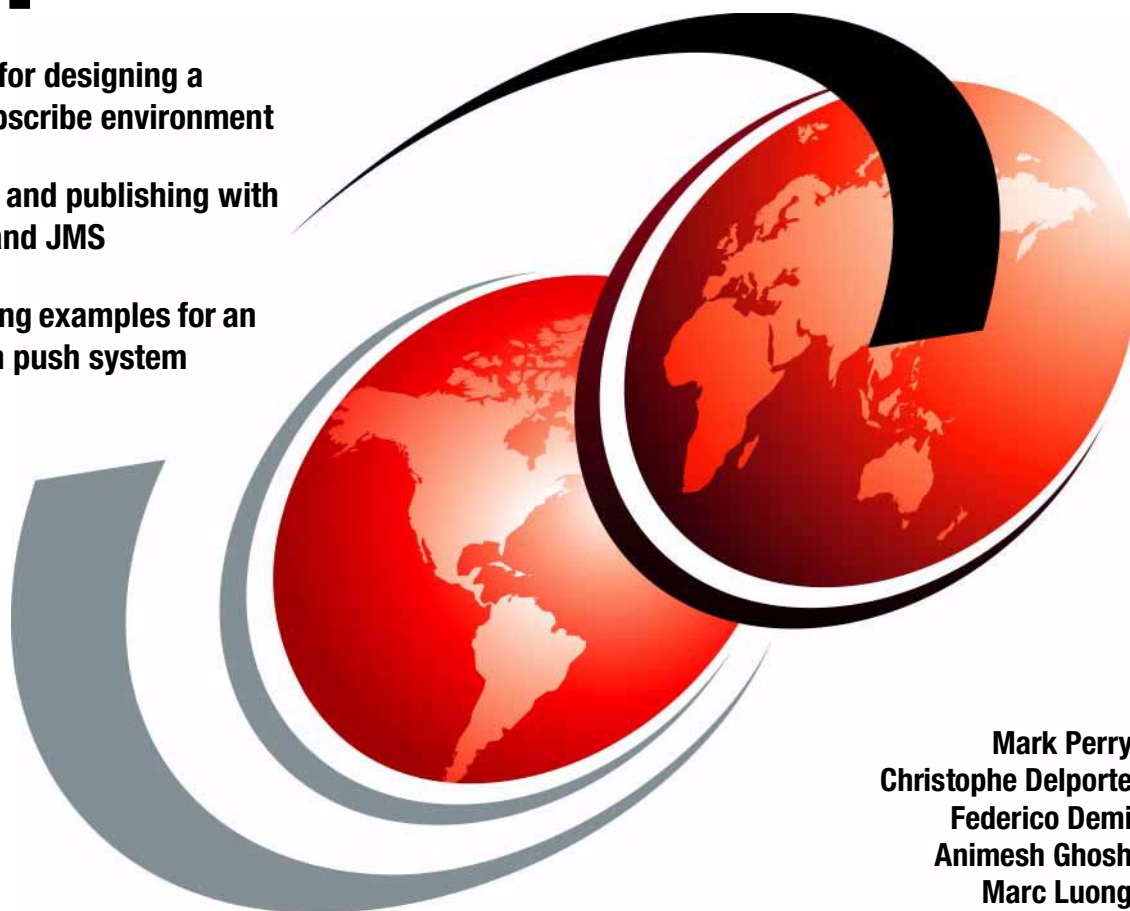


# MQSeries Publish/Subscribe Applications

Guidelines for designing a  
publish/subscribe environment

Developing and publishing with  
MQI, AMI, and JMS

Programming examples for an  
information push system



Mark Perry  
Christophe Delporte  
Federico Demi  
Animesh Ghosh  
Marc Luong





International Technical Support Organization

**MQSeries Publish/Subscribe Applications**

September 2001

**Take Note!** Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 219.

### **First Edition (September 2001)**

This edition applies to:

- ▶ MQSeries for Windows NT Version 5.2
- ▶ MQSeries Integrator for Windows NT Version 2.01

Comments may be addressed to:

IBM Corporation, International Technical Support Organization  
Dept. MP135  
IBM United Kingdom Ltd  
Hursley  
Hampshire SO21 2JN  
United Kingdom

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2001. All rights reserved.

Note to U.S Government Users – Documentation related to restricted rights – Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Preface</b> .....	ix
The team that wrote this redbook .....	ix
Special notice .....	xi
IBM trademarks .....	xii
Comments welcome .....	xii
<b>Chapter 1. Introduction to publish/subscribe</b> .....	1
1.1 What is publish/subscribe? .....	2
1.2 MQ products .....	4
1.2.1 MQSeries .....	4
1.2.2 MQSeries Integrator .....	5
1.3 Features of MQ Publish/Subscribe systems .....	6
1.3.1 Retained publications .....	6
1.3.2 Message persistence .....	6
1.3.3 Topic-based or content-based subscriptions .....	7
1.3.4 Temporary subscriptions .....	7
1.3.5 Expiration .....	7
1.4 Languages and interfaces .....	7
1.4.1 AMI .....	8
1.4.2 JMS .....	8
1.4.3 MQI .....	8
1.5 Broker networks .....	8
1.5.1 MQSeries Publish/Subscribe broker networks .....	9
1.5.2 MQSeries Integrator and mixed broker networks .....	9
<b>Chapter 2. Technical overview</b> .....	11
2.1 Queues and message headers .....	12
2.1.1 Queues .....	12
2.1.2 Message formats .....	16
<b>Chapter 3. Example application</b> .....	19
3.1 The business case .....	20
3.2 Application solution .....	20
3.2.1 Simulated public transport system .....	20
3.3 Publish/subscribe scenario 1 .....	22
3.4 Publish/subscribe scenario 2 .....	24
3.5 Publish/subscribe scenario 3 .....	24
3.6 Publish/subscribe scenario 4 .....	25
3.7 Publish/subscribe scenario 5 .....	25

<b>Chapter 4. The publish/subscribe application</b> .....	27
4.1 Software components .....	28
4.2 Environment setup .....	29
4.2.1 MQSeries Publish/Subscribe installation .....	29
4.2.2 JMS installation .....	32
4.2.3 JMS overview .....	36
4.2.4 JMS configuration, JNDI and JMSAdmin .....	39
4.2.5 Defining MQSeries required for the application .....	52
4.2.6 AMI overview .....	52
4.2.7 AMI installation .....	56
4.2.8 AMI configuration .....	60
4.3 PubLauncher .....	73
4.3.1 The properties file - pub.properties .....	74
4.3.2 PubLauncher coding logic .....	75
4.3.3 Starting the publication application .....	75
4.4 PubThread .....	76
4.4.1 PubThread coding logic .....	78
4.5 The publication messages .....	79
4.6 Publishing in C .....	80
4.6.1 Vehicle C AMI program .....	81
4.6.2 Vehicle C MQI program .....	84
4.7 Publishing in Java .....	88
4.7.1 Publishing in JMS .....	89
4.7.2 Publishing in Java AMI .....	91
4.8 Subscription .....	93
4.8.1 Setup of the environment .....	93
4.8.2 XMLParser setup .....	94
4.8.3 VAJava setup .....	94
4.9 AMI administration setup .....	94
4.10 Sample subscriber application .....	96
4.10.1 Control Program .....	97
4.10.2 XML parser program .....	102
4.10.3 GUI program .....	102
4.10.4 Parsing JMS-based published message .....	102
4.11 Comments and extensions .....	105
4.11.1 Retained publications .....	105
4.11.2 Streams .....	108
4.11.3 Broker networks .....	110
<b>Chapter 5. Migration to MQSeries Integrator</b> .....	115
5.1 Step-by-step guide .....	116
5.1.1 Step 1 - Creation of a publication queue .....	116
5.1.2 Step 2 - Creation of a simple publish message flow .....	116

5.1.3	Step 3 - Deployment to the target broker . . . . .	119
5.1.4	Step 4 - Executing example applications on MQSeries Integrator . . . . .	119
5.1.5	Step 5 - Trace analysis . . . . .	120
5.2	Comments and extensions . . . . .	120
5.2.1	Streams handling in MQSeries Integrator . . . . .	121
5.2.2	Subscription points . . . . .	122
5.2.3	MQSeries Integrator broker networks and collectives . . . . .	123
5.2.4	Topic-based security . . . . .	125
5.2.5	Example - migration of applications using streams . . . . .	130
5.2.6	Example - message translation using subscription points . . . . .	133
5.2.7	Example - MQSeries Integrator broker networks . . . . .	137
5.2.8	Example - confidential publish/subscribe environment . . . . .	138
5.3	Other forms of interoperability . . . . .	144
5.3.1	Mixed broker networks . . . . .	145
5.3.2	Migrating an MQSeries broker to MQSeries Integrator . . . . .	146
5.3.3	Example - mixed broker networks . . . . .	146
	<b>Chapter 6. Web enablement . . . . .</b>	<b>149</b>
6.1	A simple Web-based subscriber . . . . .	150
6.1.1	WebSphere Application Server configuration . . . . .	151
6.1.2	Servlet configuration . . . . .	155
6.1.3	AMI repository configuration . . . . .	158
6.1.4	Program invocation . . . . .	159
6.1.5	Discussion about the Web part of the application . . . . .	160
6.2	Comments and extensions . . . . .	163
	<b>Chapter 7. Advanced Web enablement . . . . .</b>	<b>165</b>
7.1	Concept . . . . .	166
7.1.1	A new middle tier component . . . . .	166
7.1.2	Architectural considerations . . . . .	167
7.2	Forecast application . . . . .	169
7.2.1	ForecastThread: single-threaded behavior . . . . .	170
7.2.2	ForecastThread: multithreading behavior . . . . .	171
7.2.3	The forecast message . . . . .	173
7.3	JMS Web subscriber application . . . . .	174
7.3.1	Servlet configuration . . . . .	175
7.3.2	Program invocation . . . . .	175
7.4	Program flow of the application . . . . .	178
7.4.1	Using MQSeries Integrator to tweak publication content . . . . .	180
7.4.2	Comments and extensions . . . . .	181
7.5	AMI Web application and message filtering . . . . .	182
7.5.1	Content-based subscriptions . . . . .	183
7.5.2	Content-based Web subscriber application . . . . .	183

7.5.3 Servlet configuration . . . . .	184
7.5.4 AMI repository configuration . . . . .	185
7.5.5 Program invocation . . . . .	185
7.5.6 Discussion of the application. . . . .	188
7.5.7 Subscribe on request . . . . .	188
7.6 Example - a three-tier implementation . . . . .	190
7.7 Final content-based subscriptions considerations. . . . .	191
7.7.1 Applicability . . . . .	191
7.7.2 Content-based subscription simulation . . . . .	192
7.7.3 Performance implications . . . . .	192
<b>Chapter 8. Conclusions . . . . .</b>	<b>193</b>
8.1 The technology. . . . .	194
8.1.1 Web-based applications . . . . .	194
8.1.2 Pervasive applications . . . . .	194
8.1.3 Enterprise Application Integration . . . . .	194
8.2 IBM offerings . . . . .	195
8.2.1 MQSeries Publish/Subscribe . . . . .	195
8.2.2 MQSeries Integrator . . . . .	195
8.2.3 More Information. . . . .	195
<b>Appendix A. Hardware and software environment. . . . .</b>	<b>197</b>
Hardware . . . . .	198
Software . . . . .	198
<b>Appendix B. MQSeries Publish/Subscribe administration commands .</b>	<b>199</b>
strmqbrk . . . . .	200
dspmqrk . . . . .	200
endmqbrk . . . . .	200
<b>Appendix C. MQSeries Integrator administration commands . . . . .</b>	<b>201</b>
MQSeries Integrator pub/sub admin commands . . . . .	202
Admin commands for mixed brokers . . . . .	202
mqsilistmqpubsub . . . . .	203
mqsijoinmqpubsub . . . . .	203
mqsiclearmqpubsub . . . . .	203
<b>Appendix D. The GUI-based subscriber application . . . . .</b>	<b>205</b>
AMI configuration . . . . .	206
Subscriber configuration . . . . .	206
Simple Web subscriber application . . . . .	206
AMI configuration . . . . .	206
Servlet configuration . . . . .	206
Additional WebSphere Application Server configuration. . . . .	207



Web subscriber Forecast application . . . . .	207
JMS configuration . . . . .	207
MQSeries Integrator configuration . . . . .	208
Servlet configuration . . . . .	208
Additional WebSphere Application Server configuration . . . . .	208
Advanced Web subscriber Forecast application . . . . .	208
AMI configuration . . . . .	208
Servlet configuration . . . . .	209
Additional WebSphere Application Server configuration . . . . .	209
<b>Appendix E. Additional material . . . . .</b>	<b>211</b>
Locating the Web material . . . . .	212
Using the Web material . . . . .	212
System requirements for downloading the Web material . . . . .	212
How to use the Web material . . . . .	212
<b>Related publications . . . . .</b>	<b>217</b>
IBM Redbooks . . . . .	217
Other resources . . . . .	217
Referenced Web sites . . . . .	218
How to get IBM Redbooks . . . . .	218
IBM Redbooks collections . . . . .	218
<b>Special notices . . . . .</b>	<b>219</b>
<b>Abbreviations and acronyms . . . . .</b>	<b>221</b>
<b>Index . . . . .</b>	<b>223</b>



# Preface

Publish and subscribe is an effective way of disseminating information to multiple users. This could take the form of a book publisher wanting to tell potential retail outlets what titles are in his catalog, a financial broker wanting to post stock prices to anyone in the world who may wish to invest with a particular stock, or any information provider who wants an efficient and effective process to get information to the people who need it. In these analogies, publish/subscribe applications can help to enormously simplify the task of getting business messages and transactions to a wide, dynamic, and potentially large audience in a timely manner.

These problems face many businesses and using publish/subscribe technology can provide a very manageable and extendable solution.

This redbook positions the MQSeries Publish/Subscribe to MQSeries Integrator Publish/Subscribe.

This redbook will help you create, tailor and configure an application from publishing data through to subscribing via Web pages.

This redbook gives a broad understanding of building and running an entire publish/subscribe solution.

This redbook will give you a quick start to designing and creating a solution and then migrating it from MQSeries Publish/Subscribe to MQSeries Integrator Publish/Subscribe.

The operating system used for everything described within this book is Windows 2000 Professional Edition. However, all the MQSeries applications we discuss can also run on many other platforms, and this is mentioned in many places throughout the book. Complete information about the platforms supported by the products described can be obtained from:

<http://www-4.ibm.com/software/ts/mqseries/>

## The team that wrote this redbook

This redbook was produced by a team of specialists from around the world working at the International Technical Support Organization's Hursley Center in the UK.



*The writers of this redbook - Mark, Marc, Christophe, Federico and Animesh*

**Mark Perry** is an IT Specialist and Project Leader at the International Technical Support Organization, Hursley Center. He joined IBM in 1977 and has 10 years of experience within the MQ community at Hursley, mostly as a team leader in System Test, as well as spending some time working for MQ Services in the US.

**Christophe Delporte** is an IT Specialist working for Global Services in IBM Belgium/Luxembourg. He has four years of experience with the WebSphere and DB2 family products and on the iSeries platform. He holds a degree in Civil Engineering from the Polytechnic Faculty of Mons, Belgium. He has previously participated in the production of Redbooks in connection with DB2, XML and Java.

**Federico Demi** is an IT manager working for Primeur, an IBM business partner in Italy. He has eight years of experience in middleware and the Enterprise Application Integration fields. He holds a degree in Computer Science from the University of Pisa. His areas of expertise includes the development and management of MQ and related products on large projects.

**Animesh Ghosh** is a Senior Software Engineer working for IBM Global Services in Bangalore, India. He has almost four years of experience in e-Business Enterprise Solutions. He holds a degree in Computer Applications from the College of Engineering and Technology in Bhubaneswar. He has worked at IBM for two years. His areas of expertise include Java, WebSphere family products and Database Products.

**Marc Luong** is a certified IBM IT Expert working for Software Group in IBM France. He has 25 years of experience in database design and remote server communications. Marc joined IBM in 1986. Since 1994, he has worked very closely with the IBM Hursley laboratory on MQSeries multi-platform products to implement message queueing solutions in customer sites. He holds degrees in Computer Science and Electronic Engineering from Ecole Superieure d'Informatique in Paris and the University of Paris. He has participated in several Redbooks concerning DB2 and MQSeries products.

Thanks also to the following people for their help and contributions to this project:

Alasdair Paton, Graham Winn, Kathryn McMullan, Peter Niblett, Tim Dunn, Brian Homewood, and Rachel Norris.  
MQSeries, IBM Hursley


Neil Kolban  
Technical Support, IBM Dallas

## Special notice

This publication is intended to help application developers to build and understand a complete publish/subscribe solution. The information in this publication is not intended as the specification of any programming interfaces that are provided by MQSeries, MQSeries Integrator, WebSphere Application Server or any other products mentioned in this book. See the PUBLICATIONS section of the IBM Programming Announcement for the WebSphere family for more information about what publications are considered to be product documentation.

## IBM trademarks

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

e (logo)® 

IBM ®

AIX

AS/400

CICS

DB2

DB2 Universal Database

Everyplace

IMS

IMS/ESA

iSeries

MQSeries

OS/390

Redbooks

Redbooks Logo 

RETAIN

S/390

SecureWay

SP

SupportPac

VisualAge

WebSphere

400

Lotus

Domino

## 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 Redbooks in one of the following ways:

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

[ibm.com/redbooks](http://ibm.com/redbooks)

- ▶ Send your comments in an Internet note to:

[redbook@us.ibm.com](mailto:redbook@us.ibm.com)

- ▶ Mail your comments to the address on page ii.



# **Introduction to publish/subscribe**

This chapter introduces publish/subscribe, what it is and how it can be used, along with some of the choices available to users when developing and implementing publish/subscribe applications.

## 1.1 What is publish/subscribe?

Publish/subscribe applications are intended for situations where a single message is required by, and should be distributed to, multiple users. Their big advantage over other delivery methods is that they keep the publisher separated from the subscriber. This means that the publisher in a publish/subscribe application doesn't need to have any knowledge of either the subscriber's existence or the applications that may use the published information. Likewise, the subscriber or subscriber applications don't need to know anything about the publisher application.

Put as simply as possible, a publish/subscribe application has one or more *publishers* who publish messages from an application to a *broker*, and a group of *subscribers* who subscribe to some or all of those published messages that are held on the broker. The system matches the publications to the subscribers and ensures that all the messages are made available and delivered to all the subscribers in a timely manner.

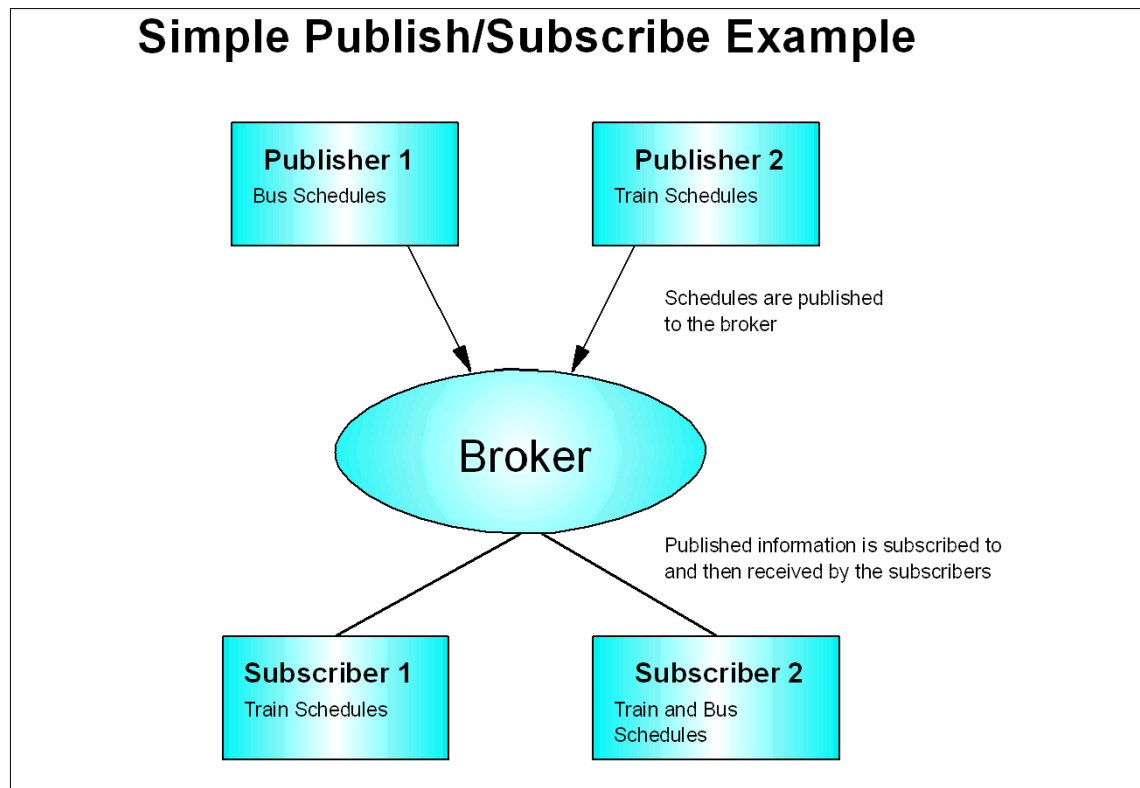


Figure 1-1 A simple pub/sub environment



Figure 1-1 shows Publisher1 and Publisher2 publishing messages concerned with schedule information onto the broker. Subscribers can choose to subscribe or unsubscribe to that information available on the broker as necessary.

Multiple brokers can simply be connected together, enabling brokers to exchange messages. This allows subscribers to one of the brokers to pick up messages that have been published to another broker, further freeing the subscriber from the constraints of using the same broker as the publisher.

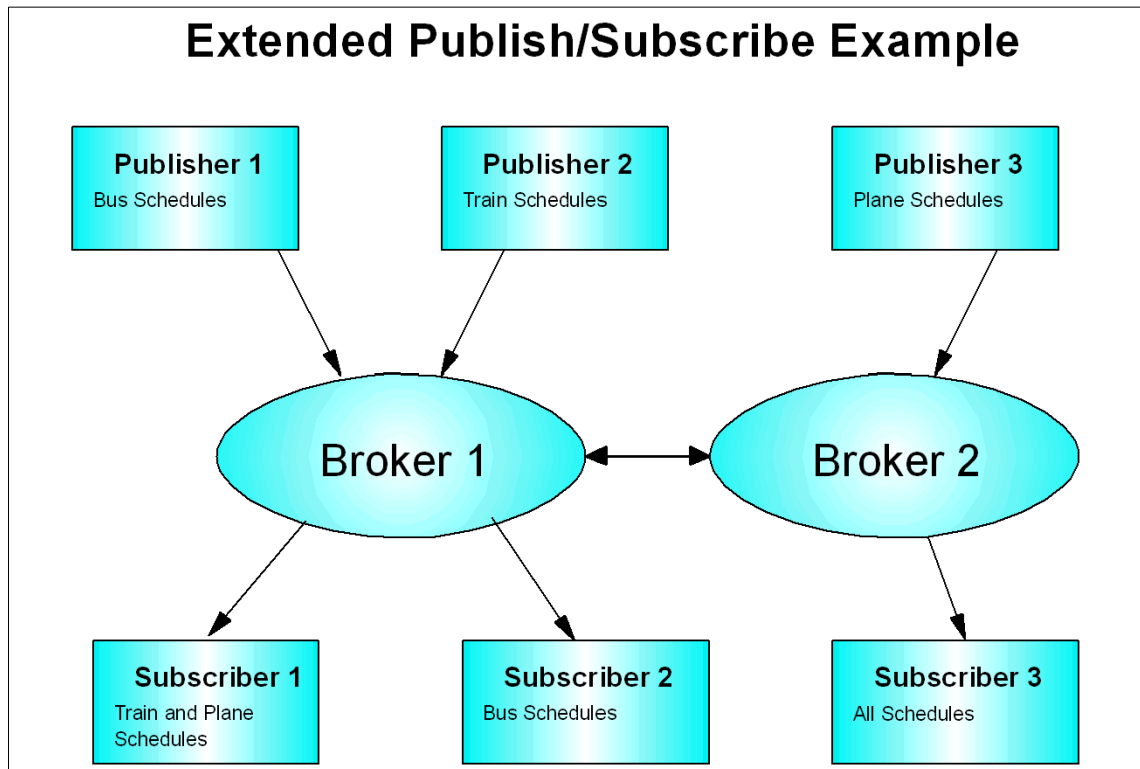


Figure 1-2 Extended pub/sub example

The subscribers are able to choose between looking at all the messages published, or just some of the messages based upon various criteria that are important to them. This is known as either content-based or topic-based subscription and is described in more detail later in this book. Publish/subscribe applications are widely used as a means of easily disseminating information to multiple users who may be interested in some or all of the information available. Additional subscribers can choose to either subscribe or unsubscribe as time

goes by, and all the subscribers are completely independent of each other. It's also worth noting that when a subscriber receives information it can then go on to publish that material itself, possibly in a modified form, either back to the broker it got it from or to another broker.

In traditional systems using point-to-point connections attempting to provide this kind of service, the matrix of connections can grow explosively, becoming very difficult to control and maintain. The root problem is that the messaging application always needs to know something about where the message is going, such as a queue manager name or a queue name. When high numbers of destinations are possible, the network view can become extremely complex.

A big advantage of a publish/subscribe system is that it can remove this unmanageable aspect of a point-to-point network and replace it with a very simple network of a publisher, a broker, and all the subscribers. New subscription clients or services can simply be added without any impact or interruption in the service to other users. This provides a superior means of streamlined and efficient integration and growth across an enterprise, and of course, since MQSeries is used as the backbone for message delivery, all the benefits and features of MQSeries are inherited.

## 1.2 MQ products

The MQSeries family consists of complementary publish/subscribe offerings:

- ▶ MQSeries
- ▶ MQSeries Integrator

### 1.2.1 MQSeries

MQSeries provides assured, once-only delivery of messages between your IT systems. It connects more than 30 industry platforms including those from IBM, Microsoft, Sun, and HP using a variety of communications protocols.

MQSeries provides rich support for applications:

- ▶ Application programming interfaces: the Message Queue Interface (MQI) and Application Messaging Interface (AMI) are supported in several programming languages.
- ▶ Communication models: point-to-point (including request/reply and client/server) and publish/subscribe are supported.
- ▶ The complexities of communications programming are handled by the messaging services and are therefore removed from the application logic.

- ▶ Applications can access other systems and interfaces through gateways and adapters to products such as Lotus Domino, Microsoft Exchange/Outlook, SAP/R3, and IBM's CICS and IMS/ESA products.

MQSeries Publish/Subscribe is available as SupportPac MA0C and can be downloaded from:

<http://www-4.ibm.com/software/ts/mqseries/txppacs/txpm1.html>

With MQSeries you can have one broker for each queue manager and the broker takes the same name as that queue manager, but the broker does not have to reside on the same machine as any of the publishers or subscribers. It can be on any machine or supported platform in your network, provided there is a route between the publisher's or subscriber's queue manager and the broker.

## 1.2.2 MQSeries Integrator

MQSeries Integrator Version 2.0 extends the capabilities of MQSeries Publish/Subscribe by supporting:

- ▶ Enhanced publish/subscribe function through exploitation of structured topic names, access control, content-based subscriptions, and subscription points.
- ▶ Enhancement of message processing through the addition of new message processing nodes to complement or replace the supplied nodes.
- ▶ Interfaces that allow messages to be enriched with information from a database, or to be stored in a database.

You can upgrade your applications, messages, and brokers to take advantage of the enhancements in MQSeries Integrator Version 2.0. You can also continue to use your existing MQSeries Publish/Subscribe applications and messages unchanged, by tailoring your Version 2.0 system to provide compatible support.

MQSeries Integrator works with MQSeries messaging, extending its basic connectivity and transport capabilities to provide a powerful message broker solution driven by business rules. Messages are formed, routed, and transformed according to the rules defined by an easy-to-use graphical user interface (GUI).

Diverse applications can exchange information in unlike forms, with brokers handling the processing required for the information to arrive in the right place in the correct format, according to the rules you have defined. The applications have no need to know anything other than their own conventions and requirements.

Applications also have much greater flexibility in selecting which messages they wish to receive, because they can specify a topic filter, or a content-based filter, or both, to control the messages made available to them.

MQSeries Integrator provides a framework that supports supplied, basic functions along with plug-in enhancements, to enable rapid construction and modification of business processing rules that are applied to messages in the system.

## 1.3 Features of MQ Publish/Subscribe systems

Here are some of the key features that can be used with MQSeries Publish/Subscribe applications.

### 1.3.1 Retained publications

By default, a broker discards a publication when it has been sent to all interested subscribers. However, a publisher can specify that it wants the broker to keep a copy of a publication, which is then called a *retained publication*. The copy can be sent by the broker to subsequent subscribers who register an interest in the topic. This means that new subscribers don't have to wait for information to be published again before they receive it.

For example, a subscriber registering a subscription to a stock price would receive the current price immediately, without waiting for the stock price to change (and hence be republished). The broker retains only one publication for each topic and subscription point, so the old publication is deleted when a new one arrives.

### 1.3.2 Message persistence

Persistent messages in MQ are logged by the queue manager and are preserved across application and queue manager restarts. Non-persistent messages may be lost if the queue manager fails, but are only delivered once at the most, whereas persistent messages are guaranteed to be delivered exactly once.

We recommend that you send all subscription registration messages as persistent messages. All subscriptions are maintained persistently by the broker and are preserved across broker restarts and system failures. Brokers maintain the persistence of publications as set by the publisher, unless changed by options specified when the subscription is registered. These options are non-persistent, persistent, persistence as queue, or persistence as publisher (the default).

When speed of delivery is the most important consideration, then you can choose to publish using non-persistent messages. This works especially well with messages that have a short life span and will rapidly be superseded, or where assured delivery is not of primary importance.

To see the maximum performance gain, you may also configure your broker to process messages outside of syncpoint. The method for doing this varies for MQSeries and MQSeries Integrator and is described in the specific product's manuals.

### **1.3.3 Topic-based or content-based subscriptions**

When you create a subscription, you can choose to subscribe to a specific topic, such as trains or buses in our example application of a transport system. But you could also subscribe to specific content, which for example might be trains that will arrive after 7pm. The difference here is that the content-based subscription is filtering out information from the topic, giving the user a much more refined result. This filtering takes place on the broker. Content-based subscription is supported only in MQSeries Integrator.

### **1.3.4 Temporary subscriptions**

It is possible that a user can create a temporary subscription by specifying a temporary dynamic queue as the subscriber queue. When the subscribing application ends, the queue is removed and the subscription therefore removed.

### **1.3.5 Expiration**

Messages can be created with an expiration date and time. This means that once that assigned time has arrived, the data is no longer available to subscribers. This is particularly relevant to messages with data pertaining to a moment in time, for example an offer of a loan rate where the rate is valid only for a period of time, or a customer enquiry where a reply is needed within a set period of time or not at all.

## **1.4 Languages and interfaces**

These are the programming choices available to you that we will cover in this book.

## 1.4.1 AMI

The Application Messaging Interface (AMI) API is one of the latest additions to the MQSeries API portfolio. AMI has very interesting features and a brand new philosophy.

AMI provides a higher level, easy-to-use interface to messaging, reducing to a minimum the number and complexity of data structures and parameters.

AMI provides verbs for specific messaging styles (send and forget, request/reply, publish/subscribe) and moves middleware-specific options out of the application and into the administration domain by using an external persistent repository, which can be easily maintained.

Programs written to the AMI interoperate seamlessly with any other MQSeries messaging API flavor (for example, Java JMS, and RPG MQI).

## 1.4.2 JMS

Java Message Service (JMS) is one of the API specifications included in the Java 2 Enterprise Edition (J2EE) specification defined by Sun. JMS provides a framework that enables the development of portable, message-based applications in the Java programming language. It defines a common set of messaging concepts that must be supported by all JMS technology-compliant messaging systems.

MQSeries complies to the JMS specification and support is provided through the MA88 SupportPac.

## 1.4.3 MQI

The Message Queue Interface is the full set of API calls that are available across the entire MQSeries family.

## 1.5 Broker networks

Geographically distributed systems or requirements for heavy workloads can be accommodated by a network of brokers having two-way MQSeries connectivity among them.

## 1.5.1 MQSeries Publish/Subscribe broker networks

MQSeries Publish/Subscribe brokers can be linked to form a hierarchy, and publications and subscriptions can flow from any broker to any other broker in the hierarchy. However an administrator can limit the scope of a publication to a subsection of the hierarchy.

Publications and subscriptions are passed from one broker to another via the hierarchy structure even if some direct MQSeries connection exists.

## 1.5.2 MQSeries Integrator and mixed broker networks

As with MQSeries Publish/Subscribe, MQSeries Integrator brokers can be connected to form networks. On top of the hierarchy topology, a new topology type is supported, called a *collective*.

A collective is a set of brokers in which the queue managers are totally connected. (that is, every queue manager is directly connected to every other queue manager). Communication between brokers is optimized within this type of topology and collectives may be connected to other stand-alone brokers or collectives in a large MQSeries Integrator topology.

A broker network can also contain both MQSeries Integrator brokers and MQSeries Publish/Subscribe brokers, without any constraints on the parent-child relationship between brokers of different types.







## Technical overview

This chapter gives an overview of the functional and programming aspects of publish/subscribe applications, and describes some of the aspects that should be understood. It will then cover the choices that can be made before beginning to write an application.

## 2.1 Queues and message headers

A publish/subscribe environment is made up of three components: the publishers, the message broker and the subscribers.

IBM provides two types of message brokers: the MQSeries Publish/Subscribe broker and the MQSeries Integrator broker.

There are also three different types of API available for writing publish/subscribe applications: MQI, AMI and JMS.

Depending on these classifications, the messages containing the commands exchanged between the three components have different types of headers and are sent to different queues.

### 2.1.1 Queues

Figure 2-1 shows a typical flow of commands between the three publish/subscribe components and the destination queues.

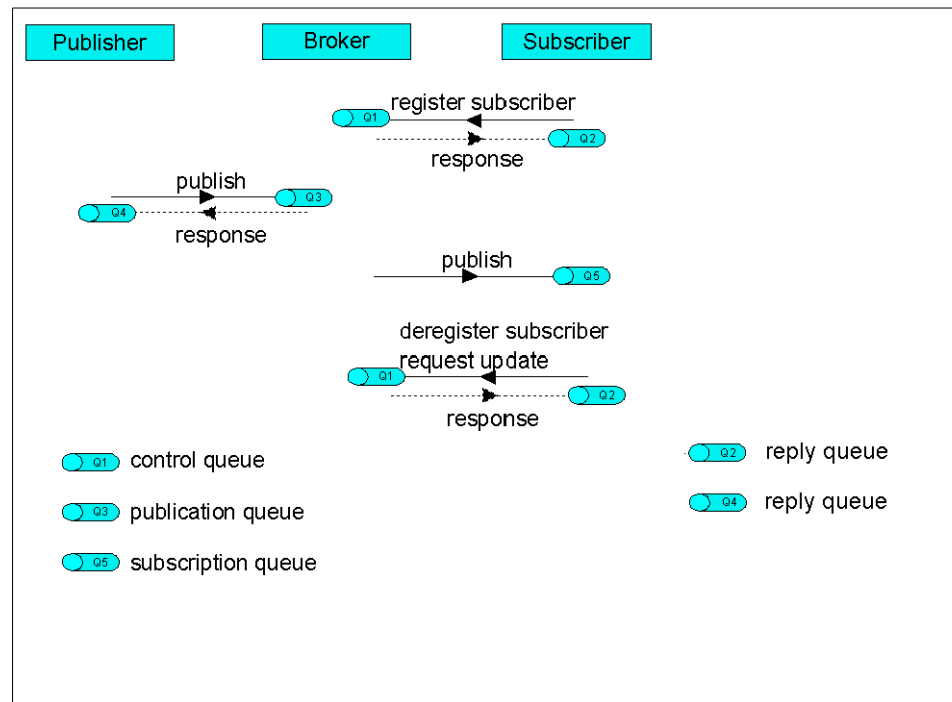


Figure 2-1 Command flow and destination queues

The commands exchanged between the publisher, the broker and the subscriber components are messages sent to queues. Table 2-1 names the queues represented in Figure 2-1 on page 12.

Table 2-1 Command queues

Queues	Pub/Sub MQSeries	Pub/Sub MQSI	JMS
Q1, the control queue	SYSTEM.BROKER.CONTROL.QUEUE	SYSTEM.BROKER.CONTROL.QUEUE	SYSTEM.BROKER.CONTROL.QUEUE
Q2, the reply queue	ReplyToQueue in the MQMD of the command message	ReplyToQueue in the MQMD of the command message	ReplyToQueue in the MQMD of the command message
Q3, the publication queue	SYSTEM.BROKER.DEFAULT.STREAM (Default)	Any Queue mentioned in the input node	SYSTEM.BROKER.DEFAULT.STREAM (Default)
Q4, the reply queue	ReplyToQueue in the MQMD of the command message	ReplyToQueue in the MQMD of the command message	ReplyToQueue in the MQMD of the command message
Q5, the subscription queue	Queue mentioned when subscribing	Queue mentioned when subscribing	SYSTEM.JMS.ND.SUBSCRIBER.QUEUE or SYSTEM.JMS.D.SUBSCRIBER.QUEUE

### The control queue

The broker control queue SYSTEM.BROKER.CONTROL.QUEUE and the other queues SYSTEM.BROKER.\* are created the first time the broker is started for MQSeries Publish/Subscribe, or on broker creation for MQSeries Integrator. The control queue is used to send all the commands to the broker apart from the publication and delete publication commands.

- ▶ in JMS, the control queue is specified in the BROKERCONQ parameter used when creating the TopicConnectionFactory. It defaults to SYSTEM.BROKER.CONTROL.QUEUE and should not be changed.
- ▶ in AMI, the user doesn't specify the control queue. AMI knows all messages and put publications must be sent to the queue SYSTEM.BROKER.CONTROL.QUEUE.
- ▶ in MQI, the control queue must be explicitly passed to the MQOPEN or MQPUT1 verb.

### The publication queue

The publication commands are not sent to the control queues as all other publish/subscribe commands. They are sent to one or more publication queues.

The queues used depend on the type of broker used.

When using the publish/subscribe functionality of MQSeries, the queues used for publications are stream queues. There is one default stream queue called `SYSTEM.BROKER.DEFAULT.STREAM`. By default, this queue is used for all publications. An administrator can create other stream queues. The first time a new stream queue is used, a **register publisher** command must be issued to make the broker aware of its new configuration or it will ignore this stream queue until a subscriber subscribes specifying this stream queue. Stream queues can be used to spread messages across more than one queue when a lot of messages are published in a short time. See 4.11.2, “Streams” on page 108.

When using the publish/subscribe functionality of MQSeries Integrator, the queues used for publications are the queue or queues mentioned in the input node or nodes of the message flow containing the publication node.

Depending on the programming method used, the publication queues are defined:

- ▶ In JMS: when creating the `TopicConnectionFactory` in JNDI (Java Naming and Directory Interface), the publication queue is indicated in the parameter `BROKERPUBQ` whose value defaults to `SYSTEM.BROKER.DEFAULT.STREAM`.
- ▶ In AMI: the Queue Name parameter of the service point used for the Publisher in the AMI repository is the publication queue.
- ▶ In MQI: you must mention the queue name with the `MQPUT` verb.

## The subscription queue

The subscription queue is the destination to which the broker sends messages matching a particular subscription. When a subscriber registers to a topic, it indicates to the broker to which queue it wants publications to be forwarded. One queue may be used by several registrations. This queue is specified in one of the parameters of the **register subscriber** command. Depending on the APIs used, this parameter is specified:

- ▶ In JMS: two approaches exist, the shared queue approach and the multiple queue approach.
  - The shared queue approach:
    - For the non-durable subscriptions, all subscribers share the queue declared with the parameter `BROKERSUBQ` of the `TopicConnectionFactory`. Its default value is `SYSTEM.JMS.ND.SUBSCRIPTION.QUEUE`. Different `TopicConnectionFactory` objects must be defined to have different subscriptions queues for the subscribers.

- For the durable subscriptions, the queue name is specified when defining the topic in JNDI. Its default value is `SYSTEM.JMS.D.SUBSCRIPTION.QUEUE`. All durable subscribers to the same topic share the same queue.
- The multiple queue approach:
  - For the non-durable subscriptions, a temporary dynamic queue is created for each subscriber based on the model queue `SYSTEM.JMS.MODEL.QUEUE`. The name of the temporary queue starts with the name of the `BROKERSUBQ` parameter of the `TopicConnectionFactory`. To indicate that the multiple queue approach is used, this parameter must end with the `*` character.
  - For the durable subscriptions, it works the same way as for the non-durable subscriptions, but the queue name prefix is mentioned here when defining the topic in JNDI.
- ▶ In AMI: the queue name is indicated in the `Queue Name` parameter of the service point used for the `Receiver Service` by the subscriber in the AMI repository. To create a temporary queue for each subscriber, the queue specified can be a model queue.
- ▶ In MQI: the subscriber's queue name is specified as a parameter when building the `register subscriber` command. If the parameter is not mentioned, the `ReplyToQ` field of the `register subscriber` command message is used.

## The replies

The response messages are optional and are only produced if the incoming command asked for one by setting the `MsgType` or `Report` fields in the incoming MQ message descriptor.

But normally, only the MQI programmers are aware of the message fields in the MQMD.

When using AMI, the methods are overloaded so that you can mention a receiver service point when wanting a response to a command sent to the broker. For example, you can use the `publish` method of a Java AMI `AmPublisher`:

- ▶ By specifying a receiver to get a response:
 

```
amPublisher.publish(amMessagePub, amReceiver, amPolicy);
```
- ▶ Or not:
 

```
amPublisher.publish(amMessagePub, amPolicy);
```

When coding with the MQ implementation of JMS, we can't decide if responses are needed or not. Report messages are used and consumed under the cover by JMS for both a successful and an unsuccessful completion of the command. The MQ queue used for these reports is SYSTEM.JMS.REPORT.QUEUE.

## 2.1.2 Message formats

Following the message descriptor (MQMD) is another header. Its type depends upon the type of broker being used:

- ▶ MQSeries Publish/Subscribe supports the MQ Rules and Format Header Version 1 (MQRFH1).
- ▶ MQSeries Integrator Publish/Subscribe supports both MQ Rules and Format Header Version 1 and 2 (MQRFH2 and MQRFH1). The MQRFH2 header is recommended because it gives the possibility of using the new features available with MQSeries Integrator, such as content filtering.

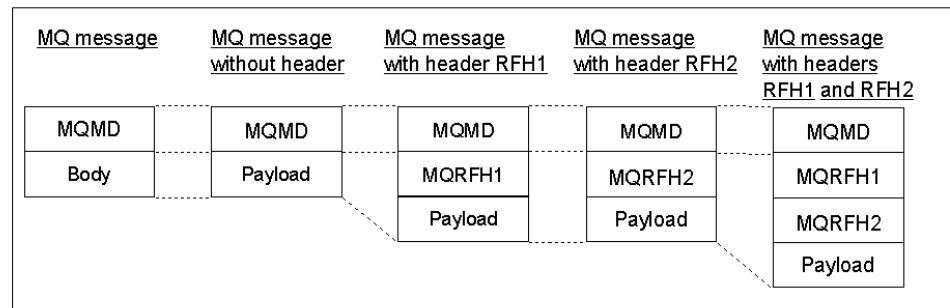


Figure 2-2 Messages headers encountered with publish/subscribe

Figure 2-2 summarizes the various types of messages headers. An MQSeries message is made up of two parts: the message descriptor (MQMD) and the message data. The message data can optionally contain one or more headers before the real payload data. The message data can also start with the MQ defined headers MQRFH1 and MQRFH2, either isolated or both.

We have a triple choice of interfaces with which to write publish/subscribe applications: MQI APIs, AMI APIs, and JMS APIs. This choice impacts the way the headers are built and the user needs to be aware of the differences.

- ▶ **MQI:** this is the only method where the application developer needs to be fully aware of these headers, since the message must be built with the correct header and then be put to the correct control or publication queue. Typically, the program is written in C MQI although other programming languages such as Java can also be used. The user has the freedom to build the header he or she wants and specifically a MQRFH1 or MQRFH2 header. Given that the

MQRFH2 headers can be built with MQI, all the possibilities of MQSeries Integrator can be used with this API.

- ▶ **AMI:** the headers are built automatically by the AMI APIs. The user can choose between both types of headers by specifying the `Service Type` parameter in the properties of the service point definition in the AMI repository. More information about AMI can be found in 4.2.6, “AMI overview” on page 52 and also in the *Application Messaging Interface*, SC34-5604. It is possible to make use of the advanced features provided with MQSI without having to manipulate the MQRFH2 header directly.
- ▶ **JMS:** by default, JMS generates a MQRFH1 header and also embeds a MQRFH2 header before the payload. Although MQRFH2 headers are created by a JMS application, it is not possible to use such MQSeries Integrator advantages as content filtering because these possibilities are not included in the JMS standard. The MQRFH2 header is used here by JMS for compliance with the JMS Message Interface, which specifies that JMS messages are composed of three parts:
  - Header - All JMS messages support the same set of header fields. Header fields contain values used by both clients and providers to identify and route messages.
  - Properties - Each message contains a built-in facility for supporting application-defined property values. Properties provide an efficient mechanism for supporting application-defined message filtering.
  - Body - JMS defines several types of message bodies, which cover the majority of messaging styles currently in use.

The JMS headers and properties that are not included in the MQMD message descriptor are placed in the MQRFH2 header.

It is also possible to change this behavior and force a JMS application to create only MQRFH1 headers for compatibility reasons with other MQ applications that don't expect an MQRFH2 header. This is done by changing the JNDI parameter of the topic `TARGCLIENT` from its default value `JMS` to the value `MQ`. Another method is to use directly the MQSeries class for JMS implementing the JMS interface to set at runtime the type of header to be generated. For more information concerning JMS, please refer to 4.2.3, “JMS overview” on page 36.







## Example application

In this chapter we give a general overview of a publish/subscribe application that we will develop and the several scenarios that will be discussed throughout the book.

The example we have chosen in order to show publish/subscribe messaging concepts, tools and techniques is the implementation of a fictional public transport system.

## 3.1 The business case

Consider a transport system where an operator has many things going on at once. There may be buses following different routes with passengers who want to know the status of their bus and when it will arrive at their stop. There may also be operators who need to know the status of the bus fleet from time to time.

More specifically the passengers would probably be interested in the current status of their bus, when it will arrive at their stop, and what buses do and will stop at their station. Most of what they will be interested in will be short-lived information. That is information that is transient and has little or no value to them once the moment has passed, since there will be more up-to-date information available or about to become available.

The operators, on the other hand, would possibly be interested in where all the buses are at a given time, whether they are on schedule or experiencing any delays, which buses have problems, and whether there are any breakdowns within the fleet. Some of this information is transient, but if it is retained it provides an historical picture of events that may be useful input to other processes.

The transport provider would like to have all the information available to the different consumers via Web pages where they can select what information they need.

As the buses travel around their routes they need to log their position each time they stop, get delayed, or break down. This information can then be made available to all the consumers on demand.

## 3.2 Application solution

We will now describe our example application in more detail.

### 3.2.1 Simulated public transport system

The sample application we describe in this book implements a simplified model of a global public transport system.

The global transport system is actually made of federated transport systems classified by *mode* (for example, trains, tube or buses) and *geography* (for example, Dover, London, or Hursley).

Each transport system is composed of a collection of independent *routes*, and a route belongs to just one transport system. Each route has the following attributes:

- ▶ *Route name*: this is unique across the transport system.
- ▶ *Number of stops*: this is the total number of stops along the route; stops are numbered starting from number 1.
- ▶ *Time between stops*: this is the number of seconds it takes for a vehicle to move from one stop to the next; we assume that all the stops on a given route are evenly spaced out along that route.

A route can accommodate a number of *vehicles*, each vehicle is identified by a name that is unique route-wide. All vehicles can contain up to 100 passengers.

Vehicles move along the route in a one-way fashion from the first to the last stop according to the time between stops attribute of the route. When a vehicle finally reaches the last stop, it disappears from the system. The only exceptions to this behavior are:

- ▶ The *breakdown* of a vehicle, which causes its early removal from the system.
- ▶ An *accident* found along the route, which delays the vehicle by a fixed amount of time (20 seconds).
- ▶ *Light traffic* conditions, which dynamically decrease the time between stops by a 25% factor.
- ▶ *Heavy traffic* conditions, which dynamically decrease the time between stops by a 25% factor.

When a vehicle reaches a stop, the following information is collected:

- ▶ Time-stamp (hh:mm:ss)
- ▶ Vehicle fill-in ratio (0 to 100)
- ▶ Traffic conditions (light, normal, heavy)
- ▶ Detection of any breakdowns
- ▶ Detection of any accidents

Several attribute values in the simulated transport system are randomly generated according to the following distribution of probability:

- ▶ Breakdowns happen 5% of the time
- ▶ Accidents happen 10% of the time
- ▶ Traffic is heavy 25% of the time, light 25% of the time, and normal for the remaining 50% of the time

- The fill-in ratio is a completely random non-negative integer less than or equal to 100.

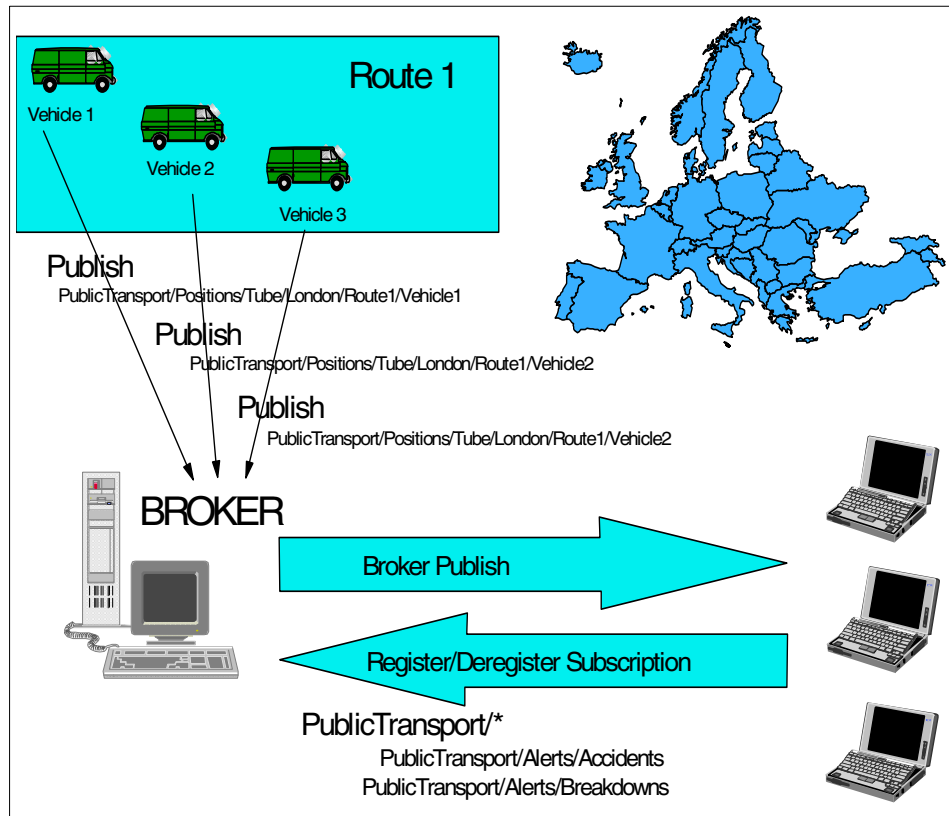


Figure 3-1 High-level overview of the example public transport system

Figure 3-1 shows a very high level overview of the example public transport system.

### 3.3 Publish/subscribe scenario 1

This scenario is based upon MQSeries Publish/Subscribe. The format of all messages exchanged by the applications written for our simulated public transport system is XML.

A vehicle publishes information about its position and state at each stop along the route, using the topic:  
PublicTransport/Positions/Mode/Geography/Route/Vehicle.

In the event of an accident or a breakdown, a message is published on topic: PublicTransport/Alerts/Accidents or PublicTransport/Alerts/Breakdowns.

