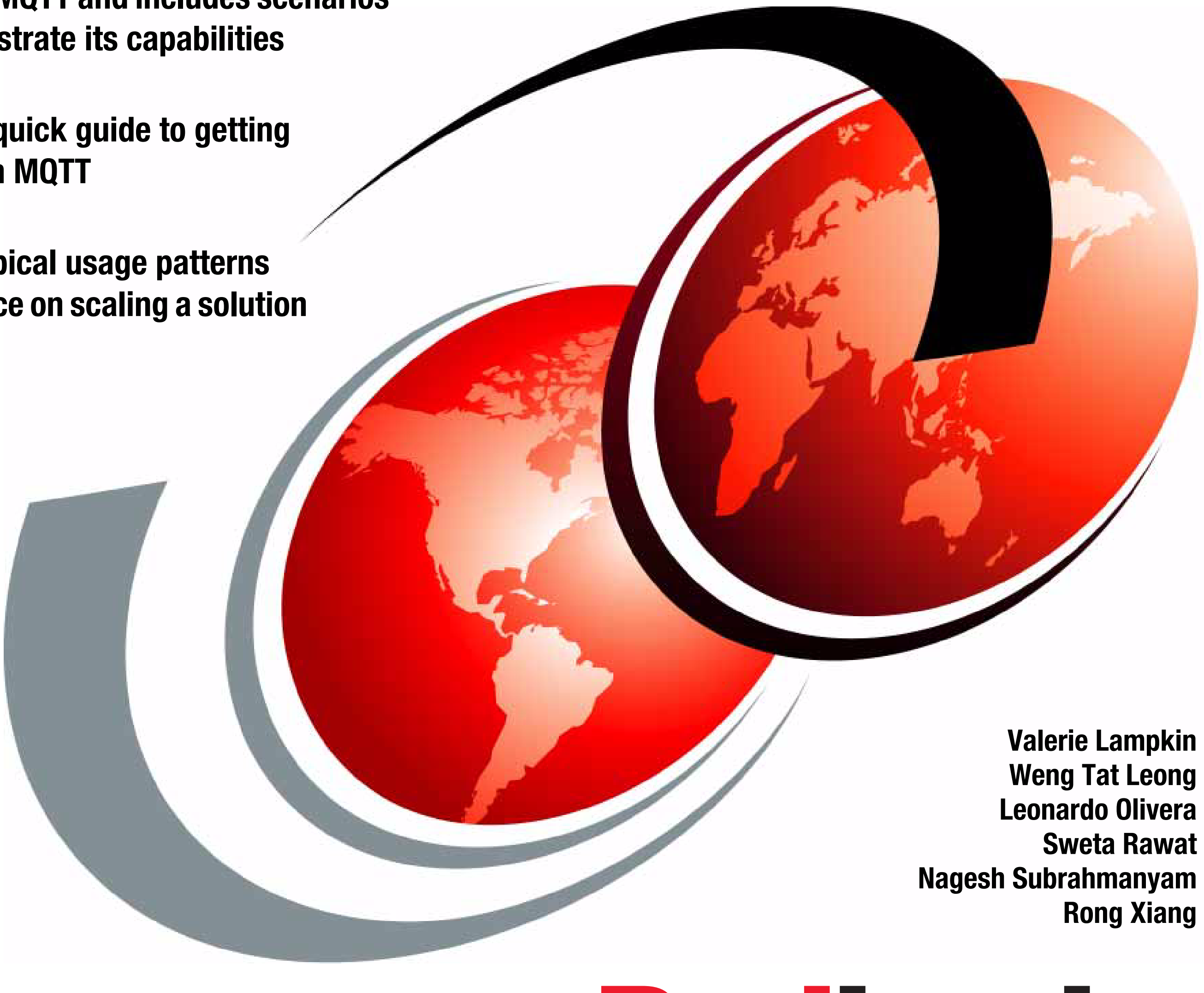
IBM® WebSphere®

Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry

Introduces MQTT and includes scenarios that demonstrate its capabilities

Provides a quick guide to getting started with MQTT

Includes typical usage patterns and guidance on scaling a solution



Valerie Lampkin
Weng Tat Leong
Leonardo Olivera
Sweta Rawat
Nagesh Subrahmanyam
Rong Xiang

Redbooks

ibm.com/redbooks

International Technical Support Organization

Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry

September 2012

SG24-8054-00

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

First Edition (September 2012)

This edition applies to Version 7, Release 1 of WebSphere MQ.

