCommArea.
MessageFormat	The physical format of the target message, CWF1.

The remaining fields in the Properties tree can be copied unchanged.

Figure 5-5 shows how to map the message payload.

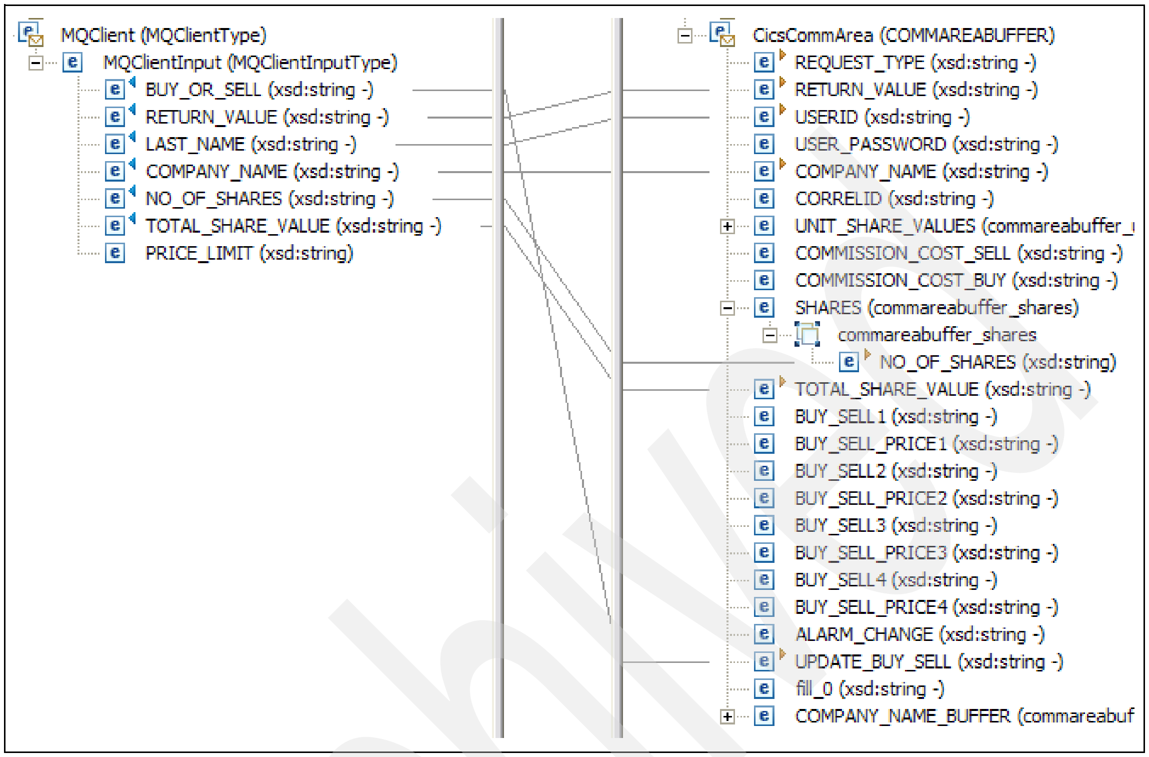

MQClient (MQClientType)
MQClientInput (MQClientInputType)
BUY_OR_SELL (xsd:string -)
RETURN_VALUE (xsd:string -)
LAST_NAME (xsd:string -)
COMPANY_NAME (xsd:string -)
NO_OF_SHARES (xsd:string -)
TOTAL_SHARE_VALUE (xsd:string -)
PRICE_LIMIT (xsd:string)
CicsCommArea (COMMAREABUFFER)
REQUEST_TYPE (xsd:string -)
RETURN_VALUE (xsd:string -)
USERID (xsd:string -)
USER_PASSWORD (xsd:string -)
COMPANY_NAME (xsd:string -)
CORRELID (xsd:string -)
UNIT_SHARE_VALUES (commareabuffer_
COMMISSION_COST_SELL (xsd:string -)
COMMISSION_COST_BUY (xsd:string -)
SHARES (commareabuffer_shares)
commareabuffer_shares
NO_OF_SHARES (xsd:string)
TOTAL_SHARE_VALUE (xsd:string -)
BUY_SELL1 (xsd:string -)
BUY_SELL_PRICE1 (xsd:string -)
BUY_SELL2 (xsd:string -)
BUY_SELL_PRICE2 (xsd:string -)
BUY_SELL3 (xsd:string -)
BUY_SELL_PRICE3 (xsd:string -)
BUY_SELL4 (xsd:string -)
BUY_SELL_PRICE4 (xsd:string -)
ALARM_CHANGE (xsd:string -)
UPDATE_BUY_SELL (xsd:string -)
fill_0 (xsd:string -)
COMPANY_NAME_BUFFER (commareabuf

Figure 5-5 Message payload map

Furthermore, the map also builds the following fields:

- REQUEST_TYPE is set to the fixed value 'Buy_Sell'.
- COMPANY_NAME is set through the user ESQL function, Map_Company_Name, which receives the source field COMPANY_NAME as its input. The sample in Example 5-5 describes the ESQL code of this user function.

Example 5-5   ESQL for Map_Company_Name

```
CREATE FUNCTION Map_Company_Name (ShortName CHAR) RETURNS CHAR
   BEGIN
      DECLARE LongName CHAR;
      IF ShortName='CAS' THEN
         SET LongName='Casey_Import_Export';
      ELSEIF ShortName='GLP' THEN
         SET LongName='Glass_and_Luget_Plc';
      ELSEIF ShortName='HWE' THEN
         SET LongName='Headworth_Electrical';
      ELSE
         SET LongName='IBM';
      END IF;
   RETURN LongName;
   END;
```

Node name: CICSRequest

This node synchronously executes the CICS target application and the response COMMAREA is returned in the response message. See Table 5-4 for a description of the property values changed from the defaults.

Table 5-4   Property values for CICSRequest

Property	Property name: value	Comment
Basic	CICS Network Applid*: SCSCERW1	Defines the name by which the CICS system is known in the network.
	CICS Program Name*: TRADERBL	The name of the CICS program that you want to invoke.
	CICS Userid*: BATCHCLI	The CICS user ID for execution.
Advanced	Include CICS request in message flow transaction: Commit	Determines whether or not the CICS transaction is part of the message flow transaction. Commit means that CICS commits when sending the reply to the node.