The subscribing application is a Java stand-alone application written to the AMI interface, while the publishing application is provided in the following flavors:

- ▶ Java JMS
- ▶ Java AMI
- ▶ C AMI
- ▶ C MQI

The logic of all these versions is identical. The publisher application publishes messages on the default broker stream without registering first. (See 4.11.2, “Streams” on page 108.)

All the messages are published in a non-retained fashion. In fact this is the only publishing style supported by JMS. We will discuss retained publications in further detail in 4.11.1, “Retained publications” on page 105.

Chapter 4, “The publish/subscribe application” on page 27 contains a detailed discussion of this scenario.

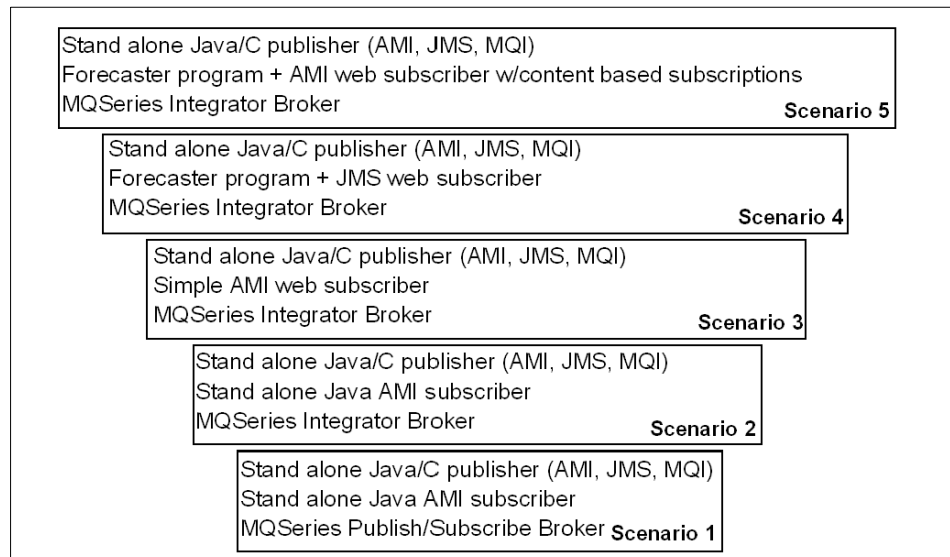


Figure 3-2 Overview of the scenarios discussed in the book

## 3.4 Publish/subscribe scenario 2

This scenario consists of the migration of the applications developed in the previous scenario to an MQSeries Integrator broker. This scenario demonstrates the API level interoperability between these two environments.

MQSeries Publish/Subscribe interoperability with MQSeries Integrator Version 2 Publish/Subscribe can be seen from three points of view:

- ▶ Migration/upgrading: migrating an established production MQSeries Publish/Subscribe broker to an MQSeries Integrator Version 2 broker (including retained publications, registered subscriptions, registered publications, and topology information).
- ▶ Broker-to-broker interoperability: setup and characteristics of mixed broker networks.
- ▶ Application-level interoperability: execution of MQSeries Publish/Subscribe applications against an MQSeries Integrator Version 2 broker.

Chapter 5, “Migration to MQSeries Integrator” on page 115 will focus mainly on application-level interoperability, but the other two aspects will be briefly discussed as well.

The execution of MQSeries Publish/Subscribe programs against an MQSeries Integrator broker is rather simple. All that is needed is to create a new local queue and to deploy a new simple message flow.

Yet even the simplest migration path to MQSeries Integrator-based publish/subscribe has several benefits, such as exploitation of the new security features that come with MQSeries Integrator.

Chapter 5, “Migration to MQSeries Integrator” on page 115 contains a detailed discussion of this scenario.

## 3.5 Publish/subscribe scenario 3

This scenario describes the migration of the stand-alone Java subscriber to a servlet running under WebSphere Application Server.

The first step towards the delivery of all the published information in a Web environment is the implementation of a simple subscriber application using a servlet running under WebSphere.

This first approach suffers from several limitations:

- ▶ Subscriptions coming from the Web are very short lived, so the use of retained publications is required:
  - This does not accommodate event information very well (accidents and breakdowns)
  - JMS cannot be used on the publishing side
- ▶ There is no immediate way in which to deliver value-added services such as computation of vehicle position forecasts to Web users.

Chapter 6, “Web enablement” on page 149 contains a detailed discussion of this scenario.

## 3.6 Publish/subscribe scenario 4

This scenario uses publish/subscribe in a more sophisticated way, in order to deliver advanced features such as real-time vehicle position monitoring and position forecasts, taking into consideration traffic conditions, accidents and breakdowns.

Both state and event information is published as non-retained by the publisher application. The only consumer of this information is a multithreaded application that is permanently subscribed to state and events messages.

This latter application re-publishes input information augmented with forecasts on a new retained topic subtree, using a particular technique that avoids the usage of a helper database to keep track and amend forecasts.

The Web subscriber can now be partially rewritten in order to access the publish/subscribe broker through JMS, which is the most natural way of doing messaging in a J2EE environment.

Chapter 7, “Advanced Web enablement” on page 165 contains a detailed discussion of this scenario.

## 3.7 Publish/subscribe scenario 5

This last scenario demonstrates how to deliver value-added information on the Web using a content-based publish/subscribe, a feature unique to MQSeries Integrator.

The Web application AMI subscriber from scenario 4 is extended in order to allow the specification of such filters as the following:

- ▶ End user views: all vehicles on a given route and stops within a range of time (for example, all tube trains leaving Kings Cross station from 17:00 to 17:12)
- ▶ Operator views: all vehicles in critical conditions (for example, all London buses currently stuck in heavy traffic, or delayed by an accident, or halted by a breakdown)

The filtering of messages will be performed centrally on the broker, optimizing the usage of network resources.

We will demonstrate that the code changes on the application required to implement these sophisticated features are minimal.

Chapter 7, “Advanced Web enablement” on page 165 contains a detailed discussion of this scenario.





## The publish/subscribe application

The publish/subscribe application in its first version is made up of two parts: the publication part and the subscription part. In this chapter we discuss each part in detail and describe the steps we followed.

## 4.1 Software components

Before describing these two parts in detail, we will describe the preliminary steps to install the complementary software components required for creating a publish/subscribe application and introduce their use:

- ▶ MQSeries Publish/Subscribe
  - Installation of the SupportPac
- ▶ Java Message Service APIs
  - Installation of the SupportPac
  - Overview of JMS
  - Configuration of JMS
- ▶ Application Messaging Interface APIs
  - Overview of AMI
  - Installation of the SupportPac
  - Configuration of AMI

After introducing these techniques, we illustrate them with our example:

- ▶ The publisher application:

The publisher application is governed by a Java application called PubLauncher. It starts a Java thread, PubThread, for each vehicle on each route.

The publication of the vehicle positions, accidents and breakdowns is made through C AMI, C MQI, Java AMI and JMS API calls.

In this part we describe:

- The controlling application: PubLauncher
  - The Java thread, called PubThread, started for each vehicle
  - The categories of messages published for each vehicle
  - How to publish messages in C AMI, C MQI, Java AMI, JMS and illustrate these techniques commenting on the publication helper modules used in the application
- ▶ The subscription part
- The following sections discuss the steps required to port the existing subscriber application to a Web based application in the WebSphere environment. We discuss:

- ▶ WebSphere Application Server configuration
- ▶ Servlet configuration
- ▶ AMI Repository configuration
- ▶ Program invocation
- ▶ Discussion of the Web part of the application

## 4.2 Environment setup

The application is running with MQSeries Version 5.2. We assume that MQSeries has already been successfully installed. Please refer to *MQSeries for Windows NT and Windows 2000 V5R2 Quick Beginnings*, GC34-5389 for more information on how to install and configure MQSeries.

To use the publish/subscribe functionality provided by MQSeries, we need to install the Pub/Sub SupportPac. The application is also using AMI and JMS API, both provided with SupportPacs as well.

The Java part of the application has been developed with VisualAge for Java Version 3.5.3. We don't cover the installation of this product here because it is adequately covered in the product documentation, but we do explain the configuration needed to use the MQ, AMI and JMS APIs with it.

For JMS, we also need a JNDI server. We describe how WebSphere Application Server and VisualAge for Java provide this functionality.

### 4.2.1 MQSeries Publish/Subscribe installation

Before using MQSeries Publish/Subscribe, please check that your MQSeries software product was successfully installed on your machine.

To run an MQSeries Publish/Subscribe broker, you need to install an additional component which can be downloaded from the Web and be installed as describe here for the Windows NT / 2000 platform:

- ▶ Go to Web site:  
<http://www.software.ibm.com/ts/mqseries/txppacs>
- ▶ Select the SupportPac **MA0C**. This SupportPac provides the MQSeries Publish/Subscribe facility for Microsoft Windows NT and 2000. Other operating systems are also supported.
- ▶ Click the **ma0c\_nt.zip** icon to download the files for MQSeries Publish/Subscribe for Windows 2000 in InfoZip compressed format.

- ▶ To install the MQSeries Publish/Subscribe, you need to uncompress the downloaded ma0c\_nt.zip file into a temporary directory, make it current and then execute the setup.exe program.
- ▶ Click **Next** to start the installation of MQSeries Publish/Subscribe as shown in Figure 4-1.

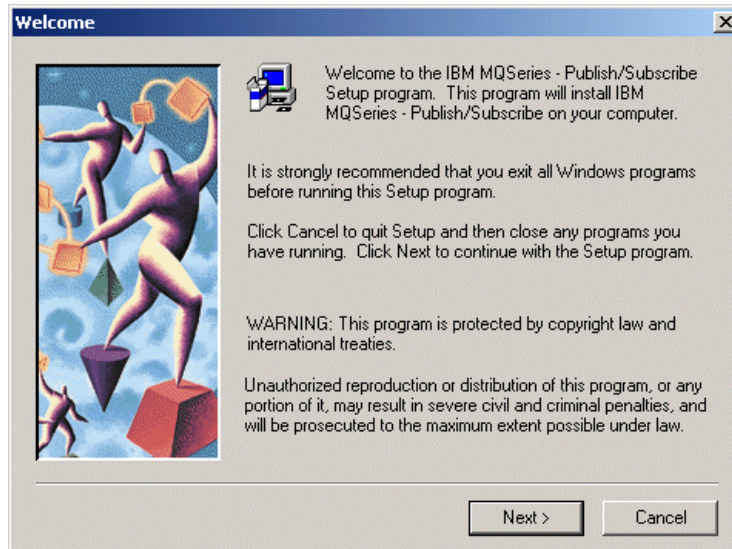


Figure 4-1 Installation of Publish/Subscribe SupportPac (1 of 4)

- ▶ Select **Yes** to continue as indicated in Figure 4-2.

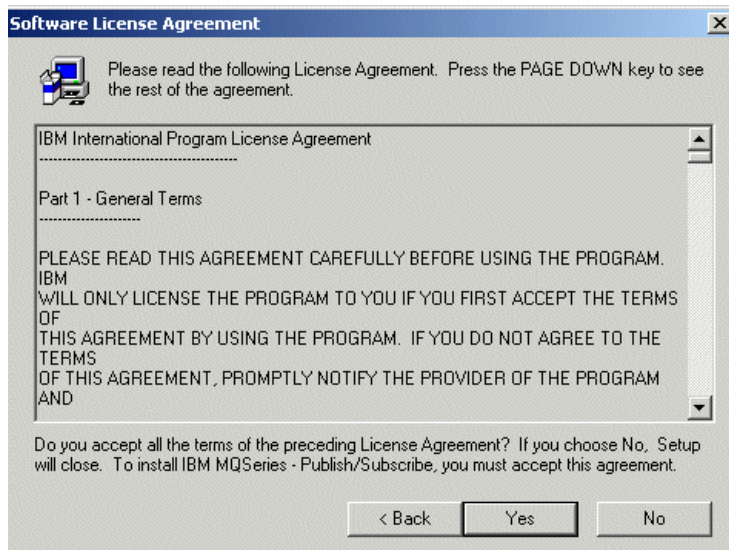


Figure 4-2 Installation of Publish/Subscribe SupportPac (2 of 4)

- ▶ MQSeries Publish/Subscribe files are copied in the destination standard folder, for example: C:\Program Files\MQSeries, where the software product of MQSeries was already installed. Click **Next** to continue your installation of MQSeries Publish/Subscribe in this directory, as illustrated in Figure 4-3.

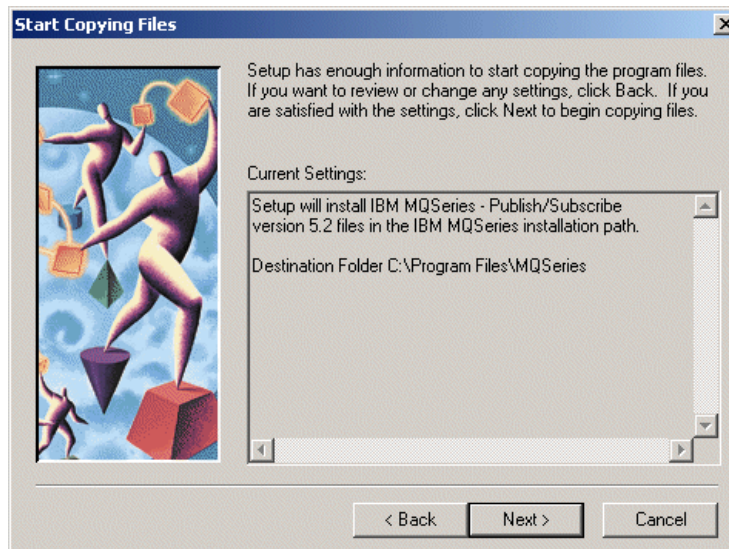


Figure 4-3 Installation of Publish/Subscribe SupportPac (3 of 4)

- ▶ Click **Finish** to confirm to terminate the installation as indicated in Figure 4-4.

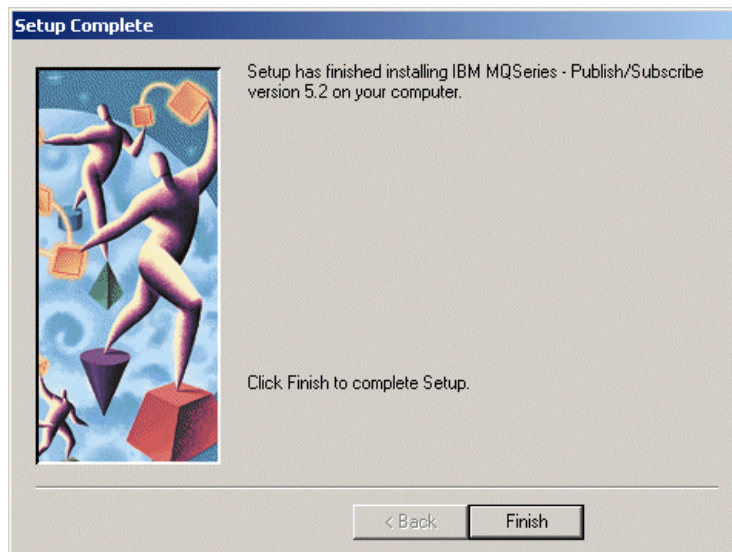


Figure 4-4 Installation of Publish/Subscribe SupportPac (4 of 4)

You have now successfully installed *MQSeries Publish/Subscribe for Windows 2000*.

## 4.2.2 JMS installation

To work with JMS, we must install the Java classes that implement the JMS interfaces defined by Sun. To make an analogy with databases, these classes can be thought of as a driver, not a JDBC driver but a JMS driver. The JMS classes for MQSeries are delivered in the free MA88 SupportPac. This SupportPac contains the MQSeries classes for Java (MQI) and the MQSeries classes for Java Messaging Service (JMS).

MA88 can be downloaded from the MQSeries internet URL:

<http://www.software.ibm.com/ts/mqseries/txppacs/>

You can download the appropriate version for your platform and install it following the installation instructions provided with the SupportPac.

For Windows 2000, we download the file `ma88_win`, uncompress it and start the `setup.exe` program. Figure 4-5 shows the welcome window of the installation.

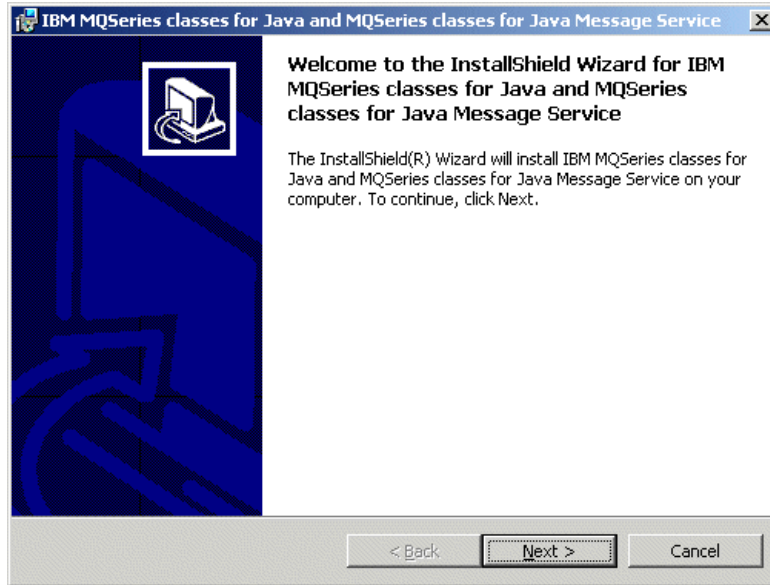


Figure 4-5 Installation of MA88 (1 of 4)

Click **Next** and choose between the **Complete** or the **Custom** installation.

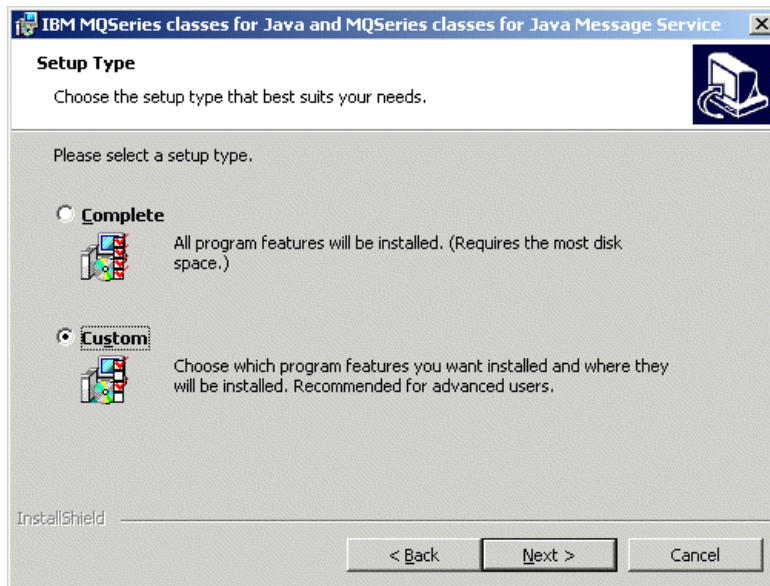


Figure 4-6 Installation of MA88 (2 of 4)



Unless you have installed MQSeries using the default directories, you must choose the **Custom** installation to specify the right directories where the new Java classes should be installed. Even if you have installed MQSeries in the default directory, we recommend that you choose the **Custom** option in order to check the installation directory.

After clicking **Next**, you can choose which Java classes you want to install, the MQSeries classes for Java or for JMS or both, as shown in Figure 4-7 on page 34.

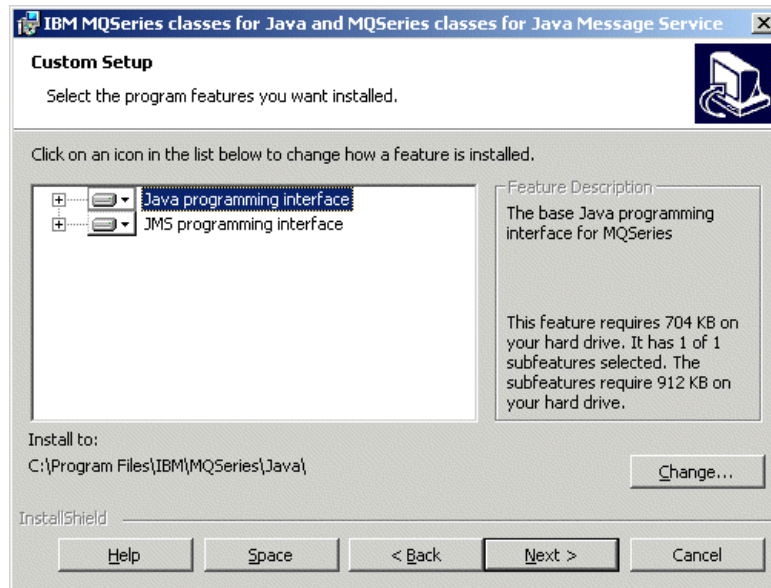


Figure 4-7 Installation of MA88 (3 of 4)

In our case we install both MQSeries classes. If MQSeries is installed in the default directory (for example C:\Program Files\IBM\MQSeries), you don't need to change the installation directory and can leave the default installation path (for example C:\Program Files\IBM\MQSeries\Java\). However we recommend that you check that the directory to which the Java classes will be installed corresponds to the existing Java subdirectory in your MQSeries installation directory.

If MQSeries has been installed into another directory, for instance C:\MQ52, then you must change the installation directory to C:\MQ52\Java\ as shown in Figure 4-8.



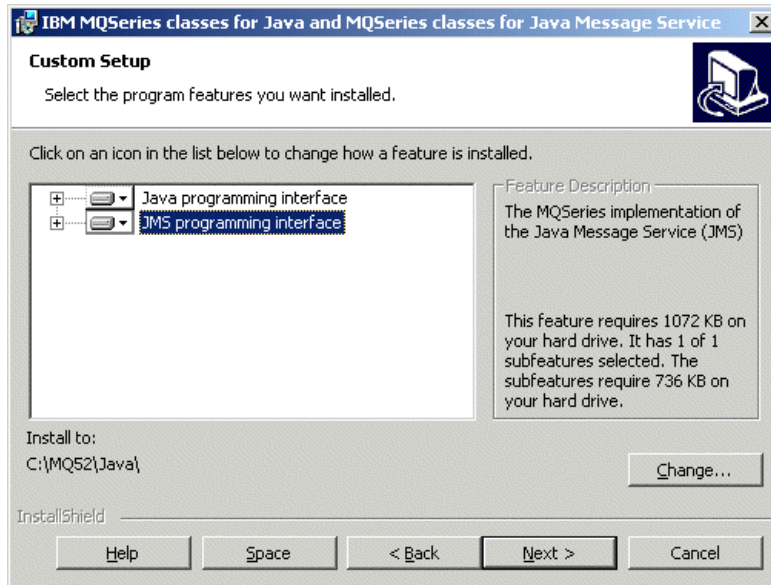


Figure 4-8 Installation of MA88 (4 of 4)

Click **Next** one last time and the MA88 SupportPac is installed automatically.

**Note:** During the installation of MA88, all the files present in the Java subdirectory of the MQSeries installation directory are placed in a backup directory and replaced with the new files contained in the SupportPac. Therefore, it is important that the AMI SupportPac is installed only after the installation of MA88. Otherwise, the JAR file containing the AMI Java classes may no longer be found in the classpath environment variable after the installation has completed.

Once the installation has completed, and in order to run publish/subscribe applications with JMS, you need to run an MQSC script, MQJMS\_PSQ.mqsc, on the queue manager that you are using to define the MQSeries queues required for JMS in its publish/subscribe mode.

This script is located in the bin subdirectory of the directory where we installed the MA88 SupportPac, for example C:\Program Files\IBM\MQSeries\Java\bin or C:\MQ52\java\bin.

To run the script on the default queue manager, execute the following command:

```
runmqsc < "C:\Program Files\IBM\MQSeries\Java\bin\MQJMS_PSQ.mqsc"
```

where C:\Program Files\IBM\MQSeries is the MQSeries installation directory.

This script creates the following queues:

- ▶ SYSTEM.JMS.ADMIN.QUEUE: the JMS publish/subscribe administration queue.
- ▶ SYSTEM.JMS.PS.STATUS.QUEUE: the JMS publish/subscribe subscriber status queue.
- ▶ SYSTEM.JMS.REPORT.QUEUE: the JMS publish/subscribe report queue
- ▶ SYSTEM.JMS.MODEL.QUEUE: the JMS publish/subscribe subscribers model queue. (This model queue is used by subscribers to create a permanent queue for subscriptions.)
- ▶ SYSTEM.JMS.ND.SUBSCRIBER.QUEUE: the JMS publish/subscribe default non-durable shared queue. (The default shared queue used by non-durable subscribers.)
- ▶ SYSTEM.JMS.ND.CC.SUBSCRIBER.QUEUE: the JMS publish/subscribe default non-durable shared queue for connectionconsumer functionality.
- ▶ SYSTEM.JMS.D.SUBSCRIBER.QUEUE: the JMS publish/subscribe default durable shared queue. (The default shared queue used by durable subscribers.)
- ▶ SYSTEM.JMS.D.CC.SUBSCRIBER.QUEUE: the JMS publish/subscribe Default durable shared queue for connectionconsumer functionality.

The installation part of the MQSeries classes for Java and JMS is now complete. In the next section, we introduce JMS and explain how to use it.

### 4.2.3 JMS overview

In this section, we will briefly describe the various concepts introduced by JMS. For more information on JMS, please refer to *Using Java*, SC34-5456.

JMS is an API specification defined by Sun to enable application portability. It is part of the J2EE specification. The JMS interfaces defined by the specification are contained in the Java package `javax.jms`. This package not only contains the Java interfaces but also a few Java classes, such as the `JMSEException` class.

Figure 4-9 on page 37 illustrates the interfaces included in the JMS package.

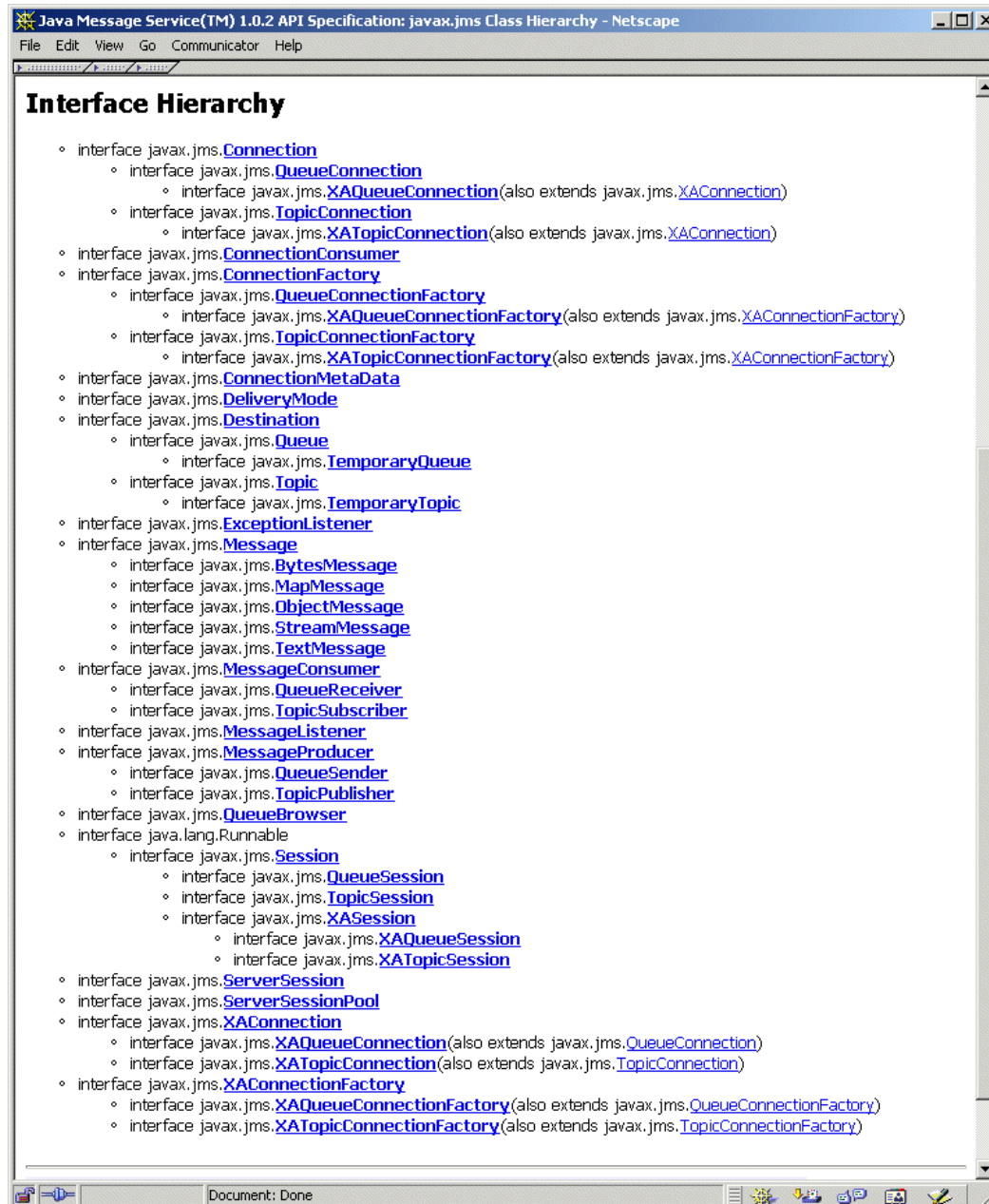


Figure 4-9 The JMS interface hierarchy

JMS defines two messaging domains, the point-to-point domain and the publish/subscribe domain, and both domains are supported by the MQSeries implementation of the JMS standard.

The six base interfaces defined by JMS are:

- ▶ The Connection interface provides access to the underlying transport and contains the parameters to connect to the queue manager.
- ▶ The Session interface is created from a Connection and corresponds to an MQSeries connection to a queue manager. It provides a transactional scope.
- ▶ The Destination interface encapsulates provider-specific addresses, since JMS does not define a standard address syntax. In MQSeries terms, it provides the information necessary to find a queue on the queue manager.
- ▶ The MessageProducer interface is created from a session by specifying the Destination and is used to send messages.
- ▶ The MessageConsumer interface is created from a session by specifying the Destination and is used to receive messages. It corresponds to an MQSeries input queue.
- ▶ The Message interface is created from a session and is used in a subclassed form for sending as well as for receiving messages.

The generic Connection, Session, MessageProducer and MessageConsumer interfaces are implemented by other interfaces specific to one of the two messaging domains as shown in Table 4-1.

*Table 4-1 Interface correspondence across messaging domains*

<b>Generic interface</b>	<b>point-to-point</b>	<b>publish/subscribe</b>
Connection	QueueConnection	TopicConnection
Session	QueueSession	TopicSession
Destination	Queue	Topic
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber

To ensure portability and independence from implementation providers, a Connection is obtained from a ConnectionFactory encapsulating a set of connection configuration parameters that have been defined by a JMS administrator. JMS establishes the convention that connection factories and other JMS administered objects are found by looking them up in a JNDI namespace.

This means that we must first register the parameters specific to an implementation of JMS by a provider into a JNDI server: a directory server (for example LDAP server) or a name server (for example the Persistent Name Server provided by WebSphere and VisualAge for Java). In other words, we start by serializing the ConnectionFactory object.

Once registered, we can retrieve this information in a JMS program by looking it up in the JNDI namespace. In other words, we deserialize the ConnectionFactory and other JMS administered objects that had previously been serialized by JNDI.

Equivalent steps are taken for destination; queue and topic are stored and retrieved in and from JNDI.

JMS providers, such as MQSeries, are expected to provide the tools an administrator needs to create and configure administered objects in a JNDI namespace. MQSeries provides a tool called the MQ JMS administration tool or JMSAdmin to store the ConnectionFactory (QueueConnectionFactory or TopicConnectionFactory) and the destination (queue or topic) in the JNDI namespace. This tool is a Java application contained in the class `com.ibm.mq.jms.admin.JMSAdmin`, which can be launched with the `JMSAdmin.bat` file located in the directory `C:\Program Files\IBM\MQSeries\Java\bin`, where `C:\Program Files\IBM\MQSeries` is the MQSeries installation directory.

#### 4.2.4 JMS configuration, JNDI and JMSAdmin

**Important:** VisualAge for Java Enterprise Edition and WebSphere Application Server provide a Persistent Name Server that can be used as JNDI server. Since as we are using VisualAge for Java for our programming and WebSphere Application Server for running the Web part of our application, we decided to use them also as the JNDI provider.

##### Setting up VisualAge for Java for use with JMS

We are using VisualAge for Java Version 3.5.3, but previous versions of VisualAge for Java should work as well.

Here are the steps to follow to work with JMS in VisualAge for Java Enterprise Edition:

1. Make sure the features IBM Enterprise Extension Libraries and IBM WebSphere Test Environment are loaded in the workspace.
2. Create a new project. We call it MQ.

3. Import in this project the JAR files mentioned hereafter from the directory C:\Program Files\IBM\MQSeries\Java, where C:\Program Files\IBM\MQSeries is the MQSeries installation directory. Figure 4-10 illustrates this import.

- com.ibm.mq.jar
- com.ibm.mqbind.jar
- com.ibm.mqjms.jar
- com.ibm.mq.iiop.jar

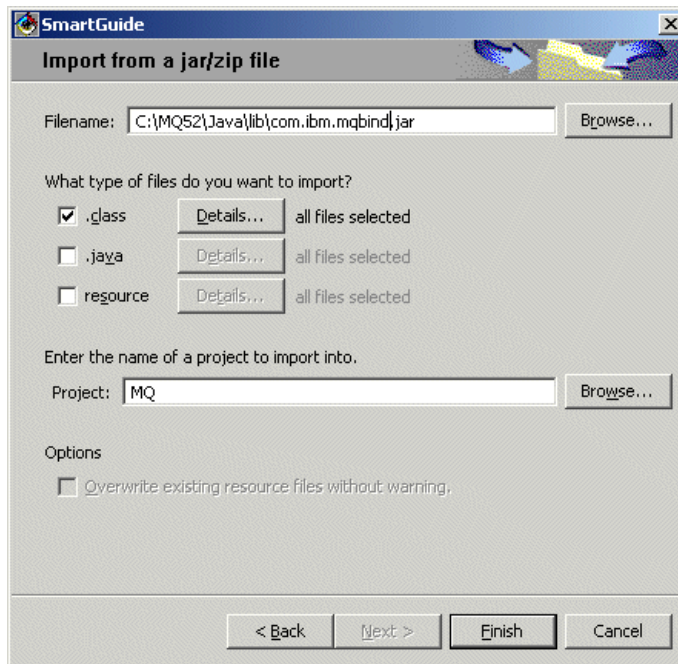


Figure 4-10 Import of JAR files in VisualAge for Java (1 of 2)

4. You should also import the following packages in your project, but depending on the version of VisualAge the packages could be added automatically in other projects provided by VisualAge for Java:

- jms.jar
- jndi.jar
- ldap.jar
- fscontext.jar
- providerutil.jar

Once all the JAR files have been imported, you should have a project similar to the project MQ shown in Figure 4-11.

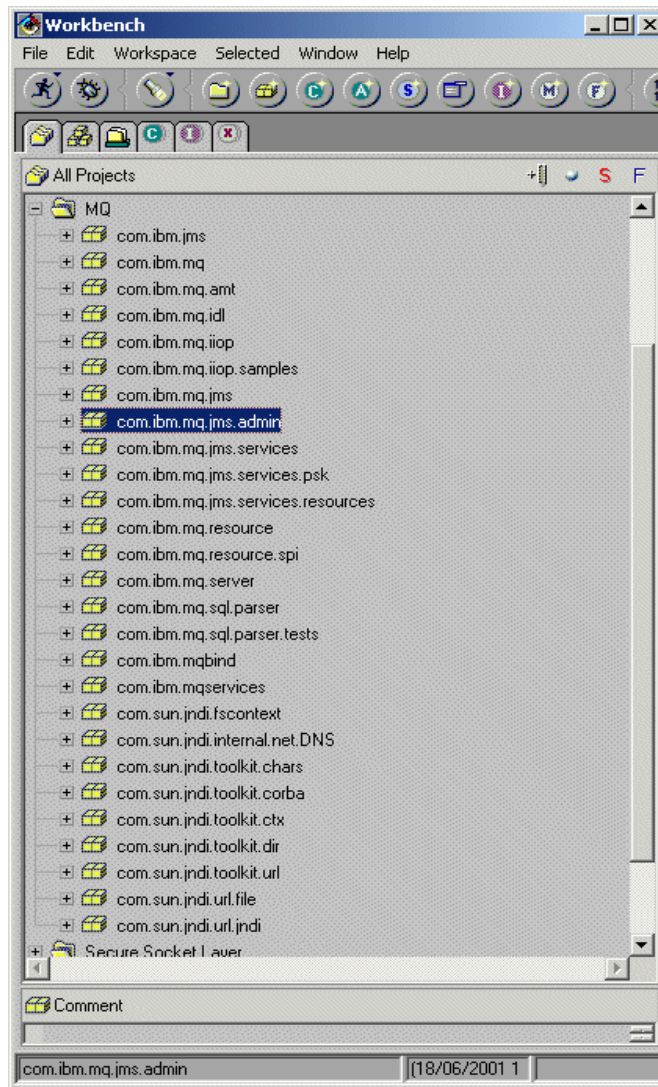


Figure 4-11 Import of jar files in VisualAge for Java (2 of 2)

- ▶ Add the directory C:\Program Files\IBM\MQSeries\Java, where C:\Program Files\IBM\MQSeries is the MQSeries installation directory, to the workspace classpath. You can access the workspace classpath from the Workbench of VisualAge for Java by clicking **Window -> Options... -> Resources**.

Figure 4-12 illustrates this step.

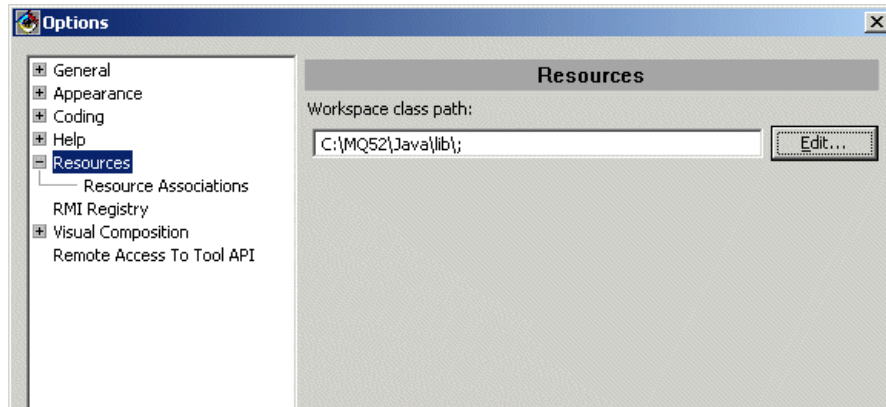


Figure 4-12 Changing the workspace classpath

5. If you want to use the binding mode when accessing the queue manager, the path environment variable must include the directory C:\Program Files\IBM\MQSeries\Java\bin, where C:\Program Files\IBM\MQSeries is the MQSeries installation directory. This directory contains the mqjbnd02.dll file required for using the binding mode. The path environment variable must be set before starting VisualAge for Java.

## Using JNDI in VisualAge for Java

The following steps describe how to use the Persistent Name Server in VisualAge for Java.

1. The use of the Persistent NameServer requires a database server to be accessible by VisualAge with JDBC. We are using DB2 Universal Database and have created a database named jndi. The workspace classpath must contain the JDBC APIs delivered with the database. When using DB2, we must add C:\SQLLIB\java\db2java.zip to the VisualAge for Java workspace classpath, where C:\SQLLIB is the installation folder of DB2.
2. Start the WebSphere Test Environment by clicking **Workspace -> Tools -> WebSphere Test Environment**.
3. Select the Persistent Name Server and enter the configuration parameters to connect to a database. Figure 4-13 on page 43 illustrates how to connect to the DB2 database named jndi.



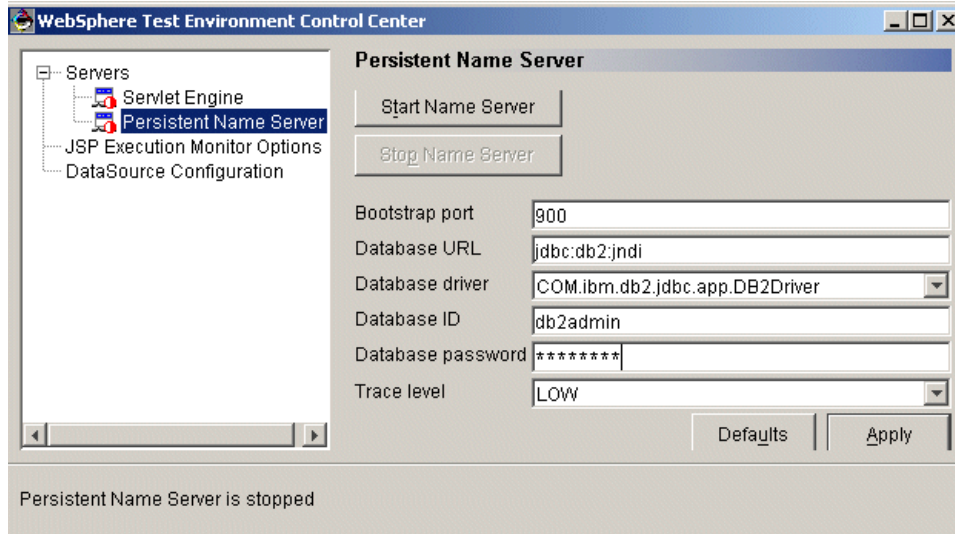


Figure 4-13 Persistent Name Server (1 of 2)

4. Start the Name Server by clicking **Start Name Server**. If the Name Server has started successfully, you should see a WebSphere Test Environment comparable to the one displayed in Figure 4-14.

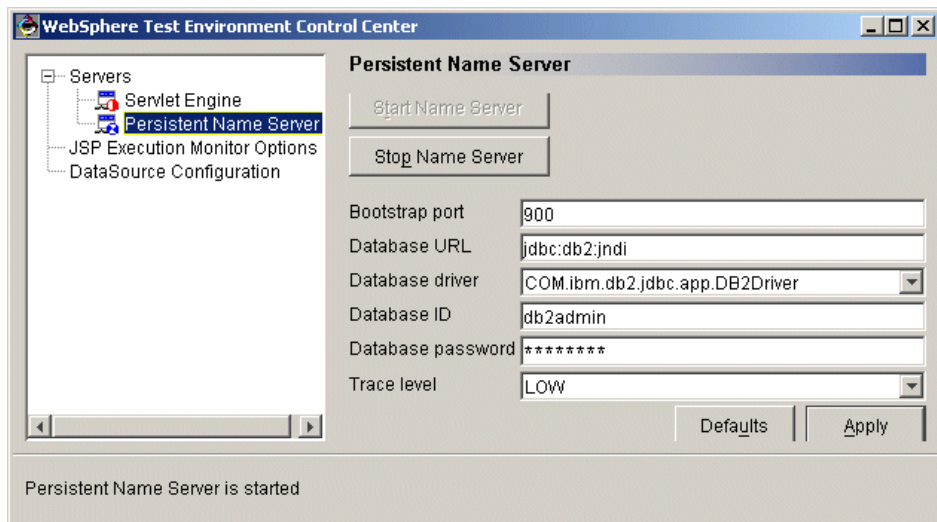


Figure 4-14 Persistent Name Server (2 of 2)

## Using JMSAdmin with VisualAge for Java

In this section, we explain how to create the connection factories and Destination JMS objects in the JNDI namespace with the JMSAdmin administration tool provided by MQSeries.

We start by updating the JMSAdmin.config file to indicate to the JMSAdmin application that it must use our Persistent Name Server. This file is located in the directory C:\Program Files\IBM\MQSeries\Java\bin, where C:\Program Files\IBM\MQSeries is the MQSeries installation directory. We must change the following two entries in this file:

- ▶ INITIAL\_CONTEXT\_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory

This value is valid for the Persistent Name Server provided by VisualAge for Java Enterprise Edition and by WebSphere Application Server. For other JNDI servers, the INITIAL\_CONTEXT\_FACTORY variable must be provided in the accompanying documentation.

- ▶ PROVIDER\_URL=iiop://hostname/

Where hostname is the name of the machine where the Persistent Name Server is running. When the Internet Inter-ORB Protocol (IIOP) is used, the default port is 900 and doesn't need to be specified. If you change the default port of the Persistent Name Server as shown in Figure 4-14 on page 43 in the bootstrap port field, you must indicate the port in the PROVIDER\_URL variable, for example iiop://hostname:901/.

When we imported all the MQSeries classes for Java and JMS as described in “Setting up VisualAge for Java for use with JMS” on page 39, we also imported at the same time the JMSAdmin application included in the package com.ibm.mq.jms.admin. Which means that we can easily run this administration tool from VisualAge for Java itself, after completing the next two steps:

1. Copy the JMSAdmin.config file in the Project Resources of the project we created in “Setting up VisualAge for Java for use with JMS” on page 39 or update the workspace classes to include the directory C:\Program Files\IBM\MQSeries\Java\bin.
2. Right-click the JMSAdmin class, as shown in Figure 4-15 on page 45, and choose **Properties** and **Classpath**.

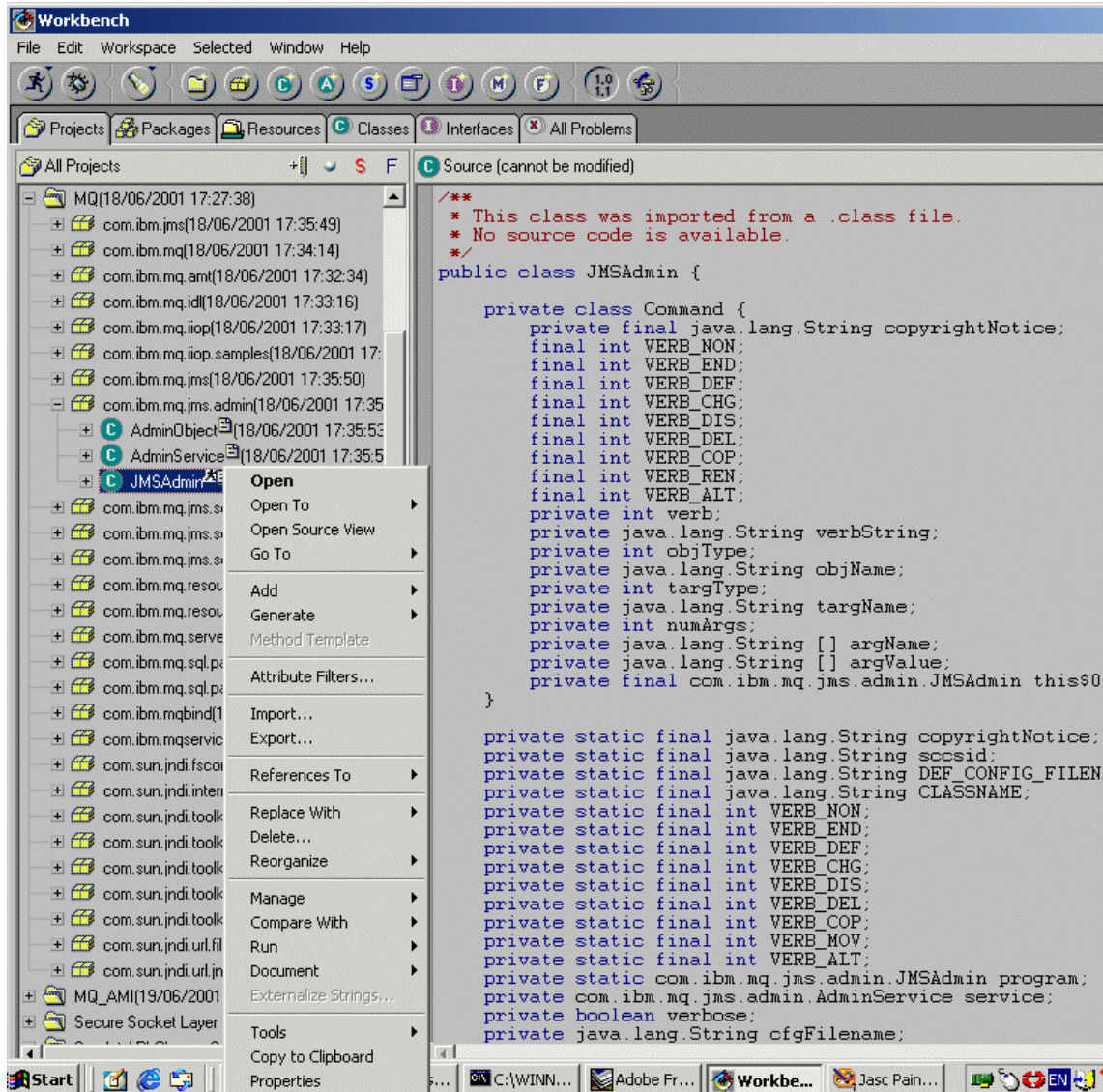


Figure 4-15 JMSAdmin (1 of 4)

Make sure the project path includes both the IBM Enterprise Extension Libraries and the IBM WebSphere Test Environment as illustrated in Figure 4-16 on page 46. Use the **Edit** button to add these two projects if needed.

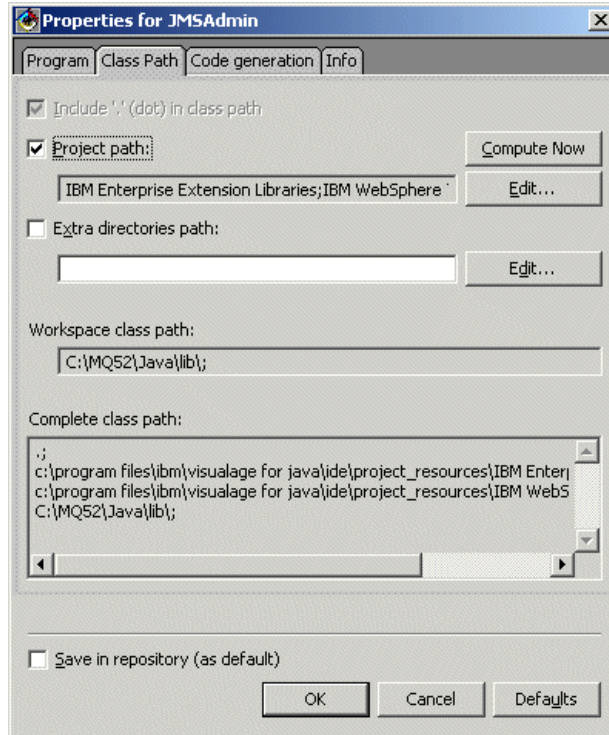


Figure 4-16 JMSAdmin (2 of 4)

We are now ready to start the JMSAdmin application in VisualAge for Java. This causes the console window of VisualAge to pop up, because the JMSAdmin application is an interactive application waiting for us to input a command. These administration commands are documented in the MQSeries book, *Using Java*, SC34-5456.

Our first step is to create a *context*, which can be thought of as a directory in the JNDI namespace. We create a context named “jms” by entering the following command: **def ctx(jms)** in the Standard In window pane of the console. To check the result, we can display the current or initial context with the command **dis ctx**. The output of this command is displayed in Figure 4-17 on page 47.