Contents

Notices

Trademarks

Preface

MQ Telemetry Transport (MQTT) is a messaging protocol that is lightweight enough to be supported by the smallest devices, yet robust enough to ensure that important messages get to their destinations every time. With MQTT devices such as smart energy meters, cars, trains, satellite receivers, and personal health care devices can communicate with each other and with other systems or applications.

This IBM® Redbooks® publication introduces MQTT and takes a scenario-based approach to demonstrate its capabilities. It provides a quick guide to getting started and then shows how to grow to an enterprise scale MQTT server using IBM WebSphere® MQ Telemetry. Scenarios demonstrate how to integrate MQTT with other IBM products, including WebSphere Message Broker. This book also provides typical usage patterns and guidance on scaling a solution.

The intended audience for this book ranges from new users of MQTT and telemetry to those readers who are looking for in-depth knowledge and advanced topics.

The team who wrote this book

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

Primary authors

The following people are the primary authors for this project.

Valerie Lampkin is a Middleware Technical Resolution Specialist in the USA. She has 13 years of experience supporting WebSphere MQ, previously as part of IBM Global Services and now with the Application and Integration Middleware Division of IBM Software Group. She holds a B.S. degree from Florida State University. She is a regular contributor to the WebSphere and IBM CICS® Support blog on IBM developerWorks®.

Weng Tat Leong is a Software Engineer in Beijing, China. He has over three years of experience in enterprise connectivity and integration. He holds a Master's degree in Control Engineering from Tsinghua University. His areas of expertise include mobile development, web development, cloud computing and enterprise integration. He has contributed a couple of articles on the IBM developerWorks website about enterprise integration and MQTT.

Leonardo Olivera is an IT Specialist at IBM Global Services, Uruguay. He has 17 years of experience in application development and systems integration. He holds a degree in Computer Science Engineering from Universidad Católica del Uruguay. His areas of expertise include Java Enterprise Edition architecture, WebSphere Portal software and Sensor and Actuators solutions. The last few years he has been working on mobile solutions for the healthcare industry.

Sweta Rawat is a software engineer in Bangalore, India. She has over five years of experience in the WebSphere MQ field. She holds a B.E degree in Electronics and Communication Engineering from Maharishi Dayanand University, India. Her areas of expertise include testing, configuring, tuning WebSphere MQ on distributed platforms with cross product enablement including IBM DB2® and Oracle. She has contributed to an IBM developerWorks article on configuring domain controller and setup of a multi-instance queue manager.

Nagesh Subrahmanyam is an IT Specialist in India. He has over 10 years of experience in the field spanning IBM z/OS®, WebSphere MQ, WebSphere Message Broker and XML technologies. He holds a Bachelor's degree in Mechanical Engineering from National Institute of Technology, Jamshedpur. His areas of expertise include Java and lately the Arduino platform. He has contributed a couple of articles on the IBM developerWorks website, and co-authored IBM Redbooks publications about XML and z/OS. He has also submitted a WebSphere MQ SupportPac.

Rong Xiang is a Staff Software Engineer in IBM Software Development Lab, China. He has four years of experience in Enterprise Integration under the JCA infrastructure and MQTT field. He holds a Master's degree in Computer Science and Technology from Xidian University, China. His areas of expertise include enterprise integration, distributed computing and M2M integration. He has contributed articles on the IBM developerWorks web site.

Figure 1 Left-to-right: Nagesh, Shawn, Leonardo, Sweta, Martin, Rong, Valerie, and Weng Tat

Additional authors

The following people contributed additional content to this project:

Gerald Kallas is working as Software IT Architect in retail space for national and international clients. He has executed a large number of successful projects in the business integration area for retail customers. Within the last couple of years he led the design of a solution called "Store Expeditor", based on requirements of a large retail enterprise. This solution has become today's IBM Lotus® Expeditor integrator, an IBM Software Group product that's especially designed for lightweight integration at remote locations.