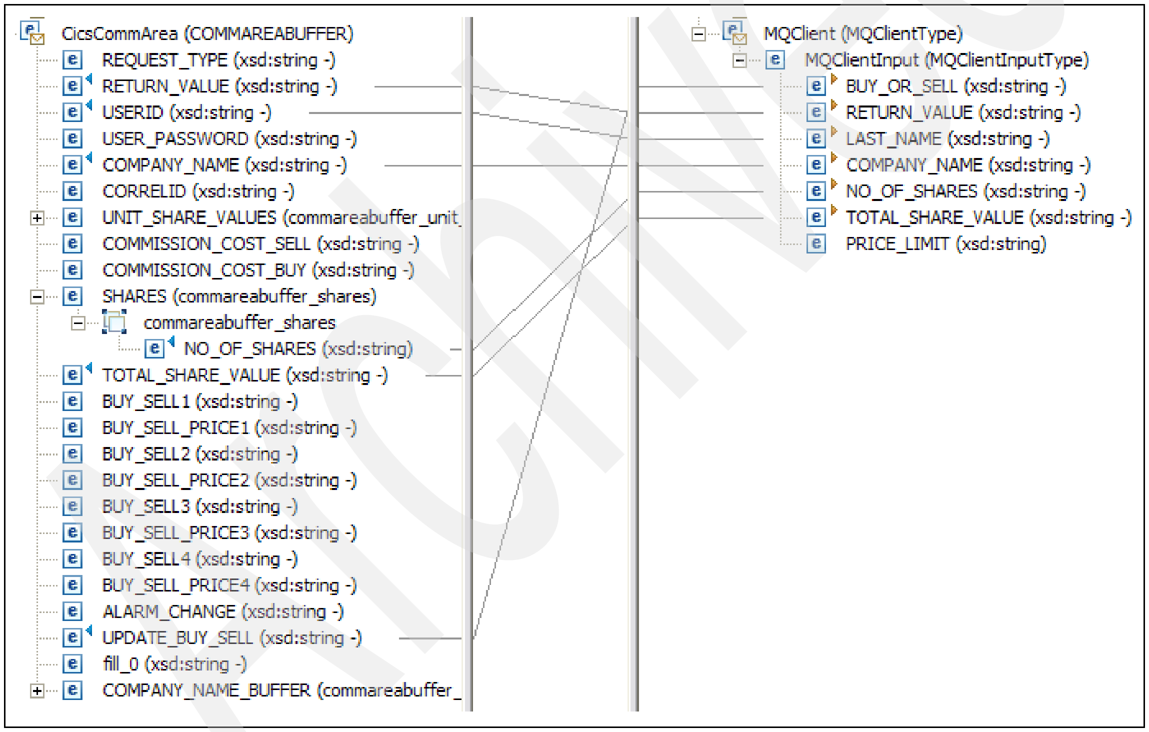
Node name: TargetToClient

The map is responsible for mapping the property values so that they match the values expected by the client application. The following three property values have to be set manually:

MessageSet	The unique message set identifier of the client message set MS_TraderMQClient_Ext.
MessageType	The message type name of the target message, MQClient.
MessageFormat	The physical format of the target message, XML1.

The remaining fields in the Properties tree can be copied unchanged.

Figure 5-6 shows how to map the message payload.



```
CicsCommArea (COMMAREABUFFER)
   REQUEST_TYPE (xsd:string -)
   RETURN_VALUE (xsd:string -)
   USERID (xsd:string -)
   USER_PASSWORD (xsd:string -)
   COMPANY_NAME (xsd:string -)
   CORRELID (xsd:string -)
   UNIT_SHARE_VALUES (commareabuffer_unit_
   COMMISSION_COST_SELL (xsd:string -)
   COMMISSION_COST_BUY (xsd:string -)
   SHARES (commareabuffer_shares)
      commareabuffer_shares
         NO_OF_SHARES (xsd:string)
   TOTAL_SHARE_VALUE (xsd:string -)
   BUY_SELL1 (xsd:string -)
   BUY_SELL_PRICE1 (xsd:string -)
   BUY_SELL2 (xsd:string -)
   BUY_SELL_PRICE2 (xsd:string -)
   BUY_SELL3 (xsd:string -)
   BUY_SELL_PRICE3 (xsd:string -)
   BUY_SELL4 (xsd:string -)
   BUY_SELL_PRICE4 (xsd:string -)
   ALARM_CHANGE (xsd:string -)
   UPDATE_BUY_SELL (xsd:string -)
   fill_0 (xsd:string -)
   COMPANY_NAME_BUFFER (commareabuffer_

MQClient (MQClientType)
   MQClientInput (MQClientInputType)
      BUY_OR_SELL (xsd:string -)
      RETURN_VALUE (xsd:string -)
      LAST_NAME (xsd:string -)
      COMPANY_NAME (xsd:string -)
      NO_OF_SHARES (xsd:string -)
      TOTAL_SHARE_VALUE (xsd:string -)
      PRICE_LIMIT (xsd:string)
```

Figure 5-6 Message payload map

The map also trims the field, COMPANY_NAME, down to three characters using the esql function left(). The code is as follows:

```
esql:left($source/CicsCommArea/COMPANY_NAME,3)
```

Node name: LogResponse

This node logs the positive response message to the VSAM data set specified in the node. See Table 5-5 for a description of the property values changed from the defaults.

Table 5-5 Property values for LogResponse

Property	Property name: value	Comment
Basic	Default File Name: //'TRADER.VSAM.ESDS'	Defines the VSAM dataset to which the node is writing.
Advanced	Retain File: Until Idle	Indicates whether the file remains open.
	Truncate: check this option	This must be selected when the data being written is greater than the record length of the VSAM dataset.
Default	MessageSet: MS_TraderMQClient_Ext	The message set describing the message that is going to be written to the VSAM file.
	Message Type: MQClient	The message type describing the message that is going to be written to the VSAM file.
	Message Format: XML1	The physical format of the message that is going to be written to the VSAM file.
	CCSID: 500	The code page you want to use when you write the message.
Request	Request Location: InputLocalEnvironment.VSAM.Request	Specifies the location of the request message in the form of an ESQL field reference.

Node name: PositiveResponse

This node delivers the positive response message to the queue specified in the ReplyToQ fields. The properties were left as the default values.

Node name: ConvertCCSID

Unlike the VSAM FileWrite node, the QSAM FileWrite node does not contain an option to specify the code page to use when you write the message. This node changes the message from Unicode to EBCDIC (CCSID 500) so that the record in the QSAM file is readable on the z/OS system. The properties were left as the default values.