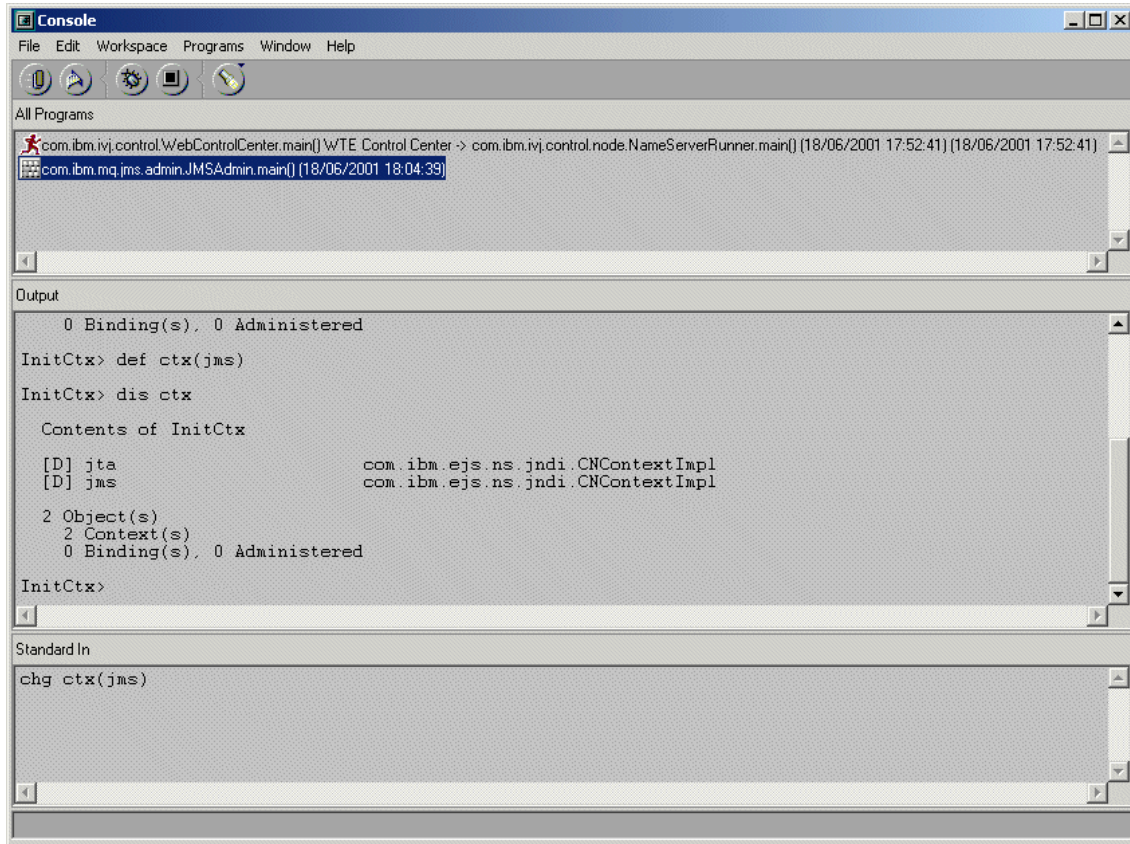


Figure 4-17 JMSAdmin (3 of 4)

We see that there are currently two defined contexts in our initial context: `jms`, the one we created, and `jta`, a context we are not concerned with at the present time.

To change from the initial context and pass into the newly created `jms` context, we can execute the command `chg ctx(jms)` displayed in the Standard In input area in Figure 4-17. This context is currently empty.

We will now create our JMS administered objects, which will allow us to work with JMS. Since this book is focusing on publish/subscribe and not on JMS, we will only discuss the publish/subscribe related objects. The point-to-point objects are created in a very similar way, simply by changing the type of object created, for example replacing a topic with a queue.

As already discussed in the 4.2.3, “JMS overview” on page 36, the first object we need to create in JNDI is a ConnectionFactory containing the information required by a JMS application to be able to connect to an MQSeries queue manager.

In the publish/subscribe messaging domain, we refer more specifically to a TopicConnectionFactory. The following command creates a TopicConnectionFactory (tcf) called ITSOPS, referring to a queue manager ITS0 running on the host machine ITS0 and listening on the TCP/IP port 1415, with a connection in client mode through the server connection channel named JVACHL, with a broker installed on that same queue manager. The broker control queue and publication queue specified are the default values for the MQSeries Publish/Subscribe broker.

```
def tcf(ITSOPS) transport(CLIENT) QMANAGER(ITS0) HOST(ITS0) PORT(1415)  
CHANNEL(JVACHL) BROKERQMgr(ITS0) BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)  
BROKERPUBQ(SYSTEM.BROKER.DEFAULT.STREAM)
```

We can also create another TopicConnectionFactory called ITSOPSBND, containing exactly the same information but defined for a connection in binding mode rather than in client mode. ITSOPSBND is created with the following command:

```
def tcf(ITSOPSBND) transport(BIND) QMANAGER(ITS0) BROKERQMgr(ITS0)  
BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)  
BROKERPUBQ(SYSTEM.BROKER.DEFAULT.STREAM)
```

It is important to understand that what these commands really do is write in a database the information necessary to connect to a queue manager so that it can be retrieved in a portable way by JNDI to be used in a JMS application. All this has no influence on the current configuration of MQSeries.

Additionally, we can also define the JNDI topic (t) objects Accident and Breakdown with the following commands:

```
def t(Accident) topic(PublicTransport/Alerts/Accidents)  
def t(Breakdown) topic(PublicTransport/Alerts/Breakdowns)
```

Figure 4-18 on page 49 displays these commands executed by the JMSAdmin application.



```

Console
File Edit Workspace Programs Window Help
All Programs
* com.ibm.ivj.control.WebControlCenter.main[] WTE Control Center -> com.ibm.ivj.control.node.NameServerRunner.main[] [26/06/2001 22:43:44] [26/06/2001 22:43:44]
com.ibm.mq.jms.admin.JMSAdmin.main[] [26/06/2001 23:16:46]

Output

5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
Starting MQSeries classes for Java(tm) Message Service Administration

InitCtx> chg ctx(jms)

InitCtx/jms> def tcf(ITSOPS) transport(CLIENT) QMANAGER(ITSO) HOST(ITSO) PORT(1414) CHANNEL(JVACHL) BROKERO
InitCtx/jms> def tcf(ITSOPSBND) transport(BIND) QMANAGER(ITSO) BROKERQMGR(ITSO) BROKERCONQ(SYSTEM.BROKER.CO
InitCtx/jms> def t(Accident) topic(PublicTransport/Alerts/Accidents)
InitCtx/jms> def t(Breakdown) topic(PublicTransport/Alerts/Breakdowns)

InitCtx/jms> dis ctx(jms)
Syntax error

InitCtx/jms> dis ctx

Contents of InitCtx/jms

a ITSOPS com.ibm.mq.jms.MQTopicConnectionFactory
a Breakdown com.ibm.mq.jms.MQTopic
a ITSOPSBND com.ibm.mq.jms.MQTopicConnectionFactory
a Accident com.ibm.mq.jms.MQTopic

4 Object(s)
0 Context(s)
4 Binding(s), 4 Administered

InitCtx/jms>
Standard In

```

Figure 4-18 JMSAdmin (4 of 4)

To end the JMSAdmin program, simply enter the **end** command.

## Using JMSAdmin with WebSphere Application Server

We can also start the JMSAdmin tool from the command line with the JMSAdmin.bat file provided with MA88 and using, for instance, WebSphere Application Server as the JNDI provider. We don't repeat here the description of JMSAdmin, since its configuration parameters and commands were already described in "Using JMSAdmin with VisualAge for Java" on page 44.

For WebSphere Application Server to work as a Persistent Name Server, nothing needs to be done. When the WebSphere AdminServer is up and running, we can access it as a Persistent Name Server. The database used is the database serving for the WebSphere Application Server repository.

In the previous section, we said that the JMSAdmin program could be run from VisualAge for Java. It is also possible to run it directly from the command line with the JMSAdmin.bat file, available in the directory C:\Program Files\IBM\MQSeries\Java\bin, where C:\Program Files\IBM\MQSeries is the MQSeries installation directory. When starting the JMSAdmin with the bat file from the DOS command line, you can run it in batch mode by specifying a script file containing all the JMSAdmin commands. Before starting JMSAdmin.bat, you must:

1. Edit the configuration file JMSAdmin.config in the same directory

As the Persistent Name Server provided by VisualAge for Java and WebSphere Application Server are similar, we must make the same changes in the configuration file JMSAdmin.config as those previously done for VisualAge for Java to make sure we can connect to the Persistent Name Server:

- INITIAL\_CONTEXT\_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory
- PROVIDER\_URL=iiop://hostname/

Where hostname is the name of the machine where the AdminServer is running. If the default port (900) has been changed to, for instance, 901, it must be specified as in the following statement:

```
PROVIDER_URL=iiop://hostname:901/
```

2. Adapt or verify the following environment variables:

- CLASSPATH: it must contain the three following packages:
  - C:\Program Files\IBM\MQSeries\Java\lib\jms.jar, where C:\Program Files\IBM\MQSeries is the MQSeries installation directory
  - C:\Program Files\IBM\MQSeries\Java\lib\com.ibm.mqjms.jar
  - C:\WebSphere\AppServer\lib\ujc.jar, where C:\WebSphere\AppServer is the WebSphere Application Server installation directory
- PATH: it must contain a JDK (Java Development Kit) or JRE (Java Runtime Environment). We can use the JRE provided with WebSphere Application Server by adding the directory C:\WebSphere\AppServer\jdk\jre\bin in the PATH environment variable, where C:\WebSphere\AppServer is the WebSphere Application Server installation directory.
- MQ\_JAVA\_INSTALL\_PATH: this environment variable is used in the provided JMSAdmin.bat file. In order to use this bat file without any modification, you need to define this variable either in your system or at least in the command line prompt where JMSAdmin is run. MQ\_JAVA\_INSTALL\_PATH must be set to C:\Program Files\IBM\MQSeries\Java\lib where C:\Program Files\IBM\MQSeries is the MQSeries installation directory.



**Tip:** If you prefer not to change the environment variables on your system, you don't need to. You can simply include the SET commands to change them directly in the batch file.

3. Create a script file containing the JMSAdmin commands to be run if you want to run JMSAdmin in batch mode. For instance, we create a file `JMSAdmin.scp` containing the following commands:

```
def ctx(jms)
chg ctx(jms)
del tcf(ITSOPS)
del tcf(ITSOPSBND)
del t(AccidentMQ)
del t(BreakdownMQ)
del t(AccidentJMS)
del t(BreakdownJMS)
def tcf(ITSOPS) transport(CLIENT) QMANAGER(ITSO) HOST(ITSO) PORT(14141)
CHANNEL(JVACHL) BROKERQMGR(ITSO) BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)
BROKERPUBQ(SYSTEM.BROKER.DEFAULT.STREAM)
def tcf(ITSOPSBND) transport(BIND) QMANAGER(ITSO) BROKERQMGR(ITSO)
BROKERCONQ(SYSTEM.BROKER.CONTROL.QUEUE)
BROKERPUBQ(SYSTEM.BROKER.DEFAULT.STREAM)
def t(AccidentJMS) topic(PublicTransport/Alerts/Accidents)
def t(BreakdownJMS) topic(PublicTransport/Alerts/Breakdowns)
def t(AccidentMQ) topic(PublicTransport/Alerts/Accidents) TARGCLIENT(MQ)
def t(BreakdownMQ) topic(PublicTransport/Alerts/Breakdowns) TARGCLIENT(MQ)
def t(Alert) topic(PublicTransport/Alerts/*)
def t(PublicTransport) topic(PublicTransport/*)
end
```

It is important not to forget the **end** command if you want to end the JMSAdmin tool properly.

4. Make sure that the WebSphere Administration Server is started. You can verify that the default port 900 is listening with the command `netstat -a` on the command line.

When all configuration steps are executed, we start the JMSAdmin tool:

- ▶ In interactive mode: by running `JMSAdmin.bat`
- ▶ In batch mode: redirecting the standard input by running the command `JMSAdmin.bat < JMSAdmin.scp`

## 4.2.5 Defining MQSeries required for the application

In order to run the Public Transport application, a few MQSeries objects (queues and server connection) must be created. In the additional material accompanying this book, we provide the MQSC script called `vehicle.mqsc` containing the definitions for these objects. You should run this script against a queue manager. You may decide to create a new queue manager or to use an existing one. Run the script with the command `runmqsc QMGR_NAME < vehicle.mqsc`, where `QMGR_NAME` is the name of the queue manager. If you want to create these objects on the default queue manager, you can use the command `runmqsc < vehicle.mqsc`.

## 4.2.6 AMI overview

The Application Messaging Interface (AMI) API is one of the latest additions to the MQSeries API portfolio, AMI has very interesting features and a different approach. Given that AMI will be used extensively throughout the book, we provide a very brief overview of its most important aspects. Complete information about this API can be found in *MQSeries Application Messaging Interface*, SC34-5604.

### Key features and concepts

The key features of AMI are:

- ▶ Provides a higher level, easy-to-use interface to messaging.
- ▶ Reduces to a minimum the number and complexity of data structures and parameters.
- ▶ Increases the number of API verbs in order to accommodate each messaging style (send and forget, request/reply, publish/subscribe) in the best way.
- ▶ Provides reasonable API structure for procedural languages like C and COBOL, as well as object-oriented ones like C++ and Java.
- ▶ Moves middleware-specific options out of the application domain and into the administration domain by using an external persistent repository.
- ▶ Interoperates seamlessly with any other MQSeries messaging API flavor (for example Java JMS, and RPG MQI).

The AMI structure and concepts revolve around three main entities: *services*, *policies* and *messages*. Let's review each of them in turn:

- ▶ A service is the abstraction of an MQSeries queue, assuming that a queue hosts semantically homogeneous sets of messages being consumed (*served*) by an application to satisfy a business need. Service definitions are stored in the repository.

- ▶ A policy is a collection of attributes that define how to handle messages from a quality-of-service point of view (for example priority, persistence, acknowledgement level), from a recovery point of view (for example error handling, and retries) or a housekeeping point of view (for example expiration).  
Policy definitions are stored in the repository.
- ▶ A message is an object containing the actual application data and some ancillary attributes (for example a correlation identifier or coded character set identifier) that the typical application can often safely ignore.

## API set structure

In this section we briefly describe the main objects of the object-oriented flavors of AMI (C++ and Java). A specific discussion of the C procedural flavor peculiarities can be found in 4.6.1, “Vehicle C AMI program” on page 81 where we explain the C version of the sample Vehicle program.

The most important class is *AmSession*. This class creates and manages all other objects providing also the transactional scope for units of work. Session objects are created using the *AmSessionFactory* class.

The *Connection* object is administered via the repository and is not exported at the application level. It deals with the details of how to actually connect to the messaging service provider.

The *AmMessage* object wraps the MQSeries message descriptor and other optional headers (for example, MQRFH rules and formatting header or MQRFH2 rules and formatting header). Application data may be stored within or outside the *amMessage* object instance.

The *AmPolicy* object wraps all options needed at open, close, send, receive, publish, subscribe times. There are several methods that can be used to override policy defaults (for example, *setWaitTime* for setting message reception timeout).

The last object is the *Service* object. There are five types of service:

- ▶ *AmSender* is the abstraction of an MQSeries output queue. It contains the actual message *send* method.
- ▶ *AmReceiver* is the abstraction of an MQSeries input queue. It contains the actual message *receive* method.
- ▶ *AmDistributionList* is a collection of AMI *amSender* objects. The main exported method is *send*.
- ▶ *AmPublisher* contains an *AmSender* object and is used to send publication messages to a publish/subscribe broker via the *publish* method.

- ▶ *AmSubscriber* contains an *AmSender* object (used to send subscription requests to a publish/subscribe broker) and an *AmReceiver* object (used to actually receive publication messages forwarded by the broker). The main methods exported are *subscribe*, *unsubscribe*, and *receive*.

## Facilities for publish/subscribe messaging

AMI is a very convenient API to use when implementing MQSeries Publish/Subscribe applications because it actually shields the developer from the mundane tasks of formatting and parsing broker headers, coping with the versioning of those headers (for example, MQSeries Integrator supports the MQRFH2 header, which is far more complex than the MQRFH header supported by MQSeries Publish/Subscribe).

Even if much of the complexity is shielded from the developers, sophisticated publish/subscribe applications can still access fringe features through lower-level API calls adding specific elements to headers.

## Repository

The AMI repository is used to store policy definitions, service definitions and service point definitions. Even if an AMI application can function without the repository (out of built-in defaults), the typical application will use one, in order to exploit its many benefits.

The repository is an XML file that is administered using the provided *AMI Tool*, a Java-based GUI currently running only on the Windows NT platform. This is not a big limitation given that the repository file can be shared between different platforms using standard file-sharing facilities or replicated via file transfer.

An alternative way of accessing the repository at runtime is via an LDAP server. This feature is supported only by AMI Version 1.2. The supported LDAP providers are Microsoft Active Directory (Windows NT/2000 only), IBM SecureWay, and DB2.

Runtime access to an LDAP directory is not supported on OS/390, but users will be able to build a local static cache (AMI 1.1 style) from data on an LDAP directory.

A typical application pattern in MQSeries is to have several queue managers hosted on different machines that are isomorphic (that is, the same queue names, same processes, same channels, same applications serving the same queues) except for the queue manager names that must be distinct to make intercommunication possible.

This pattern is easily accommodated by AMI. In fact you can make the repository definitions that are symbolic with respect to queue manager names (this way the definitions can be shared among the isomorphic queue managers), then you distribute a very small XML file on each machine named the *AMI Host File*, to define the mapping between the symbolic queue manager names in the repository and the real queue manager names on the machine, as locally applicable to that machine.

The advantage of this approach is that administration of more dynamic configuration information can be performed on the central repository image (a shared XML file or an LDAP directory), while the information locally stored on each machine is only of a highly static nature.

Applications can point to a particular repository or host using `AmSessionFactory.setRepository` and `AmSessionFactory.setLocalHost` methods. For C language applications, a similar functionality is available only via environment variables. An AMI installation provides default locations for both repository and AMI host file.

## **Policy handlers**

AMI 1.2 on the non-z/OS and OS/390 platforms supports a new feature named *Policy Handlers*. A policy handler is basically a user exit that is called by the AMI kernel at certain stages of the API computation.

The user-implemented handlers are passed context information and can modify it as appropriate. Applications of this feature are:

- ▶ Message logging/auditing
- ▶ Message encryption/decryption
- ▶ Message compression/decompression
- ▶ Message header and/or message body data
- ▶ Interfacing with other transports (for example MQSeries messages forwarded as SMTP messages)

Policy handlers can only be written in C.

## Platforms and languages currently available

Table 4-2 shows the platforms and languages where AMI is currently available.

Table 4-2 AMI language matrix

Operating System	AMI C/C++	AMI Java	AMI Cobol
IBM AIX	Yes	Yes	No
Sun Solaris	Yes	Yes	No
HP-UX (V11.0)	Yes	Yes	No
IBM OS/400	Yes	Yes	No
Microsoft Windows 2000	Yes	Yes	No
Microsoft Windows NT	Yes	Yes	No
Microsoft Windows 98	Yes	Yes	No
Microsoft Windows ME	Yes	Yes	No
IBM z/OS and OS/390	Yes	No	Yes

The AMI package can be downloaded as a SupportPac from the IBM Web site. The only exception to this is AMI for OS/390, which is built into the MQSeries for OS/390 V5.2 product.

### 4.2.7 AMI installation

**Note:** if you plan to install the SupportPac MA88 containing the Java and JMS classes for MQSeries, you should install it before installing the SupportPac MA0F containing the AMI support. Installing MA88 after MA0F may cause the AMI Java classes to no longer be included in the classpath.

In this section we describe the steps required to install the MQSeries Application Messaging Interface (AMI). AMI is available as an additional component. We describe here the installation for the Windows NT and 2000 platform:

1. Go to the following Web site:  
<http://www.software.ibm.com/ts/mqseries/txppacs/>
2. Select the SupportPac **MA0F**.
3. Click the **ma0f\_nt.zip** icon to download the MQSeries AMI for Windows 2000 in InfoZip compressed format.

- To install the AMI you need to uncompress the download file maof\_nt.zip into a temporary directory, make it current and then execute the setup.exe program.
- Click **Next** to install the AMI as shown in Figure 4-19.

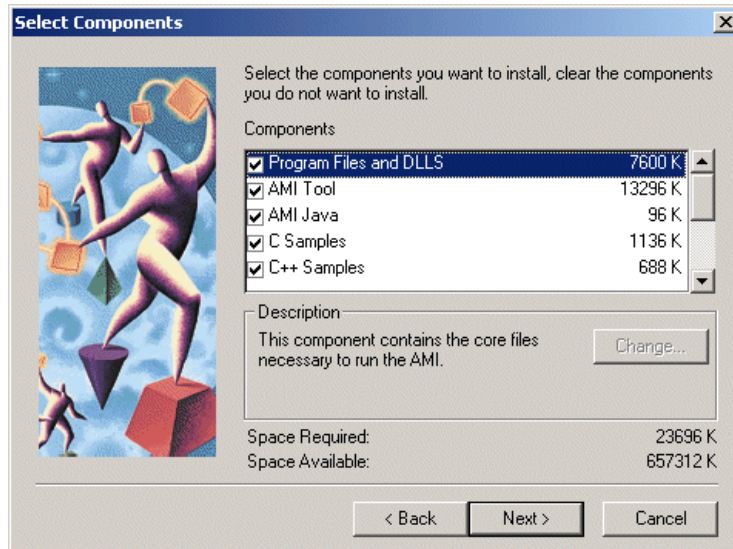


Figure 4-19 Installation of MQSeries AMI (1 of 6)

- If you don't want to type a new folder name, click **Next** to continue your installation with the standard program folder IBM MQSeries AMI as indicated in Figure 4-20.

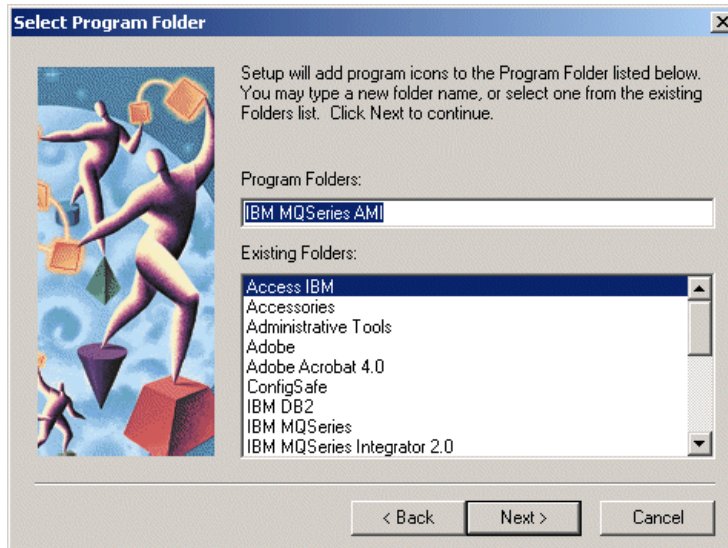


Figure 4-20 Installation of MQSeries AMI (2 of 6)

7. Choose **No** to continue as illustrated in Figure 4-21.

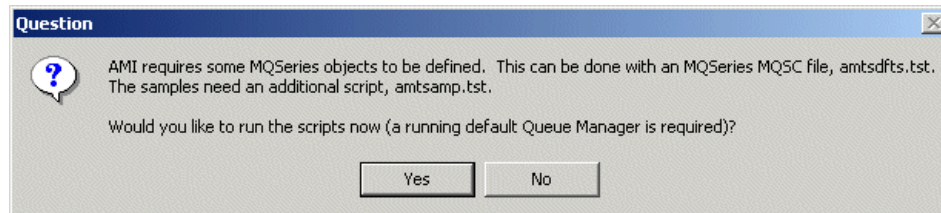


Figure 4-21 Installation of MQSeries AMI (3 of 6)

8. Click **OK** to continue your installation as displayed in Figure 4-22.

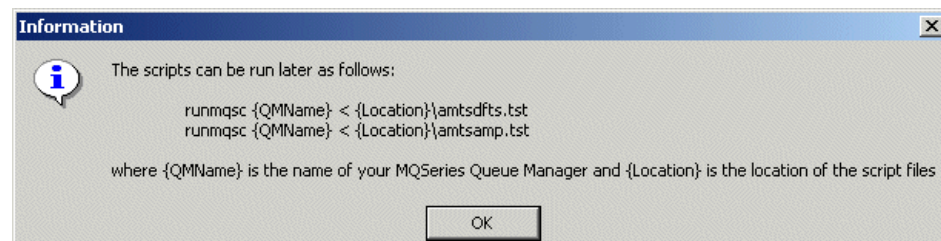


Figure 4-22 Installation of MQSeries AMI (4 of 6)



9. Click **OK** again to finish your installation of MQSeries Application Messaging Interface as shown in Figure 4-23.

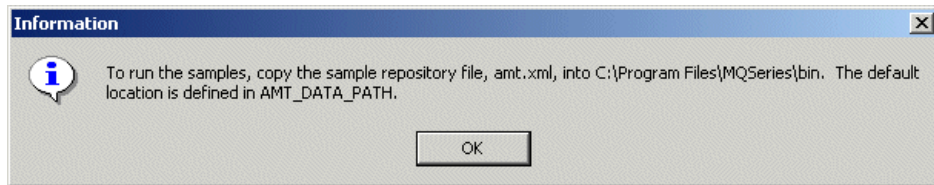


Figure 4-23 Installation of MQSeries AMI (5 of 6)

Choose **No, I will restart my computer later** to finish your installation, as indicated in Figure 4-24.

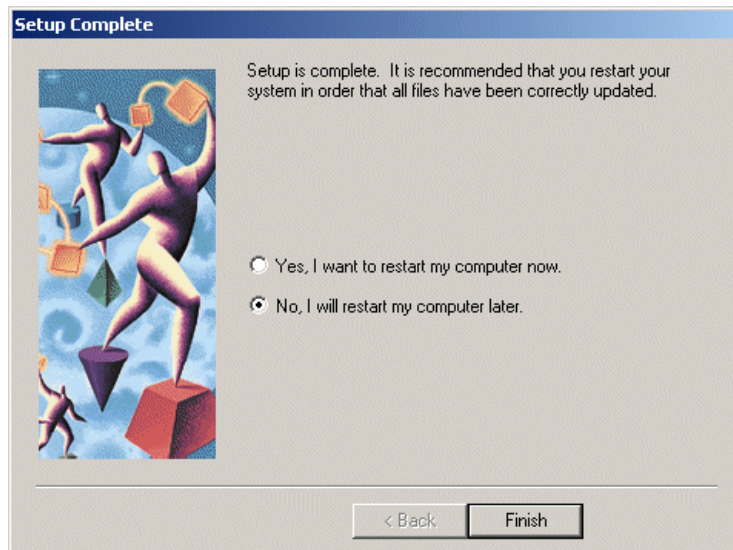


Figure 4-24 Installation of MQSeries AMI (6 of 6).

Once you have successfully installed the MQSeries Application Messaging Interface (AMI), you still need to set up the AMI sample script.

1. To set up the AMI sample script (amtsamp.tst) which is located in the subdirectory of your AMI standard installation you need to change the MQSeries subdirectory where your AMI was installed, for example C:\Program Files\MQSeries\amt\samples.
2. To run the sample script on the queue manager named YourQueueManager, use the following command: `runmqsc YourQMgrName < amtsamp.tst`
3. Your AMI system is now ready for your publish/subscribe applications.

## 4.2.8 AMI configuration

We use the Application Messaging Interface (AMI) Administration Tools for configuring the services and policies for our publish/subscribe application that we call *Global public transport system*. The AMI Administration Tools allow you to define AMI objects and save them in an AMI repository. This AMI repository is an XML file. It is by default called `amt.xml` and located in the directory `C:\Program Files\IBM\MQSeries\amt`, where `C:\Program Files\IBM\MQSeries` is the installation folder of MQSeries.

In this section, we describe how to define some of the AMI objects used by our application with the AMI Administration Tools but you don't have to go through all these steps for using our application. In the additional materials provided with this book, we provide an AMI repository file (`amt.xml`) containing all the definitions used by our application. Please see Appendix E, "Additional material" on page 211.

On the Windows 2000 platform, the AMI Administration Tools has two option panes. The left-hand navigation pane is used to select a customer application service (service points, distribution lists, subscribers and publishers) and customer application policy. The right-hand pane is used to set, update or display your input parameters for customer applications of the selected service or policy.

Using the AMI administration facility, we define and configure the publish/subscribe service and the policy. For the simplest version of our application, we performed the following tasks:

- ▶ Service for AMI application administrator:
  - Creation of the new service point for the Global public transport system application using the publish/subscribe facility.
  - Creation of the new subscribers for this application and parameter inputs for the sender service and receiver service.
  - Creation of the new publishers and parameter inputs for the sender service
- ▶ Policy for AMI application administrator:
  - Creation of the new policy for our transport application and specification of the following sets of application properties attributes: Initiation, general, send, receive, publish and subscribe.

To start the AMI Administration Tool, in the left pane of the Windows 2000 window, click **Start -> Program -> IBM MQSeries AMI -> IBM MQSeries AMI Administration Tool**. You see the first window of the AMI Administration Tool (Figure 4-25).

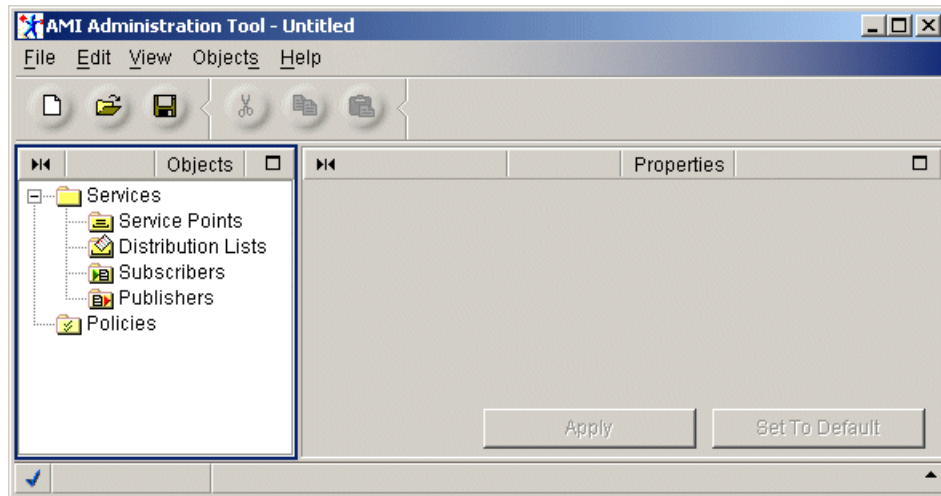


Figure 4-25 AMI administration Tool (1 of 18)

### Service point options for AMI application administrator

To create a new service point for your customer application, select **Service Points -> Create a new Service Point** (see Figure 4-26 on page 62).

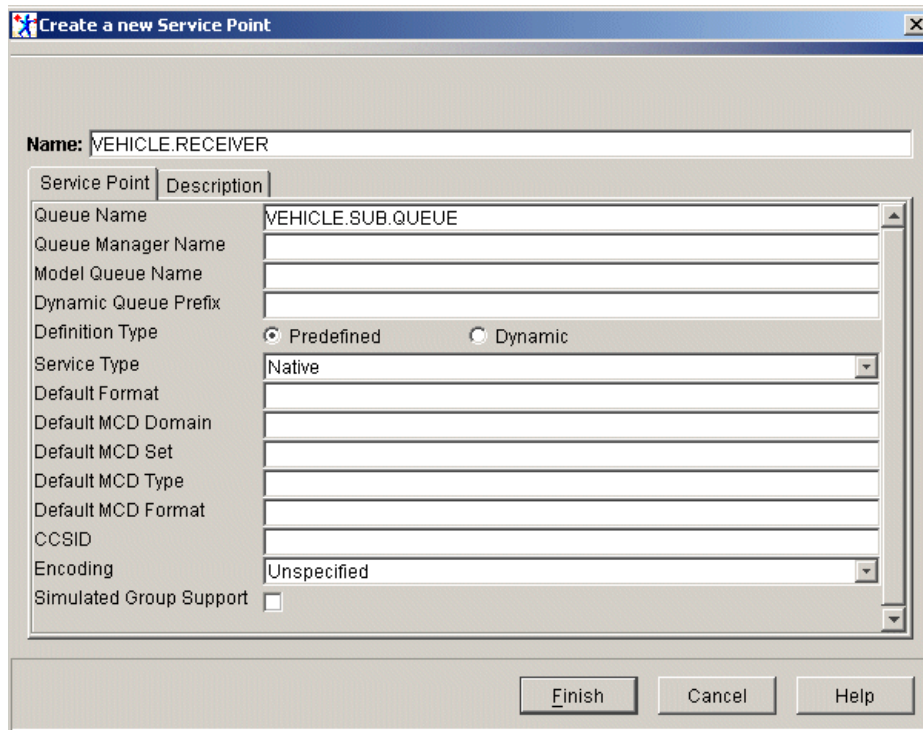


Figure 4-26 Create a new AMI service point (2 of 18)

We created the following new service points for our application (see Figure 4-27 on page 63):

- ▶ VEHICLE.RECEIVER
- ▶ VEHICLE.POSITION.PUBLISHER
- ▶ BROKER.CONTROL.QUEUE
- ▶ VEHICLE.ALERT.PUBLISHER

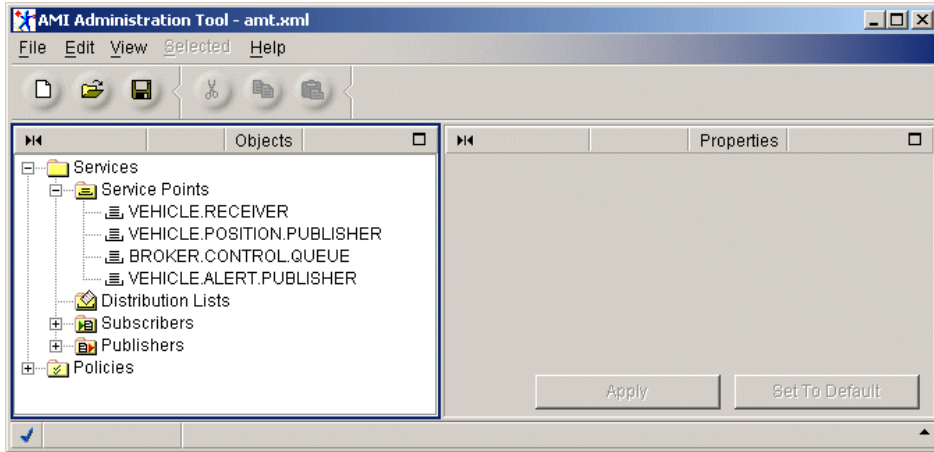


Figure 4-27 AMI administration for application (3 of 18)

We specified the application parameters for the service point of VEHICLE.RECEIVER (see Figure 4-28).

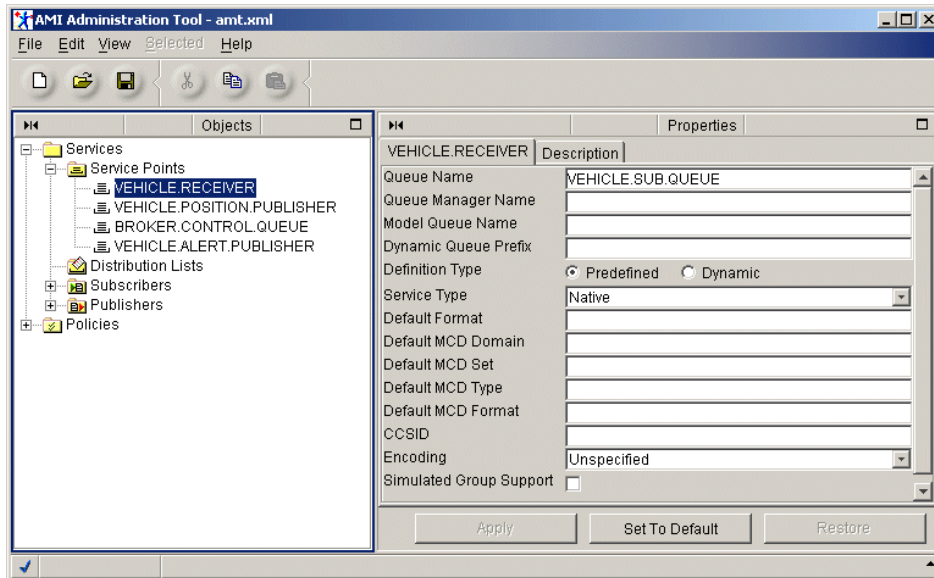


Figure 4-28 Parameters for AMI service point (4 of 18)

## Subscriber options for AMI application administrator

We created the new subscribers for our transport application and specified the input parameters for the sender service `BROKER.CONTROL.QUEUE` and receiver service `VEHICLE.RECEIVER` for the properties of our subscribers `VEHICLE.SUBSCRIBER` (see Figure 4-29).

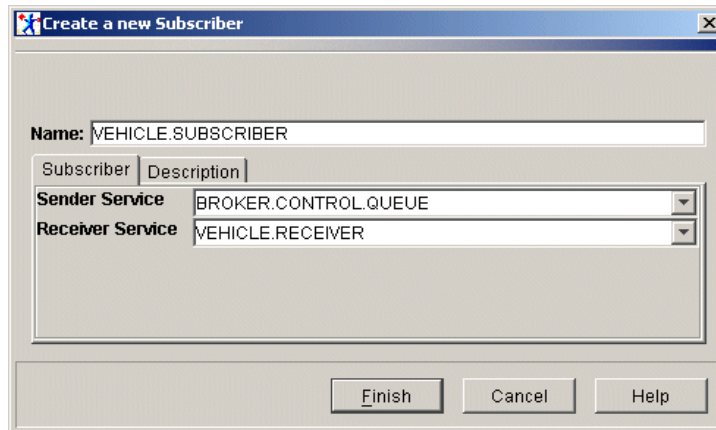


Figure 4-29 Create a new AMI subscriber (5 of 18)

For our transport application, we specified the following input parameters for the subscribers `VEHICLE.SUBSCRIBER` for our application: the sender service is `BROKER.CONTROL.QUEUE` and the receiver service is `VEHICLE.RECEIVER` (see Figure 4-30).

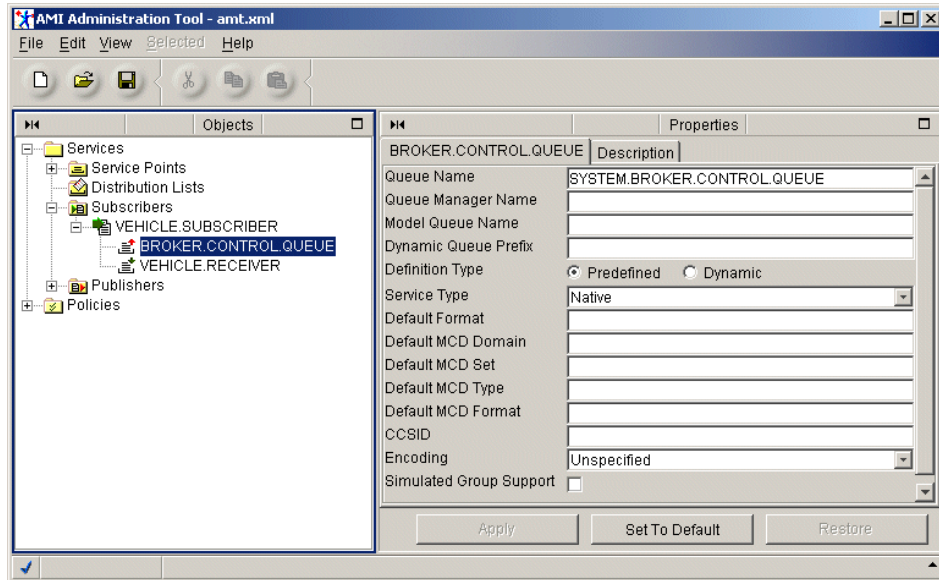


Figure 4-30 AMI Subscriber: sender service (6 of 18)

We specified the input parameters for the receiver service (see Figure 4-31).



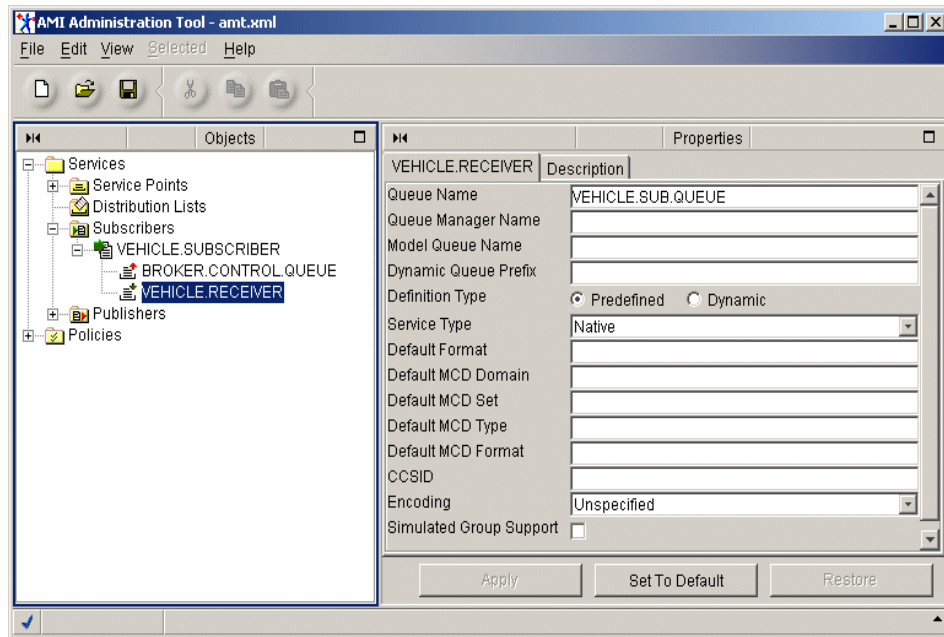


Figure 4-31 AMI Subscriber: receiver service (7 of 18)

### Publisher options for AMI application administrator

For our application, we created two new publishers (see Figure 4-32):

- ▶ VEHICLE.ALERT.PUBLISHER
- ▶ VEHICLE.POSITION.PUBLISHER

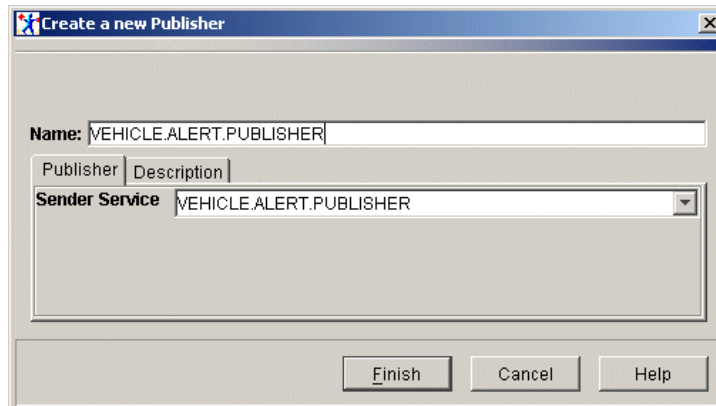


Figure 4-32 Create a new AMI publisher (8 of 18)



In this step, we created two new publishers for our application:  
VEHICLE.ALERT.PUBLISHER and VEHICLE.POSITION.PUBLISHER.

We created the sender service of the publisher VEHICLE.ALERT.PUBLISHER  
(see Figure 4-33).

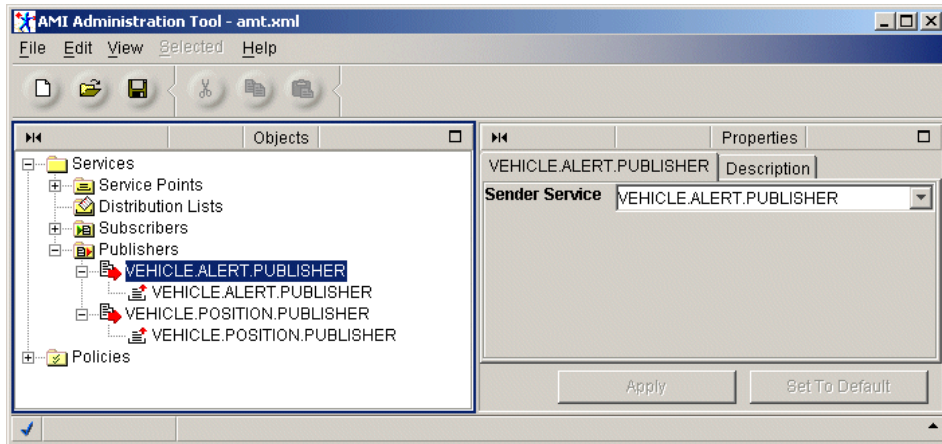


Figure 4-33 AMI Publisher: sender service (9 of 18)

We used the standard queue SYSTEM.BROKER.DEFAULT.STREAM for our  
sender service of the publishers VEHICLE.ALERT.PUBLISHER and  
VEHICLE.POSITION.PUBLISHER (see Figure 4-34).

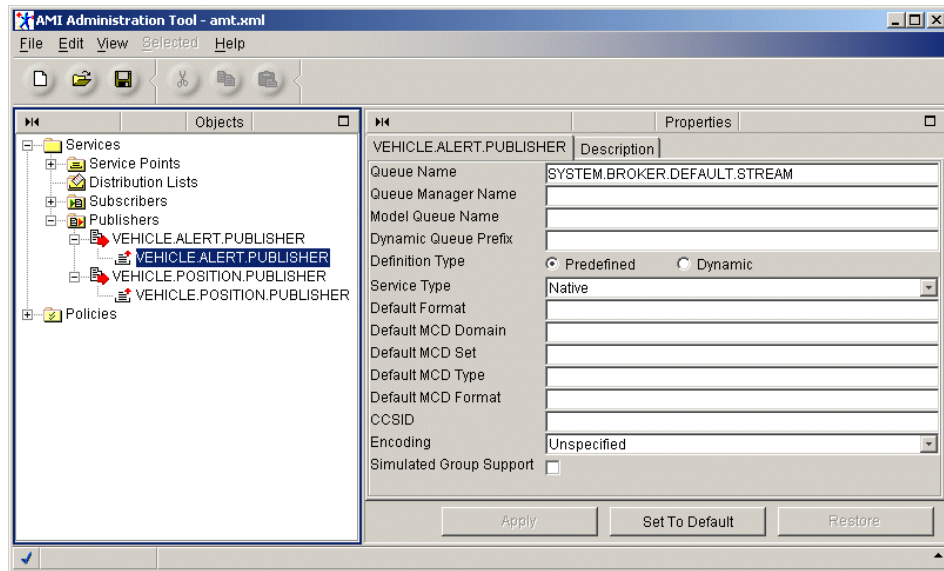


Figure 4-34 AMI Publisher: Parameters of sender service (10 of 18)

### Policy options for AMI application administrator

For our application called Global public transport system, we created three AMI policies (see Figure 4-35 on page 69):

- ▶ VEHICLE.SUB.POLICY
- ▶ VEHICLE.POSITION.PUB.POLICY
- ▶ VEHICLE.ALERT.PUB.POLICY

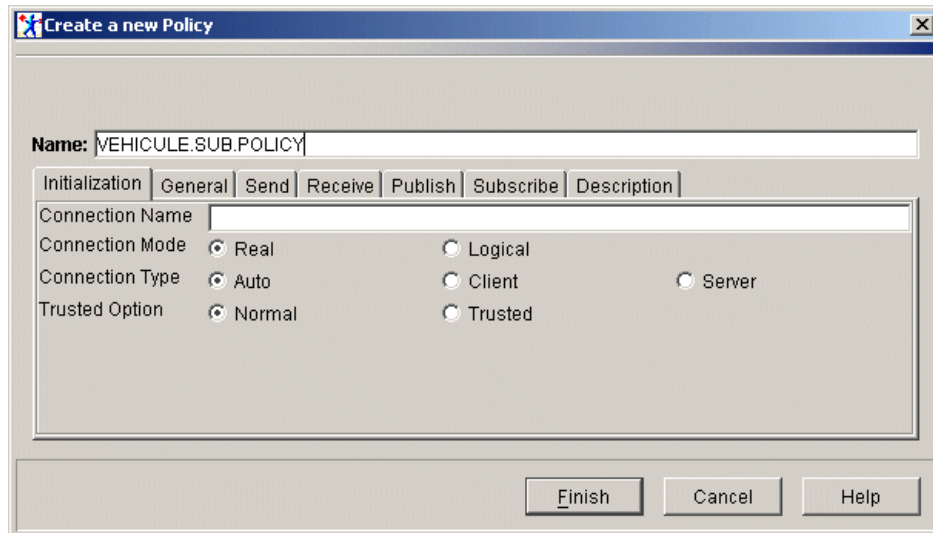


Figure 4-35 Create a new AMI policy (11 of 18)

We had three policies: VEHICLE.SUB.POLICY, VEHICLE.POSITION.PUB.POLICY, AND VEHICLE.ALERT.PUB.POLICY (see Figure 4-36).

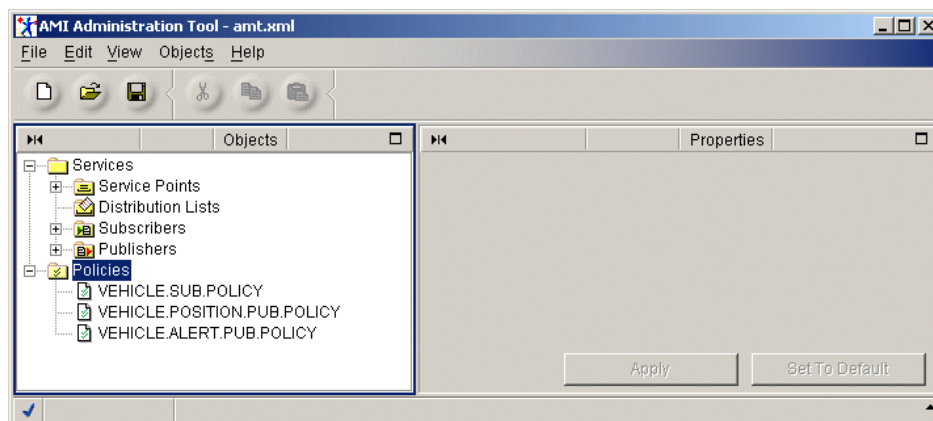


Figure 4-36 Parameters of AMI policy (12 of 18)

Each standard AMI policy has the following tabs: Initialization, General, Send, Receive, Publish, Subscribe and Description.

## AMI policy - initialization options for application

We set the parameters of the initialization options of the AMI policy for our application (see Figure 4-37).

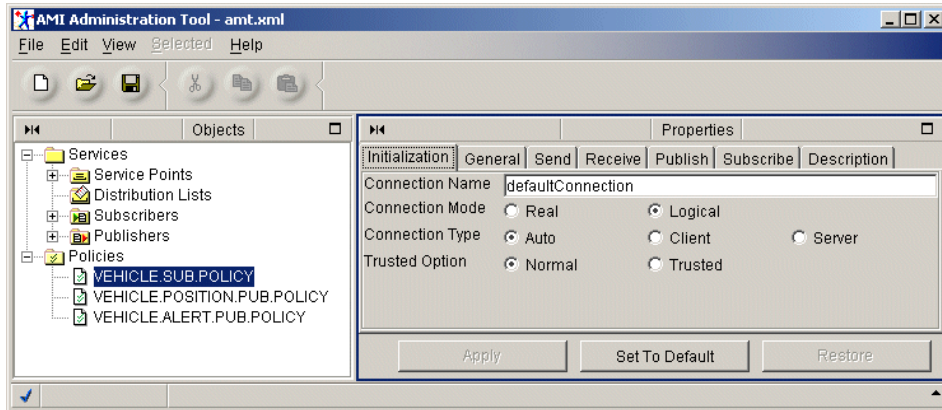


Figure 4-37 AMI policy: Parameters of initialization options (13 of 18)

## AMI policy - general options for application

We set the parameters of the general options of the AMI policy for our application (see Figure 4-38).

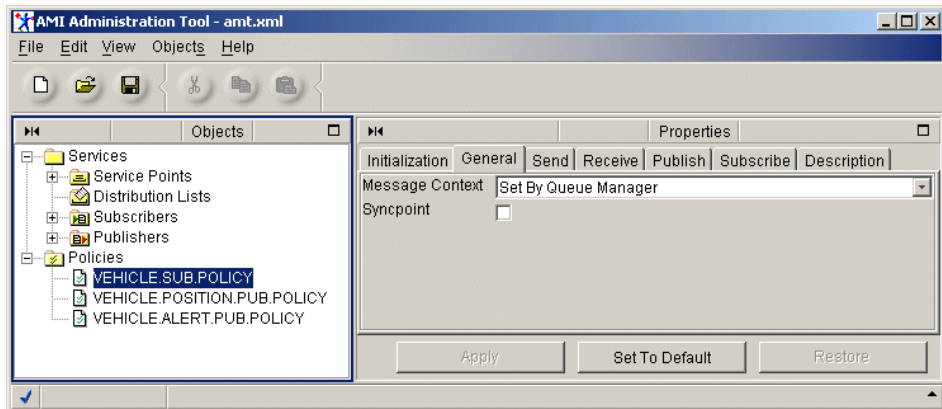


Figure 4-38 AMI policy: Parameters of general options (14 of 18)

## AMI policy - send options for application

We set the parameters of the send options of the AMI policy for our application (see Figure 4-39 on page 71).