Neeraj Krishna is a Senior Staff Software Engineer in India. He has six years of experience in the WebSphere MQ field. He holds a degree in Computer Science and Engineering from Visvesvaraya Technological University. His areas of expertise include development and support of MQTT technologies such as micro broker, MQTT clients and WebSphere MQ Classes for Java/JMS. He has written extensively on messaging.

https://www.ibm.com/developerworks/mydeveloperworks/blogs/messaging/?lang=en

Stefan Fassmann is an Open Group certified IT Architect from IBM Software Group's Lab Services in Boeblingen, Germany. With his degree in Electrical Engineering, he started working for IBM in 1996. For the last 10 years, he has been engaged in various mobile and embedded solution customer projects using IBM WebSphere and Lotus software technology. Besides his engagements in retail and E&U industry projects, Stefan works as development lead for IBM Lotus Expeditor integrator.

Additional contributors

This project was led by:

Martin Keen is a Redbooks Project Leader in the Raleigh Center. He primarily leads projects about WebSphere products and service-oriented architecture (SOA). Before joining the ITSO, Martin worked in the EMEA WebSphere Lab Services team in Hursley, U.K. He holds a Bachelor's degree in Computer Studies from Southampton Institute of Higher Education.

Dave Locke is a Software Engineer who works in the IBM Hursley development lab in the United Kingdom. He has worked for IBM for 25 years starting as a developer on the mainframe-based CICS transaction processing system, working his way down through a variety of projects on mid-range and desktop, and now focuses on small footprint pervasive systems. His main role is on messaging protocols and technologies that help connect the physical world of sensors, actuators and controllers to the more traditional digital world.

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

- Andy Piper
- T Rob
- Graham Churchill
- Matthew R Ganis
- Dougie G Lawson
- Paul J Lacy
- James Caffrey
- Andrew Banks
- Gualberto Ferrer
- Nicholas O'Leary

Thanks to the following people for supporting this project:

- Shawn Tooley, IBM Redbooks Technical Writer
- Debbie Willmschen, IBM Redbooks Technical Writer
- Alfred Schwab, IBM Redbooks Technical Editor

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

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

Stay connected to IBM Redbooks publications

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

www.ibm.com/software/support/acceleratedvalue
IBM
IBM Software Accelerated Value Program.....
Your bridge to everything IBM
Contact us at:
software_accelerated_value@us.ibm.com
WebSphere. software

1

Overview of MQTT

In this chapter we present an introduction to the MQ Telemetry Transport (MQTT) protocol and how it can be used to tie together the various types of remote smart devices and applications that are measuring, monitoring, and in some cases controlling the world today. We also introduce WebSphere MQ Telemetry, the IBM implementation of the open source MQTT protocol, which provides the software components (messaging server, MQTT clients, and so on) needed to create a complete MQTT-based messaging system.

This chapter includes the following topics:

- Building a Smarter Planet
- MQTT and WebSphere MQ Telemetry
- Benefits of using MQTT
- Where to use MQTT

1.1 Building a Smarter Planet

The emerging concept of an *Internet of Things* is a critical foundation on which the vision for a Smarter Planet will be realized. In addition, supporting the Internet of Things are new, more advanced approaches to telemetry that make it possible to connect all kinds of devices, wherever they might be, to each other, to the Internet, and to the business enterprise.

One of these advancements is the MQTT messaging protocol. It is so lightweight that it can be supported by some of the smallest measuring and monitoring devices, and transmit data over far-flung, sometimes intermittent networks. It is also open source, which makes it easy to adapt to a variety of messaging and communication needs.

Before getting into the details of MQTT, it is wise to take a brief look at the evolving world that developers using MQTT are working to connect.

1.1.1 Internet of Things

Anyone who has pointed a web browser at a search engine or social media site knows the power of the Internet to connect people to information or to other people. Yet, with the rise of various smart devices, the Internet will evolve to include what some are calling an *Internet of Things* (Figure 1-1): Billions of interconnected smart devices measuring, moving, and acting upon, sometimes independently, all the bits of data that make up daily life.