The relevant ESQL computed logic is detailed in Example 5-6.

Example 5-6 ESQL computed logic for ConvertCCSID Node

```
CREATE FUNCTION Main() RETURNS BOOLEAN
BEGIN
   CALL CopyEntireMessage();
   SET OutputRoot.MQMD.CodedCharSetId = 500;
   RETURN TRUE;
END;
```

Node name: LogRequest

This node logs the request message to the QSAM dataset specified in the node. Table 5-6 shows a description of the property values changed from the defaults.

Table 5-6 Property values for LogRequest

Property	Property name: value	Comment
Basic	Automated Mode: check this option	A new file is opened when a message arrives on the “in” terminal and there is no output file already opened.
Automated Mode	The node generates a unique output file name by appending a suffix to the Filename in the form: JulianDate.Time.SequenceNum - Dyyddd.Thhmmss.Snnnn.	
	The attributes of QSAM dataset you want to create are specified here.	

Node name: QSAMstatus

This node propagates the status message received from the LogRequest node to the specified output queue. Table 5-7 details the property changes for this node.

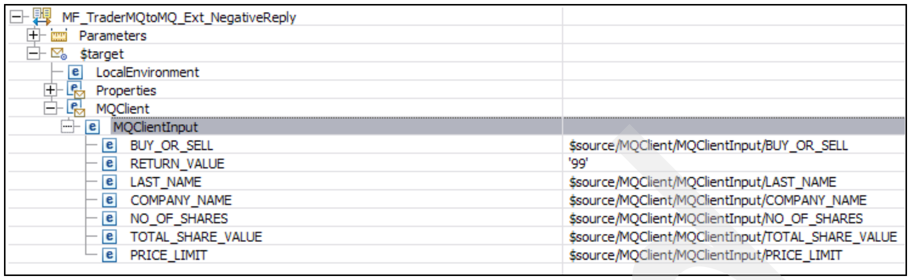
Table 5-7 Property values for QSAMstatus

Property	Property name: value	Comment
Basic	Queue Name: TRADER.QSAM.STATUS	Defines the queue to which the QSAM node status message is going to be delivered.

Node name: NegativeReply

The map is responsible for mapping the original client request message to a negative client response message. All the property values remain unchanged, similar to the MQClient message fields, with one exception. The RETURN_VALUE field is set to the value ‘99’ to indicate the message is a negative response, which means that the value of the shares were greater than the price limit.

Figure 5-7 shows how to map the message payload.



MF_TraderMQtoMQ_Ext_NegativeReply	
Parameters	
$target	
LocalEnvironment	
Properties	
MQClient	
MQClientInput	
BUY_OR_SELL	$source/MQClient/MQClientInput/BUY_OR_SELL
RETURN_VALUE	'99'
LAST_NAME	$source/MQClient/MQClientInput/LAST_NAME
COMPANY_NAME	$source/MQClient/MQClientInput/COMPANY_NAME
NO_OF_SHARES	$source/MQClient/MQClientInput/NO_OF_SHARES
TOTAL_SHARE_VALUE	$source/MQClient/MQClientInput/TOTAL_SHARE_VALUE
PRICE_LIMIT	$source/MQClient/MQClientInput/PRICE_LIMIT

Figure 5-7 Message payload flow

Node name: LogResponse1

This node logs the negative response message to the VSAM dataset specified in the node. Table 5-8 shows a description of the property values changed from the defaults.

Table 5-8 Property values for LogResponse1

Property	Property name: value	Comment
Basic	Default File Name: //'TRADER.VSAM.ESDS'	Defines the VSAM dataset to which the node is writing.
Advanced	Retain File: Until Idle	Indicates whether the file remains open.
	Truncate: Check this option	This must be selected when the data being written is greater than the record length of the VSAM dataset.
Default	MessageSet: MS_TraderMQClient_Ext	The message set describing the message that is going to be written to the VSAM file.
	Message Type: MQClient	The message type describing the message that is going to be written to the VSAM file.
	Message Format: XML1	The physical format of the message that is going to be written to the VSAM file.
	CCSID: 500	The code page you want to use when you write the message.
Request	Request Location: InputLocalEnvironment.VSAM.Request	Specifies the location of the request message in the form of an ESQL field reference.

Node name: NegativeResponse

This node delivers the negative response message to the queue specified in the ReplyToQ fields. The properties were left as the default values.

A

WebSphere ESB related scripts

This appendix assembles several elements that where used during our project to facilitate the WebSphere Enterprise Service Bus (ESB) installation and configuration.

WSADMIN scripting in batch

We often used batch executions for WebSphere Application Server administration instead of the “Admin Console”. To do so, you have to use the **wsadmin** function. Many examples are available as good starting points. You can reuse and repeat this task very easily after you have an appropriate set of scripts with variable parameters.

You can find documentation about **wsadmin** scripting in the following documents:

- *WebSphere Version 5 for z/OS- WSADMIN Primer*, WP100421
- *IBM WebSphere Application Server for z/OS Version 6.1: Using the administrative clients*, SA23-2208

You can use both JACL and JYTHON scripts. A flag indicates whether the input script is in Extended Binary Coded Decimal Interchange Code (EBCDIC) or American Standard Code for Information Interchange (ASCII) in the z/OS environment. Example A-1 shows some JCL samples that you can use for submitting **wsadmin** in batch.

Example: A-1 Execute of JACL script in ASCII mode

```
//XXXXXXXX JOB (999,POK),'VAN AERSCHOT - 2C09',CLASS=A,
//   MSGCLASS=X,NOTIFY=VANAERS,MSGLEVEL=(1,1),
//   REGION=0M
//BBOINST  EXEC PGM=IKJEFT01,DYNAMNBR=250,REGION=0M
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//SYSTSIN DD   *
 BPXBATCH SH  +
 /wassoconfig/socell/sodmnode/DeploymentManager/bin/+
 wsadmin.sh +
 -conntype soap -host 127.0.0.1 -port 29510 +
 -f /u/vanaers/wsadmin/jacl/itsoSIBSample.jacl -inline
```

We think that “Jython” is the best choice in a WebSphere environment. Jython is Python with Java, as shown in Example A-2.