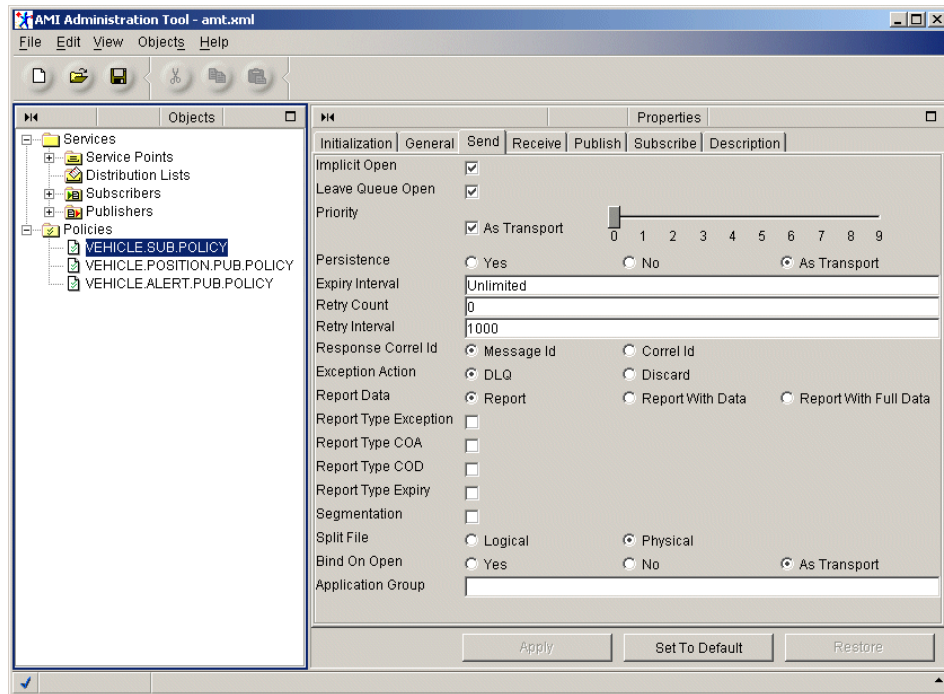


Figure 4-39 AMI policy: Parameters of send options. (15 of 18)

## AMI policy - receive options for application

We set the parameters for the receive options for the application AMI policy (see Figure 4-40 on page 72).



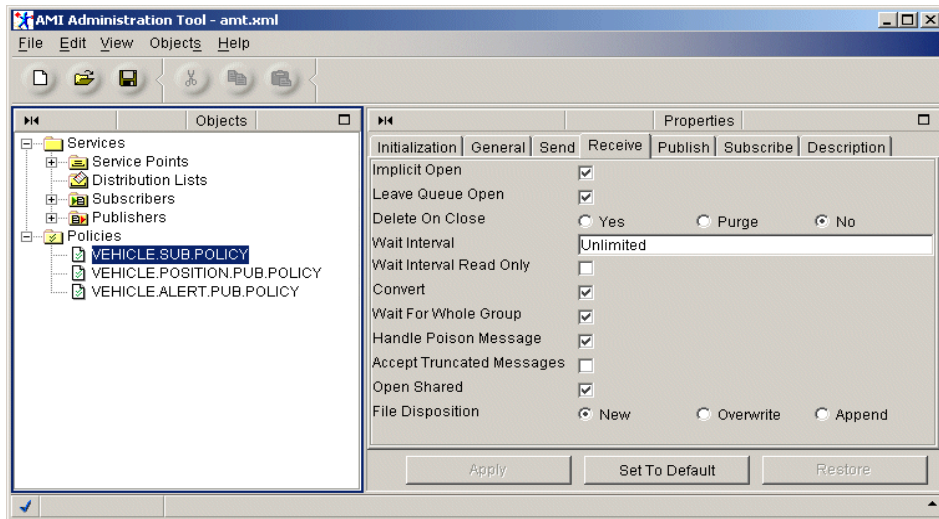


Figure 4-40 AMI policy: Parameters of receive options. (16 of 18)

## AMI policy - publish options for application

We set the parameters for the publish options for the application AMI policy (see Figure 4-41).

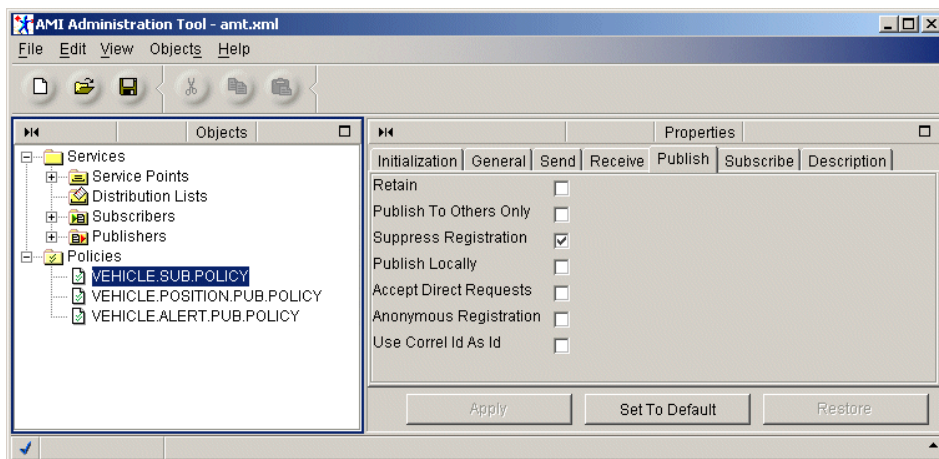


Figure 4-41 AMI policy: Parameters for publish options (17 of 18)

## AMI policy - subscribe options for application

We set the parameters for the subscribe options for the application AMI policy (see Figure 4-42 on page 73).

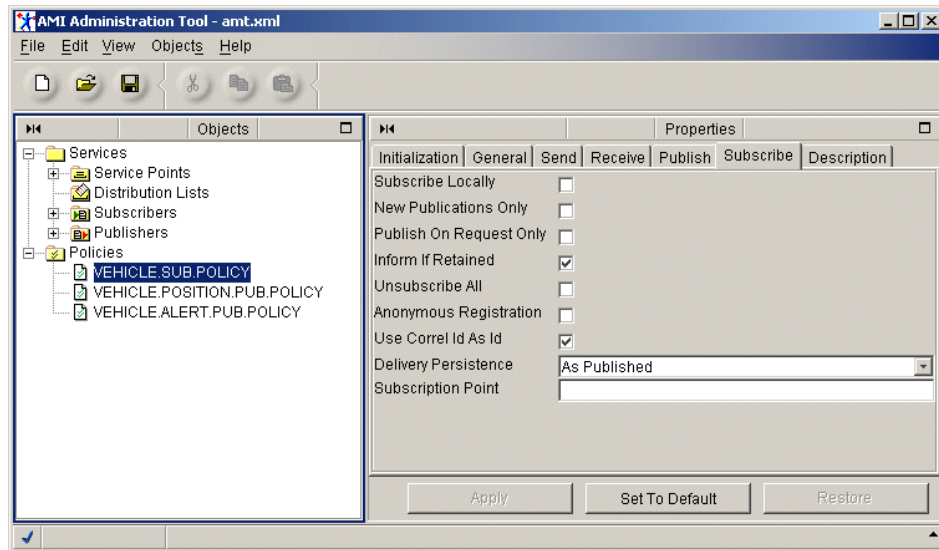


Figure 4-42 AMI policy: Parameters of subscribe options (18 of 18)

## 4.3 PubLauncher

The publication part of the application is a stand-alone Java application. It reads all of its initialization parameters from a properties file called `pubsub.properties` and starts a thread for each vehicle on each route. A thread publishes all the messages, alerts (accidents or breakdowns) and positions for this vehicle.

Figure 4-43 on page 74 illustrates the global flow of the publication application.

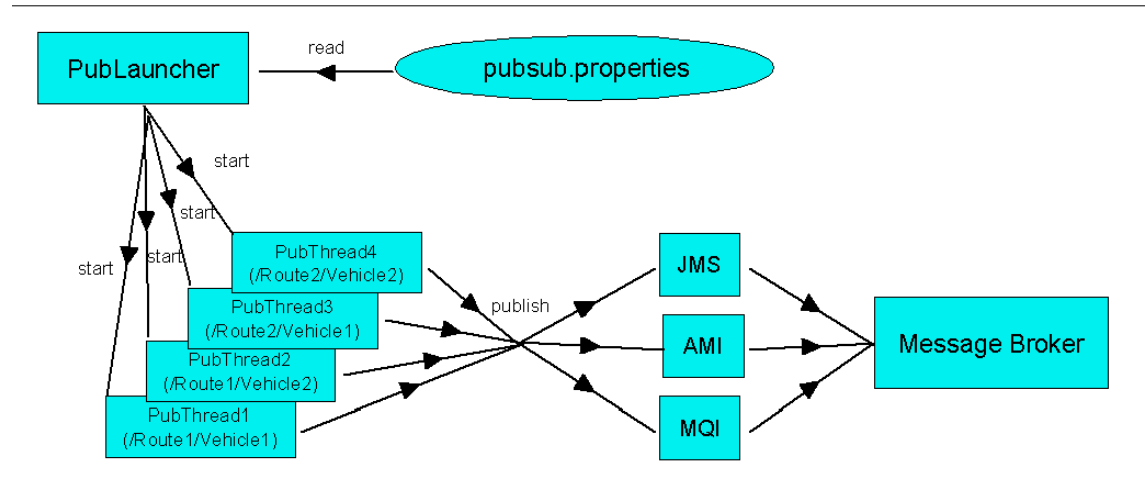


Figure 4-43 Application flow of the publication side

### 4.3.1 The properties file - pub.properties

This properties file contains the information specific to each publication module (JMS, C AMI, Java AMI, C MQI) and the parameters related to our business logic itself.

In this section, we discuss only the parameters related to the business logic and not the parameters specific to a publication method (JMS, AMI,...). These parameters are described in the section referring to the publication method itself.

Here are the entries found in the application part of the properties file:

```

pubType=1
nbrRoutes=3
nbrVehicles=3

##route1
route1=Tube/London/Piccadilly
route1NbrStops=4
route1TimeBetweenStops=4
route1Vehicle1=PiccaTrain01
route1Vehicle2=PiccaTrain02
route1Vehicle3=PiccaTrain03

##route 2
[similar to the definitions for route1]
  
```



`nbrRoutes` indicates the number of combinations between transport mode, geographical location and route we have created. Each route is then defined in the `routei` parameter, where *i* is the route number.

`nbrVehicles` is the number of vehicles that will serve each route. The design of our application only allows the same number of vehicles on each route.

For each route, we now have to define the route name, the number of stops, the time spent between two stops, and optionally the name of the vehicles.

`routei`, where *i* is the route number starting at 1, represents the route name.

`routeiNbrStops`, where *i* is the route number, is the parameter defining the number of stops (for example, bus stops) made by a vehicle on each route. Each route can have a different number of stops.

`routeiTimeBetweenStops`, where *i* is the route number, represents the time in seconds spent by a vehicle on its way between two stops.

Finally, each vehicle on each route can be given a name with the `routeiVehiclej` parameter, where *i* is the route number and *j* the vehicle number, all starting at 1. If no names are specified, names are automatically generated based on the hostname where the publication is running and on the current time.

### 4.3.2 PubLauncher coding logic

We don't explain the Java code used by the application extensively when it is not directly related to publish/subscribe functionalities. All the code can be found in Appendix E, "Additional material" on page 211. In this section we only show simplified code snippets when needed.

The PubLauncher is responsible for the three following tasks:

- ▶ Reading and parsing the properties file
- ▶ Starting the threads and providing them with the information from the properties file:

```
pubThread = new PubThread(info);  
pubThread.start();
```

- ▶ Waiting for each thread to complete and terminate:

```
pubThread.join();
```

### 4.3.3 Starting the publication application

The application uses AMI (Java and C), JMS (Java) and MQI (C).

To use AMI, the AMI repository and AMI host file must be present.

To use JNDI, a JNDI namespace containing the required JMS administered objects must be available.

The application itself is made up of the following files:

- ▶ The pub.jar file contains all the Java classes created for the publication part.
- ▶ The pub.bat file you must edit this file to indicate the installation directories of MQSeries, WebSphere (if using JNDI) and of the publication installation itself.
- ▶ The pub.properties file may be edited to suit your needs.
- ▶ The ptVehicleAMI.exe and ptVehicleMQI.exe files are placed in a directory included in the path. On Windows NT and 2000, you can put them in C:\WINNT for instance.

After editing the pub.bat and pub.properties files, if necessary, run pub.bat to start the application.

## 4.4 PubThread

Each PubThread represents a vehicle on a route. The PubThread Java class extends the thread Java class. It receives the following information before starting:

- ▶ Business information:
  - The route and the vehicle it represents.
  - The number of stops along the route.
  - The time spent by the vehicle between two stops.
- ▶ Publication information:
  - The publication method chosen (JMS, Java AMI, C AMI, C MQI).
  - The needed information relative to publication method (repository information for AMI, JNDI information for JMS, etc.). This point is covered in more detail in the section describing each publication method.

Once all PubThreads are created, they are started by the PubLauncher. At each stop, the thread publishes a position message and an accident or breakdown message depending on the situation.

We must make a distinction according to which publication method has been chosen in the properties file, either C (AMI or MQI) or Java (JMS or AMI).

In the first case, the PubThread calls a C executable file (ptVehicleAMI.exe or ptVehicleMQI.exe). PubThread consists here essentially of a Java wrapper around a C program. The logic of message generation and publication is delegated to the C program.

In the latter case, the PubThread creates the publication messages and publishes them directly through a JMS or AMI helper Java class.

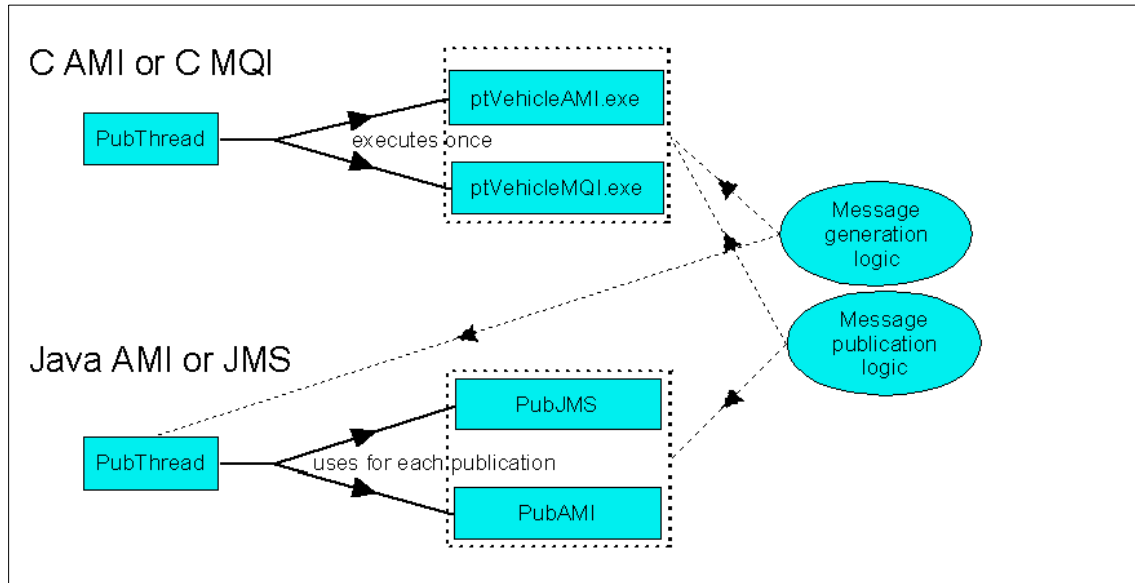


Figure 4-44 PubThread

Figure 4-44 illustrates the distinction of the logic flow for the generation of messages between the C and the Java implementation.

We have chosen to implement the publication application this way for simplicity. In our case, there is only one call to a C program in the thread (that is along a vehicle route) lifetime. If we had implemented C AMI or C MQI in the same way as Java AMI or Java JMS, we would have needed either to call a C program each time we wanted to publish a message, with an extra overload as a result, or we would have had to call a function in a C DLL file through the Java Native Interface (JNI), with a much more complex solution that was clearly outside the scope and focus of this book.

In the following subsection, we describe the class PubThread started by the PubLauncher.

## 4.4.1 PubThread coding logic

There are two important times in the lifetime of PubThread: when it is created and when it is started by the PubLauncher.

At creation time, that is in its constructor method, the thread receives the required information regarding the business logic (the vehicle and the route it represents, the characteristics of this route) and the publication logic (the publication method used, C or Java, AMI, JMS or MQI, and the information specific to the method used). Depending on the publication method chosen, the thread creates the appropriate publication helper class as shown in the following code:

```
switch (pubSubType) {
    case Const.PUBSUB_JAVA_AMI: {
        pubJava = new PubJavaAMI(amiInfo);
        break;
    }
    case Const.PUBSUB_C_AMI: {
        pubC = new PubCAMI(cInfo);
        break;
    }
    case Const.PUBSUB_JMS: {
        pubJava = new PubJMS(jmsInfo);
        break;
    }
    case Const.PUBSUB_C_MQI: {
        pubC = new PubCMQI(jmsInfo);
        break;
    }
}
```

At runtime, the thread is executing its run() method. This method distinguishes between C publication and Java publication. For an easier extension of the code, we created two Java interfaces PubJava and PubC, that are respectively implemented by the classes PubJMS, PubJavaAMI responsible for the JMS and Java AMI functionalities, and by the classes PubCAMI and PubCMQI, used for the C AMI and C MQI functionalities. While executing, the thread calls the method directly from the interface PubJava and PubC without worrying about which class is really doing the job.

If a C publication method is chosen, the thread delegates the job to the C program with the command **pubC.exe(pubData)**. This exe() method is using the runtime of the Java Virtual Machine (JVM) to call an external program.

If a Java publication is used, the thread calls the PubJava interface methods:

- ▶ `PubJava.open()` to start the connection and session with the messaging service provider (in MQSeries terms, to get a connection handle to the queue manager).
- ▶ At each stop, it uses `PubJava.publishPosition()` to send a publication indicating the position of the vehicle and if one accident or breakdown has occurred, it uses the `PubJava.publishAccident()` or `PubJava.publishBreakdown()` to send an alert publication. The messages used are described in more details in 4.5, “The publication messages” on page 79.

## 4.5 The publication messages

In the design of our application, we decided that each vehicle publishes three different XML messages: one message to indicate its position, another message when it has an accident, and a third message when the vehicle has broken down.

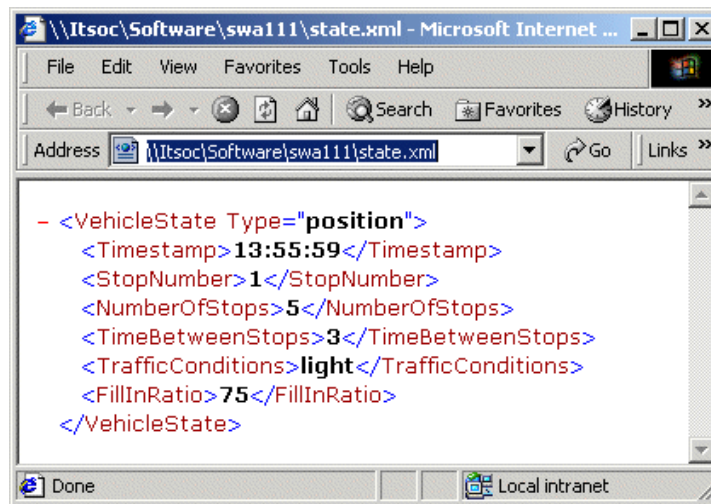


Figure 4-45 The position publication message

In Figure 4-45, we see the message generated by the publication application when a vehicle arrives at a given stop. The message indicates at what time the vehicle has reached which stop and it also contains some complementary information. The message itself contains no information regarding the vehicle or the route since it is published to a topic specific to this vehicle and route.

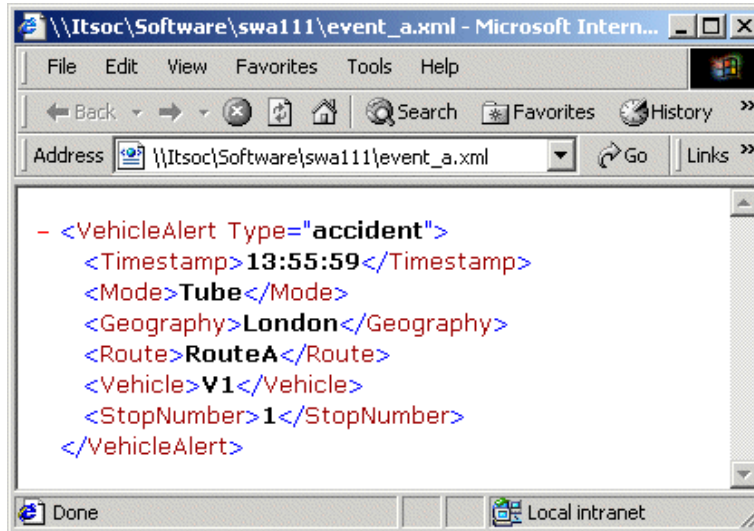


Figure 4-46 The accident publication message

In Figure 4-46, we see the accident message generated when a vehicle has had an accident on his route. It contains the name of both the vehicle and the route, because this message will be published to a general topic for all the accidents.

The breakdown publication message is very similar to the accident message. Only the Type attribute is changed, from accident to breakdown.

## 4.6 Publishing in C

We provide two distinct versions of the Vehicle programs in C. One is written to the C AMI API, and the other is written to the C MQI API.

The Vehicle program's main logic is owned by module ptVehicle.c, while support functions can be found in ptVehUty.c.

Publish/Subscribe logic is implemented by three functions: PSInitialize(), PSPublish() and PSTerminate().

The AMI implementation of these three functions is included in module ptVehAMI.c, while the MQI implementation is included in ptVehMQI.c.

A Microsoft Visual Studio project named ptVehicleAMI.dsp is used to build ptVehicleAMI.exe out of source modules: ptVehicle.c, ptVehUty.c, ptVehAMI.c.

A project named ptVehicleMQI.dsp is used to build ptVehicleMQI.exe out of source modules: ptVehicle.c, ptVehUty.c, ptVehMQI.c.

Both executables can be run from the command line passing the following parameters: Mode, Geography, Route, Vehicle, NumberOfStops, and TimeBetweenStops. See Example 4-1 and Example 4-2.

*Example 4-1*

---

```
c:\tmp>ptVehicleAMI Tube London Piccadilly UTrain001 10 5
```

---

*Example 4-2*

---

```
c:\tmp>ptVehicleMQI Trains London HogwartsExpress WizTrain077 20 6
```

---

Typically the programs will be run via the program PubLauncher, which is driven by a properties file, so none of the above parameters must actually be specified explicitly.

The programs write diagnostic information to the window, but sometimes it is better to have the same information written to a file. In order to obtain this, you should define an environment variable named VEHICLE\_TRACE (any non-null value can be assigned to this variable) prior to program execution.

File with names such as ptVehicle.processid.trc will be generate in the program current directory.

In the AMI version of the Vehicle program you can specify the name of the queue manager to use as a broker via the AMI repository or the AMI host file. The same does not apply to the MQI Vehicle program, so in order to keep the input parameters identical across program versions, ptVehicleMQI.exe will send messages to the default queue manager, or to the queue manager specified in the environment variable VEHICLE\_QMGR if defined.

In the following sections we will discuss the AMI and MQI implementation of the three main publish/subscribe related functions: PSInitialize(), PSPublish() and PSTerminate().

## 4.6.1 Vehicle C AMI program

The main distinction of the C AMI programming interface is that it is actually made of two separate sets of functions: the high-level interface and the lower-level object style interface. Calls from the two sets can be mixed in the same program.

**Restriction:** C AMI calls and C MQI calls cannot be issued from the same operating system process (even from different threads).

With the high-level interface calls, you refer to AMI entities such as Publisher and Policy by name. On the other hand the lower-level interface is based upon object handles.

Several utility calls are provided to get the handle associated to a named repository entity (for example `amSesGetPublisherHandle()` and `amSesGetPolicyHandle()`).

Our example program is simple enough to be written using calls from just the high-level interface.

## Publishing information

In every AMI application the first AMI function to call is `amInitialize()`, and the last one is `amTerminate()`.

In our Vehicle program these two calls are wrapped by `PSInitialize()` and `PSTerminate()`.

`amInitialize()` will return a session handle that is to be used in any subsequent AMI calls.

The only other call that we need to publish information is `amPublish()`, that is wrapped by `PSPublish()` in module `ptVehAMI.c`.

The `amPublish()` call issued by the program is shown in Example 4-3. Being a high-level call, the only handle needed is the session handle returned by `amInitialize()`.

*Example 4-3 The `amPublish()` call*

---

```
fSuccess = amPublish(hSession,          /* session handle */
                    pszPublisher,       /* publisher name  */
                    pszPolicy,         /* policy name     */
                    NULL,              /* no response expected */
                    strlen(pszTopic),   /* topic name length */
                    pszTopic,          /* topic name      */
                    strlen(pszMsg),     /* length of data   */
                    pszMsg,            /* publication data */
                    NULL,              /* publish message name */
                    &lCC,              /* completion code  */
                    &lRC);             /* reason code     */
```

---



The other parameters specify:

- ▶ The names of repository entities such as Publisher and Policy
- ▶ The topic on which the message is being published
- ▶ The body of the message

Two of the parameters have been left NULL; the first one is `amResponseName`. If an AMI Receiver is passed in this parameter, then the `amPublish()` API internally requests an acknowledgment by the broker that the publication data has been successfully processed. The importance of this feature greatly depends on the business scenario and the importance of the messages being published. In the sample program, we decided not to exploit it.

The second parameter being omitted is `pubMsgName`. This parameter can be used to indicate the name of an AMI Message object containing the header of the message being published and possibly the body itself (in case the length of the data passed on `amPublish()` is zero).

In order to populate an AMI Message header (for example adding extra topics or a publish/subscribe content filter), lower-level AMI calls are needed (for example `amMsgAddElement()`).

Given that our example does not have any special message header requirements, we can omit the `pubMsgName` parameter.

This example showed how you can publish information with just one simple `amPublish()` call, disregarding any middleware-related details such as:

- ▶ Queue manager connection and disconnection
- ▶ Queue opening and closing
- ▶ Broker command syntax
- ▶ Administration of environment-dependent information (for example queue manager names and queue names read from `.INI` files)

This extreme level of simplicity was obtained by the combination of benefits coming from the usage of a comprehensive external repository by AMI, and the high-level AMI C calls that automatically take care of mundane tasks such as timing the opening and closure of the relevant queue manager queues.

## Compiling and linking

AMI applications need to include `amtc.h` file. This is a self-contained definition file; no other MQSeries include file is needed.

For the link step, the import library named `amt.lib` must be added to your project library list.

## 4.6.2 Vehicle C MQI program

MQI programming interface does not provide any specialized verbs for publish/subscribe. Nevertheless it is possible to use basic MQPUT() and MQGET() verbs to interact with the broker, as long as certain conventions are followed.

Given the restriction according to which AMI calls and MQI calls cannot coexist in the same process, sometimes MQI-based publish/subscribe is the only option available to enable publish/subscribe messaging for applications already written to the MQI API.

In MQI-based publish/subscribe messaging, the following functions must be explicitly coded by the programmer:

- ▶ Queue manager connection and disconnection
- ▶ Queue opening and closing
- ▶ Formatting of publication or request messages in the correct broker syntax
- ▶ Parsing of broker responses or received publications according to the broker syntax
- ▶ Administration of environment-dependent information (for example queue manager names and queue names read from .INI files)

### **Anatomy of an MQRFH formatted message**

The most demanding chore is by no means the parsing and formatting of messages. In fact, even if the payload contained in a publication is completely opaque to the broker (that is, it is being passed to subscribers as an unchanged bitstream) it must be preceded by an MQRFH header (*rules and formatting header version 1*). See Figure 4-47 on page 85.

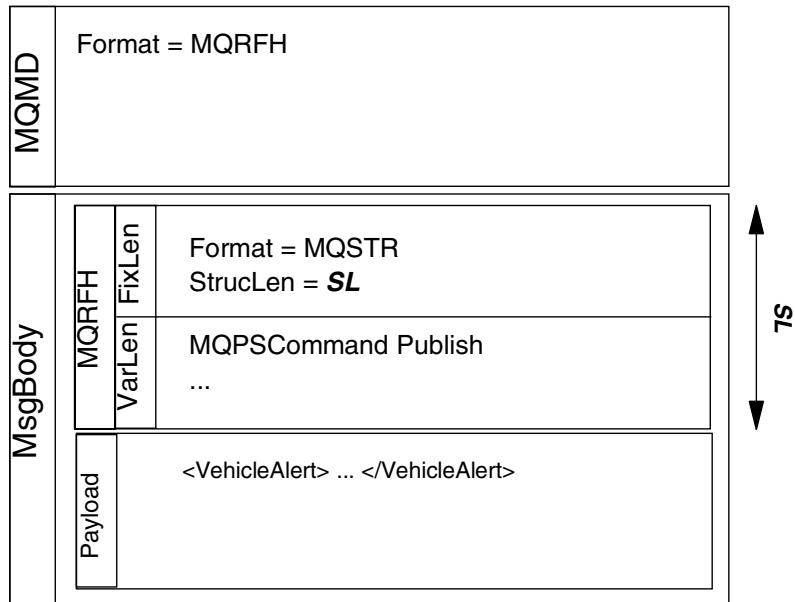


Figure 4-47 Anatomy of an MQRFH message

The same holds for messages generated by the broker as responses to commands or as forwarded publications.

Let's now take a closer look at the anatomy of a message in MQRFH format:

- ▶ There are three fields of the message descriptor that are relevant to our discussion:
  - Format: this is the name identifying the format of the message body. In this case, it is the constant value "MQHRF...".
  - Encoding: this is the encoding of the numeric information included in the message body (which is in MQRFH format).
  - CodedCharSetId: this is the coded character set identifier of the string information included in the message body (which is in MQRFH format).

- ▶ The MQRFH formatted message body has three parts:
  - Fixed-length part (the following is the formal C definition):

```
typedef struct tagMQRFH {
    MQCHAR4  StrucId;          /* Structure identifier */
    MQLONG   Version;         /* Structure version number */
    MQLONG   StrucLength;     /* Total length of MQRFH including string
                              containing name/value pairs */
    MQLONG   Encoding;       /* Numeric encoding of data that follows
                              NameValueString */
    MQLONG   CodedCharSetId; /* Character set identifier of data that
                              follows NameValueString */
    MQCHAR8  Format;          /* Format name of data that follows
                              NameValueString */
    MQLONG   Flags;          /* Flags */
} MQRFH;
```

Where:

- Format: this is the name identifying the format of the message payload, in this case it is the constant value “MQSTR...“.
  - Encoding: this is the encoding of any numeric information included in the message payload.
  - CodedCharSetId: this is the coded character set identifier of any string information included in the message payload.
  - StrucLength: total length of MQRFH fixed-length part and variable-length part (that is, excluding the message payload).
  - Flags: this field is not currently being used and must be set to zero.
- Variable-length part: the variable-length string of name/value pairs separated by blanks. This string can also be used to accommodate any attribute-tagged user data, provided that the attribute names do not begin with MQPS. MQPS tags are reserved for MQSeries Publish/Subscribe and their meaning is documented in *MQSeries Publish/Subscribe User's Guide*, GC34-5269.
  - Message payload: this is the application data that is being published or received as a publication. Its format is stated in the relevant attribute in the fixed-length part of the MQRFH header. The application data must start on a word boundary, so there can be some extra bytes between the end of the name/value string and the start of the real data. Ensure that any extra bytes are either binary zeros or blanks. Please note that the recommendation is to use a 4-byte word boundary.

**Tip:** In order to accommodate all platform requirements, ensure that the application data starts on a 16-byte boundary.

## Publishing information

Let's now discuss the code contained in module `ptVehMQI.c`.

The function `PSInitialize()` does not contain any real publish/subscribe-related code. It connects to the default queue manager (or to the one pointed to by the `VEHICLE_QMGR` environment variable) and opens the broker's default stream queue. This name is well known (`SYSTEM.BROKER.DEFAULT.STREAM`) and can be safely hardcoded in the program.

The function `PSTerminate()` closes the opened broker queue and disconnects.

The interesting code is all contained in the `PSPublish()` function, where the following steps are executed:

1. Initialize the first part of the message body area with a default `MQRFH`.
2. Override defaults for `Format`. We send XML so it will be `MQFMT_STRING`.
3. Override the default for `CodedCharSetId`. We inherit this attribute from the queue manager.
4. Compose a name/value string. Add topic and publish command details in the message body section following the `MQRFH` structure treating data as a zero-terminated string (see Example 4-4 on page 87 for one of the generated name/value strings).

### *Example 4-4* `PSPublish()` function parameters

---

```
MQPSCommand Publish
MQPSTopic   PublicTransport/Alerts/Accidents
MQSPubOpts NoReg
```

---

5. Compute and update `MQRFH StrucLength` by adding the `MQRFH` structure size, the length of the name/value string, and any extra bytes needed to align the application data on a word boundary.
6. Set the message descriptor `Format` to `MQFMT_RF_HEADER`.
7. Set the message descriptor `MsgType` to `MQMT_DATAGRAM` (we are not expecting any response from the broker).
8. Finally call the MQI `MQPUT()` verb to send the publication message. The `BufferLength` passed to `MQPUT()` is the sum of `MQRFH` fixed and variable parts plus the message payload sizes.

**Note:** When explicitly formatting an MQRFH in C, remember that the resulting bitstream is *not* a C zero terminated string, but it is a binary array of characters. Applying standard string library functions such as `strcpy()` may lead to unpredictable results.

**Tip:** When publishing without requesting an acknowledgment, it can be tricky to troubleshoot a bad published message, so it is recommended that you define a queue named `TMP.BROKER.REPLY` and insert the following temporary debug code just before the `MQPUT()` call:

```
md.MsgType = MQMT_REQUEST;  
strncpy(md.ReplyToQ, "TMP.BROKER.REPLY", MQ_Q_NAME_LENGTH);
```

The broker response messages containing information about the error will be put to the `TMP.BROKER.REPLY` queue, and can be displayed with a suitable tool (for example the MQSeries Explorer tool included in the MQSeries for Windows 2000 product).

## Compiling and linking

On top of customary MQI include files, the file `cmqpsc.h` can be included, in order to access constant definitions for well-known name/value tags. Linkage of the application is not influenced by publish/subscribe so the standard MQI rules apply.

In general, any MQI language implementation can be used to implement publish/subscribe messaging (even if no equivalent of `cmqpsc.h` is provided by IBM).

For instance a Microsoft Visual Basic application using the MQSeries COM+ interface can interact with an MQSeries Publish/Subscribe broker, as long as the format conventions discussed are respected.

## 4.7 Publishing in Java

In this section, we explain how to publish a message to a topic in Java, either with Java AMI or with JMS, and we illustrate it with our application.

To publish a message either with Java AMI or with JMS, our `PubThread` uses the following `PubJava` interface:

```
public interface PubJava {  
    public void cleanup();  
    public void open(com.ibm.itso.swa111.PubData pubData);
```

```

public void publishAccident(com.ibm.itso.swa111.PubData pubData);
public void publishBreakdown(com.ibm.itso.swa111.PubData pubData);
public void publishPosition(com.ibm.itso.swa111.PubData pubData);
}

```

The `open()` method is intended to open the session to the messaging service provider (either a JMS `TopicSession` or an AMI `AmSession`) and to create the objects depending on that session.

The purpose of the `cleanup()` method is to undo what has been done in the `open()` method, to release to messaging service provider resources, and to allow a garbage collecting of the used Java resources.

The three `publishXXXXXX()` methods we've just discussed are used to publish a message to a given topic based on the information contained in the data bean `pubData`.

In the next two sections, we discuss how a publication is sent to a and illustrate it by showing how the `PubJMS` and `PubJavaAMI` classes are implemented this `PubJava` interface.

## 4.7.1 Publishing in JMS

The JMS concepts were introduced in 4.2.3, “JMS overview” on page 36 . For a more systematic description of JMS and the JMS API, please refer to *Using Java*, SC34-5456. In 4.2.4, “JMS configuration, JNDI and JMSAdmin” on page 39, we discussed the JNDI aspects of JMS and they are not covered again in this section. Here we concentrate on the coding aspects of JMS.

1. The first step in a JMS program is to import the required packages:

```
– import javax.jms.*
```

This is the JMS interface already described.

```
– import javax.naming.* and import javax.naming.directory.*
```

These are the two packages required for JNDI.

2. We can now retrieve the information that has been stored in JNDI, that is create our `TopicConnectionFactory` and topics.
  - a. We create a hash table, put the information relative to our JNDI service provider, and get our initial context:

```

Hashtable env = new Hashtable();
env.put( Context.INITIAL_CONTEXT_FACTORY, jmsInfo[0] );
env.put( Context.PROVIDER_URL, jmsInfo[1] );
env.put( Context.REFERRAL, jmsInfo[2] );
ctx = new InitialDirContext( env );

```

When using VisualAge for Java as the Persistent Name Server, only the first two environment variables are used, with a value as follows:

```
contextFactory=com.ibm.ejs.ns.jndi.CNInitialContextFactory  
initURL=iiop://hostname/
```

These values are defined in the pub/sub properties file used by our application.

The REFERRAL variable is only used when the JNDI service provider is an LDAP server.

- b. Once we have our initial context, we can look it up for JMS administered objects that were stored in it with the JMSAdmin tool.

```
tcf = (TopicConnectionFactory)ctx.lookup(jmsInfo[3]);  
topicAccident = (Topic)ctx.lookup(jmsInfo[4]);  
topicBreakdown = (Topic)ctx.lookup(jmsInfo[5]);  
tConn = tcf.createTopicConnection();
```

We also create immediately the TopicConnection, since it doesn't cause an MQSeries connection to the queue manager to be opened at that time. The names of the TopicConnectionFactory and of the topics stored in JNDI are defined in the pub/sub properties file used by our application:

```
jndiTopicConnectionFactory=jms/ITSOPSBND  
jndiTopicAccident=jms/Accident  
jndiTopicBreakdown=jms/Breakdown
```

3. From the TopicConnection we have, we create a TopicSession and from it all the other JMS objects needed:

```
tSess = tConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE );
```

The parameter false indicates that we don't use transactions in this session.

```
publisherAccident = tSess.createPublisher(topicAccident);  
publisherBreakdown = tSess.createPublisher(topicBreakdown);
```

When the topics to which we will publish are well-known beforehand, we can store and retrieve them from JNDI and use them to create publishers so that you only need to use their publish method when publishing, without having to recreate them each time. On the other hand, if the topic is not known before runtime, as is the case when we publish the position of each vehicle to a topic specific for each vehicle, then we can dynamically create this topic from the session based on the String name of the topic and create a publisher for this topic as shown in the following code:

```
topicPosition = tSess.createTopic(pubData.topic);  
publisherPosition = tSess.createPublisher(topicPosition);
```

JMS provides various types of messages. In our application we are only using XML publications and only need messages of type TextMessage.

```
msgAccident = tSess.createTextMessage();
```



```
msgBreakdown = tSess.createTextMessage();
msgPosition = tSess.createTextMessage();
```

4. The last step to publish a message is to write the text into the message and then publish it:

```
msgPosition.setText(content);
```

where content is our XML message.

```
publisherPosition.publish(msgPosition);
```

After all the messages have been published, we can now close all open resources.

5. The resources we have to close are the TopicPublisher, the TopicSession and the TopicConnection:

```
publisherBreakdown.close();
publisherAccident.close();
publisherPosition.close();
tSess.close();
tConn.close();
```

These were all the steps needed to publish a message with JMS. We now follow a similar approach for publishing with Java AMI.

### **JMS limitations**

The JMS specification could not include all the functionalities and features offered by the existing messaging services providers or messaging-oriented middleware. Its main aim is to maximize portability by providing a simple standard interface including the most important features. In that respect, it doesn't cover all the functionalities included in MQSeries in general, and in MQSeries Publish/Subscribe in particular, such as:

- ▶ Publication of messages in retained mode (that is, full support for state publications)
- ▶ Publication/subscription in a broker network-aware fashion (for example specifying the scope of the publication/subscription).
- ▶ Content-based subscription (JMS support a form of message filtering that is managed by the client, content based subscription are instead enforced directly by the broker).

## **4.7.2 Publishing in Java AMI**

This section is not a detailed description of how AMI works. It is only a description of how to use Java AMI to publish messages.

1. The first step in an AMI program is to import the required AMI package:

```
import com.ibm.mq.amt.*;
```

2. The next step is to retrieve the information stored in the AMI repository and create the AMI objects based on this information.

We start by setting the files containing the queue manager definition and the AMI repository:

```
amSessionFactory = new AmSessionFactory();  
amSessionFactory.setLocalHost(amiInfo[0]);  
amSessionFactory.setRepository(amiInfo[1]);
```

These two files are defined in the properties file:

```
localhostFile=amthost.xml  
repositoryFile=amt.xml
```

We then create the objects based on the definitions contained in the repository (for the policies and publishers). The sessions and messages don't need any definitions from the repository:

```
amSessionEvent = amSessionFactory.createSession(SESSION_EVENT);  
amSessionState = amSessionFactory.createSession(SESSION_STATE);  
amPolicyEvent = amSessionEvent.createPolicy(amiInfo[2]);  
amPolicyState = amSessionState.createPolicy(amiInfo[3]);  
amPublisherAccident = amSessionEvent.createPublisher(amiInfo[4]);  
amPublisherBreakdown = amSessionEvent.createPublisher(amiInfo[5]);  
amPublisherPosition = amSessionState.createPublisher(amiInfo[6]);  
amMessageAccident = amSessionEvent.createMessage(MESSAGE_ACCIDENT);  
amMessageBreakdown = amSessionEvent.createMessage(MESSAGE_BREAKDOWN);  
amMessagePosition = amSessionState.createMessage(MESSAGE_POSITION);
```

The repository file contains the policies and publishers, whose names are read from the properties file:

- policyState=VEHICLE.POSITION.PUB.SUB.POLICY
- policyEvent=VEHICLE.ALERT.PUB.SUB.POLICY
- publisherPosition=VEHICLE.POSITION.PUBLISHER
- publisherAccident=VEHICLE.ALERT.PUBLISHER
- publisherBreakdown=VEHICLE.ALERT.PUBLISHER

3. After all objects are created, we must open them:

```
amSessionEvent.open(amPolicyEvent);  
amSessionState.open(amPolicyState);  
amPublisherAccident.open(amPolicyEvent);  
amPublisherBreakdown.open(amPolicyEvent);  
amPublisherPosition.open(amPolicyState);
```

**Important:** AMI handles and objects references can be used on a different thread from that on which they were first created for operations that do not involve access to the underlying message transport. However, opening a session accesses MQSeries in a way that means all operations within that session (that is, sending, receiving, publishing or subscribing) must be executed within the same thread.

4. And finally, we can publish our messages after having set their content and the topic they are to be published to:

```
amMessagePosition.addTopic(pubData.topic);  
amMessagePosition.writeBytes((content).getBytes());  
amPublisherPosition.publish(amMessagePosition, amPolicyState);
```

5. After all messages have been set, we clean up the resources by closing the publishers and the sessions:

```
amPublisherAccident.close(amPolicyEvent);  
amPublisherBreakdown.close(amPolicyEvent);  
amPublisherPosition.close(amPolicyState);  
amSessionEvent.close(amPolicyEvent);  
amSessionState.close(amPolicyState);
```

## 4.8 Subscription

The subscriber application is a stand-alone Java-based GUI window that uses the Java AMI APIs.

In the following sections we describe:

- ▶ The setup of the environment
- ▶ The AMI administration setup
- ▶ The sample subscriber application

### 4.8.1 Setup of the environment

There are some steps required to set up the environment before being able to use the AMI subscriber application. In this section, we also cover what is needed to develop the application.

Software required:

- ▶ MQSeries 5.1 or 5.2
- ▶ MQSeries Publish/Subscribe SupportPac MA0C

- ▶ AMI APIs - SupportPac MA0F\_NT
- ▶ XML Parser for Java (XML4J)
- ▶ VisualAge for Java 3.5.3

For instructions on installing these products, please refer to 4.2.7, “AMI installation” on page 56 and 4.2.1, “MQSeries Publish/Subscribe installation” on page 29.

## 4.8.2 XMLParser setup

We have used XML4J parser for XML parsing in our application. The XML4J parser can be freely downloaded from:

<http://www.alphaworks.ibm.com/tech/xml4j>

Download the XML4J-J-bin.3.1.1.zip from the above Web site and unzip in a newly created directory (we will use C:\XML4J in the rest of the chapter).

## 4.8.3 VAJava setup

We are using VisualAge for Java Version 3.5.

Here are the steps to follow to work with AMI and XML4J parser in VisualAge for Java:

1. Make sure the features IBM XML Parser for Java is loaded in the workspace.
2. Create a new project. We called it MqSubAMI.
3. Import in this project the required AMI JAR file com.ibm.mq.amt.jar from directory C:\Program Files\IBM\MQSeries\Java\Lib (where C:\Program Files\IBM\MQSeries is the MQSeries installation directory).
4. Import xml4j.jar and xerces.jar files from directory C:\XML4J\XML4J-3\_1\_1 (where C:\XML4J\ is the XML4J parser directory) into the IBM XML Parser for Java project.

## 4.9 AMI administration setup

In this section we will discuss the necessary AMI administration set up for executing the subscriber application. Please refer to 4.2.8, “AMI configuration” on page 60 for AMI Administration Tool usage details. Create a subscriber VEHICLE.SUBSCRIBER as shown in Figure 4-48 on page 95.

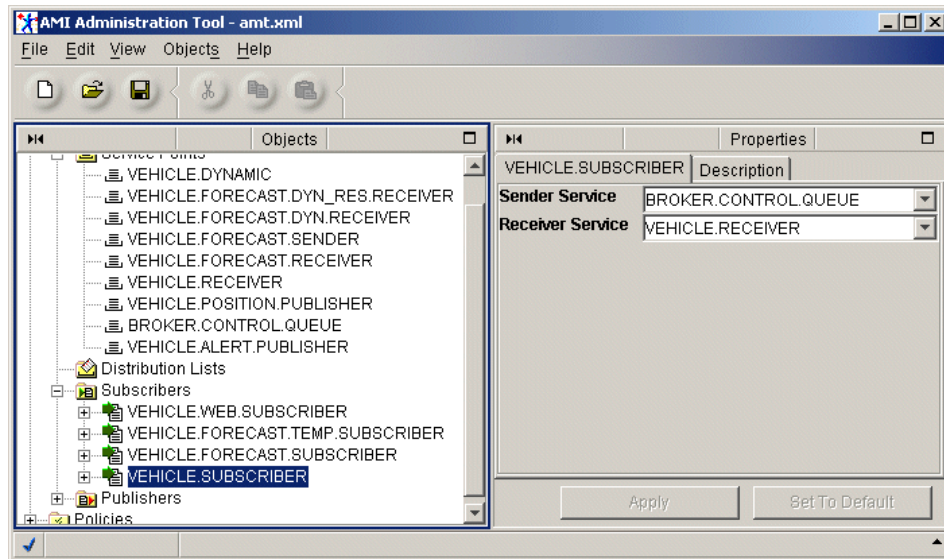


Figure 4-48 The AMI administration console displaying Subscriber setup

Next create a policy VEHICLE.SUB.POLICY as discussed in 4.2.8, “AMI configuration” on page 60.

In the Initialization tab, if you have a default queue manager running set Connection Mode to **Real**. Otherwise, set it to **Logical** and supply the connection name as specified in the local host file. In our example we have kept connection mode as **Real** and connection name as **defaultConnection**.

Then in the Subscriber tab, check **Use CorrelId as Id** as shown in Figure 4-49 on page 96. This CorrelId is used by the broker as part of the subscriber's identity.

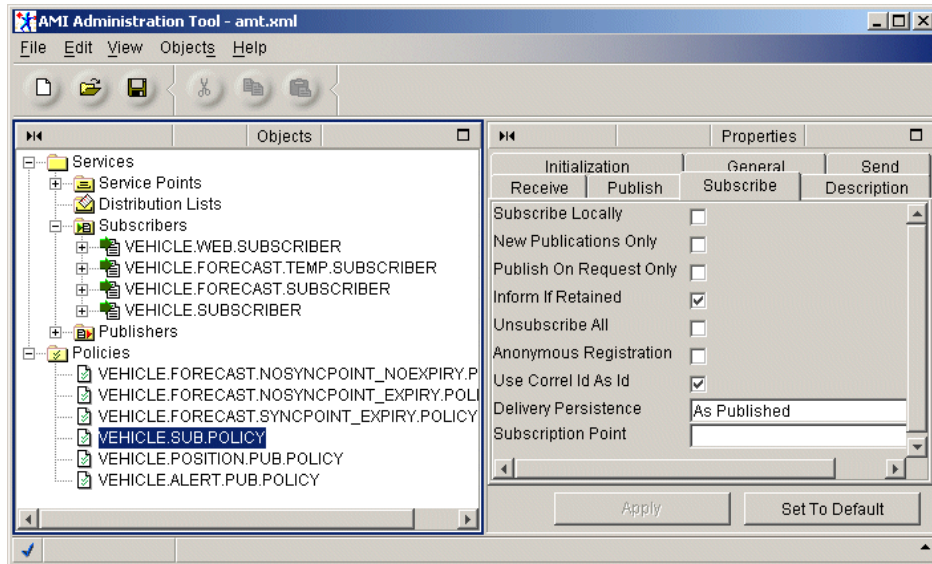


Figure 4-49 Displaying Subscriber tab on VEHICLE.SUB.POLICY

## 4.10 Sample subscriber application

The subscriber application is a simple Java AMI-based application that subscribes to all the three different messages as published by the publishing application: one message to indicate vehicle position, another message when it has an accident and the third message when the vehicle has broken down. Each of these messages has a different subtopic string with a common main topic. So it is sufficient to subscribe only the main topic name followed by /\* (forward slash and asterisk).

In this application the message topic is `PublicTransport/*`. The published messages are displayed in the window shown in Figure 4-50 on page 97.

The subscriber application has three parts:

- ▶ Control Program
- ▶ XML parser program
- ▶ GUI program

The GUI program displays the published message and invokes the Control Program. The Control Program makes the actual AMI calls, subscribes to the topic, and receives the message. The Control Program uses the XML Parser program to parse the XML message. The message parsed is shown in Figure 4-50.

MessageType	Mode	Geography	Route	Vehicle	Traffic	Stop #	# Stops	TimeBtwmStops	Fill%	TimeStamp
position	Tube	London	Bakerloo	Utrain22	normal	2	11	3	44	14:35:02
position	Tube	London	Bakerloo	Utrain21	heavy	6	11	3	80	14:35:04
position	Tube	London	Piccadilly	Utrain12	normal	3	8	4	7	14:34:56
position	Tube	London	Piccadilly	Utrain11	light	5	8	4	96	14:35:04
breakdown	Tube	London	Piccadilly	Utrain11		5				14:35:04
position	Tube	London	Bakerloo	Utrain21	heavy	7	11	3	88	14:35:07
position	Tube	London	Bakerloo	Utrain21	normal	8	11	3	21	14:35:10
position	Tube	London	Piccadilly	Utrain12	normal	4	8	4	65	14:35:10
position	Tube	London	Bakerloo	Utrain21	heavy	9	11	3	79	14:35:13
position	Tube	London	Piccadilly	Utrain12	light	5	8	4	10	14:35:14
position	Tube	London	Bakerloo	Utrain22	light	3	11	3	48	14:35:15
position	Tube	London	Bakerloo	Utrain21	heavy	10	11	3	66	14:35:16
position	Tube	London	Bakerloo	Utrain22	light	4	11	3	85	14:35:18
position	Tube	London	Piccadilly	Utrain12	normal	6	8	4	33	14:35:18

Figure 4-50 Subscriber Application view

The following section describe the third part of the subscribe application.