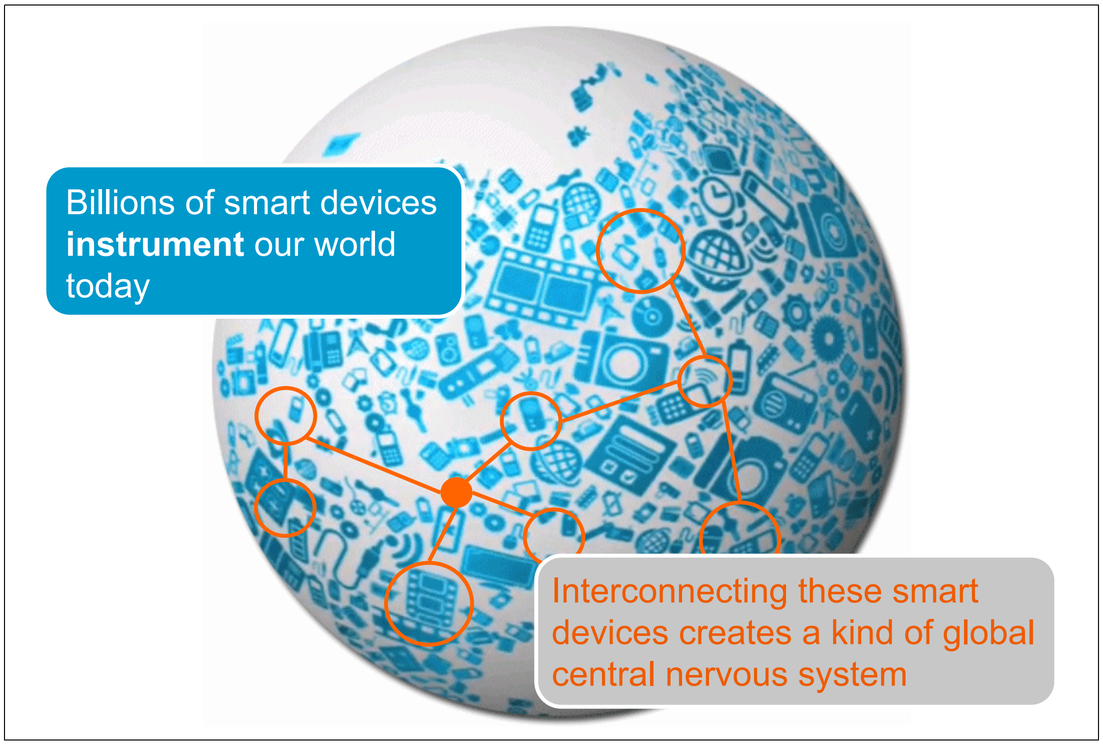

Billions of smart devices **instrument** our world today
Interconnecting these smart devices creates a kind of global central nervous system

Figure 1-1 The Internet of Things

To imagine what the Internet of Things might bring in 10 or 20 years, think about the remarkable things we already can do remotely:

- A doctor can examine a patient in a distant city and see real-time health status information, such as blood pressure, heart rate, and so on.
- An energy company can monitor oil or gas pipelines that are hundreds of miles long and remotely cut off the flow if problems are detected.
- A homeowner can view his house on a web page, complete with the status of interior devices such as the security alarm, heating system, and more.

The Internet of Things will go beyond even these examples, not just people interacting with devices but devices interacting with each other, creating what might eventually become something of a global central nervous system.

1.1.2 Smarter Planet

The IBM concept of a Smarter Planet is built on the following set of pillars called the *Three Is*, as illustrated in Figure 1-2:

- Instrumented: Information is captured wherever it exists, such as through the use of remote sensors.
- Interconnected: Information is moved from the collection point to wherever it can be usefully consumed.
- Intelligent: Information is processed, analyzed, and acted upon to derive maximum value and knowledge.

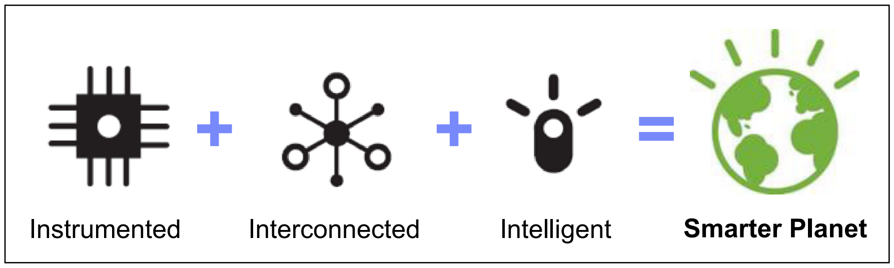


Figure 1-2 The three pillars of the Smarter Planet

The world is already increasingly instrumented, with examples ranging from tiny sensors and RFID tags in stand-alone products, through smartphones and location-aware GPS devices to notebook PCs and embedded systems. These devices typically have enough computing power to gather and transmit data, and some have enough to respond to requests to modify their behavior.

These devices also are nearly all connected to some extent. Most have, or will have, an Internet address of their own, with which they can communicate directly across local networks or indirectly by way of clouds. So the concept of the Internet of Things is already starting to emerge.