Example: A-2 Execute of Python script in EBCDIC mode

```
//XXXXXXXX JOB (999,POK),'VAN AERSCHOT - 2C09',CLASS=A,
//   MSGCLASS=X,NOTIFY=VANAERS,MSGLEVEL=(1,1),
//   REGION=0M
//BBOINST  EXEC PGM=IKJEFT01,DYNAMNBR=250,REGION=0M
//SYSPRINT DD SYSOUT=*
//SYSTSPRT DD SYSOUT=*
//STDOUT   DD SYSOUT=*
//STDERR   DD SYSOUT=*
//SYSTSIN DD   *
 BPXBATCH SH  +
 /wassoconfig/socell/sodmnode/DeploymentManager/bin/+
 wsadmin.sh -lang jython -javaoption -Dscript.encoding=Cp1047 +
 -conntype soap -host 127.0.0.1 -port 29510 +
 -f /u/vanaers/wsadmin/python/configAll.py.ebcdic  all
```

Example A-3 is a Jython script, delivered with the product, as an example that you can use to list the entire installed configuration.

Example: A-3 Python script to display the configuration

```
# This program may be used, executed, copied, modified and distributed
# without royalty for the purpose of developing, using, marketing, or distribution
#-----------------------------------------------------------------
# configAll.py - Jython implementation of example script 6
#-----------------------------------------------------------------
# The purpose of this example is to demonstrate the invocation
#  of some commands that produce a short summary of some configuration
#  and runtime information about the WebSphere installation.
#
#  This script it can be included in the wsadmin command invocation
# like this:
#     wsadmin -lang jython -f configAll.py all
# or the script can be execfiled from the wsadmin command line like
# this:
#     wsadmin> execfile ("configAll.py") or execfile("configAll.py")
#     wsadmin> configAll("all")
#
#  The script expects one parameter:
#      arg1 - a flag -- either "all" or "config"
#
#  This example demonstrates many wsadmin features:
```

```
#
#   - The use of the AdminControl object to locate running MBeans
#   - The use of the AdminControl object to getAttributes from running MBeans
#   - The use of the AdminControl object to invoke operations on running MBeans
#   - The use of the AdminConfig  object to locate configuration objects
#   - The use of the AdminConfig  object to display attributes of configuration
# objects
#   - The use of the AdminConfig getObjectName command to look up a running object.
#   - The use of the AdminControl getConfigId  command to look up a config object.
#-----------------------------------------------------------------
#
import sys, java

lineSeparator = java.lang.System.getProperty('line.separator')

def configAll(flag):
   if (flag == "config"):
      configOnly = 1
   elif (flag == "all"):
      configOnly = 0
   else:
      print 'Single argument must be "all" or "config"'
      return

   #-------------------------------------------------------------
   # set up globals
   #-------------------------------------------------------------
   global AdminControl
   global AdminConfig
   global AdminApp


   print "Installation summary: "
   print "----------------------------------------------------"
   print ""

   #-------------------------------------------------------
   # Get the cells/nodes/servers
   #-------------------------------------------------------
   print "Configured cells, nodes, and servers: "
   print "----------------------------------------------------"
   print ""
   cells = AdminConfig.list("Cell").split(lineSeparator)
   print "Number of cells: %s" % (len(cells))
```

```
for cell in cells:

   #--------------------------------------------------------
   # Get some attributes from the config for this cell --
   # the name and the security enabled flag.
   #--------------------------------------------------------
   cname = AdminConfig.showAttribute(cell, "name")
   sec = AdminConfig.list("Security", cell)
   enabled = AdminConfig.showAttribute(sec, "enabled")
   print cname + "-- security enabled: " + enabled

   #--------------------------------------------------------
   # Get a list of the nodes in this cell, and the name of each
   #--------------------------------------------------------
   nodes = AdminConfig.list("Node", cell)
   if len(nodes) == 0:
      print "  There is no node in " + cname
      continue

   nodesList = nodes.split(lineSeparator)
   print "  Number of nodes in %s: %s" % (cname, len(nodesList))
   for node in nodesList:
      nname = AdminConfig.showAttribute(node, "name")
      print "  " + nname

      #--------------------------------------------------------
      # Get a list of the servers on this node.  Use getObjectName
      # to see if there is a running server for this config object.
      #--------------------------------------------------------
      servers = AdminConfig.list("Server", node)
      if len(servers) == 0:
         print "    There is no server on " + nnmae
         continue

      serversList = servers.split(lineSeparator)
      print "    Number of servers in " + nname + ": %s" % (len(serversList))
      for server in serversList:
         print ""
         sname = AdminConfig.showAttribute(server, "name")
         runserv = AdminConfig.getObjectName(server)
         if len(runserv) > 0:
            state = AdminControl.getAttribute(runserv, "state")
            print "    Server " + sname + " is running: state is " + state
         else:
            print "    Server " + sname + " is not running"
         #----------------------------------------------------------
         # Get a list of the http transports on the server, and display
         # host and port information for them.
         #----------------------------------------------------------
         https = AdminConfig.list("HTTPTransport", server)
         if len(https) == 0:
            print "      There is no HTTP transport on " + sname
            continue

         httpsList = https.split(lineSeparator)
         print "       " + sname + " has %s HTTP transports" % (len(httpsList))
         for http in httpsList:
            addr = AdminConfig.showAttribute(http, "address")
            host = AdminConfig.showAttribute(addr, "host")
            port = AdminConfig.showAttribute(addr, "port")
            print "        port: " + port + " on host " + host
         print ""
      print ""


   #----------------------------------------------------------
   # Get a list of the ServerClusters and display it.
   #----------------------------------------------------------
   print ""
   clusters = AdminConfig.list("ServerCluster", cell)
   if len(clusters) == 0:
      print "  There is no clusters on " + cname
      continue

   clustersList = clusters.split(lineSeparator)
   print "  Number of clusters in " + cname + ": %s" % (len(clustersList))
   for cluster in clustersList:
      clname = AdminConfig.showAttribute(cluster, "name")
      memberList = AdminConfig.showAttribute(cluster, "members")
      if len(memberList) > 0:
         members = memberList[1:len(memberList)-1].split(" ")
         print "  Cluster " + clname + " has %s members" % (len(members))
         for member in members:
            mname = AdminConfig.showAttribute(member, "memberName")
            weight = AdminConfig.showAttribute(member, "weight")
            print "    Member " + mname + " has weight " + weight
      print ""

#----------------------------------------------------------
```

```
   # Get the apps
   #----------------------------------------------------------
   print "-----------------------------------------------------"
   print ""
   apps = AdminApp.list()
   if len(apps) == 0:
      print "There is no application"
   else:
      appsList = apps.split(lineSeparator)
      print "Number of applications: %s" % (len(appsList))
      print ""
      for app in appsList:
         print app

   if configOnly == 1:
      return

   #----------------------------------------------------------
   # What servers are running on each node, and what apps do they have?
   #----------------------------------------------------------
   print ""
   print "Running servers: "
   print "-----------------------------------------------------"
   print ""
   for cell in cells:
      cname = AdminConfig.showAttribute(cell, "name")
      nodes = AdminConfig.list("Node", cell)
      if len(nodes) == 0:
         print "There is no node on " + cname
         continue

      nodesList = nodes.split(lineSeparator)
      for node in nodesList:
         nname = AdminConfig.showAttribute(node, "name")
         servers = AdminControl.queryNames("type=Server,cell=" + cname + ",node=" +
nname + ",*")
         if len(servers) == 0:
            print "There is no running server on node " + nname
            continue

         serversList = servers.split(lineSeparator)
         print "Number of running servers on node %s: %s" % (nname, len(serversList))
         for server in serversList:

            #----------------------------------------------------------
            # Get some attributes from the server to display; also invoke
            # an operation on the server JVM to display a property.
            #--------------------------------------------------------------
            sname = AdminControl.getAttribute(server, "name")
            ptype = AdminControl.getAttribute(server, "processType")
            pid = AdminControl.getAttribute(server, "pid")
            state = AdminControl.getAttribute(server, "state")
            jvm = AdminControl.queryNames("type=JVM,cell=" + cname + ",node=" + nname
+ ",process=" + sname + ",*")
            osname  = AdminControl.invoke(jvm, "getProperty", "os.name")
            print "  %s (%s) has pid %s; state: %s; on %s" % (sname, ptype, pid,
state, osname)

            #--------------------------------------------------------------
            # Use getConfigId to see if there is config data for this
            # server.
            #--------------------------------------------------------------
            configserv = AdminControl.getConfigId(server)
            if len(configserv) > 0:
               print "  %s is configured." % (sname)
            else:
               print "  %s is not configured; configuration must have changed after
the server was started." % (sname)

            #--------------------------------------------------------------
            # Find the applications running on this server.
            #--------------------------------------------------------------
            apps = AdminControl.queryNames("type=Application,cell=" + cname +
",node=" + nname + ",process=" + sname + ",*")
            if len(apps) == 0:
               print "  There is no applications running on " + sname
            else:
               appsList = apps.split(lineSeparator)
               print "  Number of applications running on %s: %s" % (sname,
len(appsList))
               for app in appsList:
                  aname = AdminControl.getAttribute(app, "name")
                  print "    %s" % (aname)
            print "----------------------------------------------------------"
            print ""

            #--------------------------------------------------------------
            # Display the serverVersion information.
            #--------------------------------------------------------------
            svreport = AdminControl.getAttribute(server, "serverVersion")
            print "Server version report for this server follows:"
            print "  %s" % (svreport)
            print "----------------------------------------------------"
            print ""


#------------------------------------------------------------------
# Main
#------------------------------------------------------------------
if (len(sys.argv)) != 1:
   print 'configAll: this script requires 1 parameter: a flag that should be "all" or 
"config"'
   print ""
   print "e.g.:     configAll  all"
else:
   print '-> configAll: Jython version '
   configAll(sys.argv[0])
```