## 4.10.1 Control Program

The program reads all its initialization parameters from the properties file called subscribe.properties.

### The properties file: subscribe.properties

This properties file contains the information specific to subscription.

Here are the entries found in the application part of the properties file:

- ▶ hostfilename - Name of the host XML file
- ▶ repositoryfilename - Name of the Repository XML file.
- ▶ policyname - Name used to create the AmPolicy. This name matches the Policyname in the repository.

- ▶ `messagereceiver`- Name of the message used to receive publications. This is a user-defined tag.
- ▶ `subscribername` - Name used to create the `AmSubscriber` used for sending subscriptions and receiving publications. This name matches the Service definition in repository.
- ▶ `messagesubscriber`- Name of the message used to subscribe. This is a user-defined tag for the message.
- ▶ `sessionname` - Name of the session used to create the `AmSession`. This is also a user-defined tag for the Session.
- ▶ `topicname` - Name of the topic to which the application is subscribed. In this example we are subscribed to `PublicTransport/*`. All the topics starting with `PublicTransport` will be published.
- ▶ `waittime` - A period of time (in milliseconds) that the receiver waits for a message to become available.

## Program flow

This section is not a detailed description of how AMI works. It is only a description of how to use Java AMI to subscribe:

1. Import the required AMI package:

```
import com.ibm.mq.amt.*;
```

2. Read all the entries from the properties file `subscribe.properties`.

**Note:** The entries from the properties file are stored in the `SubscriberInitData` data bean, as follows:

```
SubscriberInitData subinit=new SubscriberInitData();
```

3. Retrieve the information stored in the AMI repository and create the AMI objects based on this information.

We start by setting the files containing the queue manager definition and the AMI repository:

```
subscriberSessionFactory = new AmSessionFactory();
subscriberSessionFactory.setLocalHost(subinit.hostfileName);
subscriberSessionFactory.setRepository(subinit.repositoryName);
```

These two files are defined in the properties file:

```
hostfilename=amthost.xml
repositoryfilename=amt.xml
```



**Tip:** The hostfile name and repository file name can also be retrieved from the environment. Typically the location of the host file and the repository file would be under the root directory of the AMI installation path, which would have been indicated by AMT\_PATH environment variable. If you desire to keep these files in a separate location, then you can set the following two environment variables and assign the name of the file with the full path:

```
set AMT_HOST= Host file Name
set AMT_REPOSITORY= repository file name
```

We then create the objects based on the definitions contained in the repository (for the policies and subscriber). The session and messages don't need any definitions from the repository:

```
subscriberSession =
subscriberSessionFactory.createSession(subinit.sessionName);
policy=subscriberSession.createPolicy(subinit.policyName);
policy.setWaitTime(new Integer(subinit.waitTime).intValue());
```

**Important:** When setting the waitInterval within the program, make sure to uncheck the Wait Interval Read Only property of VEHICLE.SUB.POLICY in the AMI administration console. This wait interval overwrites the Wait Interval property.

```
subscriber = subscriberSession.createSubscriber(subinit.subscriberName);
subscribeMsg = subscriberSession.createMessage(subinit.messageSubscriber);
```

The repository file contains the policies and subscribers, whose name are read from the properties file:

- policyname=VEHICLE.SUB.POLICY
- subscribername=VEHICLE.SUBSCRIBER

4. After all objects are created, we must open them:

```
subscriberSession.open(policy);
subscriber.open(policy);
```

5. Next, we can subscribe our messages after having set the topic to which we subscribed with a unique CorrelId:

```
subscribeMsg.addTopic(TOPIC);
correlId=generateCorrelId();
```

**Important:** In order to run multiple instances of the same subscriber using the same queue, the broker maintains the uniqueness of each application by its CorrelId, which has been generated by the application itself. (The latest release of AMI APIs (Version 1.2) supports automatic generation of the CorrelId.)

```
if (this.correlId != null )
{
    subscribeMsg.setCorrelationId(this.correlId.getBytes());
}
subscriber.subscribe(subscribeMsg,policy);
```

6. Finally, we can receive our messages, which are published on the topic to which we subscribed. Since the publishers are in a different flavor such as C-based MQI, Java AMI and JMS, typically the format of the message as published by JMS is different from MQI or Java AMI. Care has been taken in the subscriber application to interpret the message published by a JMS application. Refer to 4.10.4, “Parsing JMS-based published message” on page 102 for the JMS message format.

```
if (this.correlId != null )
{
    subscriber.receive(receiveMsg, subscribeMsg, policy);
}
else
{
    subscriber.receive(receiveMsg, policy);
}
if (receiveMsg.getFormat().indexOf("HRF2")==-1)
    msgInString =
        new String(receiveMsg.readBytes(receiveMsg.getDataLength()));
else
    msgInString=parseRFH2(receiveMsg);
```

**Important:** The wait interval may be several seconds as discussed in step 3. If no message arrives within this time interval, the subscriber program throws an exception `AMRC_NO_MSG_AVAILABLE` and exits from the blocking call. At this point the program checks if the user has requested to exit the application. If not, it goes back into a wait state.

7. The next step is to parse the message, which is in XML format.

```
xmlParser.parse(new ByteArrayInputStream(msgInString.getBytes()))
```

8. After all the messages are received, we unsubscribe to the topic we subscribed in step 5 of the Control Program:

```
subscribeMsg.addTopic(TOPIC);
subscriber.unsubscribe(subscribeMsg, policy);
if (this.correlId != null )
{
    subscribeMsg.setCorrelationId(this.correlId.getBytes());
}
```

9. After unsubscribing is done, we release the resources by closing the subscriber and the session:

```
subscriber.close(policy);
subscriberSession.close(policy);
```

Let's discuss some important methods in which we covered the above steps and operations:

- `Init()` - This method is invoked the first time to do the initialization. Inside this method, the `readProperty()` method is invoked. Step 3 of the Control Program to step 4 of the Control Program is carried out in this method.
  - `readProperty()` - This method is responsible for reading the property file as described in step 1 of the Control Program.
10. `subscribe()` - The subscription part is carried out in this method. Refer to step 5 of the Control Program.
  11. `receive()` - This method receives the published message. If the message header is `RFH2`, then it invokes the `parserRFH2()` method.
  12. `unsubscribe()` - This method does the unsubscribing operation as discussed in step 9 of Control Program
  13. `close()` - This method close all the resources as discussed in step 9 of Control Program
  14. `parseRFH2` - This method acts as a filter to the `RFH2` message header. It parses the `RFH2` message header and extracts the actual message in the

String format. Refer to 4.10.4, “Parsing JMS-based published message” on page 102 for details of the message format.

**Important:** In a scenario where both publisher and subscriber are written to the same API (for example, they are all JMS applications) no special handling for the MQRFH2 header is necessary.

### 4.10.2 XML parser program

The published message format is in XML form. To parse these XML messages we have used the SAX parser. Please refer to Appendix E, “Additional material” on page 211 for the message format.

The Control Program application supplies the published XML message to the parser and gets back the required value in a proper format.

Refer to the additional material that accompanies this redbook for XMLparser program code snippets.

### 4.10.3 GUI program

The GUI program displays the published message when any message gets published. The GUI program works as follows:

1. Invokes the Control Program that carries out the operation, as discussed in 4.10.1, “Control Program” on page 97.
2. The GUI program continuously calls the receive() method of the Control Program so that it always receives the new published message and immediately displays the same.
3. The program quits when you click the **Close** button.

**Important:** Make sure the broker is running, and the host file and repository file are placed in the proper path.

### 4.10.4 Parsing JMS-based published message

The JMS message is composed of four parts (see Figure 4-51 on page 103):

- ▶ The MQSeries Message Descriptor (MQMD)
- ▶ The MQRFH header
- ▶ The MQRFH2 header
- ▶ The message body (that is, the payload)

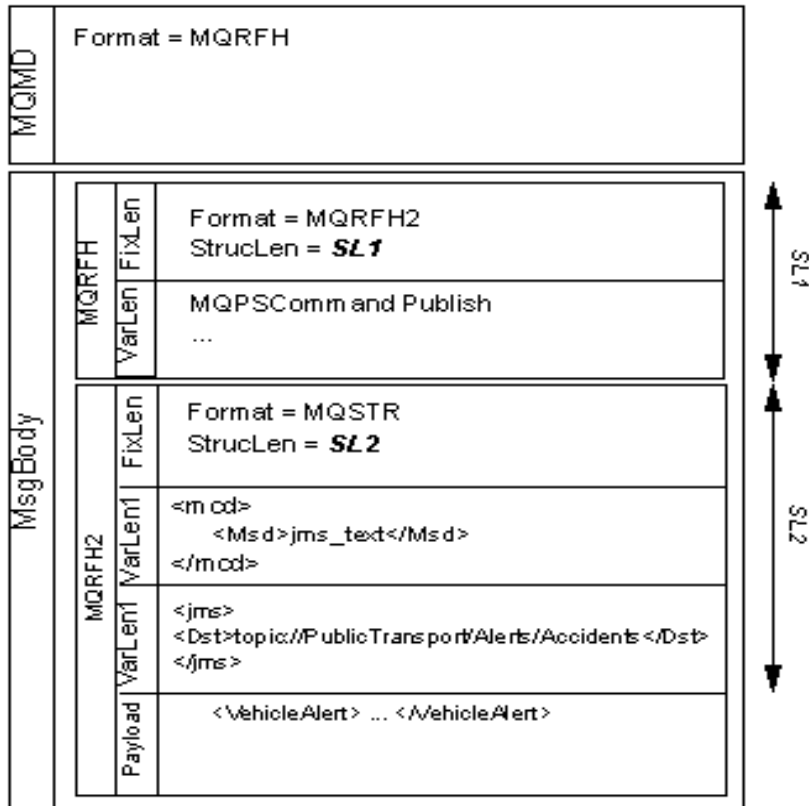


Figure 4-51 Anatomy of message format as published by JMS Pub application

Let's now take a closer look at the MQRFH2 header as used by JMS:

- ▶ There is one field of the message descriptor that is relevant to our discussion:
  - StrucLength: Total length of MQRFH2, including the NameValueData fields)
- ▶ The MQRFH2 formatted message body is made of three parts:
  - Fixed-length part
    - StrucId: this is the name identifying the format of the message payload. It is four characters in length.
    - Version: this is the encoding of any numeric information included in the message payload.

- `StrucLength`: this field gives the total length of the MQRFH2 fixed-length part and variable-length part, excluding the message payload.
  - `Encoding`: this is the encoding of any numeric information included in the message payload.
  - `CodedCharSetId`: This specifies the coded character set identifier of character strings in the data(if any).
  - `Format`: Format name of the data that follows `NameValueData`.
  - `Flag`: This field is currently not used.
  - `NameValueCCSID`: The coded character set identifier for the `NameValueData` character strings contained in this header
- Variable-length part: The fixed portion is followed by the variable portion which contains a varying number of MQRFH2 folders. Each folder contains a varying number of elements or properties. Folders are used to group together related properties. The MQRFH2 headers created by JMS can contain up to three folders:
    - `<mcd> folder`: This contains the properties that describe the format of the message. This folder is always present in a JMS MQRFH2.
    - `<jms> folder`: This contains the extra JMSX properties that cannot be fully expressed in the MQMD. This folder is always present in a JMS MQRFH2.
    - `<usr> folder`: This contains application-defined properties associated with the message. This is an optional folder. It is only present if the application has set some application-defined properties.
  - Message payload: this is the application data that is being published.

In order to extract the message payload, we need to read the value of `StrucLength` field, so that we can calculate the offset (that is, the starting point of the message payload) by this simple formula:

`offset = Total Length of message - StrucLength.`

Once we get the offset value, we can retrieve the message using AMI APIs as shown in Example 4-5.

*Example 4-5*

---

```
receiveMsg.setDataOffset(offset);
receiveMsg.readBytes(receiveMsg.getDataLength()- offset));
```

---

Refer to the additional material accompanying this redbook for code snippets of the `parseRFH2()` method.

The mapping of JMS message onto MQSeries Message format is documented in *Using Java*, SC34-5456.

## 4.11 Comments and extensions

This first scenario suffers from the following limitations:

- ▶ All the information is published as non-retained. This means that the subscriber application will receive only the publication that became available from the time it started up. This hinders any attempts to provide vehicle tracking features without using an external database.
- ▶ The rich publication data received by the subscriber application is not correlated to produce value-added information from an end-user point of view. An example of this may be the production of forecasts about expected time of arrival (ETA) of vehicles at a given point in time.

In the following sections, we will discuss potential extensions to this first scenario.

### 4.11.1 Retained publications

The main cause of the current version of the subscriber application shortcomings is to be traced back to the fact that all the position information published is actually *state information* and not *event information*.

An MQSeries Publish/Subscribe broker accommodates state information through a special form of publication named *retained*. By adopting the retained style of publications, the broker actually keeps a copy of the last publication message received for each retained topic.

Any subsequently received message on the same topic will overwrite the internally saved one.

Subscribing applications are affected in the following ways:

- ▶ After having registered a subscription, they are immediately sent all the retained publication messages held by the broker, and any other messages matching the requested topics as they become available to the broker.
- ▶ At subscription time, they can ask the broker to explicitly tag each retained message, in order to perform special handling (for example, display the information on a window, labelling it as potentially old).
- ▶ At subscription time they can ask the broker to send all the retained messages matching the topic only when explicitly asked by the subscriber.

**Note:** A common misconception of the retained publications feature leads developers to try to use it in order to enable subscribers to catch up with old publication messages that were received by the broker *prior to* subscription.

This is not correct. In fact for each single retained topic no more than *one* message is kept by the broker. All messages received before than this will not be received by subscribers.

Our example subscriber program can get the benefit of retention at publication time without the need for any code changes.

### **Retained publications with MQI**

Altering the C MQI version of the Vehicle program requires changes to the code. In fact a new name/value item must be added to the name/values string in the MQRFH header of the publication message.

The new item is the following publication option: MQPSPubOpts RetainPub. In C this would look like the following:

```
strcat(pszNameValueString, MQPS_PUBLICATION_OPTIONS_B);  
strcat(pszNameValueString, MQPS_RETAIN_PUBLICATION);
```

We do not provide a modified version of the program in the additional material that accompanies this redbook.

### **Retained publications with JMS**

JMS specifications does not include special provisions for state information. As we will discuss later, this does not impede JMS from subscribing to a topic that is retained.



**Restriction:** JMS does *not* support retained publications.

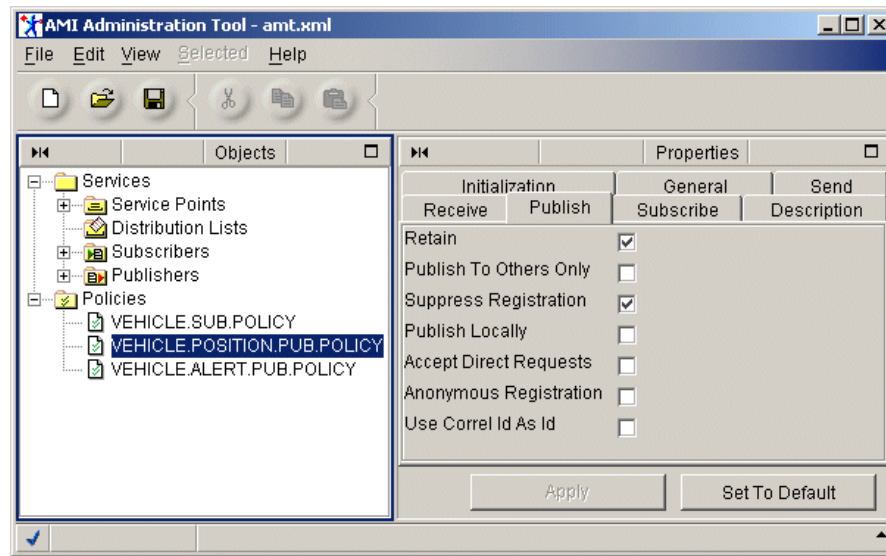


Figure 4-52 Making a topic retained in the AMI Repository

## Retained publications with AMI

Making a topic retained for an AMI publishing application is as simple as changing a setting in the repository file.

With reference to our example environment these are the steps to follow:

1. Start the AMI Tool
2. Open the amt.xml file containing the repository
3. Select **VEHICLE.POSITION.PUB.POLICY** policy icon
4. Select the **Publish** tab
5. Check the **Retain** option

Now alter the pub.properties file in order to activate the AMI C or AMI Java publisher and run the publisher program once, waiting until it terminates.

At this point start the subscriber, and you should see position information received even if no one is currently publishing it. This information is in fact of a retained nature.

If you restart the publisher you will see that the freshly published information will join the old one on the subscriber window, as it become available to the broker.

From the user point of view, the added value of our subscriber application is higher. In fact, as soon as it is started it immediately displays the last known state of the system and then updates it with any new information received.

A few issues still remain open:

- ▶ There is no simple way to use alert information to augment the position information (for example Vehicle X was at stop Y at time T, but then had an accident).
- ▶ There is no simple way to produce a single view of current positions and forecasted ones.

In this context *no simple way* means that to implement the requested features it would take a database, but then the subscriber application would increase in complexity (for example global transaction handling) and would become less eligible to be transformed in a lightweight Web application.

In the following sections we will evolve the application in a way that will cope with all the limitations of the current version.

## 4.11.2 Streams

All our publishing applications send messages to queue `SYSTEM.BROKER.DEFAULT.STREAM.QUEUE`. This is the broker default *stream*, and is automatically defined by the broker the first time it starts up.

In general a stream is a local queue used by the broker to receive publications. In this section we will discuss how streams can be used in MQSeries Publish/Subscribe.

**Restriction:** Stream queues must not be cluster queues.

### Topic namespace partitioning

When a publisher publishes messages or a subscriber registers a subscription, these operations explicitly or implicitly (in case of the default stream) specify a stream name to which to refer.

The stream name can be viewed as a high-level qualifier for the topic name. So, if an application subscribes to the topic `Stocks/*` on stream `NYSE` it will not receive messages published on topic `Stock/JunkBonds` on stream `NASDAQ`.

The topic namespace is thus actually partitioned in classes associated to the available streams.

## Parallel computation

Each stream queue is served by a dedicated broker thread, so streams can be used as a means to implement parallel computation within the broker and increase the overall message throughput.

## Access control

A publisher can publish on a stream queue only if it has MQSeries put authority on that queue.

A subscriber can receive publications only if:

- ▶ It has MQSeries put authority on `SYSTEM.BROKER.CONTROL.QUEUE`, used for registering the subscription.
- ▶ It has put authority on the queue where the publication message will be sent.
- ▶ It has browse authority on the stream queue.

Using separate streams for different confidentiality levels (and setting the MQSeries authorities accordingly) is an effective way of implementing access control on the publish/subscribe domain using standard MQSeries facilities.

**Note:** the access control checks on the subscriber queue (put authority checked) and the stream queue (browse authority checked) are performed by the broker on behalf of the application, even if the subscriber does not browse messages from the stream queue directly and does not put messages on its subscriber queue.

## Impact on the applications

Explicitly specifying a non-default stream entails adding an MQPSSstreamName name/value pair to the MQRFH header. At the application level the impact depends on the API flavor used:

- ▶ When using the MQI, a change in the code building the MQRFH header is needed.
- ▶ When using AMI the stream name can be specified using the method `addElement` of the `AmMessage` object. For the C language a helper macro `AmMsgAddStreamName` is provided.
- ▶ When using JMS you can set the stream name altering the definition of the `TopicConnectionFactory` object `BROKERPUBQ` parameter in the JNDI repository.

## **Publishers and subscribers decoupling**

The use of streams has sometimes been criticized because it looked like a feature coupling together publisher and subscribers by sharing a queue name, and thus losing some of the benefits that come from publish/subscribe itself.

This is not correct. In fact from this point of view streams can be seen simply as high-level qualifiers for topics.

The physical representation of a stream happen to be an MQSeries queue having the same name of the stream. The stream name is used also as a queue name by publishers to send messages to the broker.

Subscribers specify a stream name to qualify their subscription topics but actually access only a well-known broker queue at subscription registration time (SYSTEM.BROKER.CONTROL.QUEUE) and their own subscriber queue to receive publication messages.

Streams also play a very important role in broker networks. This aspect is discussed in the following section.

### **4.11.3 Broker networks**

In order to accommodate geographically distributed publishers and subscribers, or particularly heavy workloads, it is sometimes needed or recommended that you not use just one broker but a full set of them.

Brokers can be connected in a network. The only supported topology is the hierarchy (that is, the tree structure).

#### **Broker hierarchies**

A broker network is a tree of interconnected brokers. All brokers apart from the root broker are created specifying the parent broker.

From a generic broker to all neighbors (that is, parent broker plus child brokers) there must be two-way MQSeries connectivity. The configuration of such connectivity must be done using MQSeries standard tools and practices external to MQSeries Publish/Subscribe.

While the only broker topology supported is the hierarchy, there are no topological constraints on how to implement the underlying queue manager intercommunication. For example a child broker may communicate with its parent broker via a hub, of which the two broker queue manager are spokes.

Even if two distinct non-descendent brokers have direct MQSeries connectivity between each other, they exchange MQSeries Publish/Subscribe information only via their closest common ancestor broker (in the worst case, this will be the root broker).

Each broker communicates to his neighbors information about supported streams. When a broker receives a subscription it forwards it to all the broker supporting the stream associated to the subscription.

Broker-to-broker subscriptions are consolidated by the subscribing broker.

### **Example**

Broker1 receives two subscriptions (see Figure 4-53):

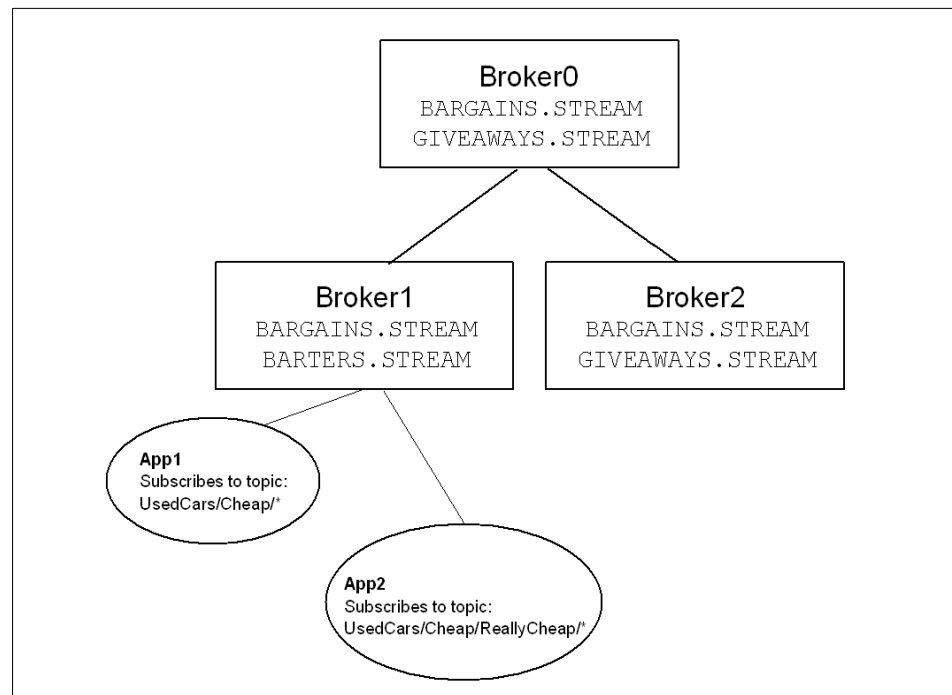


Figure 4-53 Simple broker hierarchy

- ▶ A subscription from application App1 on topic UsedCars/Cheap/\* at stream BARGAINS.STREAM
- ▶ A subscription application App2 on topic UsedCars/Cheap/ReallyCheap/\* at stream BARGAINS.STREAM.

When the first subscription is received, the broker registers a subscription with Broker0 on topic UsedCars/Cheap/\*, because this broker supports the stream BARGAIN.STREAM. At Broker0 a similar subscription is registered with Broker2 for the same reason.

When the second subscription is received, no interbroker subscription is registered, because there is one already in place for the same stream and a wider topic set.

The handling of publication messages is straightforward. Each broker sends publication messages to all the matching subscribers, irrespective of whether they are user applications or neighbor brokers.

The only caveat for interbroker publication or subscription message exchanges is that messages are put with broker authority, and not publisher or subscriber authority. This means that the access control checks for publishers and subscribers are performed only at their local broker.

From an MQSeries intercommunication point of view, brokers address queues owned by neighbor brokers using *explicit addressing* (that is, they open queues specifying a queue name and also a queue manager name).

To ensure that this addressing works in a classic distributed queue management environment, it's best to name the transmission queues the same as their target queue managers name, or provide a queue manager alias such as:

```
DEFINE QREMOTE (TARGET.QMGR)
          RNAME()
          RQMNAME(TARGET.QMGR)
          XMITQ(WEIRD.NAME.TARGET.QMGR.XMITQ)
```

On the other hand, if brokers are hosted by queue managers residing on an MQSeries queue manager cluster, no particular administrative action is needed.

## Building the hierarchies

Brokers are added to a network from the root down, but are removed from the leaves up.

In the rare circumstances where this is not feasible (for example a wrong administrative action deleted an intermediate queue manager hosting a broker), there is a special command named `c1rmqbrk` that can be used to fix the hierarchy, but its usage should be restricted to abnormal situations.

The first time that a broker is started using the `strmqbrk` command, you can specify a parent broker using the `-p` option. This value is retained by the broker but to change it you cannot simply pass a different value. You should use the `c1rmqbrk` command.

To remove a leaf broker from a hierarchy you should use the `dltmqbrk` command after having quiesced the broker itself, all local publish/subscribe applications and all inbound MQSeries channels from neighbor brokers.

### **Example**

The hierarchy in Figure 4-53 on page 111 can be built as follows:

```
strmqbrk -m Broker0
strmqbrk -m Broker1 -p Broker0
strmqbrk -m Broker2 -p Broker0
```

### **Streams**

Streams play a key role in broker networks. In fact subscriptions are propagated from one broker only to those neighbors that support the stream included in the subscription (possibly an implicit `SYSTEM.BROKER.DEFAULT.STREAM`).

Let's suppose that the hierarchy in Figure 4-53 on page 111 is extended with two full broker subtrees rooted on Broker1 and Broker2, and for some business reason you want to segregate messages on `BARGAINS.STREAM` to the originating subtree.

To obtain this effect it is enough to remove `BARGAINS.STREAM` from Broker0, effectively stopping subscription propagations to and from Broker1 and Broker2.

Streams in large geographically distributed broker networks are also important as a means to differentiate quality of service in a stream-wise manner (for example messages to non-critical streams are mapped to channels running only overnight).

### **Impact on the applications**

As we have seen, broker networks are easy to implement, and are largely an administrative issue.

Most MQSeries Publish/Subscribe applications will not be affected by the inclusion of their broker in a hierarchy.

On the other hand, hierarchy-aware applications can explicitly set the scope of their publication or subscription to be local to their broker (that is, not to be propagated to neighbor brokers), by specifying the publication or subscription option `Local` in the `MQRFH` header.







# Migration to MQSeries Integrator

This chapter describes the migration of the example applications written for MQSeries Publish/Subscribe to an MQSeries Integrator broker, and demonstrates the API level interoperability between these two environments.

A general knowledge of MQSeries Integrator and a working MQSeries Integrator test environment (that is a working Configuration Manager, User Name Server and Broker) are required to carry out the tasks described in this chapter.

Please refer to MQSeries Integrator documentation for complete details about the product. We recommend reading Appendix A, “Planning for migration and integration” in *MQSeries Integrator Introduction and Planning*, GC34-5599. In particular compare your requirements against the migration inhibitors checklist.

If you need a quick start with the main MQSeries Integrator concepts and features, refer to *Business Integration Solutions with MQSeries Integrator*, SG24-6154.

## 5.1 Step-by-step guide

This section demonstrates a typical procedure for running an application written for an MQSeries Publish/Subscribe broker on an MQSeries Integrator broker.

### 5.1.1 Step 1 - Creation of a publication queue

The applications written for the simple MQSeries Publish/Subscribe scenario access the following broker queues:

- ▶ `SYSTEM.BROKER.DEFAULT.STREAM`: this queue is used by the publisher to send publication messages.
- ▶ `SYSTEM.BROKER.CONTROL.QUEUE`: this queue is used by the subscriber to register its subscription.

The MQSeries Integrator broker uses the same control queue as the MQSeries Publish/Subscribe broker. On the other hand, there is no equivalent for streams.

In order to ensure that no changes are required to the publishing applications, we define a new local queue on the MQSeries Integrator broker queue manager named as the MQSeries Publish/Subscribe default stream queue. Using MQSC commands this can be done as follows:

```
DEFINE QLOCAL(SYSTEM.BROKER.DEFAULT.STREAM) REPLACE
```

### 5.1.2 Step 2 - Creation of a simple publish message flow

If you imported all the definitions needed to run the MQSeries Integrator samples you have a message flow named Default Publish/Subscribe in your workspace. If you don't, you can import it from the file named `SamplesWorkspaceForImport` in the `examples` subdirectory of MQSeries Integrator installation directory.

This is a very simple message flow made of two nodes:

- ▶ An MQInput node named `Get next message`, which gets messages from a queue that by default is `SYSTEM.BROKER.DEFAULT.STREAM`.
- ▶ A Publication node named `Route to matching subscribers`, that actually publishes the message on the MQSeries Integrator broker.

In order to have a clearer view of how this message flow works, we will modify it by inserting a Trace node in between the two existing nodes. To do this first you need to check out the message flow by right-clicking the message flow icon and selecting the **Check Out** option.

To add the Trace node, drag it to the message flow graph area from the IBMPrimitives folder. To wire it, remove the existing connection between the original nodes and create two new connections, one from the MQInput node to the Trace node and one from the Trace node to the Publications node.

The final result of the above operations should be similar to Figure 5-1.

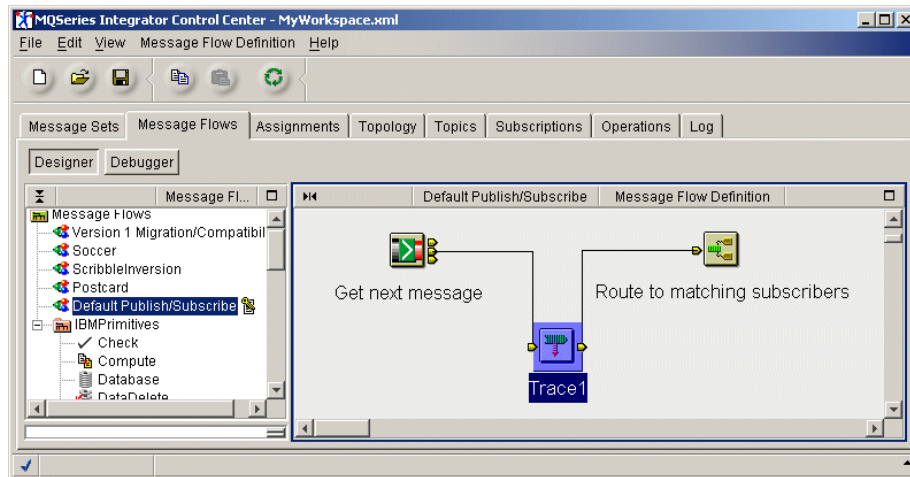


Figure 5-1 Trace enabled Default Publish/Subscribe message flow

Now we can configure the trace node to write the whole message tree to a text file, labeling each line with a timestamp. This can be obtained by entering the Trace node parameters as shown in Figure 5-2 on page 118.

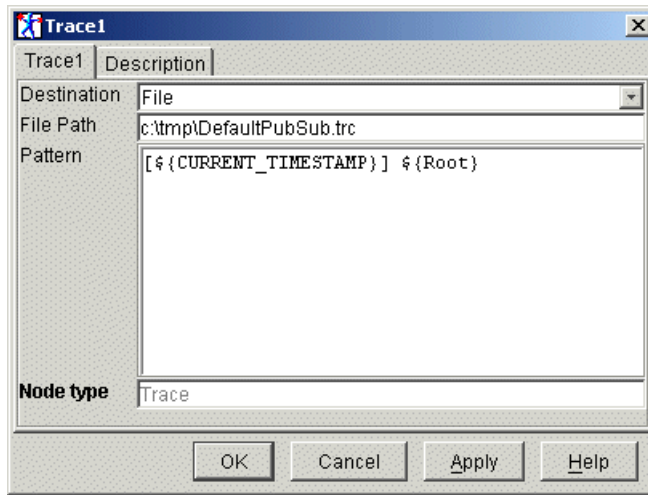


Figure 5-2 Tracing message tree to a text file

Given that the payload of the messages generated by the example applications is in XML format, we will configure the MQInput node accordingly (see Figure 5-3). This does not actually impact the way messages are published, but makes the information in the trace easier to understand.

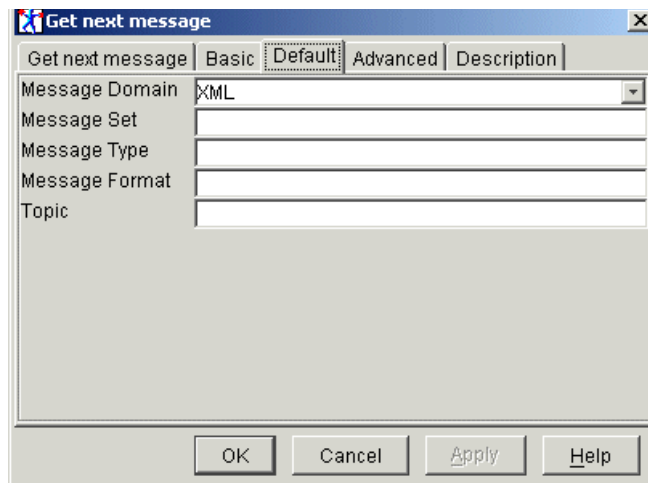


Figure 5-3 Suggesting the domain on the input node

The message flow is now ready. Right-click on it and select and click **Check In** to commit your changes in the Configuration Manager database.

### 5.1.3 Step 3 - Deployment to the target broker

To deploy the message flow on the target broker group, use the Assignments tab of the Control Center. Select and click **Check out**, drag the modified message flow onto the default execution group box on the right pane, and check in the default execution group.

Start the deployment operation using the menu option **File->Deploy->Delta configuration (all types)**. The progress and outcome of the operation can be monitored using the Log tab of the Control Center.

All the above deployment actions are completely standard, and apply to any MQSeries Integrator message flow to be deployed.

### 5.1.4 Step 4 - Executing example applications on MQSeries Integrator

The MQSeries Integrator broker is now ready to run with the example applications.

Run the stand-alone Java subscriber, then go to the Subscriptions tab of the MQSeries Integrator Control Center and click the **Query** (that is, refresh) button. You should see something similar to Figure 5-4.

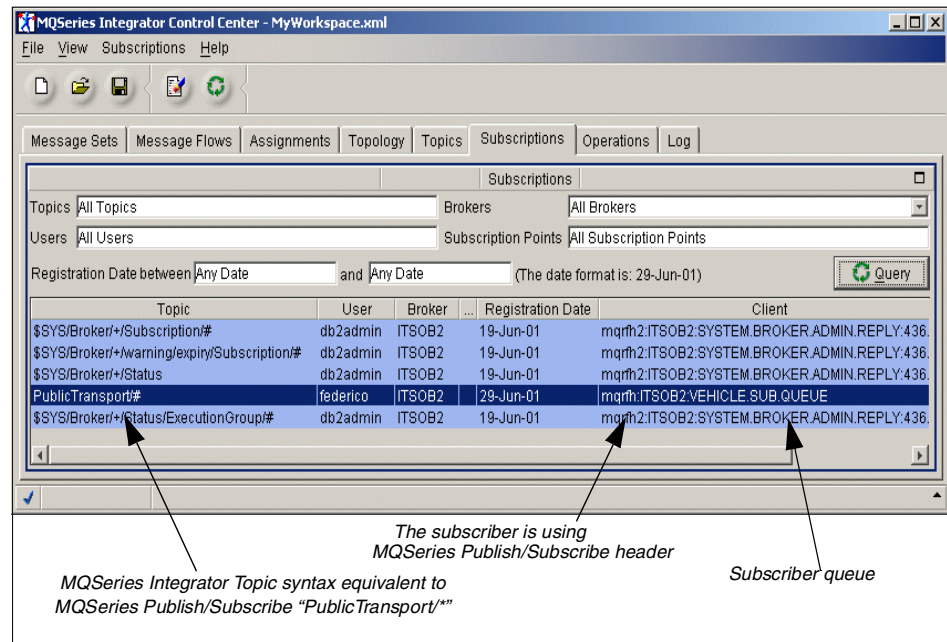


Figure 5-4 MQSeries Integrator Control Center Subscriptions tab

This means that a subscription on topic `PublicTransport/*` has been successfully registered by an application originally written for an MQSeries Publish/Subscribe broker, whose subscription queue is `VEHICLE.SUB.QUEUE`.

You are now ready to run the publisher application using any of the available language and API options available.

### 5.1.5 Step 5 - Trace analysis

If you are using a non-JMS flavor of the publisher application, the trace will show an MQRFH header followed by the message body. If you run the JMS publisher, the trace will show an MQRFH header embedding an MQRFH2 header, embedding the real XML application message.

## 5.2 Comments and extensions

We showed that any existing application written on MQSeries Publish/Subscribe can be run on MQSeries Integrator without requiring any code changes. In fact the Publish node honors all the information contained on the MQRFH header added by MQSeries Publish/Subscribe applications.

Even this simple migration path to MQSeries Integrator Version 2-based publish/subscribe has several benefits:

- ▶ ACLs can be associated to topics also for MQRFH publications.
- ▶ Message content can be altered before publication from within the publication message flow.
- ▶ Improved administrative tooling is available (for example, the Subscription view in the control manager can be used to monitor subscriptions and to forcibly remove them).

Moreover, messages published by MQSeries Publish/Subscribe applications can be subscribed to MQSeries Integrator applications (that is, applications using an MQRFH2 header).

In the following subsections we will discuss several more advanced aspects of MQSeries Integrator-based publish/subscribe:

- ▶ Support for MQSeries Publish/Subscribe streams: this feature as such is not supported by MQSeries Integrator, but a minimal support for compatibility purposes is guaranteed.
- ▶ Support for subscription points: this is a very powerful MQSeries Integrator-only feature.

- ▶ MQSeries Integrator broker networks and MQSeries Integrator broker collectives.
- ▶ MQSeries Integrator Topic-based security.

We will demonstrate each feature with an example.

## 5.2.1 Streams handling in MQSeries Integrator

Strictly speaking, MQSeries Publish/Subscribe streams are *not* supported by MQSeries Integrator, but a limited functionality is provided for applications originally written for MQSeries Publish/Subscribe.

There are three possible scenarios:

1. An application uses the MQ Publish/Subscribe default stream. There is no special handling for this case, the MQSeries Integrator topic that is built is the same as the original one.
2. An application specifies a non-default value for MQPSSStreamName in the MQRFH header: the MQSeries Integrator topic becomes `$/SYS/STREAM/StreamName/TopicName`.
3. An application publishes on a queue (for example, MY.STREAM) without specifying any MQPSSStreamName parameter in the header, but the MQSeries Integrator Publish node in the message flow serving MY.STREAM has the Implicit Stream Naming parameter set to true. The MQSeries Integrator topic becomes `$/SYS/STREAM/MY.STREAM/TopicName`.

MQRFH subscribers never specify or receive the `$/SYS/STREAM/StreamName` topic prefix. On the other hand, MQRFH2 subscribers need to explicitly specify the prefix in the subscription request, but they will not receive it in the published messages forwarded by the broker.

In MQSeries Publish/Subscribe, streams are mainly used for:

- ▶ Mapping publish/subscribe access controls to normal MQSeries queue-level access controls
- ▶ Enabling parallel processing within the broker

MQSeries Integrator provides a topic-level ACL feature and multiple instances of message flows (on the same or different queues) that satisfy the above requirements in a different and more flexible way.

Our example applications publish only on the default stream, so the topic that is built is the same as the original MQSeries Publish/Subscribe one. One way to check this is to inspect the Subscriptions tab of the Control Center while the subscriber application is running (see Figure 5-4 on page 119), where the registered subscription appears simply as PublicTransport/#. The hash sign is the equivalent of the “\*” wildcard in MQSeries Publish/Subscribe. Relevant character conversions are automatically applied by MQSeries Integrator.

See 5.2.5, “Example - migration of applications using streams” on page 130 for a detailed example.

## 5.2.2 Subscription points

In MQSeries Integrator all messages are published, or subscribed to, using a subscription point name and a topic name. The default subscription point is “” (empty string).

Subscription points are used to publish the same message on the same topic in several different formats (for example, some message elements, such as the decimal point sign or date can be internationalized to suit the user).

Subscriber applications will then specify a topic and (optionally) the requested subscription point.

A typical MQSeries Integrator message flow publishing messages in a subscription point manner will contain several Publication nodes wired to the output of distinct Compute nodes dealing with the construction of the various message formats.

This functionality is fully available only for applications using the MQRFH2 header. Messages with an MQRFH header can be published on an arbitrary service point, but MQRFH subscribers will have access only to those published on the default subscription point.

**Note:** Subscription points do not partition the topic name space, but the set of the published messages.

To clarify this we will use a short example.

Let’s suppose we have an application that publishes confidential messages on topic SpooksInfo/Nukes using two different subscription points: one named CIA.SP supporting the American English language, and one named KGB.SP supporting the Russian language.



When we use an ACL to restrict subscription access to the confidential SpooksInfo/Nukes topic, we do this irrespective of the subscription points the messages will be published to.

The general idea is that the *semantics* of a published message is associated with its topic, while the set of allowed *syntactical realizations* are associated with the supported subscription points.

See 5.2.6, “Example - message translation using subscription points” on page 133 to see subscription points at work.

### 5.2.3 MQSeries Integrator broker networks and collectives

An MQSeries Integrator broker network is a hierarchy of nodes. Each node can be a simple MQSeries Integrator broker or a *collective* of MQSeries Integrator brokers.

From a topological point of view, a collective is a completely connected subgraph that participates in the hierarchy as a single entity. Some of the brokers inside a collective can be connected to brokers outside the collective, acting as gateways on behalf of all collective members (for example, Broker1 in Figure 5-5).

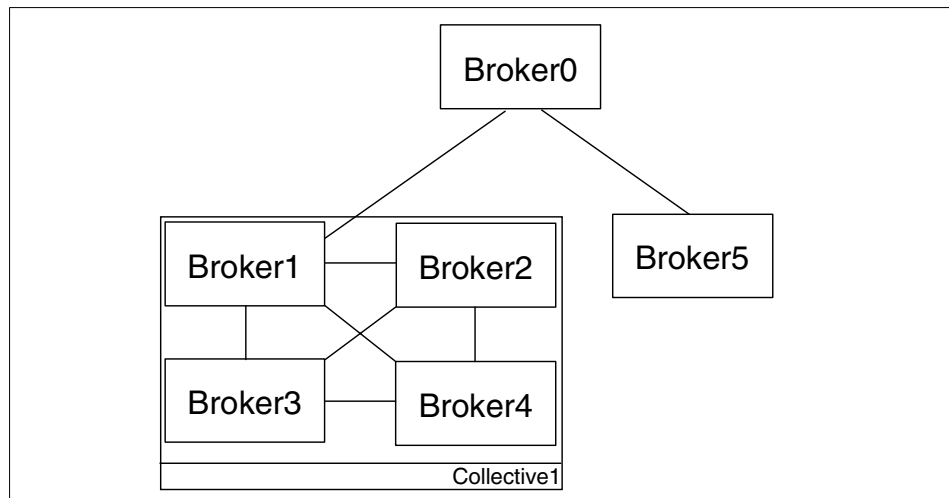


Figure 5-5 Pure MQSeries Integrator broker network with a collective

From an MQSeries intercommunication point of view, two connected MQSeries Integrator brokers, within or outside a collective need two-way MQSeries connectivity.

**Tip:** A natural fit for MQSeries networks hosting brokers that are members of the same MQSeries Integrator collective is the MQSeries Clustering feature.

Similar to what happens for MQSeries Publish/Subscribe broker networks, when a subscription is received by one broker, it causes a set of proxy subscriptions to be created from that broker against neighbor brokers. This way if a message is published anywhere in the network on a topic matching the subscription, it will be forwarded to the subscribed application (via its subscription broker).

As in MQSeries Publish/Subscribe broker networks, the broker takes care to avoid unnecessary subscriptions against neighbor brokers, in order to optimize the flow of messages on the networks. The message flows needed to support interbroker communications are automatically maintained by MQSeries Integrator. The administrator only needs to ensure that the broker runs at least an execution group.

The main tool for controlling the flow of information in an MQSeries Publish/Subscribe broker network is the use of streams. In MQSeries Integrator broker networks the access to single topics is instead regulated via ACLs, which operate on a domain-wide basis.

When groups of publishers and subscribers using different brokers are highly coupled on particular topic subtrees, it is a good idea to group the involved brokers in a collective, because the intra-collective communications are always direct (that is, without any intervening broker).

All interbroker subscriptions are always topic based, without any content filter specification. This means that the selection of the subset of messages matching a particular user-defined filter happens only on the broker owning the real user subscription.

It should now be apparent that an accurate design of the topic hierarchy is needed in order to exploit advanced publish/subscribe features, such as broker networks and collectives.

An overly coarse topic granularity may imply a systematic usage of content filtering that in a networked environment will not be used by the brokers to minimize the number of messages sent to their neighbors.

On the other hand the topic tree should include elements that are general enough to enable topic affinities to emerge and be exploited with collectives.

See 5.2.7, “Example - MQSeries Integrator broker networks” on page 137 for a discussion of an MQSeries Integrator broker network involving streams.

## 5.2.4 Topic-based security

MQSeries Integrator topics are hierarchies of topic qualifiers. The flat syntactical representation of them is a list of slash separated qualifiers.

In Figure 5-6 you can see a simplified view of the topic tree associated with our example application. The root topic of each hierarchy is the "" topic, that is the empty string.

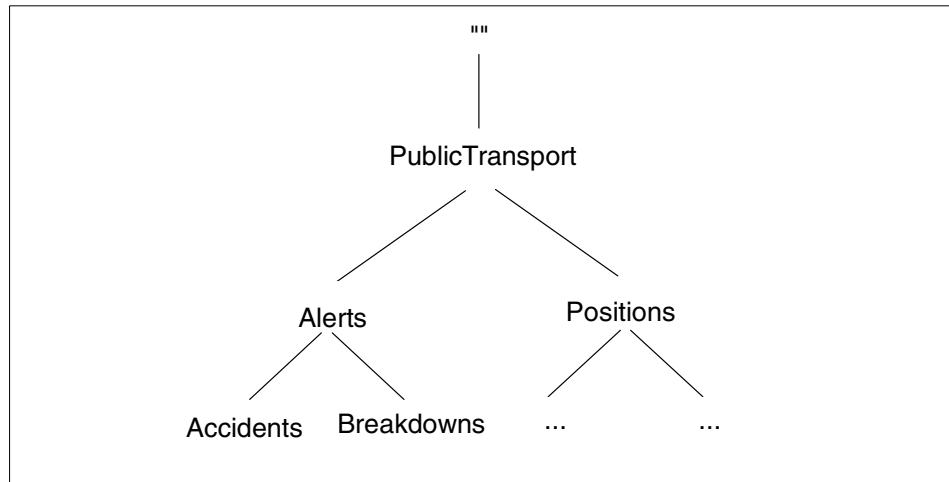


Figure 5-6 Example program topic tree

The flat syntactical representation of one of the supported topics is `PublicTransport/Alerts/Accidents`. Please notice that the root topic is not explicitly represented.

It is possible to associate ACLs to a subset or to all the topics supported by a hierarchy, using the Control Center. The ACLs specifications are then deployed to the target brokers that are in charge of enforcing them.

The Control Center and the target broker receive security domain-specific information from the User Name Server.

The following sections will explain the relationships between all MQSeries Integrator components involved in topic-based security, and the mechanics of topic ACLs.

## The User Name Server

The *User Name Server (UNS)* is an optional component of MQSeries Integrator that polls the operating system users and groups database at prefixed intervals (the default is 60 seconds) and extracts a list of available user IDs and a list of groups associated with each user ID.

This information is sent to all the MQSeries Integrator components (UNS client components) that registered for it with the UNS. These components are:

- ▶ The Configuration Manager
- ▶ One or more brokers

The link between one MQSeries Integrator component and the UNS is specified by passing the name of the UNS queue manager on the relevant parameter at the time the component is created.

The UNS is hosted by a queue manager, possibly the same one hosting the Configuration Manager and/or a broker.

The UNS does not need any support database, but requires two-way MQSeries connectivity with all its client components.

Typically there will be just one UNS per MQSeries Integrator domain, and it will be located on a machine having access to the operating system's user and groups database; for example, on Windows NT/2000 it will be a machine able to access the Security Account Manager (SAM) of the security domain specified when creating the Configuration Manager.

For more details on UNS location, see "Heterogeneous networks" on page 128.

## ACLs and inheritance

Using the Control Center topics view you can grant to each topic and principal (that is, user or group) the following capabilities:

- ▶ Publish: the user can publish messages on the topic.
- ▶ Subscribe: the user can subscribe to messages published on the topic.
- ▶ Persistent: the user can specify the subscription option requesting delivery of publication messages as persistent.

This is the only tool allowed in MQSeries Integrator to build a topic ACL.

The list of principals displayed by the Control Center is built from the information sent by the UNS to the Configuration Manager.

There is also an extra MQSeries Integrator generated principal named “Public Group” to which all users implicitly belong. This group cannot be deleted.

Every topic in the topic namespace has got an explicit or implicit ACL associated with it. An explicit ACL is one that the administrator defined using the Control Center for that topic. An implicit ACL is one that is inherited from a parent topic.

If no parent topic has an explicit ACL, then the ACL is inherited from the root topic. The root topic always has an ACL that by default enables all members of the Public Group to publish, subscribe, and request persistent delivery.

The inheritance mechanisms plays a key role for dynamic topics, that is topics that are programmatically generated by applications.

In our example application, the administrator could restrict the access to the topic to PublicTransport/Positions/Trains without specifying all the dynamically generated Route and Vehicle topic qualifiers, but when an application subscribes to topic PublicTransport/Positions/Trains/London/HogwartsExpress/WizTrain077 the ACL that is checked by the broker is the one inherited from PublicTransport/Positions/Trains.

## **Planning introduction of topic-level security**

When you plan to support publish/subscribe messaging in MQSeries Integrator environment you may not be concerned by topic security at first, but you may decide to introduce it afterwards.

**Tip:** It is recommended that you always install a UNS and create brokers and Configuration Manager accordingly, even when the topic security feature is not being used.

UNS overhead is minimal (and its polling interval can be tuned to be high), but when the info structure is in place, switching on topic-level security is as easy as creating the first ACL.

Until that time all access control checks will transparently resolve to the ACL inherited from the root topic, which by default grants full publish and subscribe capability to all users on all topics.

**Important:** When starting to use topic-level security in a production environment we recommend that you alter the default ACL for the root topic to a more conservative setting.

## Heterogeneous networks

MQSeries messaging access control uses principals from the operating system of the platform where the queue manager is running. In a heterogeneous network, these principals will come from distinct security domains (for example, a password file on a UNIX box and a SAM database on Windows NT domain).

In order to use the user ID stored in the message descriptor for access control (for example, specifying PUTAUT CTX on the definition of a receiver channel), there must be some form of coordination between the administrators in the creation of principals on the relevant security domains.

Similarly MQSeries Integrator access control mechanisms are designed to run in a heterogeneous environment where there are policies that guarantee a consistent naming convention and principal replication (where needed) across different security domains.

This means that if we restrict the observation to the principals involved in MQSeries Integrator-related activities, we can consider these principals as belonging to a unique namespace.

Let's give a few simple examples of what this implies.

### ***Example 1***

Let's suppose that:

- ▶ All the MQSeries Integrator brokers and the Configuration Manager run on a single Windows 2000 domain.
- ▶ All the applications use a domain user ID (and not a user ID local to their machines).
- ▶ The UNS runs on a box from where it can access the domain-wide user ID and group definitions.

This is really a single namespace. The UNS will detect any changes in the domain principals and will communicate them to the brokers and the Configuration Manager.