The next steps, then, are gathering all of the data that is collected by these small, medium, or even large devices, routing that data to where it is best interpreted, and using the world's vast computational resources to understand what is happening and respond as necessary to make life better.

1.1.3 Telemetry and the Internet

Telemetry technology allows things to be measured or monitored from a distance. In addition, today, improvements in telemetry technology make it possible to interconnect measuring and monitoring devices at different locations and to reduce the cost of building applications that can run on these smart devices to make them even more useful.

People, businesses, and governments are increasingly turning to smart devices and telemetry to interact more smartly with the world. A man shopping for groceries wants to know what is currently in his pantry at home. A woman heading to Barcelona wants to know if flights into that city are currently delayed by weather. A motorist driving across town wants to know if the main highway to get there is still blocked by the car crash that was reported on the morning news. A doctor with a patient due to arrive in the office at 3 p.m. wants to know, in the morning, whether the patient's blood pressure is stable enough to make the trip safely.

Information to help with each of these decisions can come, or could someday come, from a variety of smart meters and devices.

Yet challenges lie in getting the information from the devices to the people and applications that want to use it, in time for them to use it effectively and, ideally, with the added ability for them to reply to the devices with new instructions or requests. If the devices are widely distributed geographically, or if they have limited storage or computational abilities, the challenges increase considerably, as do costs.

Fortunately, these challenges are being overcome through the use of improved telemetry technologies and communication protocols that are making it possible to send and receive this information reliably over the Internet, even if the network is unsteady or the monitoring device has little processing power.

MQTT provides telemetry technology to meet the information challenges of today's Internet users.

1.2 MQTT and WebSphere MQ Telemetry

IBM WebSphere MQ has long served as a reliable, universal messaging backbone enabling any-to-any connectivity. It runs on wide variety of platforms, has a number of language bindings, and a stable, backward-compatible API. It has become the accepted method of gluing disparate applications together.

The piece that has been missing until recently is the ability to reliably connect the edges, the frontiers of the data network. Systems already exist that understand what actions to take based on the status of remote devices. However, communicating that status to the system has been a challenge, particularly if the network is constrained or if the device lacks the computational power required for traditional messaging.

With MQTT, smart energy meters, industrial control systems, satellite receivers, healthcare monitoring devices, and sensors on everything from planes to trains to automobiles can communicate with each other and with other systems or applications.

1.2.1 MQ Telemetry Transport

MQTT is an extremely simple and lightweight messaging protocol. Its publish/subscribe architecture is designed to be open and easy to implement, with up to thousands of remote clients capable of being supported by a single server. These characteristics make MQTT ideal for use in constrained environments where network bandwidth is low or where there is high latency and with remote devices that might have limited processing capabilities and memory.

The MQTT protocol includes the following benefits:

- Extends connectivity beyond enterprise boundaries to smart devices.
- Offers connectivity options optimized for sensors and remote devices.
- Delivers relevant data to any intelligent, decision-making asset that can use it.
- Enables massive scalability of deployment and management of solutions.

MQTT minimizes network bandwidth and device resource requirements while attempting to ensure reliability and delivery. This approach makes the MQTT protocol particularly well-suited for connecting machine to machine (M2M), which is a critical aspect of the emerging concept of an Internet of Things.

The MQTT protocol includes the following highlights:

- Open and royalty-free for easy adoption

  MQTT is open to make it easy to adopt and adapt for the wide variety of devices, platforms, and operating systems that are used at the edge of a network.

- A publish/subscribe messaging model that facilitates one-to-many distribution

  Sending applications or devices do not need to know anything about the receiver, not even its address.

- Ideal for constrained networks (low bandwidth, high latency, data limits, and fragile connections)

  MQTT message headers are kept as small as possible. The fixed header is just two bytes, and its on demand, push-style message distribution keeps network utilization low.

- Multiple service levels allow flexibility in handling different types of messages

  Developers can designate that messages will be delivered *at most once*, *at least once*, or *exactly once*.

- Designed specifically for remote devices with little memory or processing power

  Minimal headers, a small client footprint, and limited reliance on libraries make MQTT ideal for constrained devices.

- Easy to use and implement with a simple set of command messages

  Many applications of MQTT can be accomplished using just CONNECT, PUBLISH, SUBSCRIBE, and DISCONNECT.