Example A-4 shows the Jython script for setting and listing the endpoints in a server.

Example: A-4 Jython script for setting and listing the endpoints in a server

```
#-------------------------------------------------------------------------------------
---
#                 setepandhttp6.py
#
#   this script can display and set the endpoints of an application server
------------
#   set depends on parameter  -setports (display is default)
#   dry run (no final save) if parameter -save NOT specified
#   parameters
#     -setports        set endpoints  (default display)
#     -save            save configuration (default nosave)
# ---> for ITSOPOK
#     -range           (3 digits) start of portnr - 2 digits are added see logic
#-------------------------------------------------------------------------------------
---
import sys
def wsadminToList(inStr):
        outList=[]
        if (len(inStr)>0 and inStr[0]=='[' and inStr[-1]==']'):
                tmpList = inStr[1:-1].split(" ")
        else:
                tmpList = inStr.split("\n")  #splits for Windows or Linux
        for item in tmpList:
                item = item.rstrip();        #removes any Windows "\r"
                if (len(item)>0):
                        outList.append(item)
        return outList
#endDef
#-------------------------------------------------------------------------------------
---
#   this file allows the change of the Endpoints of an Application Server
#   the jacl file can be executed with connection SOAP
#                                                 NONE
#-------------------------------------------------------------------------------------
---
print "->setepandhttp6 PSSC JYTHON version "
argerr = 0
errstr = ""
newPort = ""
i = 0  #forStart
argc = len(sys.argv)
while ( i < argc ):  #forTest
        arg = sys.argv[i]
        #set j [expr {${i} + 1}]
        #puts "3"
        #set arg2 [lindex $argv $j]
        #puts "-----argument $i  $arg   $j  $arg2 "
        if (arg == "-server"):
                i = i+1;
                if (i < argc):
                        serverName = sys.argv[i]
                else:
                        argerr = 2
                #endElse
        elif (arg == "-node"):
                i = i+1;
                if (i < argc):
                        nodeName = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-cell"):
                i = i+1;
                if (i < argc):
                        cellName = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-teamnr"):
                i = i+1;
                if (i < argc):
                        teamnr = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-range"):
                i = i+1;
                if (i < argc):
                        range = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-team"):
                i = i+1;
                if (i < argc):
                        teamnr = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-servnr"):
                i = i+1;
                if (i < argc):
                        servnr = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-boot"):
                i = i+1;
                if (i < argc):
                        bootPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-http"):
                i = i+1;
                if (i < argc):
                        httpPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-https"):
                i = i+1;
```

```
                if (i < argc):
                        httpsport = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-iiop"):
                i = i+1;
                if (i < argc):
                        iiopPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-iiops"):
                i = i+1;
                if (i < argc):
                        iiopsPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-drs"):
                i = i+1;
                if (i < argc):
                        drsPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-soap"):
                i = i+1;
                if (i < argc):
                        soapPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-jmsq"):
                i = i+1;
                if (i < argc):
                        jmsqPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-jmsd"):
                i = i+1;
                if (i < argc):
                        jmsdPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-sib"):
                i = i+1;
                if (i < argc):
                        sibPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-sibs"):
                i = i+1;
                if (i < argc):
                        sibsPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-sibm"):
                i = i+1;
                if (i < argc):
                        sibmPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-sibms"):
                i = i+1;
                if (i < argc):
                        sibmsPort = sys.argv[i]
                else:
                        argerr = 3
                #endElse
        elif (arg == "-setports"):
                setports = "YES"
        elif (arg == "-save"):
                savehere = "YES"
        else:
                argerr = 10
        #endElse
        i += 1  #forNext
#endWhile  (#endFor)
if (argerr > 0):
        print "-> setepandhttp6 ending with error "+argerr+" for "+errstr
        sys.exit()
#endIf
cellId = AdminConfig.getid("/Cell:/")
cellName = AdminConfig.showAttribute(cellId, "name")
```

```
if ( not  ("servnr" in locals().keys() or "servnr" in globals().keys())  ):
        servnr = "1"
#endIf
print "->setepandhttp6 parameters seem to be OK"
if ( ("teamnr" in locals().keys() or "teamnr" in globals().keys())  ):
        teamstr = "Team"
        teamstr += teamnr
        if ( not  ("server" in locals().keys() or "server" in globals().keys())  ):
                serverName = teamstr
                serverName += "Serv0"
                serverName += servnr
        #endIf
        if ( not  ("node" in locals().keys() or "node" in globals().keys())  ):
                nodeName = teamstr
                nodeName += "Node01"
        #endIf
        if ( not  ("cell" in locals().keys() or "cell" in globals().keys())  ):
                cellName = teamstr
                cellName += "NDCell01"
        #endIf
#endIf
servid = AdminConfig.getid("/Cell/"+cellName+"/Node/"+nodeName+"/Server/"+serverName
)
if (servid == ""):
        print "->setepandhttp6 invalid Server "+serverName+" in node("+nodeName+")
cell("+cellName+")"
        sys.exit()
#endIf
print "->setepandhttp6 for server("+serverName+") in node("+nodeName+")
cell("+cellName+")"

if ( ("setports" in locals().keys() or "setports" in globals().keys())  ):
      if ( ("range" in locals().keys() or "range" in globals().keys())  ):
                print "->setepandhttp6 ports will be changed to range " +range
                httpPort = range
                httpPort += "18"
                httpsPort = range
                httpsPort += "19"
                bootPort = range
                bootPort += "11"
                iiopPort = range
                iiopPort += "11"
                iiopsPort = range
                iiopsPort += "12"
                soapPort = range
                soapPort += "10"
                drsPort = range
                drsPort += "13"
                jmsqPort = range
                jmsqPort += "16"
                jmsdPort = range
                jmsdPort += "17"
                dcsPort = range
                dcsPort += "15"
                sibPort = range
                sibPort += "20"
                sibsPort = range
                sibsPort += "21"
                sibmPort = range
                sibmPort += "22"
                sibmsPort = range
                sibmsPort += "23"
                sasslPort = range
                sasslPort += "24"
                WC_admPort = range
                WC_admPort += "25"
                WC_admsPort = range
                WC_admsPort += "26"
                WC_defPort = range
                WC_defPort += "27"
                WC_defsPort = range
                WC_defsPort += "28"
                CServAPort = range
                CServAPort += "29"
                CMutlAPort = range
                CMutlAPort += "30"
        else:
                print "->setepandhttp6 individual ports will be changed"
                #endElse
#endIf
i = 0
#puts "**** [$AdminConfig showall $servid]"
#--------HTTP(S)_ADDRESSES-----------
wc = AdminConfig.list("WebContainer", servid )
transportAttr = AdminConfig.showAttribute(wc, "transports" )
transports = transportAttr[1:len(transportAttr)-1].split(" ")
for transport in transports:
   #print "->setepandhttp6 2" + transport
        address = AdminConfig.showAttribute(transport, "address" )
        #print "->setepandhttp6 3" + address
        port = AdminConfig.showAttribute(address, "port")
        host = AdminConfig.showAttribute(address, "host")
        ssl = AdminConfig.showAttribute(transport,"sslEnabled")
        print "-> *** HTTP_ADDRESS current port "+port+" on host "+host+" sslenabled
"+ssl
        if (ssl == "true"):
               addresshttps = address
        else:
               addresshttp = address
        #endElse
#endFor
if (host == ""):
        host = "*"
#endIf
if ( ("setports" in locals().keys() or "setports" in globals().keys())  ):
        if ( ("httpPort" in locals().keys() or "httpPort" in globals().keys())  ):
                print "-> ***-> HTTP_ADDRESS new port "+httpPort+" on host "+host
                AdminConfig.modify(addresshttp, [["host", host], ["port", httpPort]]
)
        #endIf
        if ( ("httpsPort" in locals().keys() or "httpsPort" in globals().keys())  ):
                print "-> ***-> HTTPS_ADDRESS new port "+httpsPort+" on host "+host
                AdminConfig.modify(addresshttps, [["host", host], ["port",
httpsPort]] )
        #endIf
#endIf
#-------------------------------------
print "-> *** EndPoint Addresses ****"
#------------------------------------------------------------------ -----------