## **Example 2**

Let's suppose that:

- ▶ The publishing application connects to an IBM AIX broker using the UNIX user ID john.
- ▶ The subscribing application connects to Windows 2000 broker using Windows domain user mary.
- ▶ The two brokers are connected together in the MQSeries Integrator topology.
- ▶ The Configuration Manager is a Windows 2000 machine on the same domain as all the MQSeries Integrator brokers that run on a single Windows 2000 domain.
- ▶ The UNS runs on a Windows 2000 machine, and monitors the Windows 2000 domain principals.

In this environment in order to be able to include john in an ACL that was created using the Control Center and to enable the Windows 2000 broker to correctly process messages published by john, the user john must be defined in the Windows NT domain, even if it will never be explicitly used.

Strictly speaking this is not a single namespace, but the Windows 2000 domain principals are a superset of the principals involved in MQSeries Integrator activities on the heterogeneous network.

The general rule is that the UNS should run on the machine holding the definitions of all the MQSeries Integrator-related principals. Given that the Configuration Manager component of MQSeries Integrator runs only on the Windows NT/2000 platform, the UNS usually runs on this platform as well, typically on the same domain as the Configuration Manager.

It should be now apparent that having a single instance of the UNS per MQSeries Integrator domain is not a limitation, provided that it is located on a node that can access a security domain containing all the principals used on the heterogeneous network (some of them natively belonging to that security domain, while others were created only to replicate a foreign principal).

**Restriction:** In contrast with what happens for MQSeries access control, MQSeries Integrator UNS does not collect Windows NT/2000 SID or Windows NT/2000 domain-qualified user ID.

In 5.2.8, “Example - confidential publish/subscribe environment” on page 138 we show some of the security concepts discussed in this section, using the Vehicle application scenario.

## 5.2.5 Example - migration of applications using streams

An example of a publish/subscribe application using streams is the soccer sample included in MQSeries Publish/Subscribe SupportPac.

The sample is made of two applications:

- ▶ The match simulator program (amqsgam.exe) that publishes non-retained messages on topics Sport/Soccer/Event/MatchStarted, Sport/Soccer/Event/ScoreUpdate and Sport/Soccer/Event/MatchEnded at stream SAMPLE.BROKER.RESULTS.STREAM.
- ▶ The results service program (amqsres.exe) that subscribes to Sport/Soccer/Event/\* at stream SAMPLE.BROKER.RESULTS.STREAM and uses some service-retained topics to handle recovery logic.

We will describe how to migrate this MQSeries Publish/Subscribe sample to MQSeries Integrator.

### Initial configuration

In order to run the sample on a queue manager hosting an MQSeries Integrator broker, you first need to execute the following MQSC scripts that come with the sample: amqsfmda.tst, amqsgama.tst, amqsresa.tst.

Publication messages will be sent by the game simulator to the stream queue SAMPLE.BROKER.RESULTS.STREAM. This queue has no special meaning for MQSeries Integrator and by default is not served by any user message flow or internal MQSeries Integrator component.

Using the Control Center we will create a modified version of the Default Publish/Subscribe message flow named WMQ Pub/Sub Soccer Sample, whose input node is configured to read from the SAMPLE.BROKER.RESULTS.STREAM queue (see Figure 5-7 and the WMQPubSubSoccerSampleMsgFlow.xml file in the additional material accompanying this redbook).



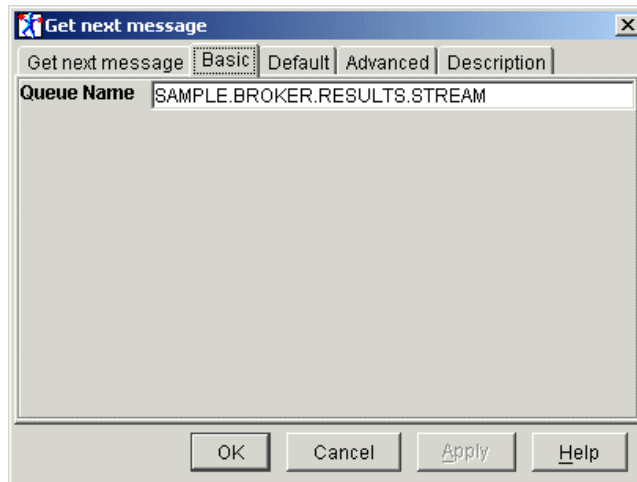


Figure 5-7 MQSeries Pub/Sub Soccer Sample message flow MQInput node

The modified message flow is then deployed to the target broker.

## Running the samples

Now we run the results service application specifying the broker to connect to (for example, amqsres ITSOB3), using the Subscription tab in the Control Center we can confirm the application started and successfully registered a subscription (see Figure 5-8 on page 132).

The topic reported by the Control Center is:  
\$SYS/STREAM/SAMPLE.BROKER.RESULTS.STREAM/Sport/Soccer/Event/#.

The \$SYS/STREAM/SAMPLE.BROKER.RESULTS.STREAM/ prefix has been added by MQSeries Integrator because the subscription came from an MQRFH client providing a non-default value (actually SAMPLE.BROKER.RESULTS.STREAM) for the MQPSSstreamName parameter.

The results service application is now waiting for results from instances of match simulator applications. Let's start a simulated game passing the name of two teams and a broker (for example, amqsgam Beauxbatons Hogwarts ITSOB3).

Since messages are displayed by the match simulator, *nothing* actually happens in the results service window!

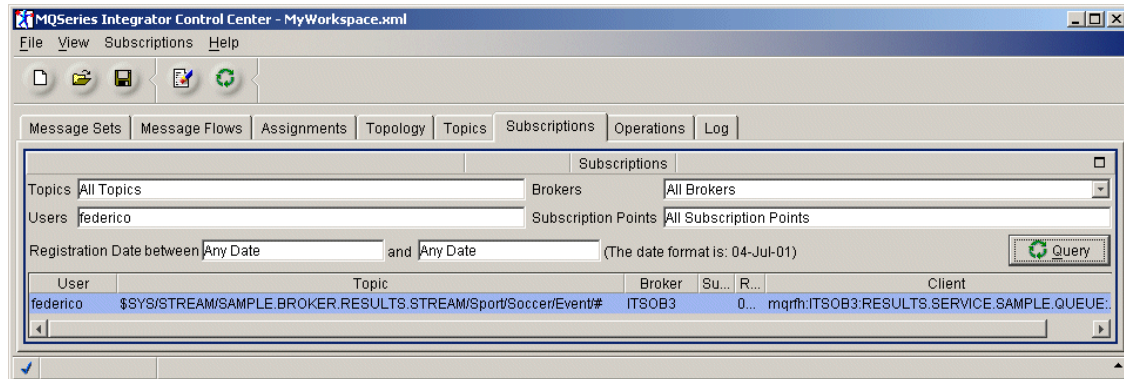


Figure 5-8 Checking results service subscription

## Revising the configuration

Given that the subscription side of the application seems correct (see Figure 5-8), we will now have a closer look at the publishing side: the match simulator program.

We stop the message flow and start a simulated game. This way messages on queue `SAMPLE.BROKER.RESULTS.STREAM` will not be consumed. Using the MQSeries Explorer tool we browse the name/value pairs associated with one of the messages. They look like this:

```
MQPSCmd Publish
MQPSPubOpts NoReg
MQPSTopic Sport/Soccer/Event/MatchStarted
```

Each message is a publication specifying a topic and the fact that the publisher is not registered and a specific topic. The stream on which the message is published is not specified in the MQRFH name value, but is implicitly determined by the act of putting the messages to a specific queue.

When a message like this is received by MQSeries Integrator, it is treated as if published to the default stream (MQSeries Integrator could not determine a non-default stream name anyway), so the topic that is built for subscription matching is:

```
Sport/Soccer/Event/MatchStarted
```

Clearly this does not match the results service subscription on the topic:

```
$SYS/STREAM/SAMPLE.BROKER.RESULTS.STREAM/Sport/Soccer/Event/#
```

To force stream-aware behavior when no stream name is contained in the published messages, we check the **Implicit Stream Naming** option on the Publish node (see Figure 5-9) and redeploy the message flow.

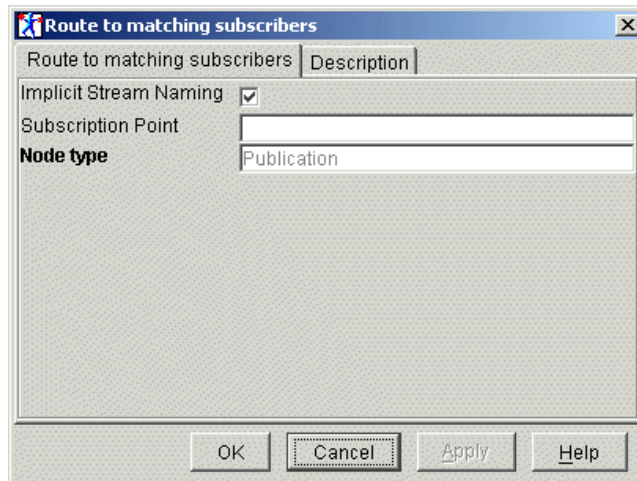


Figure 5-9 Forcing implicit stream naming on the Publish node

We now restart both the results service and the match simulator service applications, and the results are actually displayed as expected.

## 5.2.6 Example - message translation using subscription points

This example shows the usage of MQSeries Integrator subscription points. The reason for showing this feature is the translation of TrafficConditions XML tag values into French, supporting both MQRFH and MQRFH2 publishers and subscribers.

### Advanced publish/subscribe message flow

The message flow we built (see Figure 5-10) is included in the additional material as the AdvancedPubSubMsgFlow.xml export file.

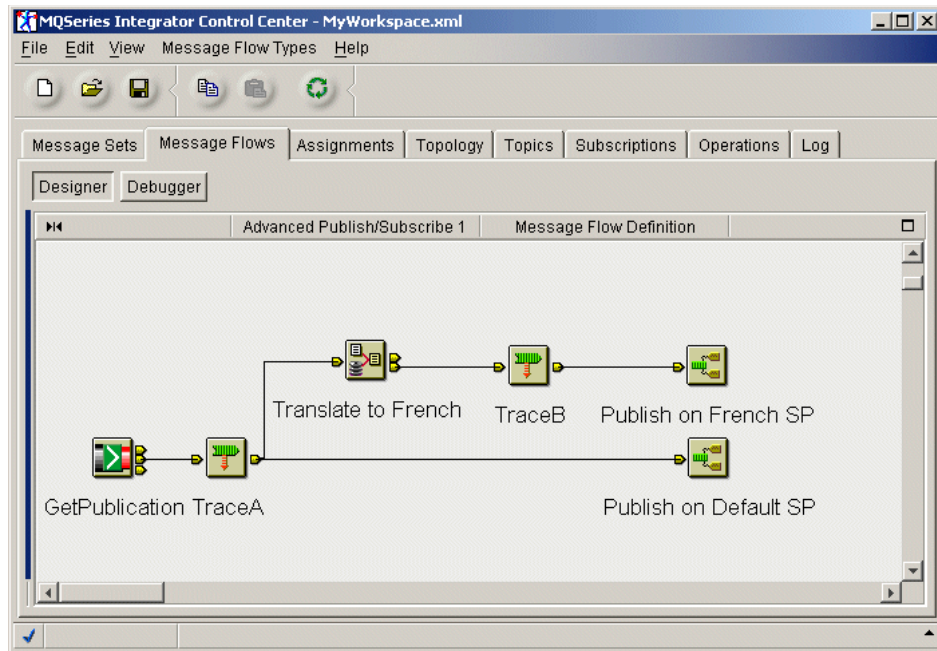


Figure 5-10 The Advanced Publish/Subscribe message flow

In order to exploit subscription points, we must subscribe to messages using an MQRFH2 header. All MQRFH subscribers will only have access to the default subscription point. To ensure backward compatibility with all the applications written so far, we will use the default subscription point for the unchanged English language publications, while we will use a new FRENCH.SP subscription point for the French language messages.

The structure of the message flow is simple. It is made of an MQInput node getting publication messages from SYSTEM.BROKER.DEFAULT.STREAM, and forwarding them in parallel to two targets:

- ▶ One is the Publication node publishing on the default subscription point.
- ▶ One is the Compute node handling the translation to French, followed by a Publication node publishing on the FRENCH.SP subscription point.

### Running publishers and subscribers

To run the example, import the AdvancedPubSubMsgFlow.xml file into your workspace and deploy it to your broker (if you already have the Default Publish/Subscribe message flow serving SYSTEM.BROKER.DEFAULT.STREAM queue, remember to remove it).

In order to show the MQRFH2 subscription with subscription points we use the SupportPac IA71 Static subscriber registration utility that can be downloaded from:

<http://www-4.ibm.com/software/ts/mqseries/txppacs/ia71.html>

**Note:** Only versions higher than 1.0 of SupportPac IA71 include support for subscription points.

Using the utility we subscribe to MQRFH2 messages in English with queue VEHICLE.MQRFH2.EN.SUB.QUEUE and to MQRFH2 messages in French using VEHICLE.MQRFH2.FR.SUB.QUEUE and FRENCH.SP subscription point (see Figure 5-11).

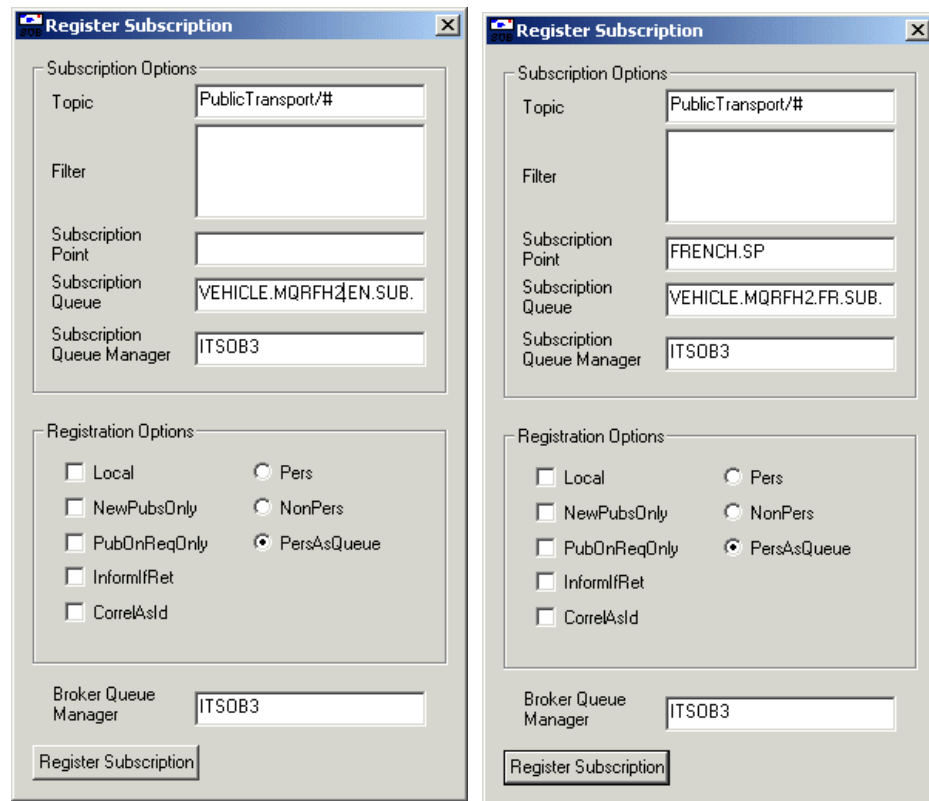


Figure 5-11 Creating static subscriptions

We create an MQRFH subscription by starting the example subscriber application developed in Chapter 4, “The publish/subscribe application” on page 27, which by default receives messages on VEHICLE.SUB.QUEUE.

The MQSeries Integrator broker now has three subscriptions registered on topic PublicTransport/#:

- ▶ An MQRFH subscription on the default subscription point
- ▶ An MQRFH2 subscription on the default subscription point
- ▶ An MQRFH2 subscription on FRENCH.SP subscription point

To confirm this, we use the Subscriptions tab of the Control Center and see what depicted in Figure 5-12.

We now start the publishing application (any API flavor of it is supported) and we will see messages appearing on the subscriber application window. Using the MQSeries Explorer tool we can check the contents of the other two queues. They contain the same number of messages (in MQRFH2 format) and those in VEHICLE.MQRFH2.FR.SUB.QUEUE will be translated in French.

The message flow worked as expected because:

- ▶ MQRFH2 subscribers received the publications in the MQRFH2 format (even if the original one was MQRFH1) and in the requested language
- ▶ MQRFH subscribers worked as before
- ▶ All subscribers received only one copy of each published message

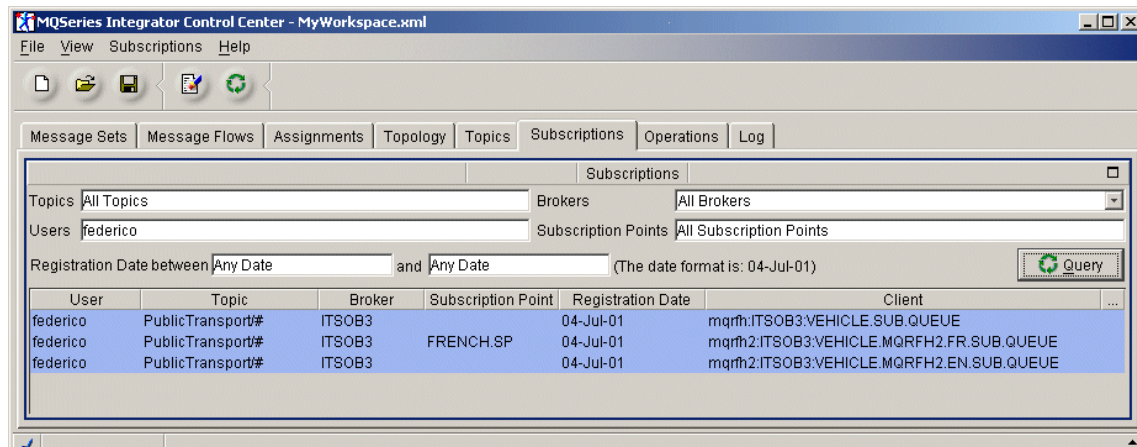


Figure 5-12 Heterogeneous subscriptions

## 5.2.7 Example - MQSeries Integrator broker networks

Let's suppose that the British transport system is made up of just the London underground and the Dover hovercraft, and that the whole system is controlled by a central administrative body.

These are the main characteristics of the hypothetical system:

- ▶ Hovercrafts publish essential real-time information about their status.
- ▶ There are only a few dozen scheduled hovercrafts trips per day, all on a single route from Dover to Calais.
- ▶ Underground trains publish detailed real-time information about their status.
- ▶ There are more than 100 underground trains continuously shuttling back and forth on a dozen routes each having dozens of stops.
- ▶ All the published information is consumed by travel agents located in London, apart from a few statistical topics that are subscribed by the central administrative body.

Now we will discuss a proposed MQSeries Integrator architecture that may accommodate these requirements in an optimal way (see Figure 5-13).

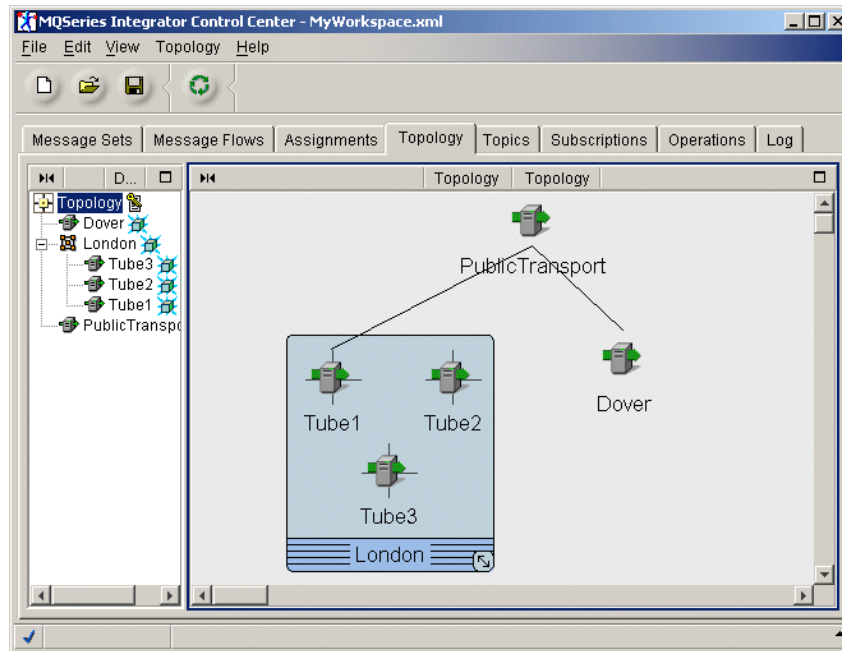


Figure 5-13 A complex MQSeries Integrator topology view with Control Center

Given that the London underground system produces the biggest share of published messages and London site run almost all the subscriber applications, we decide to create a collective for the underground system. This collective will also run travel agent subscriber applications.

In order to guarantee an adequate level of service, we decide to support the remote Dover site using a locally running broker.

The central body is supported by a dedicated broker that is also the root of a hierarchy.

These are the main features of the proposed architecture:

- ▶ Most of the messages *never leave the collective*. This is a good design point given that intra collective communications are highly optimized.
- ▶ Most of Dover messages are subscribed by applications connected to brokers belonging to the collective, by positioning the central body broker between the collective and the Dover broker we potentially minimize the number of interbroker subscriptions (the gain will become bigger as we extend the network to support trains, ships, etc.).

## 5.2.8 Example - confidential publish/subscribe environment

Let's suppose that the example vehicle applications will be used not only for normal public transport subsystems, but also by the military community to keep track of military convoy movements.

The specifications mandate a strict access control over the publish/subscribe activity on some military reserved topics, without disrupting the service for the already deployed non-confidential subsystems (for example, Tube and Trains).

### Environment setup

The Windows 2000 principals involved in this example are the following:

- ▶ `mqm`: this is the MQSeries-generated group. Members of this group have complete authority on MQSeries resources.
- ▶ `MilitaryPersonnel`: members of this group consume confidential information.
- ▶ `IntelligenceAgency`: members of this group provide confidential information.
- ▶ `JoeGI`: this user belongs to the `MilitaryPersonnel` and `mqm` groups.
- ▶ `JamesB`: this user belongs to the `IntelligenceAgency`, `MilitaryPersonnel` and `mqm` groups.
- ▶ `Goldfinger`: this user only belongs to the `mqm` group.



All the position information about military convoys will be published on subtopics of PublicTransport/Positions/Convoy.

We used the Control Center to restrict publication access on this topic to group IntelligenceAgency (as in Figure 5-14), and subscription access to MilitaryPersonnel.

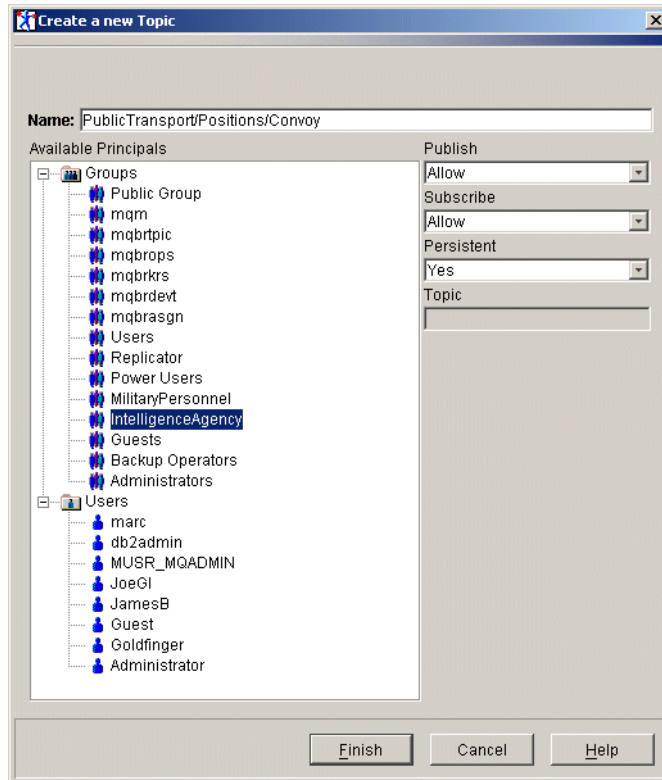


Figure 5-14 Restricting publication access to a confidential topic

Similarly we denied publish and subscribe access to the protected topic to PublicGroup. The reason for this will become apparent as we proceed in the example discussion.

The final results shown by the Control Center should be like those in Figure 5-15 on page 140. The new ACL can now be deployed to the broker by clicking **File->Deploy->Delta configuration (all types)**.

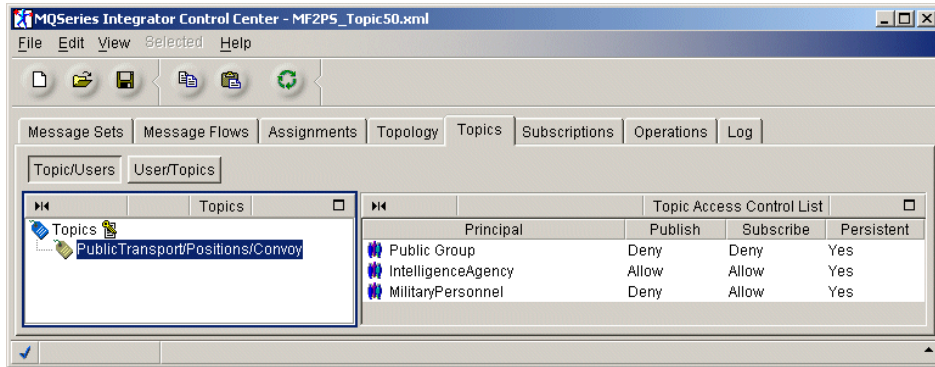


Figure 5-15 An ACL as displayed by the Control Center

When setting ACLs we only used group principals. Similar results could be obtained by using the user ID directly, but in a production environment this entails a bigger administrative overhead. No functionality is lost by using just group principals. In fact, the UNS delivers updated group membership information to all its client brokers for them to perform the actual user access control.

## Running the publisher and the subscriber applications

Before starting the Vehicle publisher, edit its property file to include at least a new vehicle on a route with the Convoy mode. Since some of the features we will discuss are based on the fact that the position information is published as retained, you should:

- ▶ Edit the publisher program properties file and set the pubType property to 2 (this will force the publisher to use AMI).
- ▶ Check that the VEHICLE.POSITION.PUB.POLICY policy in your AMI repository id set to publish as retained.

A suitable properties file named `convoy_pub.properties` is included in the additional material accompanying this redbook. You can rename it `pub.properties` and copy it into your publisher application directory.

Let's now review the results obtained by running the publisher and the subscriber application, logging to the system using the three involved user IDs in turns.

### JamesB results

All the publication messages produced by user JamesB are received and displayed by the subscriber GUI. These messages can be classified in for groups:

- ▶ Public positions: the user can publish and subscribe to these messages because the ACL inherited from the root topic allows him to do so.

- ▶ Classified positions: the user can publish and subscribe to these messages because the ACL inherited from `PublicTransport/Positions/Convoy` allows him to do so.
- ▶ Alerts (public): the user can publish and subscribe to these messages because the ACL inherited from the root topic allows him to do so.

### ***JoeGI results***

Not all the publication messages produced by user JoeGI are received and displayed by the subscriber GUI:

- ▶ Public positions: published and received.  
The user can publish and subscribe to these messages because the ACL inherited from the root topic allows him to do so.
- ▶ Classified positions: not published (and thus not received).  
The ACL inherited from `PublicTransport/Positions/Convoy` does not allow the user to publish on these type of messages, but he can subscribe to them. This is demonstrated by the fact that as soon as the subscriber application starts it receives the *retained* classified position messages previously published by the authorized user JamesB.
- ▶ Alerts (public): the user can publish and subscribe to these messages because the ACL inherited from the root topic allows him to do so.

The failed attempt by JoeGI to publish on a topic he is not authorized leaves an audit trail message in the Windows 2000 Event Log (Figure 5-16 on page 142).

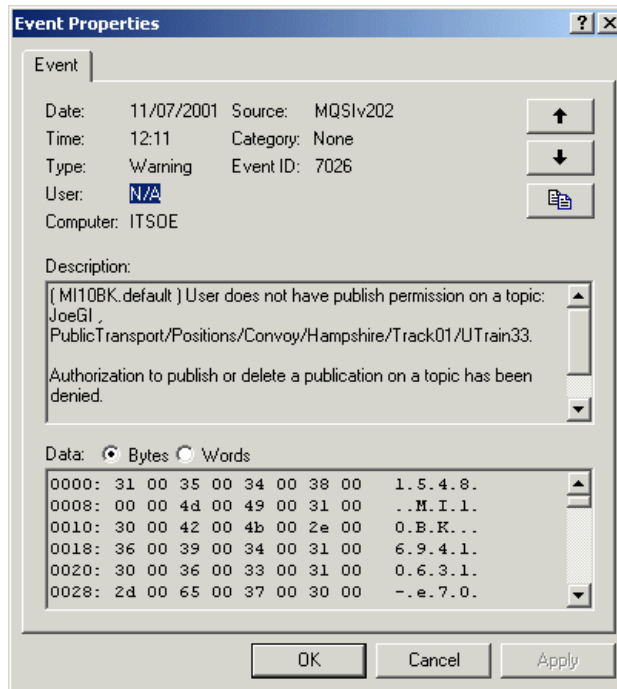


Figure 5-16 Topic access denied Windows 2000 Event Log message

A similar error message is not produced by the subscription side, but even if the subscriber application subscribes to topics matching the pattern `PublicTransport/*`, the broker will quietly forward only the publication messages on the subtopics to which to subscriber has subscription access.

### **Goldfinger results**

Not all the publication messages produced by user Goldfinger are received and displayed by the subscriber GUI:

- ▶ **Public positions: published and received.**  
The user can publish and subscribe to these messages because the ACL inherited from the root topic allows him to do so.
- ▶ **Classified positions: not published (and thus not received).**  
The ACL inherited from `PublicTransport/Positions/Convoy` does not allow the user to publish on these type of messages, nor he can subscribe to them This is demonstrated by the fact that as soon as the subscriber application starts it receives some retained position messages, but not the convoy-related ones (previously published by the authorized user JamesB), because an ACL prevents the user from receiving them.

- ▶ Alerts (public): the user can publish and subscribe to these messages because the ACL inherited from the root topic allows him to do so.

The test creates an audit trail similar to the one created for the JoeGI test.

## Final considerations

This example shows several interesting points about ACL inheritance.

The first thing to notice is that all the access control checks in the example are based upon inherited ACL, in particular:

- ▶ Applications publishing or subscribing to alert information inherit the ACL from the topic root, that is no explicit ACL has been created on this topic subtree.
- ▶ Applications publishing or subscribing to classified positions always deal with subtopics of the `PublicTransport/Positions/Convoy` topic, and inherit the ACL from this latter topic.
- ▶ Applications publishing or subscribing to public positions inherit the ACL from the topic root, that is no explicit ACL has been created on this topic subtree.

Another interesting point is that the ACL element preventing `Goldfinger` from publishing or subscribing to classified positions is the `PublicGroup` explicitly being denied access to the topic `PublicTransport/Positions/Convoy`.

If we omitted this element from the ACL, the `Goldfinger` ACL on that topic would have been inherited from the root topic that by default enables access to everyone.

**Tip:** When you are restricting access to a topic, always deny access to the `Public Group` within the topic ACL, or ensure that the `Public Group` rights on ancestor topics (for example, the root topic) are set to a safe value.

An easy way to lock `Goldfinger` out of the whole publish/subscribe domain would be to deny him access to the root topic as shown in Figure 5-17 on page 144.

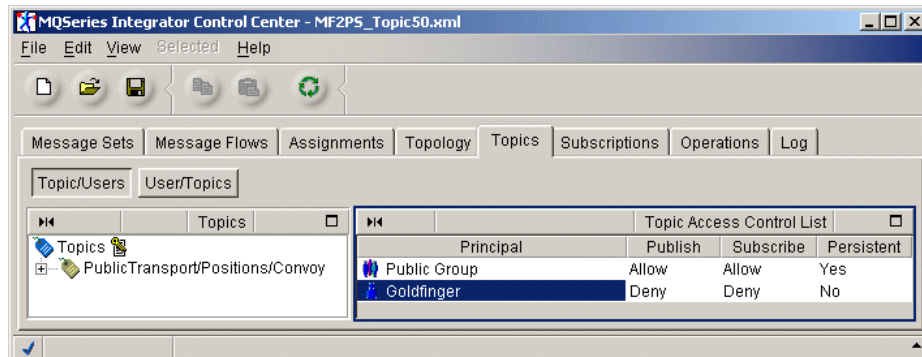


Figure 5-17 Locking Goldfinger out of the publish/subscribe domain

Another anomaly weakens the security of our example: alerts referring to military convoys vehicles in the real world would be as confidential as the classified position messages; unfortunately there is no way to enforce the intuitively needed access control check, because all alert messages share the same PublicTransport/Alerts topic.

In general, there is no way in MQSeries Integrator to implement content-based access control on single publish/subscribe messages.

**Tip:** consider security-related issues when designing the topic hierarchy.

## 5.3 Other forms of interoperability

MQSeries Publish/Subscribe interoperability with MQSeries Integrator Publish/Subscribe can be seen from three points of view:

- ▶ Application-level interoperability: execution of MQSeries Publish/Subscribe applications against an MQSeries Integrator.
- ▶ Migration/upgrading: migration of an established production MQSeries Publish/Subscribe broker to MQSeries Integrator (including retained publications, registered subscriptions, registered publications and topology information).
- ▶ Broker-to-broker interoperability: setup and management of mixed MQSeries Publish/Subscribe and MQSeries Integrator broker networks.

The first aspect has been completely covered by the initial part of this chapter. We will now touch quickly on the other two aspects.

### 5.3.1 Mixed broker networks

A mixed broker network is a hierarchy of MQSeries Publish/Subscribe brokers and MQSeries Integrator brokers. There are no constraints on the position of a particular type of broker in the hierarchy.

For example an MQSeries Integrator broker can be the child of an MQSeries Publish/Subscribe or vice versa.

Most of the general concepts and building techniques of MQ Publish/Subscribe broker networks still apply to mixed broker networks.

To prepare an MQSI broker to belong to an MQ Publish/Subscribe network, there are some preliminary actions to perform:

- ▶ You must create the default stream queue `SYSTEM.BROKER.DEFAULT.STREAM` and deploy a message flow to serve it. For example you can use the Default Publish/Subscribe message flow provided in the MQSeries Integrator samples workspace import file.
- ▶ For each stream that MQSeries Integrator broker will support, you must create a local queue named after the stream and deploy a message flow to serve it. For example you can modify the Default Publish/Subscribe message flow provided in the MQSeries Integrator samples workspace import file.
- ▶ You must create the local queue on which the broker will receive messages from neighbor brokers. This queue should be named `SYSTEM.BROKER.INTER.BROKER.COMMUNICATION`. The message flow serving this queue will be automatically created and deployed by MQSeries Integrator.

Now that the MQSeries Integrator broker internal configuration is ready, we must graft the broker to an existing hierarchy tree.

If the broker will be included in the network as a leaf node, you simply need to use the `mqsijoinmqpubsub` command to specify the name of the parent broker.

If the broker is included as intermediate node in a network, you must:

- ▶ Sever existing MQSeries Publish/Subscribe broker connections using the MQSeries Publish/Subscribe command `c1rmqbrk`.
- ▶ Create all the connections from child brokers to the new MQSeries Integrator broker using MQSeries Publish/Subscribe command `strmqbrk` using the `-p` option to pass the name of the parent broker
- ▶ Specify the name of the MQSeries Integrator broker parent using the `mqsijoinmqpubsub` command.

Subscriptions and publications propagation works in a mixed broker network in the same manner as in a pure MQSeries Publish/Subscribe broker network.

See 5.3.3, “Example - mixed broker networks” on page 146 for a discussion of a mixed broker network.

### 5.3.2 Migrating an MQSeries broker to MQSeries Integrator

The migrating process of MQSeries Publish/Subscribe broker to MQSeries Integrator broker will migrate the following items:

- ▶ Information about neighbor brokers
- ▶ Subscriptions
- ▶ Local publishers identities
- ▶ Retained messages on the default stream and all other user-defined streams

To migrate an MQSeries Publish/Subscribe broker you will have to:

- ▶ Define the MQSeries Integrator broker using the command `mqsicreatebroker` specifying the optional `-m` option that requests the migration
- ▶ Use the MQSeries Publish/Subscribe command `migmqbrk`
- ▶ Deploy message flows to serve the default stream and all other user-defined streams, as discussed in 5.3.1, “Mixed broker networks” on page 145

**Important:** the migration of an MQSeries Broker to MQSeries Integrator is *not* reversible.

### 5.3.3 Example - mixed broker networks

In a production environment the architecture Figure 5-5 on page 123 may have been the last step of a staged migration to MQSeries Integrator from an MQSeries Publish/Subscribe environment.

If this was the case, maybe the Dover site would have been the last to be migrated, given that its requirements were less demanding than the London site.

During the transition period, Dover MQSeries Publish/Subscribe broker would have been reporting to the already migrated PublicTransport broker without disruption of the service.

The root MQSeries Integrator broker would have defined a set of message flows servicing the stream queue supported by Dover, in order to publish the information on the general MQSeries Integrator broker domain network.



The Control Center would not have shown the MQSeries Publish/Subscribe broker at Dover, but relevant topological information could have been retrieved by running the command

```
mqsilistmqpubsub PublicTransport
```

at the PublicTransport MQSeries Integrator broker.





## Web enablement

This chapter describes the migration of the Java stand-alone subscriber application discussed in Chapter 3, “Example application” on page 19 to the WebSphere Application Server.

The publishing side of the application remains unchanged.

## 6.1 A simple Web-based subscriber

This chapter discusses the migration of a stand-alone application to a Web-based application with minimal changes to the existing application.

The best way to do this is to adopt a servlet-based solution, in which the servlet will cater to the browser request, contact the application bean and pass the necessary information. In turn the bean will process and return the result and the servlet will pipe back the output to the browser as a response.

Hence the core business logic and processing steps of the application bean remain the same except for a few changes to the interface of the bean, in order to accommodate servlet-specific calls.

In this Web scenario we demonstrate the use of a temporary dynamic queue as a subscription queue. When a temporary dynamic queue is specified as a subscription queue, the MQSeries Integrator Broker automatically deletes the subscription associated to it as soon as it discovers that the queue has been deleted (that is, the owning subscriber application is terminated without unsubscribing).

**Important:** Automatic subscription deregistration works only if the temporary dynamic subscription queue is local to the broker queue manager (otherwise, the broker would not be able to tell that it is a temporary dynamic queue) and is not accessed via an alias.

In order to keep the subscription lifetime to a minimum (optimizing the broker performance), the subscribing applications should explicitly deregister under normal circumstances. The automatic deregistration feature is intended only to avoid orphaned subscriptions in abnormal situations.

**Note:** In this Web-based subscriber application, we are only demonstrating the usage of a temporary dynamic queue in the AMI environment. The same application can be executed with a local queue as the subscriber queue by changing the subscriber and policy in the AMI repository.

The steps required to port an existing subscriber application to a Web-based application in the WebSphere environment include:

- ▶ WebSphere Application Server configuration
- ▶ Servlet configuration
- ▶ AMI Repository configuration
- ▶ Program invocation
- ▶ Discussion of the Web part of the application

## 6.1.1 WebSphere Application Server configuration

We will create a Web application named “itso” and configure the servlet.

The following section covers the steps to create a Web application. Refer to the *WebSphere V3.5 Handbook*, SG24-6161 for more information.

1. Start the WebSphere Application Server Administrator console by clicking **Start->Programs -> IBM WebSphere -> Application Server 3.5 -> Administrator's Console**.
2. Create a Web application named itso by clicking **Wizard -> Create a Web Application**. Leave the default option as it is and click **Next** as shown in Figure 6-1.

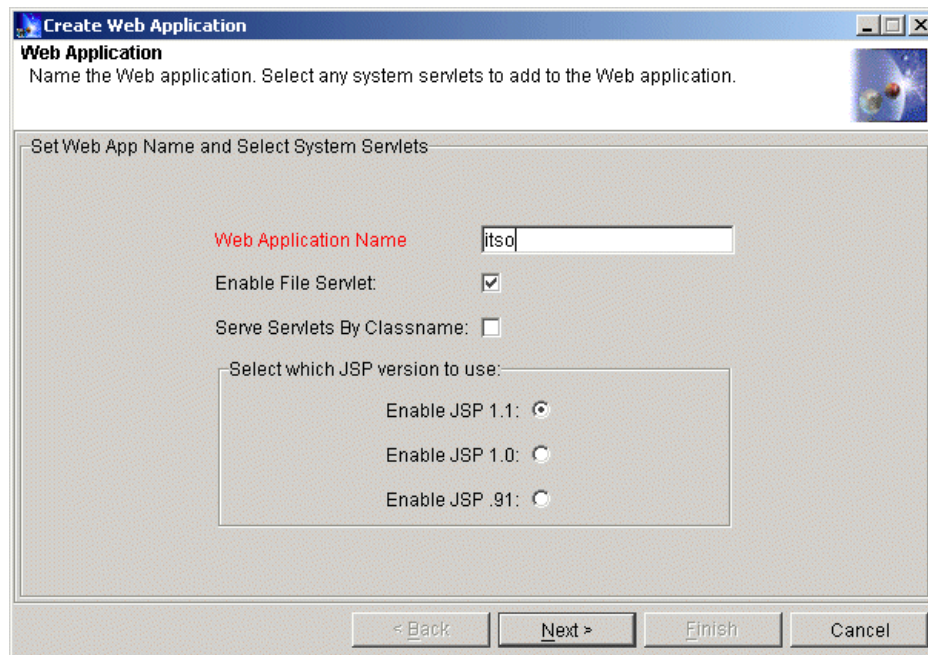


Figure 6-1 Step (1-4) to create a Web application

3. Select **Default Servlet Engine** as the parent Servlet Engine and click **Next** to display the window shown in Figure 6-2 on page 152.

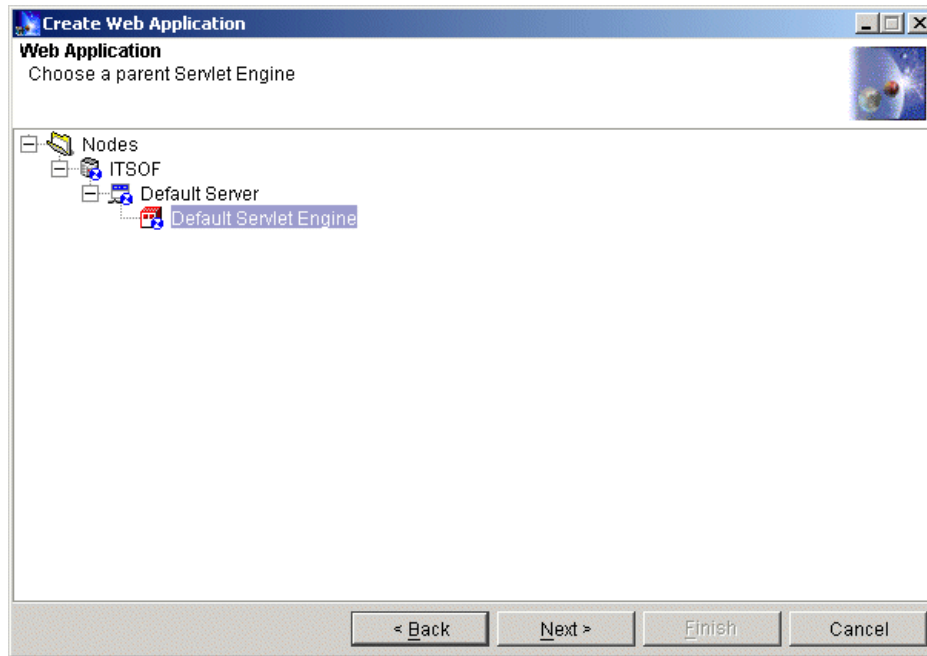


Figure 6-2 Step (2-4) to create a Web application

4. Change the Web Application Web Path to /its0 as shown in Figure 6-3 and click **Next**.

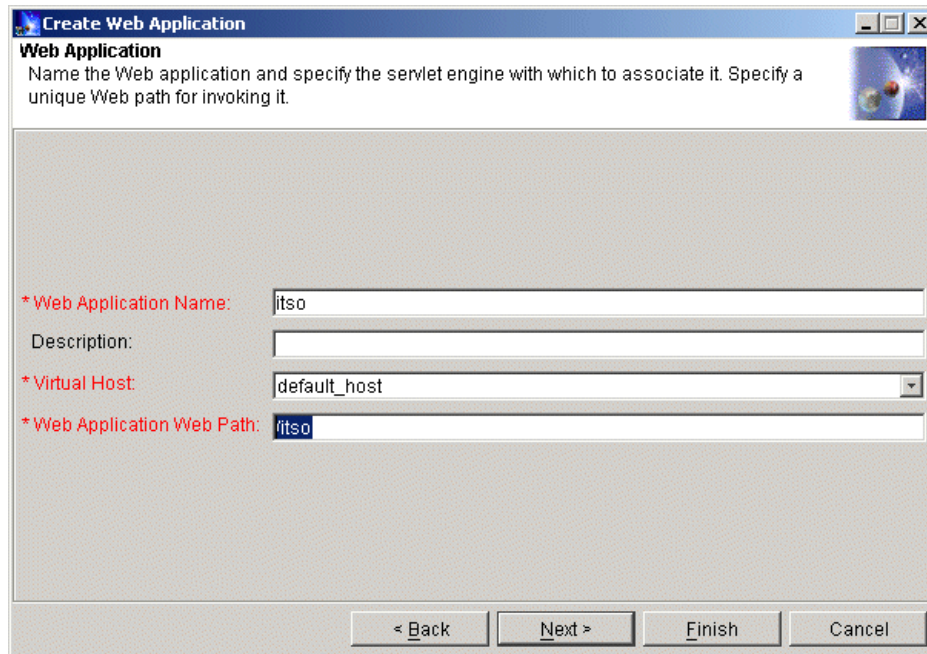


Figure 6-3 Step (3-4) to create a Web application

5. By default WebSphere specifies a default classpath and document root. Leave the default and click **Finish** as shown in Figure 6-4 on page 154. The Web application `itso` will be created.

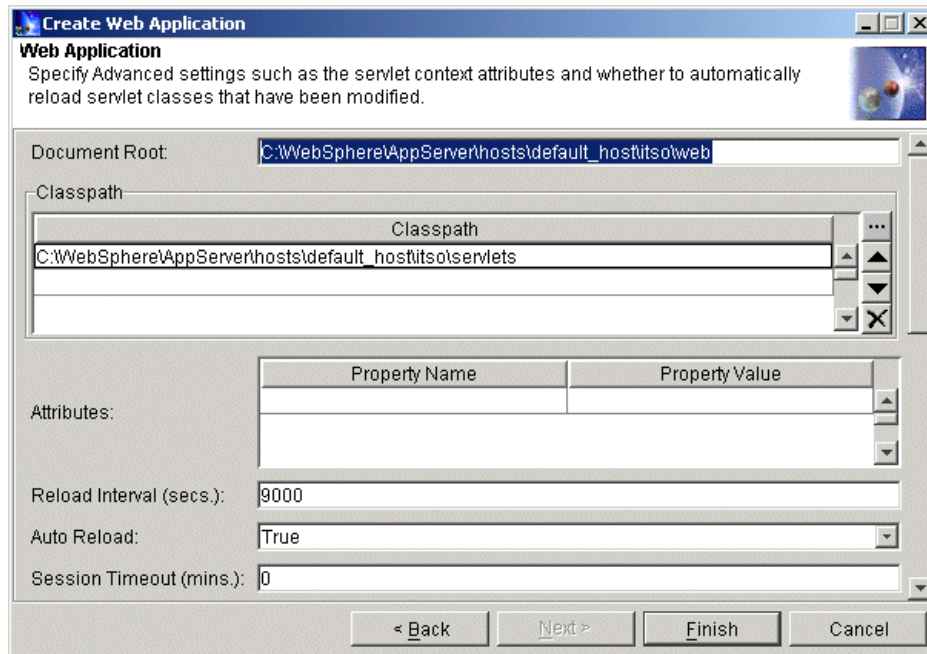


Figure 6-4 Step (4-4) to create a Web application

6. Since we have chosen the default classpath and document root, we must create the required folders as follows:
  - a. Create a folder called “itso” in the C:\WebSphere\AppServer\hosts\default\_host path, where C:\WebSphere\AppServer is the WebSphere root.
  - a. Create servlets and a folder called “Web” under the itso folder. The servlets folder should contain all the Java class files, whereas the Web folder should have the HTML pages, since we chose the **Enable File Servlet** option in step 1.
7. Copy the JAR file that contains the application class files into the C:\WebSphere\AppServer\hosts\default\_host\itso\servlets folder, where C:\WebSphere\AppServer is the WebSphere root directory. Copy the HTML files into the C:\WebSphere\AppServer\hosts\default\_host\itso\web folder. Refer to Appendix D, “The GUI-based subscriber application” on page 205 for the required JAR file and HTML file name.



**Important:** Users have the ability to decide where to put servlets and other class files, which may be different from the Web application default classpath.

## 6.1.2 Servlet configuration

The next step is to configure the servlet:

1. Right-click the **ITSO Web** application and create a servlet, as shown in Figure 6-5.

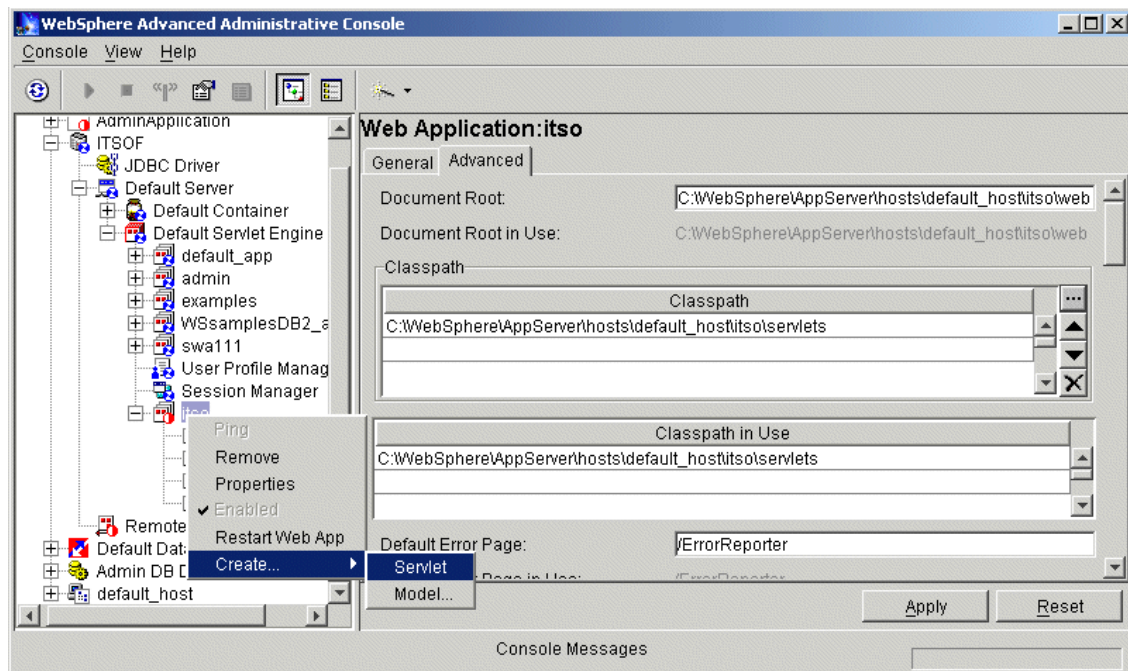


Figure 6-5 Step(1-2) to create and configure a servlet

2. In this step supply the servlet name as `WebSubscriber` and servlet class name as `com.ibm.itso.swa111.subscribe.web.WebSubscribe` as shown in Figure 6-6 on page 156 and click **Add**. Then add the servlet Web path name as `WebSubscriber` and click **OK**. The `WebSubscriber` servlet is created.

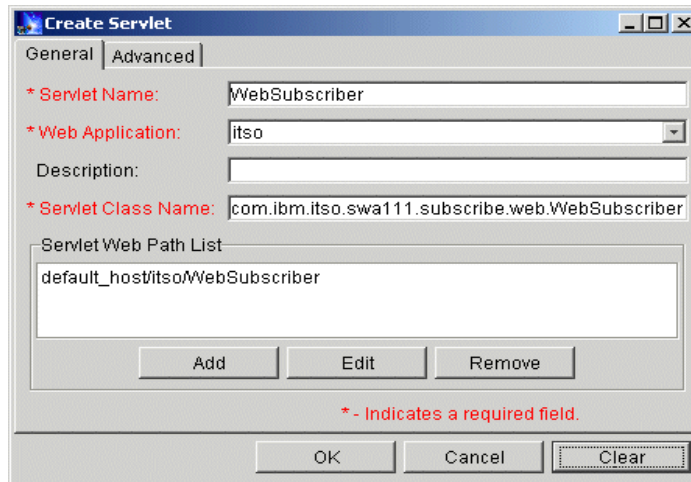


Figure 6-6 Step (2-2) of servlet creation on WebSphere

3. In this step, we assign the initial parameter to the servlet as shown in Figure 6-7 on page 157. The init parameter entries are shown in Table 6-1 on page 157.

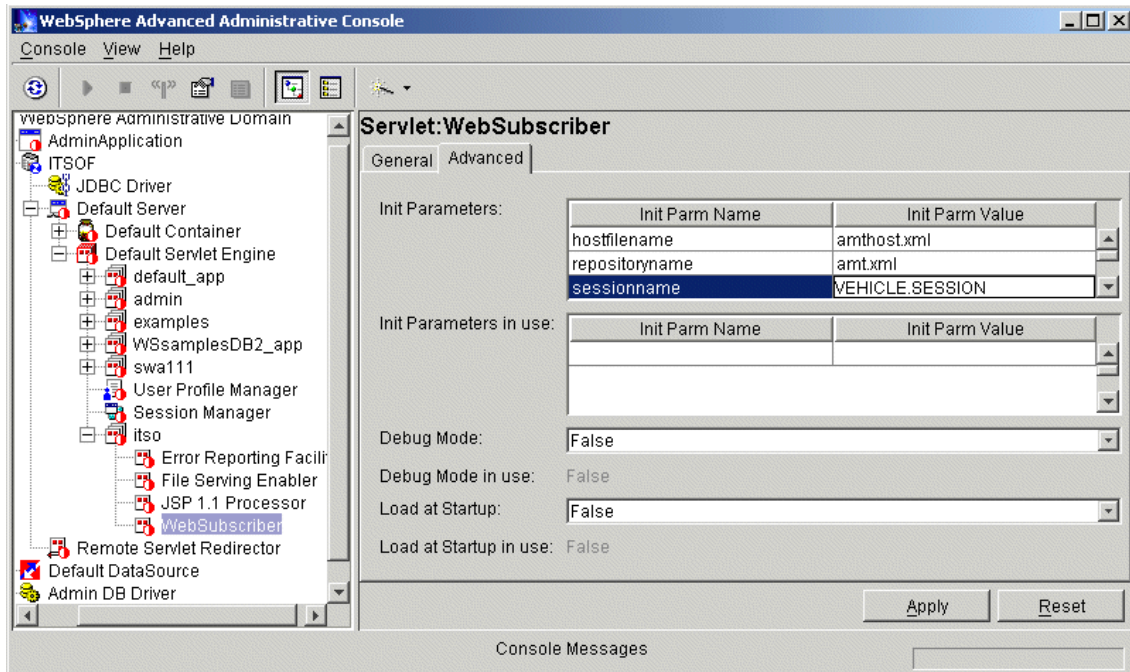


Figure 6-7 Setting up the Init parameter of servlet

Table 6-1 Init parameter list of the servlet WebSubscriber

Init Param Name	Init Param Value
hostfilename	amthost.xml
repositoryname	amt.xml
sessionname	VEHICLE.WEB.SESSION
policyname	VEHICLE.SUB.POLICY
subscribername	VEHICLE.SUBSCRIBER
messagereceiver	VEHICLE.WEB.RECEIVER.MSG
messagesubscriber	VEHICLE.WEB.SUBSCRIBER.MSG
WAITTIME	2000
TOPICNAME	PublicTransport/*

**Note:** The init parameter list is similar to the entries of subscriber.properties file as discussed in 4.10, “Sample subscriber application” on page 96. When the servlet loads in the application server, all the init parameters are passed to the servlet.

### 6.1.3 AMI repository configuration

Since we are using a dynamic queue for subscription, we define a new policy VEHICLE.WEB.SUB.POLICY and a new subscriber VEHICLE.WEB.SUBSCRIBER. In the following sections we discuss the important properties to be configured in the policy and subscriber.

1. Create a service point VEHICLE.WEB.SUB.DYNAMIC, which will represent a dynamic queue. We will use SYSTEM.DEFAULT.MODEL.QUEUE as the model queue name and VEHICLE.WEB.DYNAMIC.\* as the dynamic queue prefix. If the last non-blank character in positions 1 to 33 of the prefix is '\*', then the '\*' is replaced by a string that guarantees that the name generated is unique. Ensure the definition type of the model queue is temporary.
2. Create Subscriber VEHICLE.WEB.SUBSCRIBER. Assign BROKER.CONTROL.QUEUE as the sender service and VEHICLE.WEB.SUB.DYNAMIC as the receiver service.
3. Create policy VEHICLE.WEB.SUB.POLICY.
  - a. Specify the persistence of the message to No as shown in Figure 6-8 on page 159. This is an important step in configuring a temporary dynamic queue to use as the subscriber queue.
  - b. Make sure the Wait Interval Read Only property is unchecked.
  - c. Set the Use CorrelId As Id property to true.

**Important:** Since the broker does not publish a persistent message to a temporary dynamic queue, the subscriber sets the send message type to non-persistent in order to also receive the persistent message.

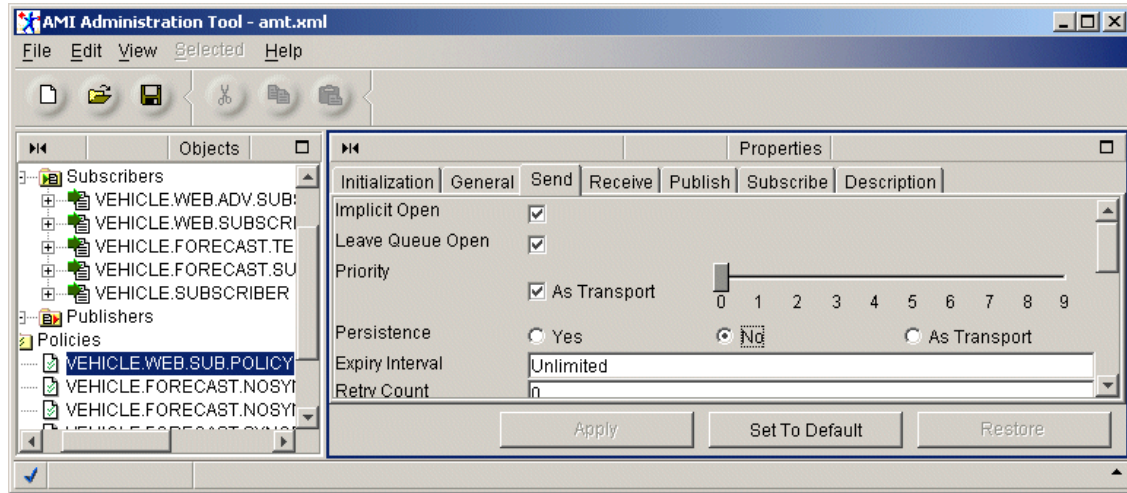


Figure 6-8 Configuring VEHICLE.WEB.SUB.POLICY f

### 6.1.4 Program invocation

Copy the Start.html file into the Web folder of the Web application itso. Then type the following URL in the browser:

`http://hostname/its0/Start.html`

where hostname is the name of the server on which the WebSphere Application Server is running. Refer to the additional materials accompanying this redbook for code snippets.

The output is shown in Figure 6-9 on page 160.

**Sample Subscriber Application**

MessageType	Mode	Geography	Route	Vehicle	Traffic	Stop #	# Stops	TimeBtwnStops	Fill%	TimeStamp
position	Tube	London	Piccadilly	UTrain12	normal	1	10	6	71	21:04:30
position	Tube	London	Bakerloo	UTrain22	light	1	10	4	76	21:04:30
position	Tube	London	Piccadilly	UTrain23	normal	1	10	4	23	21:04:30
position	Tube	London	Piccadilly	UTrain11	heavy	1	10	6	50	21:04:30
position	Trains	London	HogwartsExpress	WizTrain31	light	1	10	5	35	21:04:30
position	Tube	London	Bakerloo	UTrain21	normal	1	10	4	29	21:04:30
position	Tube	London	Piccadilly	UTrain13	normal	1	10	6	47	21:04:30
position	Trains	London	HogwartsExpress	WizTrain32	light	1	10	5	94	21:04:30
position	Trains	London	HogwartsExpress	WizTrain33	normal	1	10	5	78	21:04:30
breakdown	Tube	London	Piccadilly	UTrain13	null	1	null	null	null	21:04:30
accident	Tube	London	Piccadilly	UTrain11	null	1	null	null	null	21:04:30
position	Tube	London	Bakerloo	UTrain22	normal	2	10	4	93	21:04:34
position	Tube	London	Bakerloo	UTrain23	normal	2	10	4	15	21:04:34
position	Tube	London	Bakerloo	UTrain21	light	2	10	4	10	21:04:34
position	Trains	London	HogwartsExpress	WizTrain31	light	2	10	5	24	21:04:35
accident	Trains	London	HogwartsExpress	WizTrain31	null	2	null	null	null	21:04:35
position	Trains	London	HogwartsExpress	WizTrain32	light	2	10	5	63	21:04:35
position	Trains	London	HogwartsExpress	WizTrain33	heavy	2	10	5	99	21:04:35
accident	Trains	London	HogwartsExpress	WizTrain32	null	2	null	null	null	21:04:35
accident	Trains	London	HogwartsExpress	WizTrain33	null	2	null	null	null	21:04:35
position	Tube	London	Piccadilly	UTrain12	light	2	10	6	60	21:04:36
position	Tube	London	Bakerloo	UTrain22	normal	3	10	4	82	21:04:38

Figure 6-9 Output of simple Web-based subscriber application

Click the **Refresh** button to see the latest messages.

### 6.1.5 Discussion about the Web part of the application

This Web-based application is a straightforward migration of the simple stand-alone subscriber as discussed in 4.10, “Sample subscriber application” on page 96, in which the main appeal is to keep the core functionality the same as the existing one.

As in the application discussed in that section, the subscriber application receives messages instantly as they become available on the broker.

The advantage of the stand-alone GUI application is that it subscribes once and then is always in receive mode, but the disadvantage is that it lacks the ability to reach out to the wider audience as a Web-equivalent application would.

Porting the stand-alone application to the Web is a generalized choice, but in this case we have to compromise on certain behavioral aspects of the application, such as not having any automatic update to the message window on the Web.

In a typical Web scenario, the pull model is very much the preferred choice over a push model. In the former model, the end user's request pulls the information from the server, so we can say the pulled information is a snapshot of the information available at the moment of request. In the latter model, the server maintains a dedicated client connection and is responsible for pushing all information to the client.

In our example, we follow the pull model. These are the steps involved:

1. On each request, the user subscribes to the topic PublicTransport/\* .
2. Depending on the availability of messages, the user may receive messages for that subscribed topic at that instant of time.
3. Subsequently the user unsubscribes.

We implement a servlet-based solution in which the steps are demonstrated in Figure 6-10 sequentially.

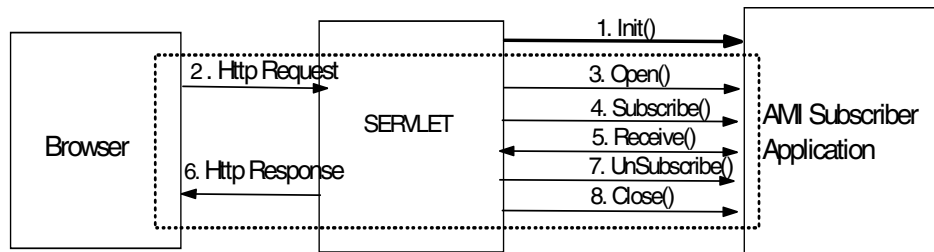


Figure 6-10 Stages involved in a servlet -based Web subscriber application

**Note:** In Figure 6-10, the dotted portion illustrates the steps that are repetitive. The solid arrow 1.init() is out of the dotted portion, thus illustrating the fact that it is a one-time operation.

The following are the stages involved in a Web-based subscriber application:

1. The servlet gets loaded into the application server, retrieves all the init parameter values, and assigns it to a WebSubscriberInit data bean. Subsequently it invokes the init() method of the AMI subscriber application and passes the WebSubscriberInit data bean (shown as a solid arrow and marked 1.Init() in Figure 6-10). The WebSubscriberInit data bean carries the

information relevant to the AMI Subscriber application to create an AMI session.

```
super.init(config);
initData = new WebSubscriberInit();
initData.hostfileName=getInitParameter("hostfilename");
initData.repositoryName=getInitParameter("repositoryname");
initData.sessionName=getInitParameter("sessionname");
initData.policyName=getInitParameter("policyname");
initData.subscriberName=getInitParameter("subscribername");
initData.messageReciver=getInitParameter("messagereceiver");
initData.messageSubscriber=getInitParameter("messagesubscriber");
initData.waitTime=getInitParameter("waittime");
initData.topicName=getInitParameter("topicname");
sub=new com.ibm.itso.swa111.subscribe.SubscriberAMI();
sub.init(initData);
```

Refer to 4.10, “Sample subscriber application” on page 96 for details on the operation of an AMI subscriber application.

2. The Web user request for subscription in the form of an HTTP GET request. Inside the servlet the GET request will be delegated to the performTask() method.

```
public void doGet(HttpServletRequest request,HttpServletResponse response)
throws javax.servlet.ServletException, java.io.IOException {
performTask(request, response);
}
```

3. In the performTask() method, an AMI session is created by invoking the open() method of the AMI subscriber application.

```
sub.open();
```

4. Then subscribe to the topic by invoking the subscribe() method of the AMI subscriber application:

```
sub.subscribe(initData.getTopicName());
```

5. The servlet invokes the receive() method of the application bean and receives messages. The receive() method will be invoked continuously until there are no messages left in the subscriber queue. The receive() method returns the result in the form of a SubscriberData data bean. The servlet returns the HTML output built with the received message:

```
sd=new com.ibm.itso.swa111.subscribe.SubscriberData();
sd=sub.receive();
```

6. Soon after the previous step, the servlet unsubscribes the requested user by invoking the unsubscribe() method of the AMI subscriber application:

```
sub.unsubscribe(initData.getTopicName());
```



7. Then the session is closed by invoking the `close()` method of the AMI subscriber application:

```
sub.close();
```

## 6.2 Comments and extensions

This second application scenario demonstrates how easily information can be delivered to Web users via publish/subscribe messaging. The key feature needed to easily obtain good results is again *retained publications*.

The reason for this is that the Web application subscription lifetime is negligible, so the information displayed will be the one already retained by the broker at the moment the subscription request is received.

This second application scenario still suffers from several limitations:

- ▶ Unfortunately this approach is not a good fit for event information such as the Alert and Breakdown messages. Typically these messages will not be shown on the Web interface because the probability is small that a subscription for them is in place when they reach the broker.
- ▶ The other limitation is that the publishing side cannot be implemented using JMS (which does not support retained publications) when the subscribing side is Web-based.
- ▶ Given that JMS is part of the J2EE specification, the pattern where both the producer and the consumer of messages is a JMS application is expected to be a typical one.
- ▶ There is not an immediate way to implement the Web application in order to deliver such value-added services as computation of vehicle position forecasts.

All these issues will be addressed by extending the current version of the application in Chapter 7, “Advanced Web enablement” on page 165.

The Web application subscriber we presented in this chapter is not backward compatible with MQSeries Publish/Subscribe brokers, because we used a temporary dynamic queue as subscription queue. This feature is only supported by MQSeries Integrator brokers.

If for some reason the application must also support MQSeries Publish/Subscribe brokers (ignoring the benefits coming from MQSeries Integrator listed in Chapter 5, “Migration to MQSeries Integrator” on page 115), all that’s needed is a modification to the AMI repository definition for the receiver specified for the `VEHICLE.WEB.SUBSCRIBER` subscriber object.





## Advanced Web enablement

In the first part of this chapter we will extend the example application discussed in Chapter 6, “Web enablement” on page 149, in order to satisfy these new requirements:

- ▶ Support for JMS publishers
- ▶ Support for JMS subscribers
- ▶ Publication of comprehensive position forecast data for each vehicle, taking into consideration current traffic conditions, accidents and breakdowns

In the second part of this chapter we will extend the Web part of the application further, in order to support such complex queries as:

- ▶ Listing of all vehicle that are expected to reach a certain stop within a specified range of time
- ▶ Listing of all vehicles that just broke down or are significantly delayed by accidents
- ▶ Any combination of the above

## 7.1 Concept

The most demanding requirement to accommodate is the support for JMS publishing applications, because as we discussed in the previous chapter, Web subscriptions are extremely short lived, so the use of retained publication is almost unavoidable. On the other hand, JMS cannot publish in a retained fashion.

The second difficult requirement to accommodate is the publication of forecasts. The problem here is that to compute new forecasts (for example, predicting the effect of an accident on a route), access to the most current forecast is sometimes needed, that is, forecasts must be retained.

If forecasts are retained, the existing publisher is not a good candidate to publish them (it can be a JMS application), nor is the subscriber (it can be a JMS application, and being Web-based it must be as lightweight as possible).

The correct approach to satisfy the new requirements respecting all the above constraints seems to be the implementation of a new application, the *Forecast* application, which resides between the existing publisher application and the Web subscribers.

### 7.1.1 A new middle tier component

The Forecast application is implemented as a permanently subscribed application receiving position and alert messages published by the vehicles. Given that this application is always subscribed, the information can be published in a non-retained fashion without the risk of the broker discarding it for the lack of matching subscribers. Thus, JMS publishers can be fully supported.

All the positions and alerts are used by the Forecast application to produce a message for each vehicle containing the expected time of arrival (ETA) of the vehicle for all the remaining stops along the route. This message is published as retained (newer forecasts overwrite older ones) on the new subtopic root `PublicTransport/Forecasts`, including the name of the vehicle, the route, the geography, and the mode in the full topic name.

The publication of retained forecasts not only serves the unique needs of short-lived subscriptions coming from the Web, but also plays the role of a context holder for the Forecast application itself. In fact when an old forecast is to be amended by a new one, to accurately compute the new one the Forecast application sometimes needs to subscribe temporarily to some of the retained topics it previously published on (that is, the current forecast for the vehicle for which the new forecast is to be computed).

This technique uses the broker almost as a database, making it possible to write the Forecast application in a simple way, because:

- ▶ The application is stateless (the state is persistently maintained by the broker).
- ▶ The application only deals with one resource manager (the broker) and not with a relational database as well; this usually gives performance gains and avoids extra complexities such as dealing with more complex transactional semantics (for example, global units of work).

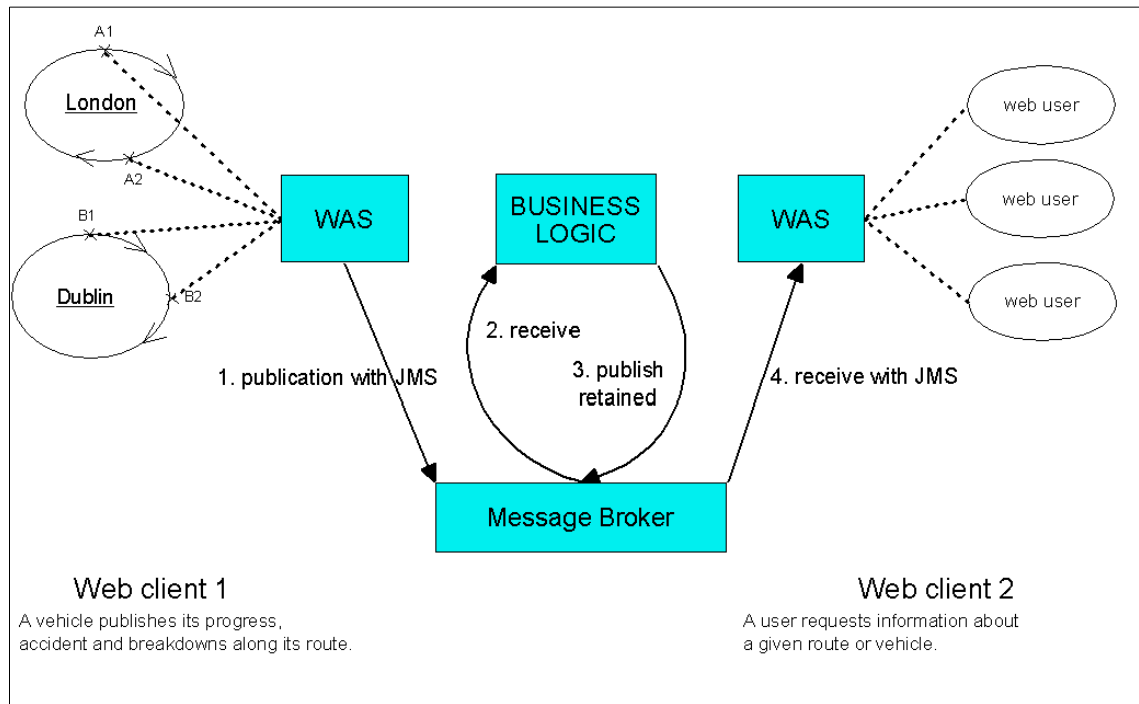


Figure 7-1 Use of JMS in the application server

## 7.1.2 Architectural considerations

With the above discussed new approach we are able to use the J2EE compliant JMS API on both extremities of the messaging flow. In Figure 7-1, we see a possible real-life implementation where each vehicle could send its position through mobile devices to WebSphere Everyplace Suite and WebSphere Application Server publishes this information with the JMS API.

A back-end component applies the business rules on the published information and publishes the newly computed data as retained information with AMI or MQI.

Web users use their browser to query the current and expected positions of their vehicle and when the next vehicle is expected to arrive at their stop. The application server then interrogates the message broker with JMS to get the required information.

The application we produced is very similar to the one described, apart from the publication part where the vehicles and their interaction with a first application server is simulated by the publisher application that publishes directly the positions and alerts messages on the target broker.

Figure 7-2 gives a general view of the main components making up the application described in this chapter.

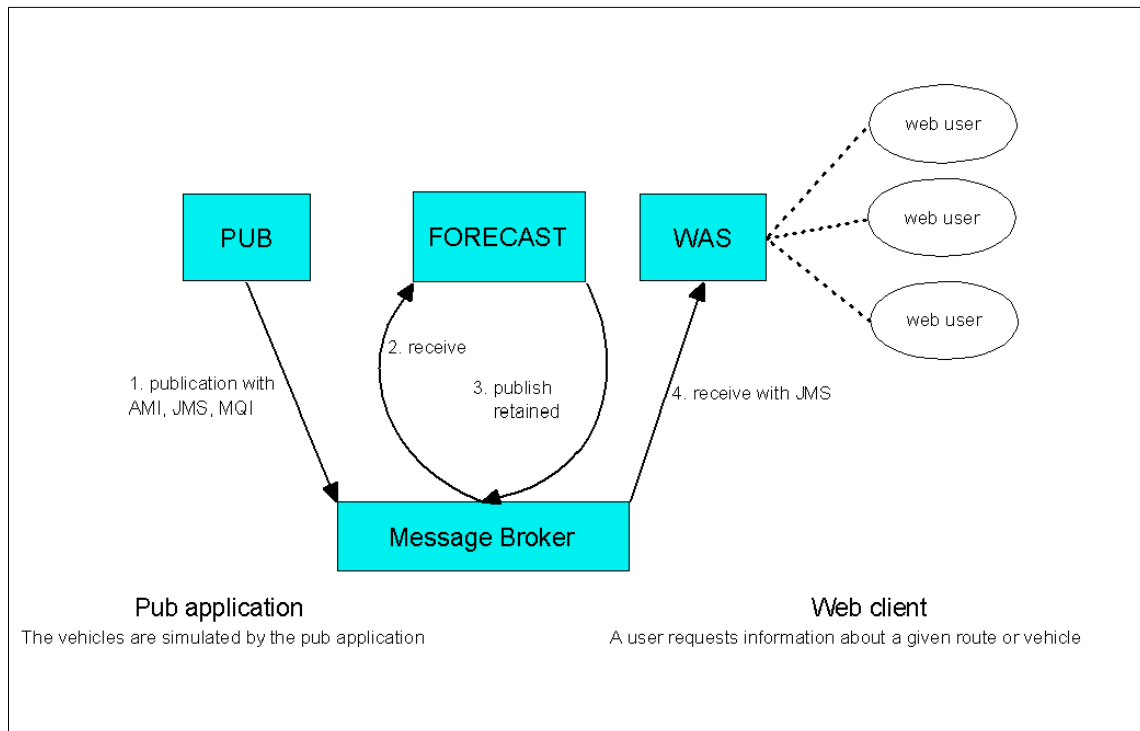


Figure 7-2 Components used in the global application

Since the Forecast application needs to publish retained messages, we decided to use the AMI API for the messaging part of the application.

7.2, "Forecast application" on page 169 describes in more detail the Forecast application.

In 7.3, “JMS Web subscriber application” on page 174 and 7.5, “AMI Web application and message filtering” on page 182, we discuss how the newly supported features impact the subscription side of the application.

## 7.2 Forecast application

The Forecast application is a stand-alone Java application. It is designed as a multithreaded application, but given that the multithreading makes the application slightly more complex because of the need to obtain and release locks and get acknowledgments from the broker, we first discuss the Forecast application when there is only a single thread receiving the messages published by the vehicles, computing the forecast for this vehicle, and publishing the forecast messages.

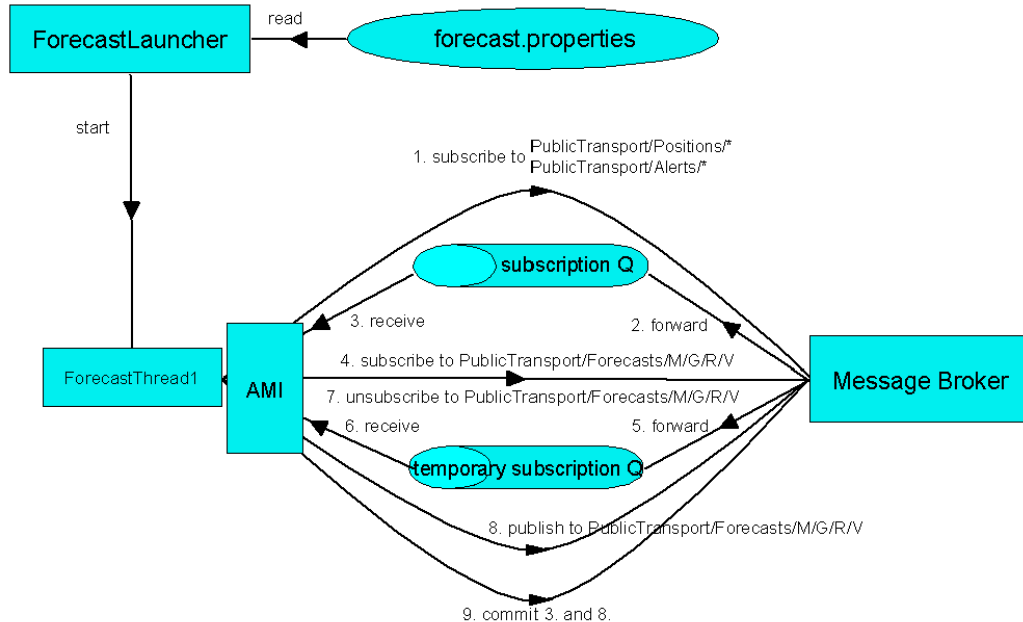


Figure 7-3 Forecast application logic with a single thread

Figure 7-3 shows the application logic and the message flow of the Forecast application when a single threaded application is running.

The application is started by the Java class `ForecastLauncher`. It reads the AMI information in the `forecast.properties` file and also the number of threads needed in the application. We decided to provide the possibility of a multithreaded application so that the application can scale with the load.

The `ForecastLauncher` starts the desired number of threads, called `ForecastThread`, for doing the work.

### 7.2.1 ForecastThread: single-threaded behavior

We describe each step indicated in Figure 7-3 on page 169.

1. The thread subscribes to all the publications sent by the vehicles, both positions and alerts. We subscribe to the topics `PublicTransport/Positions/*` and `PublicTransport/Alerts/*`. We don't want to subscribe to `PublicTransport/*` as a third topic branch called `PublicTransport/Forecasts`. Note that in the multithreaded implementation, only the first thread subscribes. We subscribe every time the application is started and we don't unsubscribe, so that the Forecast applications can handle all the messages even if it is not running.
2. The message broker forwards the vehicle publications to our subscription queue. This subscription queue (`VEHICLE.FORECAST.SUB.QUEUE`) is defined in the AMI service point `VEHICLE.FORECAST.RECEIVER` created in the AMI repository.
3. The thread gets a message from the subscription queue using the AMI policy `VEHICLE.FORECAST.SYNCPOINT_EXPIRY.POLICY`. This policy specifies that the message is to be taken under syncpoint and that the wait time for getting the message is five seconds.

We must now make a distinction based on the type of messages received:

- ▶ If a position message is received, the Forecast application can forecast the future positions of the vehicle (the total number of stops and the time spent between stops is included in the message).
- ▶ If a breakdown message is received, the application has enough data to simply publish a last message containing the time and position where the breakdown happened.
- ▶ If an accident message is received, then the application can't create a forecast message based on this accident message only. It needs to know the current forecasts for this vehicle and relay them in accordance with the accident information.

For a position and a breakdown message, we skip the following steps 4 to 7 and go directly to step 8, whereas when an accident message is received, we must go through all steps.



4. The thread temporarily subscribes to the forecast topic of that vehicle `PublicTransport/Forecasts/Mode/Geography/Route/Vehicle`. The subscriber used is `VEHICLE.FORECAST.TEMP.SUBSCRIBER`, corresponding to a service point `VEHICLE.FORECAST.DYN.RECEIVER` defined as a temporary dynamic queue starting with a prefix `VEHICLE.FORECAST` and built on the model queue `SYSTEM.DEFAULT.MODEL.QUEUE`.
5. The temporary queue is created and the broker forwards the forecast message to it.
6. The thread gets the forecast message from the temporary queue.
7. The thread unsubscribes from that topic. The temporary queue gets deleted.
8. The thread now has enough data to compute the forecast for the vehicle. It creates the forecast message and publishes it to the forecast topic for that vehicle (`/PublicTransport/Forecasts/Mode/Geography/Route/Vehicle`). The AMI policy used for the publication is the same policy as the one used for receiving the message at step 3: `VEHICLE.FORECAST.SYNCPOINT_EXPIRY.POLICY`, which means that the publication also is under syncpoint.
9. We commit the transaction started at step 3, that is the reception of the position or alert message and the publication of the forecast message.

**Important:** The Forecast application can potentially receive messages with a `MQRFH1` header, a `MQRFH2` header or both. Therefore when parsing the message, we must first check the type of header.

The `MQRFH1` header is removed automatically so that we only have the payload data.

If a `MQRFH2` header is present (that is, the message publisher was a JMS application), we must remove it by reading at the right offset the length of the `MQRFH2` header and skipping it.

## 7.2.2 ForecastThread: multithreading behavior

We can change the `nbrThreads` property in the property file of the application to a value greater than 1, in order to make it run in multithreaded mode. The multithreaded behavior is roughly the same as if there were just one thread.

But when more than one thread is running, we must ensure that two different threads are not handling two messages for the same vehicle at the same time, for example, a position message and an accident or breakdown message.

Therefore, we must add a few steps in the logic described in the previous section. Figure 7-4 shows the logic flow of the Forecast application when in multithreading operation mode.

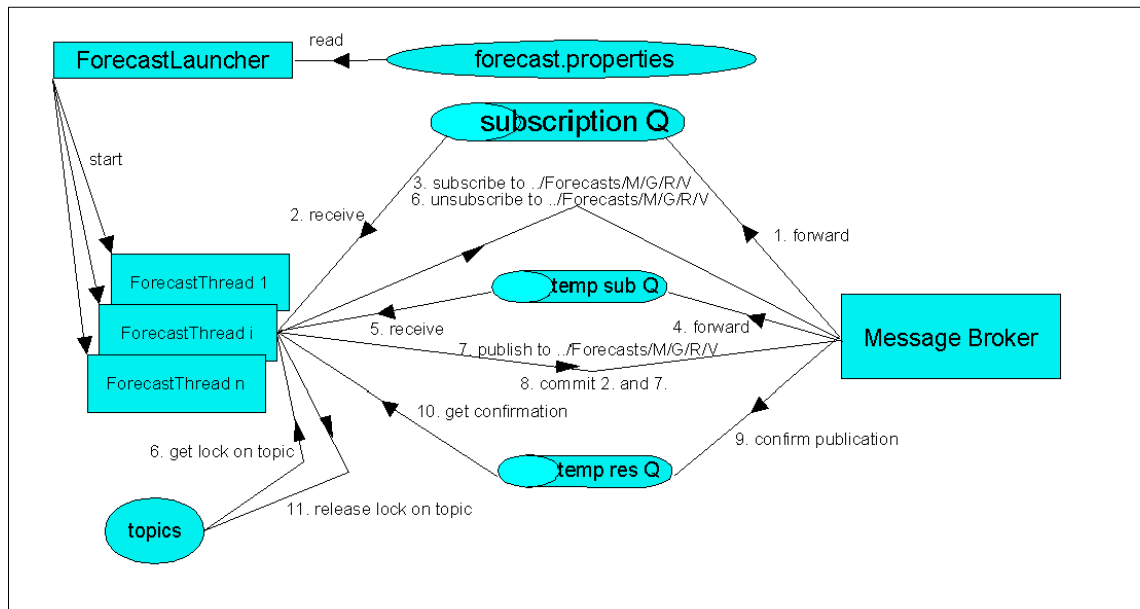


Figure 7-4 Forecast application logic with multiple threads

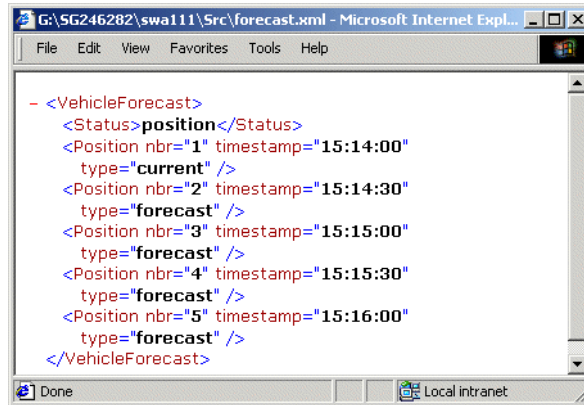
Only the first thread subscribes to the position and alert topics, because all threads are consuming from the same subscription queue and are considered as one single subscriber. We certainly don't want the messages to be duplicated for each thread.

1. Similar to single-threaded behavior, the broker forwards the publications published by the publisher application to the subscription queue used by the Forecast application.
2. All threads are consuming in parallel the messages from the subscription queue.
3. Similar to single-threaded behavior, when the publication is an accident, the thread subscribes to the forecast topic for receiving the current forecast for the vehicle that had an accident.
4. Similar to single-threaded behavior, the broker forwards the forecast message to a temporary dynamic queue. Each thread has its own temporary dynamic queue.
5. Similar to single-threaded behavior, the thread consumes the forecast message from the temporary queue and the queue is closed and deleted.

6. Now to an additional step: the thread gets a lock on the topic to which the forecast message for the vehicle will be published. For this lock, we use synchronized access to a singleton class shared between the threads. In this class, we check out and check in the topic string to which we publish in order to get and release the lock.
7. The thread publishes the forecast message. In contrast with the single-threaded model, we specify a receiver service point in the overloaded publish Java method of the AmPublisher class to ask a confirmation from the broker that the publication message has been handled. We will need this broker response in order to be able to release the lock on the topic. The receiver defined in the AMI repository is VEHICLE.FORECAST.DYN\_RES.RECEIVER. It indicates that a temporary dynamic queue is to be used, built on the default model queue SYSTEM.DEFAULT.MODEL.QUEUE and whose name must be prefixed with VEHICLE.FORECAST.RESPONSE.
8. Similar to single-threaded behavior, the reception of the position or alert publication and the publication of the forecast are committed.
9. Multithreaded applications now have three additional steps. After committing the publication, the broker consumes the publish command on the publication queue and sends the confirmation to the temporary dynamic queue. Each thread uses its own temporary dynamic queue, the same for a thread lifetime.
10. The second additional step: the response sent by the broker is received by the thread.
11. The third additional step: after receiving the broker confirmation, the thread releases the lock he had acquired on the topic. Note that when running in multithreaded mode, the Forecast application requires the broker to be running, since it needs to receive responses from it.

### 7.2.3 The forecast message

The forecast publication message used by the application needs to contain the stop number and the estimated time of arrival, for each stop ahead on the route of the vehicle. Figure 7-5 on page 174 shows an example of such a forecast publication message.



```
- <VehicleForecast>
  <Status>position</Status>
  <Position nbr="1" timestamp="15:14:00"
    type="current" />
  <Position nbr="2" timestamp="15:14:30"
    type="forecast" />
  <Position nbr="3" timestamp="15:15:00"
    type="forecast" />
  <Position nbr="4" timestamp="15:15:30"
    type="forecast" />
  <Position nbr="5" timestamp="15:16:00"
    type="forecast" />
</VehicleForecast>
```

Figure 7-5 forecast XML message

Since each vehicle on each route has its own topic, we decided not to duplicate this information by adding it in the message itself. We'll see in 7.4.1, "Using MQSeries Integrator to tweak publication content" on page 180 that this design choice has some important impact.

## 7.3 JMS Web subscriber application

The Web subscriber application is an extension of the one we discussed in Chapter 6, "Web enablement" on page 149 and is an ideal J2EE implementation.

In this case the Web user will get the opportunity to subscribe to a particular topic string as well as to all topics, rather than having the only option of subscribing to the root topic, as discussed in Chapter 4, "The publish/subscribe application" on page 27 and Chapter 6, "Web enablement" on page 149.

We will adopt the same servlet-based solution as discussed in Chapter 6, "Web enablement" on page 149. In this case the servlet will also cater for the browser request, contact the subscriber application bean, and pass the request. In turn, the bean will process and return the result, and the servlet pipes back the output to the browser as a response. In this case the subscriber application bean is a pure JMS implementation.

In the following sections we discuss:

- ▶ Servlet configuration
- ▶ Program invocation
- ▶ Program flow
- ▶ Using MQSeries Integrator to tweak publication content

### 7.3.1 Servlet configuration

Refer to 6.1.2, “Servlet configuration” on page 155 for help with servlet configuration. The init parameters that are required are:

1. `jmscontext` - The value of Initial context factory `com.ibm.ejs.ns.jndi.CNInitialContextFactory`. This value is valid for the persistent Name Server provided by WebSphere Application Server.
2. `jmsurl` - `iiop://hostname/`, where `hostname` is the name of the machine where the persistent service is running.
3. `jmsrefferal` - In our example, this parameter is not relevant. If we use LDAP services, then we need this parameter.
4. `jmstcf` - `TopicConnectionFactory` name.
5. `jmswaittime` - A period of time (in milliseconds) that the JMS subscriber waits for a message to be available.

Refer to 7.5.3, “Servlet configuration” on page 184 for init parameter values.

### 7.3.2 Program invocation

Copy the sample `PTInput.html` file into the `Web` folder of the Web application `itso` and type the following URL in the browser:

```
http://hostname/itso/PTInput.html
```

You will see a window like the one shown in Figure 7-6 on page 176.

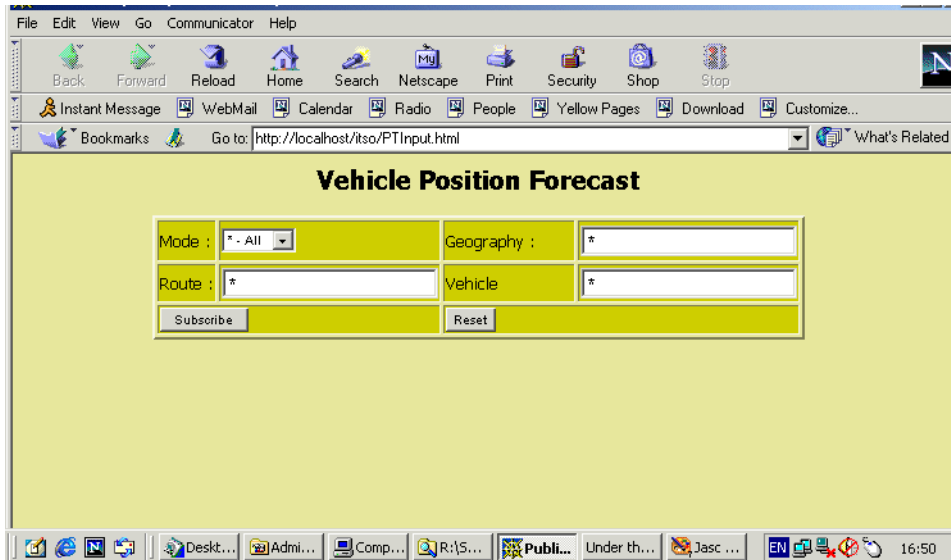


Figure 7-6 Forecast Input window

This input Web page prompts the user to subscribe to a specific topic string. By default all the options are marked as “\*” (asterisk), which means the topic string is the same as `PublicTransport/Forecasts/*`. Otherwise, the topic string will be formed as per the following entries:

The hierarchy of fields in the topic string is

`PublicTransport/Forecasts/Mode/Geography/Route/Vehicle`

If `Mode=Tube`, `Geography=London`, `Route=RouteA`, `Vehicle=UT221`, then the topic string would be `PublicTransport/Forecasts/Tube/London/RouteA/UT221`

If `Mode=*` and the rest of the fields have some value other than `*`, then the topic string would be `PublicTransport/Forecasts/*`.

We click the **Subscribe** button to subscribe to that topic and it receives a message (if there is any retained message published to the broker for that specific topic) as shown in Figure 7-7 on page 177.

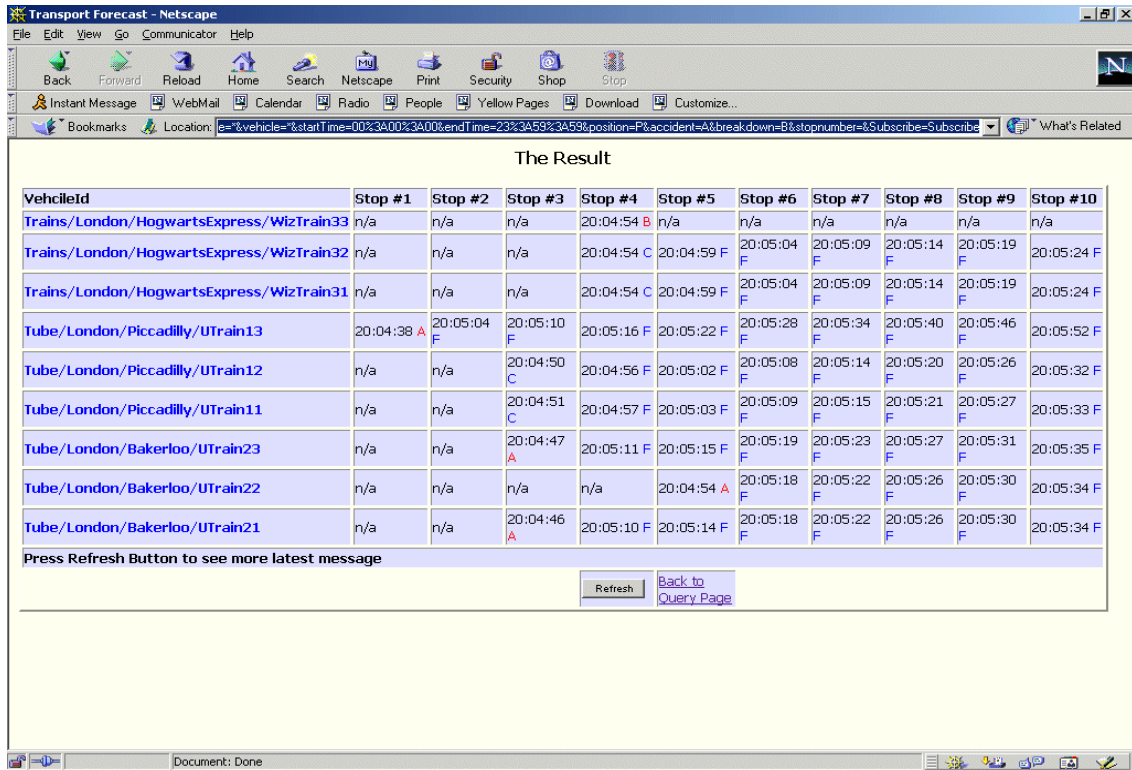


Figure 7-7 Topic-based Forecast message subscription

The output message displays each vehicle's expected time of arrival (marked as 'F') at that stop number as well as the current position (marked as 'C'). It also displays the accident and breakdown messages (marked as 'A' and 'B' respectively). The vehicle ID is in the format Mode/Geography/Route/Vehicle, which can be extracted from the topic (which is in the format PublicTransport/Forecasts/Tube/London/Route) to which the user subscribed. Please see 7.4.1, "Using MQSeries Integrator to tweak publication content" on page 180 for a detailed discussion of the extraction of the vehicle ID from the topic.

We can either re-subscribe with the old topic string by clicking the **Refresh** button, or we can go back to the previous window shown in Figure 7-6 on page 176 to subscribe to a new fresh topic.

## 7.4 Program flow of the application

In this section we discuss the program flow of our subscriber application.

The forecast messages that are published to the broker are in retained form. Hence the subscriber application always gets a message for the topic to which it subscribed, provided there was a publication on that topic before.

The approach to this servlet-based Web solution is similar to the one we have adopted in Chapter 6, “Web enablement” on page 149. Let's discuss the program flow.

1. The servlet gets loaded into the application server, and the `init()` method gets executed.

- a. It retrieves the init parameter values and assigns it to a data bean `WebSubscriberInit`.

```
super.init(config);
initData = new com.ibm.itso.swall11.subscribe.web.WebSubscriberInit();
initData.jmsContext=getInitParameter("jmscontext");
initData.jmsUrl=getInitParameter("jmsurl");
initData.jmsReferral=getInitParameter("jmsreferral");
initData.jmsTcf=getInitParameter("jmstcf");
```

- b. Subsequently it invokes the `init()` method of the JMS subscriber application and passes the data bean, which carries the information relevant to the JMS subscriber application to initialize JMS environment.

```
sub=new com.ibm.itso.swall11.subscribe.SubscriberJMS();
sub.forecastInit(initData);
```

- c. The servlet uses the MQ connection pool available underlying the MQ transport layer. Hence we can leverage the power of the MQ connection pooling by having multiple sessions running together.

```
token = MQEnvironment.addConnectionPoolToken();
```

- d. The `forecastInit()` method of the JMS application bean sets up the JMS environment.

```
CTX_FACTORY = subinit.getJmsContext();
INIT_URL = subinit.getJmsUrl();
env = new Hashtable();
env.put( Context.INITIAL_CONTEXT_FACTORY, CTX_FACTORY );
env.put( Context.PROVIDER_URL, INIT_URL );
env.put( Context.REFERRAL, subinit.getJmsReferral() );
Context ctx = new InitialDirContext( env );
tcf = (TopicConnectionFactory)ctx.lookup(subinit.getJmsTcf());
```



2. When a user subscribes, the servlet performs the following steps:
  - a. Invokes the `forecastOpen()` method of the subscriber application bean. Inside the open method, we create a topic connection and then a session. Then we start the connection:

```
tConn = tcf.createTopicConnection();
tSess = tConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE );
tConn.start();
```

- b. Next the servlet invokes the `forecastSubscribe()` method of subscriber application bean. Inside this subscribe method, we create the topic and create a subscriber:

```
t = tSess.createTopic(topicName);
tSub = tSess.createSubscriber(t);
```

- c. Next the `getForecastMessage()` of the subscriber application bean is invoked, which returns the message to the servlet:

```
msg = (TextMessage)tSub.receive(MESSAGE_WAIT_TIME);
```

**Important:** The `MESSAGE_WAIT_TIME` parameter is the time period for which the subscriber application waits for the arrival of the next message. In a Web scenario where response time is a key factor, the `MESSAGE_WAIT_TIME` value should be kept to a minimum, keeping in mind the end users' expectations of response time.

The above step continues until there are no more messages in the subscriber queue. The servlet returns the HTML output built with the received message as shown in Figure 7-7 on page 177.

3. After the message is displayed we unsubscribe by invoking `sub.forecastUnsubscribe(topicname)`, which does the following operation in the subscriber application bean:

```
tsub.close()
```

4. Finally we close the session and the connection by invoking `sub.forecastClose()`. We close the session and the topic connection:

```
tSess.close()
tConn.close()
```

**Important:** We always close the topic connection at the end of the session and request a fresh connection in each new request. Creating a topic connection for each request is expensive in a normal scenario when there is no connection pooling, whereas in our example we are using `MQConnectionPool`, which is readily available in the MQSeries transport layer.

**Note:** We must remove the MQ connection pool token from the environment when it is no longer being used. In the `destroy()` method of the servlet we can add the following code:

```
MQEnvironment.removeConnectionPoolToken(tokenname);
```

### 7.4.1 Using MQSeries Integrator to tweak publication content

The JMS Web subscriber needs to parse the topic name in order to extract the Mode, Geography, Route and Vehicle fields. When the messages are published by a JMS application, the topic name is stored in a field of the JMS embedded MQRFH2 header. Unfortunately in our case the messages are published by the Forecast application that is written to the AMI API (because it publishes in retained mode), so this information is not available to the JMS subscriber.

**Note:** The subscription topic pattern is honored, so only the right messages are received by the JMS subscriber, but the matched topic cannot be programmatically accessed.

A simple way around this problem is to add a new tag containing the topic inside the body of the message, that is, where even a JMS application can have access.

We decided not to modify the existing Forecast application (in a production environment this could be a legacy application that *could not* be modified), and have MQSeries Integrator do the work.

To achieve this, we built a very simple message flow that is an extension of the default publish/subscribe message flow (see Figure 7-8 on page 181) including a compute node that adds a tag name `VehicleId` to the body of the message, taking it from the MQRFH header `Topic` field.

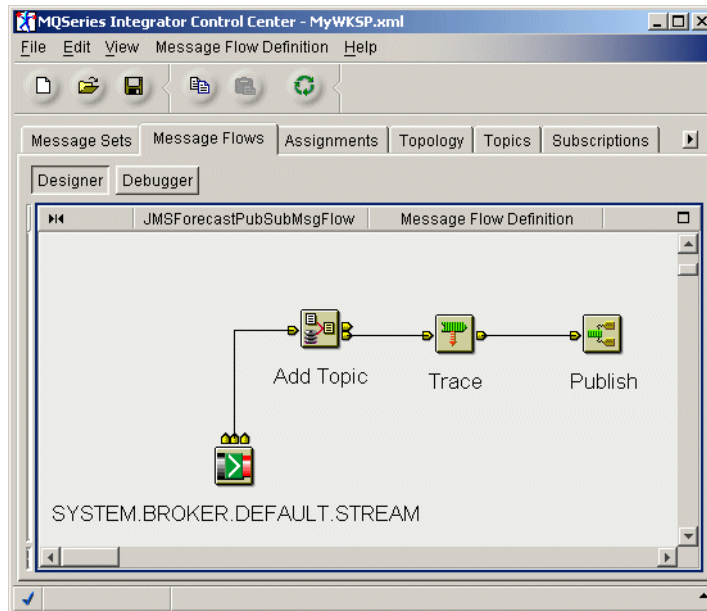


Figure 7-8 MQSeries Integrator publish/subscribe message flow with message augmentation

This message flow is included in the redbook additional material in the export file named JMSForecastPubSubMsgFlow.xml.