nodeid = AdminConfig.getid("/Cell/"+cellName+"/Node/"+nodeName )
serverEntriesAttr = (AdminConfig.list("ServerEntry", nodeid))
#print "###1" + serverEntriesAttr +"/"
serverEntries =
serverEntriesAttr.split(java.lang.System.getProperty("line.separator"))
for serverEntry in serverEntries:
   #print "###1A" + serverEntry +"/"
        sName = AdminConfig.showAttribute(serverEntry, "serverName" )
        #print "###1B" + sName +"/"
        if (sName == serverName):
                print " ***-> Special entries for server "+sName
                specialEndPoints = AdminConfig.showAttribute(serverEntry,
"specialEndpoints" )
                #print "###1BA" + specialEndPoints +"/"
                specialEndPoints = AdminConfig.showAttribute(serverEntry,
"specialEndpoints")[1:len(specialEndPoints)-1].split(" ")
```

```
                for specialEndPoint in specialEndPoints:
                        endPointNm = AdminConfig.showAttribute(specialEndPoint, 
"endPointName" )
                        #print "###1BAA" + endPointNm
                        newPort = ""
                        epoint = AdminConfig.showAttribute(specialEndPoint, 
"endPoint" )
                        port = AdminConfig.showAttribute(epoint,"port")
                        host = AdminConfig.showAttribute(epoint,"host")
                        print "   "+endPointNm+"  current port "+port+" on host 
"+host+" "
                        if ( ("setports" in locals().keys() or "setports" in 
globals().keys())  ):
                                if (endPointNm == "JMSSERVER_DIRECT_ADDRESS"):
                                        if ( ("jmsdPort" in locals().keys() or 
"jmsdPort" in globals().keys())  ):
                                                 newPort = jmsdPort
                                        else:
                                                 newPort = ""
                                        #endElse
                                elif (endPointNm == "JMSSERVER_QUEUED_ADDRESS"):
                                        if ( ("jmsqPort" in locals().keys() or 
"jmsqPort" in globals().keys())  ):
                                                 newPort = jmsqPort
                                        else:
                                                 newPort = ""
                                        #endElse
                                elif (endPointNm == "ORB_LISTENER_ADDRESS"):
                                        if ( ("iiopPort" in locals().keys() or 
"iiopPort" in globals().keys())  ):
                                                 newPort = iiopPort
                                        else:
                                                 newPort = ""
                                        #endElse
                                elif (endPointNm == "ORB_SSL_LISTENER_ADDRESS"):
                                        if ( ("iiopsPort" in locals().keys() or 
"iiopsPort" in globals().keys())  ):
                                                 newPort = iiopsPort
                                        else:
                                                 newPort = ""
                                        #endElse
                                elif (endPointNm == "SOAP_CONNECTOR_ADDRESS"):
                                        if ( ("soapPort" in locals().keys() or 
"soapPort" in globals().keys())  ):
                                                 newPort = soapPort
                                        else:
                                                newPort = ""
                                        #endElse
                                elif (endPointNm == "SIB_ENDPOINT_ADDRESS"):
                                        if ( ("sibPort" in locals().keys() or 
"sibPort" in globals().keys())  ):
                                                newPort = sibPort
                                        else:
                                                newPort = ""
                                        #endElse
                                elif (endPointNm == "SIB_ENDPOINT_SECURE_ADDRESS"):
                                        if ( ("sibsPort" in locals().keys() or 
"sibsPort" in globals().keys())  ):
                                                newPort = sibsPort
                                        else:
                                                newPort = ""
                                        #endElse
                                elif (endPointNm == "SIB_MQ_ENDPOINT_ADDRESS"):
                                        if ( ("sibmPort" in locals().keys() or 
"sibmPort" in globals().keys())  ):
                                                newPort = sibmPort
                                        else:
                                                newPort = ""
                                        #endElse
                                elif (endPointNm == 
"SIB_MQ_ENDPOINT_SECURE_ADDRESS"):
                                        if ( ("sibmsPort" in locals().keys() or 
"sibmsPort" in globals().keys())  ):
                                                newPort = sibmsPort
                                        else:
                                                newPort = ""
                                        #endElse
                                elif (endPointNm == "BOOTSTRAP_ADDRESS"):
                                        if ( ("bootPort" in locals().keys() or 
"bootPort" in globals().keys())  ):
                                                newPort = bootPort
                                        else:
                                                newPort = ""
                                        #endElse
                                elif (endPointNm == "DCS_UNICAST_ADDRESS"):
                                        if ( ("dcsPort" in locals().keys() or 
"dcsPort" in globals().keys())  ):
                                                newPort = dcsPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm ==
"SAS_SSL_SERVERAUTH_LISTENER_ADDRESS"):
                                        if ( ("sasslPort" in locals().keys() or
"sasslPort" in globals().keys())  ):
                                                newPort = sasslPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm == "WC_adminhost"):
                                        if ( ("WC_admPort" in locals().keys() or
"WC_admPort" in globals().keys())  ):
                                                newPort = WC_admPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm == "WC_adminhost_secure"):
                                        if ( ("WC_admsPort" in locals().keys() or
"WC_admsPort" in globals().keys())  ):
                                                newPort = WC_admsPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm == "WC_defaulthost"):
```

```
                                        if ( ("WC_defPort" in locals().keys() or
"WC_defPort" in globals().keys())  ):
                                                newPort = WC_defPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm == "WC_defaulthost_secure"):
                                        if ( ("WC_defsPort" in locals().keys() or
"WC_defsPort" in globals().keys())  ):
                                                newPort = WC_defsPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm ==
"CSIV2_SSL_SERVERAUTH_LISTENER_ADDRESS"):
                                        if ( ("CServAPort" in locals().keys() or
"CServAPort" in globals().keys())  ):
                                                newPort = CServAPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                elif (endPointNm ==
"CSIV2_SSL_MUTUALAUTH_LISTENER_ADDRESS"):
                                        if ( ("CMutlAPort" in locals().keys() or
"CMutlAPort" in globals().keys())  ):
                                                newPort = CMutlAPort
                                        else:
                                                newPort = ""
                                        #endElse
                                else:
                                        print "-> setepandhttp6 bypass "+endPointNm+"
NOT considered <-"
                                        newPort = ""
                                #endElse
                                if ( ("newPort" in locals().keys() or "newPort" in
globals().keys())  ):
                                        if (newPort == port):
                                                print "-> setepandhttp6 bypass
change <-"
                                        elif (newPort == ""):
                                                print "-> setepandhttp6 NO new value
for "+endPointNm+" bypass"
                                        else:
                                                print "-> setepandhttp6 ******
"+endPointNm+" old port("+port+") new port("+newPort+") on host("+host+") <-"
                                                AdminConfig.modify(epoint, [["host",
host], ["port", newPort]] )
                                                print "-> setepandhttp6 change OK
<-"
                                        #endElse
                                #endIf
                        #endIf
                #endFor
        #endIf
#endFor
if ( ("savehere" in locals().keys() or "savehere" in globals().keys())  ):
        print "-> setepandhttp6 saving"
        try:
        print "-> setepandhttp6 try to save with synclevel"
        sync1 = AdminControl.completeObjectName("type=NodeSync,node="+nodeName+",*" )
        print "-> psscserver6 saving with synclevel "+sync1+" "
        AdminControl.invoke(sync1, "sync" )
        print "-> setepandhttp6 synclevel accepted"
        except:
      print "-> setepandhttp6 Unexpected error:", sys.exc_info()[0]
      print "-> setepandhttp6 synclevel denied"
        else:
           AdminConfig.save( )
        print "-> setepandhttp6 after save"
#endIf
print "***----HTTP(S) and EndPoint display/settings ENDED(JYTHON)-------------------"
#    ---------- new ports ------------------
```