**Note:** Before trying to run the JMS subscriber, remember to deploy the message flow JMSForecastPubSubMsgFlow.xml to your broker.

If you already have a deployed publish/subscribe message flow serving the queue SYSTEM.BROKER.DEFAULT.STREAM, remember to remove it before deploying the new one.

## 7.4.2 Comments and extensions

This Web application scenario demonstrates publish/subscribe messaging in a J2EE environment. Because of retained publication, the JMS-based subscriber application always receives the messages.

This application scenario still suffers from the following limitations:

- ▶ In JMS Web application, the filter specified with a topic pattern is not specific enough to get only the information that is needed for a particular end user purpose. The content-based filtering is not supported.

- ▶ When the JMS subscriber (which receives retained messages) subscribes to a given topic during the course of publication of message on the same topic, the broker publishes all the retained messages to the subscriber queue. As a result, the subscriber gets more than one message for the same topic, which is legal but not desirable.

Suppose the publisher is publishing 10 messages for a given topic, At the time of fifth message publication, the subscriber subscribes, hence it receives the last retained message, that is, the fifth message. It then starts receiving the rest of the messages during the course of publication (message 6 to message 10). In our example, the subscriber would receive all forecasted information about the same vehicle, which is unclear to the user.

Unfortunately, in JMS there is no easy way to filter the unwanted messages other than applying a custom filter condition while receiving the message. In our example we implemented the following logic at the JMS subscriber application end.

```
if (timestamp of received message) <= (timestamp of subscription)
```

Then we allow the messages. Failing this condition, we stopped receiving messages.

This issue will be addressed by extending the current version of the application in 7.5, “AMI Web application and message filtering” on page 182.

## 7.5 AMI Web application and message filtering

In the previous section we discussed JMS-based publisher and Web subscriber applications in which the subscriber application can subscribe to a specific topic rather than subscribe to the root topic. In this section, we will extend the Web part of the application further to content-based subscription. The content-based subscription will support complex queries like:

1. Listing of all vehicle that are expected to reach a certain stop within a specified range of time
2. Listing of all vehicles that just broke down or are significantly delayed by an accident
3. Any combination of the above

## 7.5.1 Content-based subscriptions

The main issue in the JMS Web application is that often the filter specified with a topic pattern is not narrow enough to get only the information that is needed for a particular end-user purpose. For example, there is no way using only the topic filters to receive only the information related to a particular stop on a particular route.

To overcome this limitation, we can modify the subscriber application in order to exploit the content-based subscription feature that comes with MQSeries Integrator. Unfortunately the JMS specification does not support this feature, so we will rewrite the relevant application modules using AMI.

**Attention:** Don't confuse JMS built-in message filtering, which is performed by the JMS API on the application side, with MQSeries Integrator content-based subscriptions, which are evaluated on the broker side (that is, before sending the messages to the target applications), thus optimizing the usage of network resources.

The general idea of content-based subscriptions is that of giving the application the possibility of passing a complex boolean expression when submitting the subscription request. The expression will reference syntactical elements of the message.

In MQSeries Integrator the language used to specify filters is the ESQL, which is an extension of the SQL language used for relational databases.

The main thing to remember when using content-based subscriptions is that the broker must be able to correctly parse published messages in order to evaluate filter expressions against them.

In our example application, this is not a problem because we only deal with XML messages that are natively supported by MQSeries Integrator, but in general you will need to provide the relevant MQSeries Integrator Message Repository definitions before using this feature.

## 7.5.2 Content-based Web subscriber application

We use AMI to build the content-based Web subscriber application. The approach for this servlet-based solution is similar to simple Web subscriber application as discussed in Chapter 6, "Web enablement" on page 149 and in 7.3, "JMS Web subscriber application" on page 174. The presentation logic is the same as in the previous example. The only extended feature is content-based subscription.

In this application, the user is prompted for input in two levels. The first level for topic selection was discussed in examples in 7.4, “Program flow of the application” on page 178, and the second level is for query setting. The user can specify start time, end time, and stop number, which can have a combination of message types within the available choices (Position/Accident/Breakdown).

In the following sections we discuss:

- ▶ Servlet configuration
- ▶ AMI repository configuration
- ▶ Program invocation
- ▶ Discussion about the application

### 7.5.3 Servlet configuration

Refer to 6.1.2, “Servlet configuration” on page 155 for servlet configuration information. Create a servlet entry `ForecastAdvServlet` in the `itso` Web application. Configure the `init` parameter of the servlets as described in Table 7-1.

Table 7-1 *ForecastAdvServlet Init Param list*

Init Param Name	Init Param Value
hostfilename	amthost.xml
repositoryname	amt.xml
sessionname	VEHICLE.WEB.ADV.SESSION
policyname	VEHICLE.SUB.POLICY
subscribername	VEHICLE.WEB.ADV.SUBSCRIBER
messagereceiver	VEHICLE.WEB.ADV.RECEIVER.MSG
messagesubscriber	VEHICLE.WEB.ADV.SUBSCRIBER.MSG
waittime	2000
control_queue	BROKER.2.CONTROL.QUEUE
sub_receiver	VEHICLE.2.RECEIVER

The param list contains the same entries discussed in Chapter 6, “Web enablement” on page 149 and two extra entries, `control_queue` and `sub_receiver`, which hold the sender service and receiver service names of the subscriber respectively.

## 7.5.4 AMI repository configuration

The content-based subscription is supported only in MQSeries Integrator broker and the message is in the MQSI V2 format or in the RFH2 format. So we define a new service point as `BROKER.2.CONTROL.QUEUE` for the `SYSTEM.BROKER.CONTROL.QUEUE`, in which Service Type is either MQSeries Integrator V2 or RF Header V2. Then we create a new subscriber point entry `VEHICLE.WEB.ADV.SUBSCRIBER` in the repository and assign `BROKER.2.CONTROL.QUEUE` as the sender service and `VEHICLE.2.RECEIVER` as the receiver service. There is no need to define a new policy for this application. We use the old policy `VEHICLE.SUB.POLICY`, which was used for the simple Web subscriber application discussed in Chapter 6, “Web enablement” on page 149.

## 7.5.5 Program invocation

We copy the `ForeceastAdv.html` file into the Web folder of the Web application `itso` and type the following URL in the browser:

```
http://hostname/itso/ForecastAdv.html
```

where `hostname` is the name of the machine on which the WebSphere Application Server is running. We get a window like the one shown in Figure 7-9 on page 186.

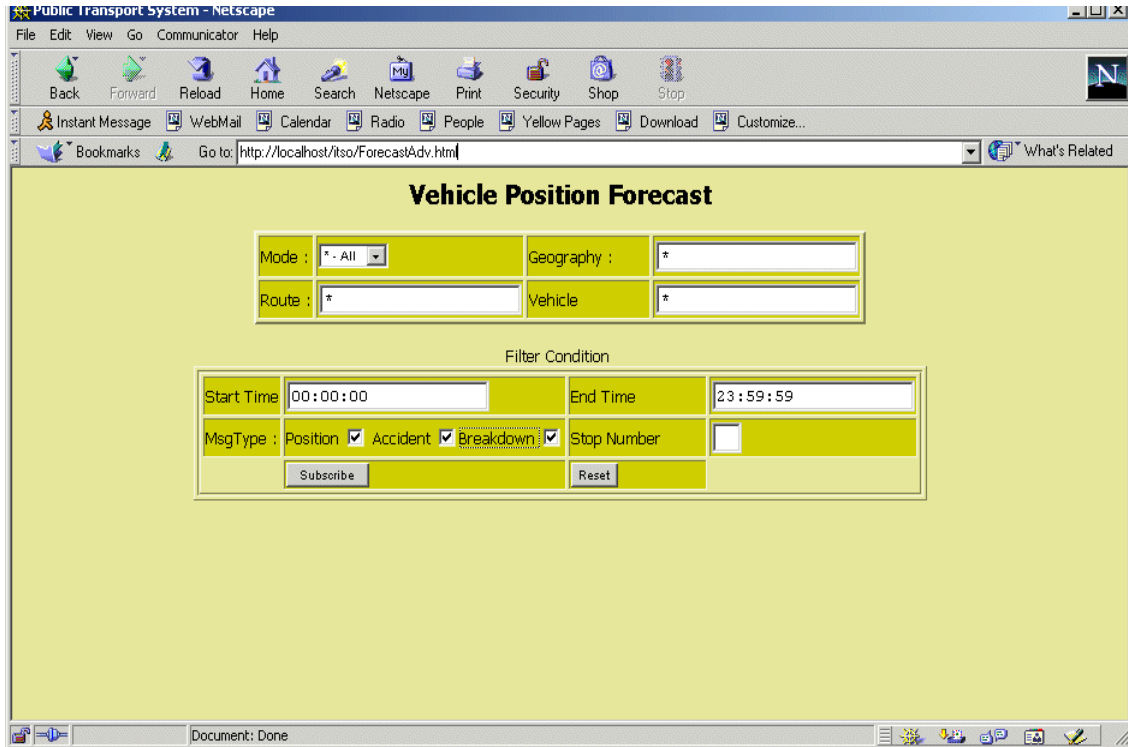


Figure 7-9 Content-based forecast message input window

The default value for Start Time is '00:00:00' and for End Time is '23:59:00'. This is the maximum time limit that can be specified. By default the all the message types are checked and the stop number is blank. We accept the defaults and click the **Subscribe** button to see the output shown in Figure 7-10 on page 187.



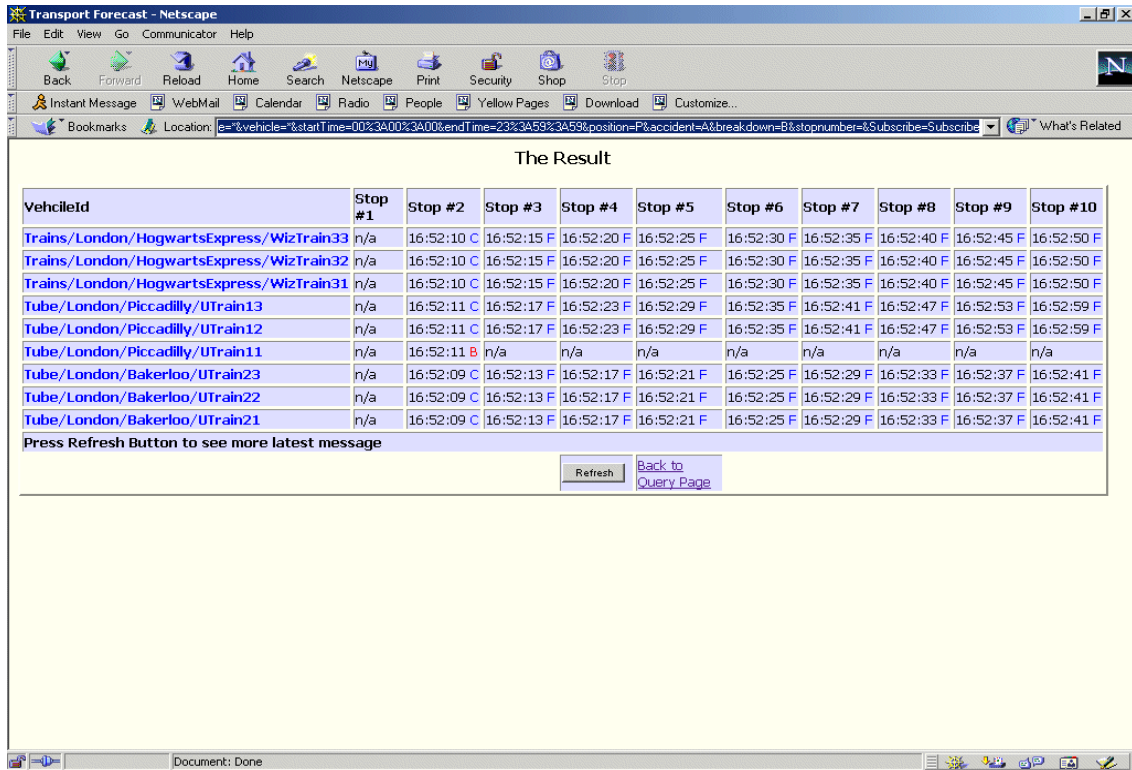


Figure 7-10 Content-based forecast message subscription

The presentation logic is the same as discussed in the previous JMS-based Web subscriber example.

We can either re-subscribe with the old topic and filter condition by clicking the **Refresh** button, or we can go back to the previous Input window shown in Figure 7-9 on page 186 to subscribe to a new topic and filter condition.

**Note:** For content-based subscriptions, the publication message has to be in the RFH2 message format. Please refer to “The Forecast application” on page 216 for more details.

## 7.5.6 Discussion of the application

This application has the same program flow as discussed in previous JMS-based subscriber applications. The following are the important steps involved in this application:

1. Applying a filter condition for content-based subscription.

Since the evaluation of filters by the broker slows down the time spent by the broker on each message (for example, depending on the filter complexity it may double it), we ignore content-based subscription for the following conditions and subscribe as per the topic selection:

- If the default values are selected for StartTime, EndTime, MessageType
- If Start Time and End Time is blank
- If all the Message Type are checked

Otherwise we construct the query string and add a filter to the subscribe message. The query string contains the equivalent ESQL statement, which is similar to SQL.

2. If we set Start Time=01:12:00, End Time=05:23:00 and we checked the message type for 'Position' and 'Breakdown', the query string would be as follows:

```
(Root.XML.VehicleForecast.Position.timestamp>= '01:12:00' and  
Root.XML.VehicleForecast.Position.timestamp <='05:23:00' ) and  
(Root.XML.VehicleForecast.Status='position' or  
Root.XML.VehicleForecast.Status='breakdown' )
```

If the query string is formed, we add that to the subscribe message just before subscribing to the selected topic.

```
subscribeMsg.reset();  
subscribeMsg.addFilter(queryString);
```

Otherwise we just subscribe to the topic without adding any filter.

## 7.5.7 Subscribe on request

In this part we discuss the issue that was raised in 7.4.2, “Comments and extensions” on page 181, where the JMS-based subscriber application receives more than one message if it subscribes during the course of message publication. We address this issue in this AMI-based solution by following the steps as discussed below.

1. In order to populate the AMI message header for features like Publish On Request, Request Updation and Use Correl Id as ID, we make use of lower-level AMI calls addAmElement(amElement). We define the following AmElement in subscriber application bean:

```

AmElement pubOnRequestOnly = new
AmElement(AmConstants.AMPS_REGISTRATION_OPTIONS,AmConstants.AMPS_PUB_ON_REQ
UEST_ONLY);
AmElement requestUpdate = new
AmElement(AmConstants.AMPS_COMMAND,AmConstants.AMPS_REQUEST_UPDATE);
AmElement correlIdasId= new
AmElement(AmConstants.AMPS_REGISTRATION_OPTIONS,AmConstants.AMPS_CORREL_ID_
AS_ID);
AmElement subQueue=null;

```

2. We create a sender session and a receiver session with Subscriber Sender Service name and Receiver Service name. Then we retrieve the Receiver Service queue name and we create an AmElement.

```

amSender = subscriberSession.createSender(subinit.getControl_queue());
amReceiver = subscriberSession.createReceiver(subinit.getSub_queue());
String q = amReceiver.getQueueName();
subQueue = new AmElement(AmConstants.AMPS_Q_NAME, q);

```

3. During subscription, we do the following steps:

- a. Reset the subscribe message:

```
subscribeMsg.reset();
```

- b. If the request is for content-based subscription, then we add the supplied query string as a filter:

```
subscribeMsg.addFilter(queryString);
```

- c. We subscribe on request only. We add the correlIdasId and pubOnRequestOnly AmElement to the Subscribe Message header.

```

topicElement = new AmElement(AmConstants.AMPS_TOPIC,topicName);
subscribeMsg.addTopic(topicName);
subscribeMsg.addElement(correlIdasId);
subscribeMsg.addElement(pubOnRequestOnly);
correlId=generateCorrelId();
    if (correlId != null )
    {
        subscribeMsg.setCorrelationId(this.correlId.getBytes());
    }

```

```
subscriber.subscribe(subscribeMsg, policy);
```

- d. Immediately after subscription, we reset the subscribe message once again and add requestUpdate Element and set the Subscriber Queue name to the Subscribe Message header. Then we send the Subscribe Message to the broker by the amSender service. By this approach we request the broker to publish the updated message to the subscriber queue (which is set in the MessageHeader) thereby ensuring that only the updated message is received for a given topic in the subscriber queue.

```
subscribeMsg.reset();
```

```
subscribeMsg.addElement(requestUpdate);
subscribeMsg.addTopic(topicName);
```

3. If the request is for content-based subscription, then we add the following lines in which we add the query string as a filter:

```
subscribeMsg.addFilter(queryString);
subscribeMsg.addElement(correlIdasId);
subscribeMsg.setCorrelationId(this.correlId.getBytes());
subscribeMsg.addElement(subQueue);
amSender.send(subscribeMsg);
```

4. Finally we receive the output as shown in Figure 7-8 on page 181.

This Web-based subscriber application in AMI has demonstrated the power of content-based subscription as well as receiving updated messages for a given topic.

## 7.6 Example - a three-tier implementation

All the example application usage cases discussed so far used just one broker for simplicity. This is not a limitation. For example, we were able to easily set up a test environment with three interconnected brokers (see Figure 7-11).

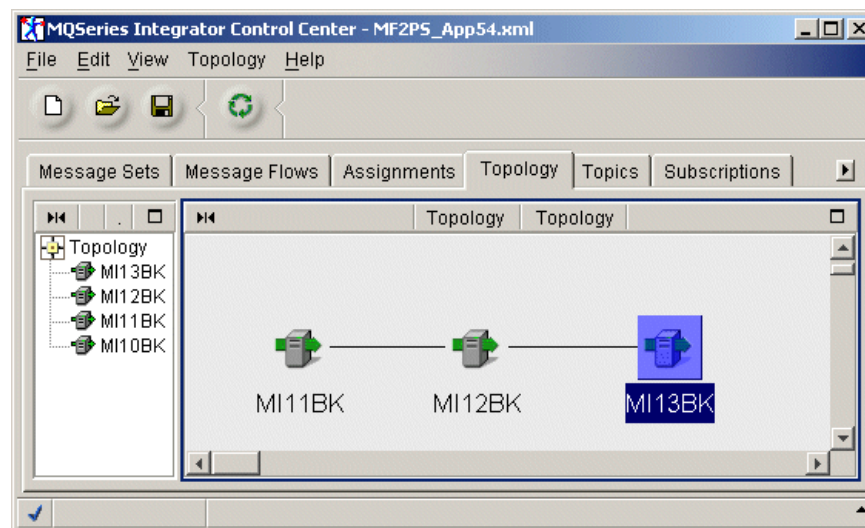


Figure 7-11 A three tier broker network

- Broker MI11BK runs the publisher application; in a real-world scenario this might have been the field broker serving a multitude of publishing mobile devices embedded in vehicles.

- ▶ Broker MI12BK runs the Forecast application; in a real-world scenario this might have been the back office broker enforcing corporate message transformation and routing rules.
- ▶ Broker MI13BK runs the subscriber application; in a real-world scenario this might have been the front office broker serving requests coming from users using different client technologies (for example, legacy cobol applications, Web browser or messaging enabled pervasive devices).

To implement this scenario we followed these steps (we do not go into the details of each step, because they are already covered in the book):

1. We created four queue managers (one for the Configuration Manager and the User Name Server plus one for each broker).
2. We defined the Configuration Manager queue manager as the full repository of a MQSeries queue manager cluster to which all the other queue managers were subsequently joined.
3. We defined the relevant MQSeries Integrator components: Configuration Manager, User Name Server and brokers.
4. We defined the broker topology shown in Figure 7-11 on page 190.
5. We deployed the Default Publish/Subscribe message flow to broker MI11BK and MI12BK (MI13BK does not need one because it only hosts subscribers).
6. We ran the example application as usual.

## 7.7 Final content-based subscriptions considerations

Content-based subscriptions are a very powerful feature. In the following sections we review the main things to consider before implementing a subscriber application based upon them.

### 7.7.1 Applicability

If you compare the simple AMI Web subscriber discussed in 7.5, “AMI Web application and message filtering” on page 182 with the AMI Web subscriber we just described, you notice how little the code of the two programs varies (we basically add a dynamically built ESQL filter expression to our existing subscription request) and how big is the difference in terms of functionality.

Generally the adoption of content-based subscriptions provides for streamlined applications and sometimes avoids the use of databases used only to locally filter transient data.

Many publish/subscribe applications can then benefit from content-based subscriptions.

## 7.7.2 Content-based subscription simulation

Content-based subscriptions are a unique feature of MQSeries Integrator and are not supported by MQSeries Publish/Subscribe.

In some special cases you can simulate a content-based subscription with a topic-based one simply designing to topic three in order to include the value on which to filter.

For example suppose that you are publishing weather forecasts on topic *Weather/Tomorrow/City*, and you want to receive the data of only the sunny cities.

One way of doing this is to subscribe to *Weather/Tomorrow/#* with filter `Forecast='sunny'`; another way of obtaining the same result using only an extended topic hierarchy is to subscribe to *Weather/Tomorrow/#/sunny*.

In general complex filter expressions (for example, involving the logical operator NOT) operating on fields having non-discrete values (for example, the US Dollar versus English Pound exchange rate) cannot be simulated just with an ad hoc designed topic hierarchy.

## 7.7.3 Performance implications

The adoption of content-based subscriptions can have mixed impacts on the overall system performance:

- ▶ The evaluation of filter by the broker slows down the time spent by the broker on each message (for example, depending on the filter complexity it may double).
- ▶ The subscriber application throughput and the network load can significantly benefit from the reduced number of messages that are forwarded by the broker.

So there is no golden rule. It very much depends on network speed, filter complexity, and filter selectivity.



# Conclusions

This brief chapter will quickly review publish/subscribe applications.

## 8.1 The technology

This redbook contains many different examples of publish/subscribe technology applications. It should now be apparent how its applicability spans domains and is by no means confined to niche applications dealing with stock trading.

Here are some emerging areas of applicability for the publish/subscribe paradigm.

### 8.1.1 Web-based applications

We have discussed this type of application thoroughly in the book. Typically the content will be retained by the broker and the subscriptions will be very short lived.

Moreover Web portal technologies can leverage publish/subscribe features in order to attain a higher level of content personalization for the Web users.

### 8.1.2 Pervasive applications

Complex applications residing on mobile devices can benefit from a publish/subscribe broker to significantly reduce the complexity of message routing and messaging administration. In such an environment, content-based subscription can be used to further filter the amount of data to be received.

### 8.1.3 Enterprise Application Integration

Heterogeneous application environments in large enterprises can exploit the services of a publish/subscribe broker in order to dynamically accommodate everchanging business requirements, without the intervention of the application codes.

In such environments, typically subscriptions will never expire and the exchanged messages will be persistent. Maybe these are the environments where the benefits of having a publish/subscribe broker running over a proven and robust messaging provider such as MQSeries are most evident.

One of the key features of an enterprise-wide publish/subscribe backbone is to have fine-grained control over the corporate topic hierarchy. MQSeries Integrator offers such a feature.



## 8.2 IBM offerings

These are IBM's current offerings in the publish/subscribe arena.

### 8.2.1 MQSeries Publish/Subscribe

MQSeries Publish/Subscribe is a product extension that can be downloaded free of charge from the IBM Web site. It enables customers to build a topic-based publish/subscribe infrastructure. Publisher and subscriber applications can run on any of the more than 35 MQSeries supported platforms, while the broker can currently run only on the AIX, HP-UX, Solaris, Linux, Windows NT and Windows 2000 platforms.

MQSeries Publish/Subscribe brokers support queue-level security and parallel computation, and can be interconnected in hierarchies.

### 8.2.2 MQSeries Integrator

MQSeries Integrator is a full-fledged message broker. Its publish/subscribe capabilities can be freely combined with other powerful features, such as message transformation, content-based routing, context-based routing, and database integration.

Topic-based as well as content-based subscriptions are available.

The security model implemented by MQSeries Integrator is both queue-based and topic-based, so the administrator can create an ACL for each restricted topic.

The MQSeries Integrator publish/subscribe implementation is highly scalable. You can combine multiple brokers in tightly coupled subnetworks (*collectives*) in order to obtain maximum performance.

MQSeries Integrator brokers and collectives can be combined with MQSeries Publish/Subscribe brokers in mixed broker networks.

### 8.2.3 More Information

For more information and all the latest developments in IBM's messaging and Publish/Subscribe products, please go to:

<http://www-4.ibm.com/software/ts/mqseries/>





**A**

## **Hardware and software environment**

During the writing of this redbook, we used five Intel machines to develop and test the sample publish/subscribe applications discussed in this book.

The following sections contain a detailed account of the hardware and software we used.

## Hardware

The PCs we used were:

- ▶ IBM PC - Model 6579
- ▶ Intel Pentium III running at 864 MHz
- ▶ 522 MB of real memory and 768 MB of virtual memory.
- ▶ 22 GB hard disk

All the machines were on a Windows 2000 domain, and were connected together by a TCP/IP token-ring network.

## Software

The operating system and other software on the machines was:

- ▶ Microsoft Windows 2000 Professional V5.0 2195 Build 2195
- ▶ IBM MQSeries for Windows 2000 V5.2
- ▶ IBM MQSeries Integrator for Windows 2000 v2.0.2
- ▶ IBM DB2 V7.2
- ▶ IBM WebSphere Application Server V3.5.4
- ▶ IBM HTTP Server V1.3.12.3 Support
- ▶ Netscape Communicator V4.77
- ▶ MS Visual C++ Standard Edition V6.0
- ▶ IBM VisualAge for Java for Windows V3.5.3
- ▶ IBM Developer Kit Java (JDK) V1.2.2
- ▶ IBM MQSeries Classes for Java and Java Message Service (MA88)
- ▶ IBM MQSeries Publish/Subscribe (MA0C)
- ▶ IBM MQSeries Application Messaging Interface (MA0F) V1.11



## MQSeries Publish/Subscribe administration commands

This appendix is a brief overview of the main MQSeries Publish/Subscribe administration commands, please refer to *MQSeries Publish/Subscribe User's Guide*, GC34-5269 for more detailed information.

**Note:** The user ID invoking these MQSeries Publish/Subscribe commands needs to be a MQSeries administrator.

## strmqbrk

The **strmqbrk** command is used to start a broker. The first time this command is run on a queue manager, all the relevant MQSeries objects are automatically created.

```
strmqbrk -m MYQMGRNAME  
strmqbrk -p MYPARENTQMGRNAME -m MYQMGRNAME
```

You can use the **-p** option to specify the name of the parent queue manager in a broker network, once you specified such value it is retained by the broker upon restart.

## dspmqbrk

The **dspmqbrk** command is used to check the status of the broker. Possible states are: *starting*, *running*, *stopping*, *quiescing*, *not active* and *ended abnormally*.

```
dspmqbrk -m MYQMGRNAME
```

## endmqbrk

The **endmqbrk** command is used to stop a broker. There are two options: **-c** requests a controlled shutdown (default), **-i** requests an immediate shutdown.

```
endmqbrk -i -m MYQMGRNAME
```

.



# MQSeries Integrator administration commands

This appendix is a brief overview of the main MQSeries Integrator administration commands. Please refer to *MQSeries Integrator Administration Guide*, SC34-5792 for more detailed information.

The most commonly used commands on Windows 2000 or NT are also available by clicking **Start -> Programs -> IBM MQSeries Integrator 2.0 -> Command Assistant**.

Please note that:

- ▶ The MQSeries Integrator command assistant does not provide support for starting and stopping the MQSeries Integrator components. You must issue the MQSeries Integrator administration commands `mqsisstart` and `mqsisstop` from the command line.
- ▶ The user ID invoking MQSeries Integrator commands must be part of relevant MQSeries Integrator groups.

## MQSeries Integrator pub/sub admin commands

Here is a comprehensive list of the available MQSeries Integrator commands. We provide examples only for those dealing with the integration with MQSeries Publish/Subscribe brokers.

- ▶ Generic start/stop command for the MQSeries Integrator components (brokers, configuration manager and user name server):
  - **mqsistart**
  - **mqsistop**
- ▶ List and trace commands:
  - **mqsilist**
  - **mqsiformatlog**
  - **mqsisilcc**
  - **mqsireporttrace**
  - **mqsireadlog**
- ▶ MQSeries Integrator component creation commands:
  - **mqsicreateconfigmgr**
  - **mqsicreatebroker**
  - **mqsicreateusername server**
- ▶ MQSeries Integrator component deletion commands:
  - **mqsidedeleteconfigmgr**
  - **mqsidedeletebroker**
  - **mqsidedeleteusername server**
- ▶ MQSeries Integrator commands used to interface MQSeries Publish/Subscribe brokers:
  - **mqsilistmqpubsub**
  - **mqsijoinmqpubsub**
  - **mqsiclearmqpubsub**

## Admin commands for mixed brokers

Here are some examples of MQSeries Integrator commands that are used when dealing with mixed broker networks, which are networks made of MQSeries Integrator brokers and MQSeries Publish/Subscribe brokers.



## **mqsilistmqpubsub**

The command **mqsilistmqpubsub** is used to display the status and supported streams of the MQSeries Publish/Subscribe neighbor brokers of the specified MQSeries Integrator Broker.

```
mqsilistmqpubsub MYBROKERNAME
```

## **mqsijoinmqpubsub**

The command **mqsijoinmqpubsub** is used to join the specified MQSeries Integrator broker as a child to an MQSeries Publish/Subscribe broker.

```
mqsijoinmqpubsub MYBROKERNAME -p MYPARENTQUEUEMANAGERNAME
```

The **-p** option identifies the MQSeries Publish/Subscribe broker that will be the parent of the MQSeries Integrator broker.

The successful completion of this command only indicates that the MQSeries Integrator broker has accepted the request, not that the required action has completed.

## **mqsiclearmqpubsub**

The command **mqsiclearmqpubsub** is used to remove an MQSeries Publish/Subscribe broker as a neighbor of an MQSeries Integrator broker.

```
mqsiclearmqpubsub MYBROKERNAME -n NEIGHBORQUEUEMANAGERNAME
```

To complete this action, you must also issue the MQSeries Publish/Subscribe command **clrmqbrk** against the MQSeries Publish/Subscribe broker.

When both clear commands have completed, all publish and subscribe traffic between the two brokers ceases.





**D**

## **The GUI-based subscriber application**

The GUI subscriber application is a stand-alone Java program using AMI for subscription hardware.

## AMI configuration

Copy the amt.xml and amthost.xml files from the AMIRepository directory to the amt subdirectory located in the MQ installation directory (for example, C:\Program Files\IBM\MQSeries\amt or C:\Program Files\MQSeries\amt). If another amt.xml or amthost.xml file is already present in this directory, you may want to rename the previous version of these files first.

Edit amthost.xml to adapt the defaultConnection attribute of the queue manager's element to the name of your queue manager.

Run the vehicle.mqsc script on your queue manager to create three queues (named VEHICLE.xxx) and a server connection channel.

## Subscriber configuration

Edit the two entries SUBLIB and MQ of the subgui.bat batch file in the subgui directory:

- ▶ Set the SUBLIB value to the complete name of the directory where you copied this application, that is the directory containing the subscribe.properties file.
- ▶ Set the MQ value to the MQ installation directory.
- ▶ Start the GUI-based subscriber application by double-clicking the subgui.bat file.

## Simple Web subscriber application

The simple Web subscriber application will be ported to the WebSphere Application Server. It uses Java AMI program for subscription.

## AMI configuration

Check that the default model queue SYSTEM.DEFAULT.MODEL.QUEUE is defined.

## Servlet configuration

The servlet configuration has been discussed in detail in 6.1.2, "Servlet configuration" on page 155, so please follow those steps.

## Additional WebSphere Application Server configuration

The following JAR files are required to execute the sample Web application.

- ▶ com.ibm.mq.amt.jar
- ▶ xerces.jar

We can either add these JAR file paths to the application server classpath or to the classpath of the Web application named itso:

### 1. Add to application server classpath

- Open the admin.config file of the WebSphere Application Server from WebSphere installation path\appserver\bin. Edit the entry com.ibm.ejs.sm.adminserver.classpath and add the path of the AMI JAR file that is located at MQInstallpath\java\lib\ . For example, c:\MQSeries\java\lib\com.ibm.mq.amt.jar.

- Also add the path of the xerces.jar file where you have unzipped the file. For example, c:\websub.

```
com.ibm.ejs.sm.adminserver.classpath=c:/MqSeries/java/lib/com.ibm.mq.amt.jar;c:/MqSeries/java/lib;c:/websub/xerces.jar;
```

- Restart the application server.

### 2. Add to Web application (itso) classpath

- Add the path of the AMI JAR file, which is located at MQInstallpath\java\lib\ . For example, c:\MQSeries\java\lib\com.ibm.mq.amt.jar.

- Add the path of the xerces.jar file.

- Restart the Web application.

- Type the following URL:

```
http://hostname/itso/start.html
```

to start the application, where hostname is the name of the machine on which WebSphere Application Server is running.

## Web subscriber Forecast application

The Web subscriber Forecast application will be ported to the WebSphere Application Server. It uses JMS for subscriptions.

### JMS configuration

Follow the steps as discussed in “JMS configuration” on page 214.

## MQSeries Integrator configuration

Start Broker ConfigMgr from Windows Services. Import the JMSForecastPubSubMsgFlow.xml file, which is provided in the MQSeries Integrator directory. Start the MQSeries Integrator broker from Windows Services and deploy the broker.

## Servlet configuration

The servlet configuration has been discussed extensively in 6.1.2, “Servlet configuration” on page 155. Follow the steps discussed there and create a servlet entry as ForecastServlet and servlet class name as com.ibm.itso.swa111.subscribe.web.ForecastServlet under the Web application itso folder. Enter the init parameter entries as discussed in 7.3.1, “Servlet configuration” on page 175. The value of the init parameters can be obtained from the pub.properties file.

## Additional WebSphere Application Server configuration

Follow either of the steps as discussed in “Additional WebSphere Application Server configuration” on page 207 and add the following JAR files:

- ▶ com.ibm.mq.jar
- ▶ com.ibm.mq.iiop.jar
- ▶ com.ibm.mqbind.jar
- ▶ com.ibm.mqjms.jar
- ▶ xerces.jar

Type the following URL:

**`http://hostname/itso/PTInput.html`**

where hostname is the name of the machine on which WebSphere Application Server is running.

## Advanced Web subscriber Forecast application

This advanced Web subscriber Forecast application demonstrates content-based subscription. It uses Java AMI for subscription.

## AMI configuration

Check that the default model queue SYSTEM.DEFAULT.MODEL.QUEUE is defined.

## Servlet configuration

The servlet configuration has been discussed extensively in 6.1.2, “Servlet configuration” on page 155. Follow the steps discussed there and create a servlet entry as `ForecastAdvServlet` and servlet class name as `com.ibm.itso.swa111.subscribe.web.ForecastAdvServlet` under the Web app `itso` folder. Enter the init parameter entries found in 7.3.1, “Servlet configuration” on page 175.

## Additional WebSphere Application Server configuration

Follow the steps discussed in “Additional WebSphere Application Server configuration” on page 207 and add the following JAR files:

- ▶ `com.ibm.mq.amt.jar`
- ▶ `xerces.jar`

Type the following URL:

**`http://hostname/itso/ForecastAdv.html`**

where `hostname` is the name of the machine on which WebSphere Application Server is running.







## **Additional material**

This redbook refers to additional material that can be downloaded from the Internet as described below.

## Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

<ftp://www.redbooks.ibm.com/redbooks/SG246282>

Alternatively, you can go to the IBM Redbooks Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246282.

## Using the Web material

The additional Web material that accompanies this redbook includes the following file:

<i>File name</i>	<i>Description</i>
<b>sg246282.zip</b>	Zipped code

## System requirements for downloading the Web material

The following system configuration is recommended:

<b>Hard disk space:</b>	20 MB minimum
<b>Operating System:</b>	Windows
<b>Processor:</b>	500 MHz or higher
<b>Memory:</b>	256 MB, preferably 512 MB

## How to use the Web material

In this chapter we will describe what you need to do in order to make the different application components run.

Create a directory on your workstation, and unzip the contents of the Web material zip file into this folder.

### Preliminaries

It is assumed that:

- ▶ MQSeries Version 5.2 is installed, a queue manager has been created and is currently running. We refer to this queue manager as ITS0 but the name has no importance.

- ▶ The SupportPacs MA88 and MA0F are installed.
- ▶ WebSphere is installed and running (this is only needed for JMS).
- ▶ A JDK or JRE is installed on your machine. To check if a JDK or JRE is installed, you can run the command `java -fullversion`. If you receive an error message because the command is unknown, then your computer is not aware of any JDK installed.

In that case, you can use the JDK installed with WebSphere Application Server by editing the PATH environment variable of your computer and add the following entry:

```
C:\WebSphere\AppServer\jdk\bin;
```

where C:\WebSphere\AppServer is the installation directory of WebSphere Application Server.

If you are not using WebSphere Application Server and no JDK or JRE is installed, you can also download and install any other JDK.

- ▶ Either MQSeries Integrator or MQSeries Pub/Sub (SupportPac MA0C) must be installed.
  - if MQSI is chosen, you must:
    - Create the queue SYSTEM.BROKER.DEFAULT.STREAM with the MQ script default\_stream.mqsc provided in the MQSC directory.
    - Import the example message flows provided with MQSeries Integrator (from C:\mqsi20\examples\SamplesWorkspaceForImport, where C:\mqsi20 is the installation directory of MQSeries Integrator) and deploy the default publish/subscribe message flow on the queue manager.
  - if MQSeries Publish/Subscribe is chosen, the MQSeries Pub/Sub SupportPac (MA0C) must be installed and the broker must be started with the `strmqbrk` command.

## The publisher application

The publisher application is a Java program using either AMI (C or Java), JMS or C MQI for publishing messages. All the other components of the application require the publisher.

### **AMI configuration**

1. Copy the amt.xml and amthost.xml files from the AMI Repository directory to the amt subdirectory located in the MQ installation directory (for example, C:\Program Files\IBM\MQSeries\amt or C:\Program Files\MQSeries\amt). If another amt.xml or amthost.xml file is already present in this directory, you may want to rename the previous version of these files first.

2. Edit amthost.xml to adapt the defaultConnection attribute of the queuemanagers element to the name of your queue manager.
3. Run the vehicle.mqsc script on your queue manager to create three queues (named VEHICLE.xxx), and a server connection channel.

### **JMS configuration**

1. Start the JNDI server (WebSphere Application Server Admin Server, VisualAge Persistent Name Server, LDAP Server, etc.).
2. Copy the JMSAdmin.scp script provided in the pub directory to the \java\bin subdirectory located in the MQSeries installation directory (for example, C:\Program Files\MQSeries\Java\bin).
3. Edit the script JMSAdmin.scp provided in the pub directory to replace the QMANAGER and BROKERQMGR parameter values with your queue manager name, the HOST parameter value with your host name, the PORT value with the port on which the queue manager listens (for example, 1414) in the two **def tcf** commands.
4. Edit the JMSAdmin.config file in the same directory to specify the necessary parameters in order to reach your JNDI server.

If you are using WebSphere Admin Server on your local machine, the parameters are:

```
INITIAL_CONTEXT_FACTORY=com.ibm.ejs.ns.jndi.CNInitialContextFactory
PROVIDER_URL=iiop://localhost/
```

5. You should now run the command **JMSAdmin.bat < JMSAdmin.scp > JMSAdmin.out** from the same to define the objects contained in the JMSAdmin.scp script.

To avoid any problems of path or classpath when running JMSAdmin.bat, we provide a JMSAdmin2.bat file that you could of easier use.

Copy JMSAdmin2.bat into the same directory as JMSAdmin.bat.

Edit the SET MQ entry in this batch file to point to the installation directory of MQSeries.

Open a command prompt, change the current directory to the directory containing the JMSAdmin2.bat file and run the command: **JMSAdmin2.bat < JMSAdmin.scp > JMSAdmin.out**

Check the JMSAdmin.out file to see if the script was correctly executed.

The following output indicates a successful completion:

```
5648-C60 (c) Copyright IBM Corp. 1999. All Rights Reserved.
Starting MQSeries classes for Java(tm) Message Service Administration
```

```
InitCtx> Context not found, or unremovable
```



If the queue manager is on the same machine as where the pub.bat batch file is run, and if you are using JMS, you can use JMS in binding mode by changing the jndiTopicConnectionFactory property in pub.properties from jms/ITSOPS to jms/ITSOPSBND.

5. Start the publisher application by double-clicking the pub.bat file.
6. The program stops automatically when all the messages for all the vehicles have been published.

## **The Forecast application**

This standalone application is only using AMI and Java. It subscribes to the publications sent by the publisher application, waits for receiving the publications and publishes forecast messages.

### ***AMI configuration***

The required steps have been done when configuring the publisher application.

### ***Forecast configuration***

1. Check that the default model queue SYSTEM.DEFAULT.MODEL.QUEUE is defined.
2. Go in the forecast directory and edit the two entries FORECASTLIB and MQ in the forecast.bat batch file:
  - Set the FORECASTLIB value to the complete name of the forecast directory (indicate the full path of the directory containing the forecast.properties file).
  - Set MQ value to the MQ installation directory.
3. Check the properties file forecast.properties and decide if you want to run the application in single or multi-threaded mode (nbrThreads property).
4. Start the forecast application by double-clicking the forecast.bat file.
5. To end the program, type **end** on the command line and press Enter.

### ***Measure for content-based subscription***

For content-based subscription, we need to do the following changes in the forecast.properties and pub.properties file.

1. Update the following entry in the forecast.properties file:

```
publisherForecast=VEHICLE.FORECAST.PUBLISHER to  
publisherForecast=VEHICLE.FORECAST.2.PUBLISHER
```

2. Update the following entries in the pub.properties file:

```
publisherAccident=VEHICLE.ALERT.PUBLISHER to  
publisherAccident=VEHICLE.ALERT.2.PUBLISHER  
publisherBreakdown=VEHICLE.ALERT.PUBLISHER to  
publisherBreakdown=VEHICLE.ALERT.2.PUBLISHER
```

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this redbook.

## IBM Redbooks

For information on ordering these publications, see “How to get IBM Redbooks” on page 218.

- ▶ *MQSeries Primer*, a redpaper found at <http://www.ibm.com/redbooks>
- ▶ *Business Integration Solutions with MQSeries Integrator*, SG24-6154
- ▶ *MQSeries Version 5.1 Administration and Programming Examples*, SG24-5849

## Other resources

These publications are also relevant as further information sources:

- ▶ *Using Java*, SC34-5456
- ▶ *Application Messaging Interface*, SC34-5604
- ▶ *IBM MQSeries for Windows NT and Windows 2000 V5R2 Quick Beginnings*, GC34-5389
- ▶ *IBM MQSeries Integrator for Windows NT V2.01 Quick Beginnings*, GC34-5389
- ▶ *IBM MQSeries Publish/Subscribe User's Guide*, GC34-5269
- ▶ *IBM MQSeries An Introduction to Messaging and Queueing*, GC33-0805
- ▶ *IBM MQSeries Application Programming Guide*, SC33-0807
- ▶ *IBM MQSeries System Administration*, SC33-1873
- ▶ *IBM MQSeries Planning Guide*, GC33-1349
- ▶ *IBM MQSeries Integrator Introduction and Planning*, SC34-5599
- ▶ *IBM MQSeries Integrator Using the Control Center*, SC34-5602
- ▶ *IBM MQSeries Integrator Administration Guide*, SC34-5792
- ▶ *IBM MQSeries Integrator Programming Guide*, SC34-5603

## Referenced Web sites

These Web sites are also relevant as further information sources:

- ▶ MQSeries manuals  
<http://www-4.ibm.com/software/ts/mqseries/library/manuals/>
- ▶ MQSeries downloads and SupportPacs  
<http://www-4.ibm.com/software/ts/mqseries/downloads/>
- ▶ MQSeries APARs and Fix packages  
<http://www-4.ibm.com/software/ts/mqseries/support/fixes/>

## How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

## IBM Redbooks collections

Redbooks are also available on CD-ROMs. Click the CD-ROMs button on the Redbooks Web site for information about all the CD-ROMs offered, as well as updates and formats.



# Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

The following terms are trademarks of other companies:

Tivoli, Manage. Anything. Anywhere., The Power To Manage., Anything. Anywhere., TME, NetView, Cross-Site, Tivoli Ready, Tivoli Certified, Planet Tivoli, and Tivoli Enterprise are trademarks or registered trademarks of Tivoli Systems Inc., an IBM company, in the United States, other countries, or both. In Denmark, Tivoli is a trademark licensed from Kjøbenhavns Sommer - Tivoli A/S.

C-bus is a trademark of Corollary, Inc. in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States and/or other countries.

PC Direct is a trademark of Ziff Communications Company in the United States and/or other countries and is used by IBM Corporation under license.

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks of Intel Corporation in the United States and/or other countries.

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

SET, SET Secure Electronic Transaction, and the SET Logo are trademarks owned by SET Secure Electronic Transaction LLC.

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

# Abbreviations and acronyms

<b><i>ACL</i></b>	Access Control List
<b><i>AMI</i></b>	Application Messaging Interface
<b><i>IBM</i></b>	International Business Machines Corporation
<b><i>ITSO</i></b>	International Technical Support Organization
<b><i>JMS</i></b>	Java Messaging Service
<b><i>JNDI</i></b>	Java Naming Directory Interface
<b><i>MQ</i></b>	MQSeries
<b><i>MQI</i></b>	Message Queuing Interface
<b><i>UNS</i></b>	User Name Server
<b><i>XML</i></b>	Extensible Markup Language



# Index

## A

- access 138
- access control 112
- acknowledgments 169
- ACL 139
- adapters 5
- administration 44
- administrator 14
- alert 108, 144
- AMI 12, 15, 28, 52, 56
- AMI administration 94
- AMI Tool 54
- amt.lib 83
- amtc.h 83
- API 12
- Application Messaging Interface 4, 8
- attribute 86
- audit 143
- authorities 109

## B

- backup 35
- bean 98, 161, 179
- bitstream 88
- blocking 101
- bootstrap 44
- broker ix, 3, 5, 9, 12, 24, 84
- broker control queue 13
- business logic 74
- business rules 167

## C

- C MQI 16
- channels 113
- child 110
- classes 32
- classpath 42
- cluster 108
- clustering 124
- collective 9, 123, 138
- commit 171
- Compiling 88

- component 126
- compute 166
- configuration 145
- configure 29
- connection 4, 38, 161
- ConnectionFactory 38, 48
- console 46
- content 3, 7
- content filtering 16
- content-based subscriptions 5
- context 46, 166
- Control Center 119, 125
- conventions 84
- conversions 122
- correlation identifier 53
- CorrelId 95, 99, 100
- coupling 110
- Custom 34

## D

- database 5, 167
- DB2 42, 54
- debug 88
- decoupling 110
- default 48
- default port 44
- definitions 60
- deploy 119
- deregistration 150
- deserialize 39
- design 138
- Destination 38
- diagnostic 81
- distributed 2
- domain 126, 129
- durable subscriptions 15
- dynamic queue 7

## E

- environment 51, 99
- ESQL 188
- event 25
- example 118

exception 101  
expiration date and time 7  
Explorer 88  
export 181

## **F**

filter 26, 188  
filtering 7, 26  
flow 116  
forecast 25, 169  
functionality 121

## **G**

gateways 5  
General 69  
generic 38, 110  
global units of work 167  
groups 126

## **H**

handles 93  
header 16, 121  
hierarchy 9, 110, 113, 145  
hostfile 99  
HP 4  
hub 110

## **I**

IIOp 44  
Implicit Stream Naming 133  
import 116  
inheritance 127  
install 29  
interbroker 112  
interoperability 24  
interpret 100  
invocation 150  
isomorphic 54

## **J**

J2EE 36  
JAR 35  
Java 28, 88  
Java Message Service 8  
Java Native Interface 77  
JDBC 32, 42  
JDK 50

JMS 12, 14, 17, 28  
JMSAdmin 39, 49, 50  
JMSX 104  
JNDI 17, 29, 38, 76  
JRE 50

## **K**

kernel 55

## **L**

LDAP 54  
leaves 112  
Libraries 39  
lifetime 163, 173  
linking 88  
logic 23, 172

## **M**

MA0C 5, 29  
MA0F 56  
MA88 8, 32  
Mapping 121  
message 2  
Message Queue Interface 4, 8  
MessageConsumer 38  
MessageProducer 38  
method 53, 76, 101, 162  
Microsoft Active Directory 54  
migration 24, 115, 146  
model queue 36  
MQI 12, 15  
MQInput 118  
MQMD 15  
MQPS 86  
MQRFH 53, 54, 84, 85, 180  
MQRFH1 16  
MQRFH2 16, 54  
mqsc 35  
MQSeries 4  
MQSeries Clustering feature 124  
MQSeries Integrator 4  
Multiple brokers 3  
multithreaded 25

## **N**

network 4, 5, 110, 129  
node 123, 129, 134

- nodes 5
- non-durable 14
- Non-persistent 6

## O

- object-oriented 53
- objects 93, 99
- offset 104
- optimize 124
- overhead 127
- overview 22

## P

- parallel processing 121
- parent 110, 145
- parsing 84, 101
- partition 109
- pattern 55
- payload 86, 104
- performance 7
- persistent 6
- persistent messages 6
- Persistent Name Server 39, 42
- point-to-point 4, 47
- policies 53, 60, 68, 99
- policy definitions 54
- policy handler 55
- port 50
- portability 91
- principals 128
- properties 74, 92, 98
- PubJava 88
- PubLauncher 28
- publication node 14
- publish ix, 91
- Publish/Subscribe ix, 1
- publisher ix, 2, 12, 60
- PubThread 28, 76

## Q

- queries 165
- queue manager 5
- quiesced 113

## R

- real-time 25
- receiver 128

- receiver service 65
- recovery 130
- Redbooks Web site 218
  - Contact us xii
- register 6, 14
- registration 135
- repository 54, 99
- responses 15, 85
- retained 20, 91, 105, 142
- retained publications 6, 24
- retained topic subtree 25
- RFH2 101
- Route 116
- rules 5

## S

- sample 130
- scope 9
- SecureWay 54
- security 129, 144
- Security Account Manager 126
- semantics 123, 167
- serialize 39
- service 53
- service definitions 54
- service point 14, 17, 54, 60
- services 60
- servlet 24, 150
- session 38, 89, 93
- simulator 130
- SMTP 55
- spokes 110
- state 25
- static 55
- stream 87, 108, 109
- stream queues 14
- structured topic names 5
- subscribe ix, 3, 100
- subscriber 2, 12, 60, 99
- subscription 122, 134
- subscription points 5
- subtrees 113, 124
- Sun 4
- SupportPac 28, 29
- syncpoint 7, 170

## T

- target 168

TCP/IP 48  
temporary subscription 7  
TextMessage 90  
thread 28, 73  
timing 83  
topic 3, 7, 39, 47, 88, 89, 96  
topology 110  
trace 118  
transform 5, 108  
transient 20  
translation 133  
transmission queues 112

## **U**

unsubscribe 3  
upgrade 5  
User Name Server 125  
utility calls 82

## **V**

Visual Basic 88  
Visual Studio 80  
VisualAge 29, 39

## **W**

waitInterval 99  
WebSphere 28  
WebSphere Application Server 24  
workloads 8  
wrapper 77

## **X**

XML 22, 54, 79, 87, 90, 94, 102  
XML4J 94





## MOSeries Publish/Subscribe Applications

(0.2" spine)  
0.17" x 0.473"  
90 x 249 pages







# MQSeries Publish/Subscribe Applications

## **Guidelines for designing a publish/subscribe environment**

Publish and Subscribe is an effective way of disseminating information to multiple users. Publish/Subscribe applications can help you to enormously simplify the task of getting business messages and transactions to a wide, dynamic and potentially large audience in a timely manner.

## **Developing and publishing with MQI, AMI, and JMS**

This redbook positions the MQSeries Publish/Subscribe to MQSeries Integrator Publish/Subscribe.

## **Programming examples for an information push system**

It will help you create, tailor and configure an application from publishing data through to subscribing via Web pages.

The books provides a broad understanding of how to build and run an entire publish/subscribe solution.

It will give you a quick start to designing and creating a solution and then migrating it from MQSeries Publish/Subscribe to MQSeries Integrator Publish/Subscribe.

## **INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION**

### **BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE**

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)