Executing zWESBConfig with BPXBATCH

You can execute the zWESBConfig phase with BPXBATCH, if the called shell is slightly changed and accepts an additional parameter “runtime”. The JCL invokes the changed shell “zWESBConfigPOK.sh”, as shown in Example A-5.

Example: A-5 BPXBATCH for executing the configuration script

```
//XXXXXXXX JOB (999,POK),'xxxxxxxxxx - 2C08',CLASS=A,
//   MSGCLASS=X,NOTIFY=VANAERS,MSGLEVEL=(1,1),
//   REGION=0M
/*JOBPARM S=SC48
//INST1 EXEC PGM=IKJEFT01,REGION=0M
//STDENV DD *
_CBINSTALL=/usr/lpp/zWESBSO/V6R0
_CBCONFIG=/wassoconfig/socell/sodmnode
_ASDIR=DeploymentManager
//SYSTSPRT DD  SYSOUT=*
//SYSTSIN  DD  *
 BPXBATCH SH  +
   $_CBCONFIG+
   /$_ASDIR+
   /bin/zWESBConfigPOK.sh +
    -response /u/larryc/DmgrDB2.rsp +
    -runtime $_CBCONFIG/$_ASDIR +
    -augment
```

Example A-6 shows the changes for the shell script.

Example: A-6 Changed zWESBConfigPOK.sh

```
#!/bin/sh
if Ý ! $1 ¨; then
   syntax=1
else
  runtimeValue=false
  smprootValue=false
  for arg in $@
  do
     if Ý "$arg" = "-runtime" ¨; then
        runtimeValue="true"
     elif Ý "$runtimeValue" = "true" ¨; then
        TARGET_RUNTIME_DIR="$arg"
        runtimeValue="false"
     fi
  done
fi

.......
if Ý $syntax ¨; then
   echo '----------------------------------------------------------'
.......
   echo '----------------------------------------------------------'
exit 8
fi
echo $PATH
if Ý ! $TARGET_RUNTIME_DIR ¨; then
   syntax=1
   echo "Error: -runtime value not specified."
fi
# run the setupCmdLine.sh in present working directory.
echo issuing cd $TARGET_RUNTIME_DIR/bin
cd  $TARGET_RUNTIME_DIR/bin
echo setupCmdLine.sh
. ./setupCmdLine.sh
```

B

Additional material

This IBM Redbooks publication refers to additional material that you can download from the Internet as described in the following sections.

Locating the Web material

The Web material associated with this IBM Redbooks publication is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG247335`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** option and open the directory that corresponds with the IBM Redbooks form number, SG247335.

Using the Web material

The additional Web material that accompanies this IBM Redbooks publication includes the following files:

File name	*Description*
SG247335.zip	The readme.txt file, contained within the zip file, lists the different structures and source code files used with our sample scenarios, which are described in Chapter 4, “Sample scenarios” on page 195. It also contains the complete project interchange files for the projects containing the developed samples.

How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this IBM Redbooks publication.

IBM Redbooks

For information about ordering these publications, see "How to get IBM Redbooks" on page 330. Note that some of the documents referenced here might be available in softcopy only. IBM Redbooks for further information are:

- *ABCs of System Programming Volume 5*, SG24-5655
- *Architecting High Availability Using WebSphere V6 on z/OS*, SG24-6850
- *Getting Started with WebSphere Enterprise Service Bus V6*, SG24-7212
- *IMS Connectivity in an On Demand Environment: A Practical Guide to IMS Connectivity*, SG24-6794
- *System Programmer's Guide to: Workload Manager*, SG24-6472
- *z/OS Getting Started: WebSphere Process Server and WebSphere Enterprise Service Bus V6*, SG24-7378

Other publications

The following publications are also relevant as further information sources:

- *IBM WebSphere Application Server for z/OS Version 6.1: Using the Administrative Clients*, SA23-2208
- *TCP/IP Tutorial and Technical Overview*, GG24-3376
- *WebSphere Enterprise Service Bus for z/OS V6.0.1 Reference*, SC34-6768
- *WebSphere Message Broker for z/OS V6.0 Program Directory, Program Number 5655-M74*
- *WebSphere MQ for z/OS Concepts and Planning Guide V6.0*, GC34-6582
- *WebSphere MQ for z/OS System Setup Guide V6.0*, SC34-6583
- *WebSphere MQ for z/OS V6.0 Program Directory*, GI10-2584

Online resources

The following Web sites are also relevant as further information sources:

- TechDocs support site:

  http://www.ibm.com/support/techdocs/atsmastr.nsf/Web/Techdocs
- WebSphere Application Server for z/OS information center:

  http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp?topic=/com.ibm.websphere.zseries.doc/info/welcome_nd.html
- WebSphere Business Process Management information center:

  http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp
- WebSphere Enterprise Service Bus for z/OS:

  http://www.ibm.com/software/integration/wsesb/library/infocenter/

How to get IBM Redbooks

You can search for, view, or download IBM Redbooks, IBM Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy IBM Redbooks or CD-ROMs, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

IBM

Redbooks

Implementing an ESB using IBM WebSphere Message Broker V6 and WebSphere ESB V6 on z/OS

(0.5” spine)
0.475”<->0.875”
250 <-> 459 pages

Implementing an ESB using IBM WebSphere Message Broker V6 and WebSphere ESB V6 on z/OS

Configuration and customization

Qualities of service

Deployment scenarios

This IBM Redbooks publication is designed for IT architects and IT specialists that are dealing with IBM WebSphere Message Broker and IBM WebSphere Enterprise Service Bus (ESB) solutions.

This book illustrates how to configure an ESB using either IBM WebSphere Message Broker V6 for IBM z/OS or IBM WebSphere Enterprise Service Bus V6. It makes various traditional z/OS transactions and data available as a Web service through the broker, including IBM DB2 data, batch programs, and IBM CICS/IMS transactions.

In this book, there is more focus on WebSphere Enterprise Service Bus installation and customization than on WebSphere Message Broker because the latter has already been documented in other places.

This book describes multiple scenarios that show how to integrate applications using a mix of MQ and SOAP protocols using both WebSphere Message Broker, also known as WebSphere Message Broker, and WebSphere Enterprise Service Bus.

High availability using clustering of brokers in multiple logical partitions (LPARs) on z/OS is also addressed in this book, with details on which feature you can use and configure to improve continuous operation, such as shared queues, shared ports, and Sysplex Distributor.

SG24-7335-00 ISBN 0738486086