- Built-in support for loss of contact between client and server

  The server is informed when a client connection breaks abnormally, allowing the message to be re-sent or preserved for later delivery.

1.2.2 WebSphere MQ Telemetry

WebSphere MQ Telemetry is a feature of WebSphere MQ that extends the universal messaging backbone with the MQTT protocol to a wide range of remote sensors, actuators and telemetry devices. It enables instrumented devices that are located virtually anywhere in the world to connect to each other, and to enterprise applications and web services, with WebSphere MQ. The use of MQTT extends WebSphere MQ to remote devices and enables massive scalability. A WebSphere MQ Server can handle up to 100,000 concurrent MQTT connections.

WebSphere MQ Telemetry includes the following key components:

- The MQ Telemetry service that runs on the WebSphere MQ server
- MQ Telemetry clients that are distributed to remote devices and applications

MQ Telemetry uses the MQTT protocol to send and receive messages between devices or applications and the WebSphere MQ queue manager. From the WebSphere MQ queue manager, the messages can be exchanged with other messaging applications, such as similar telemetry applications, MQI, JMS, or enterprise messaging applications.

MQ Telemetry version note: MQ Telemetry was added as an installable feature of WebSphere MQ 7.0.1 before being fully integrated into WebSphere MQ V7.1.

1.2.3 Basic concepts of MQTT

The MQTT protocol is built upon several basic concepts, all aimed at assuring message delivery while keeping the messages themselves as lightweight as possible.

Publish/subscribe

The MQTT protocol is based on the principle of publishing messages and subscribing to topics, which is typically referred to as a *publish/subscribe* model. Clients can subscribe to topics that pertain to them and thereby receive whatever messages are published to those topics. Alternatively, clients can publish messages to topics, thus making them available to all subscribers to those topics.

Topics and subscriptions

Messages in MQTT are published to *topics*, which can be thought of as *subject areas*. Clients, in turn, sign up to receive particular messages by subscribing to a topic. *Subscriptions* can be explicit, which limits the messages that are received to the specific topic at hand or can use wildcard designators, such as a number sign (#) to receive messages for a variety of related topics.

Quality of service levels

MQTT defines three *quality of service* (QoS) levels for message delivery, with each level designating a higher level of effort by the server to ensure that the message gets delivered. Higher QoS levels ensure more reliable message delivery but might consume more network bandwidth or subject the message to delays due to issues such as latency.

Retained messages

With MQTT, the server keeps the message even after sending it to all current subscribers. If a new subscription is submitted for the same topic, any *retained messages* are then sent to the new subscribing client.

Clean sessions and durable connections

When an MQTT client connects to the server, it sets the *clean session flag*. If the flag is set to true, all of the client's subscriptions are removed when it disconnects from the server. If the flag is set to false, the connection is treated as durable, and the client's subscriptions remain in effect after any disconnection. In this event, subsequent messages that arrive carrying a high QoS designation are stored for delivery after the connection is reestablished. Using the clean session flag is optional.

Wills

When a client connects to a server, it can inform the server that it has a *will*, or a message, that should be published to a specific topic or topics in the event of an unexpected disconnection. A will is particularly useful in alarm or security settings where system managers must know immediately when a remote sensor has lost contact with the network.

1.2.4 Comparison between MQTT and HTTP

Although comparison is often made between MQTT and other common protocols, the most useful comparison is with HTTP, for the following reasons:

- HTTP is the most widely used and available protocol. Almost all computing devices with a TCP/IP stack have it. In addition, because HTTP and MQTT are both based on TCP/IP, developers need to choose between them.
- The HTTP protocol uses a request/response model, which is currently the most common message exchange protocol. MQTT uses a publish/subscribe pattern. Developers need to understand the relative advantages of each type of model.

Quick comparison

Table 1-1 provides a quick comparison to help developers choose the most suitable messaging protocol for applications.

Table 1-1 Quick comparison of MQTT and HTTP

	MQTT	**HTTP**
Design orientation	Data centric	Document centric
Pattern	Publish/subscribe	Request/response
Complexity	Simple	More complex
Message size	Small, with a compact binary header just two bytes in size	Larger, partly because status detail is text-based
Service levels	Three quality of service settings	All messages get the same level of service
Extra libraries	Libraries for C (30 KB) and Java (100 KB)	Depends on the application (JSON, XML), but typically not small
Data distribution	Supports 1 to zero, 1 to 1, and 1 to n	1 to 1 only

Detailed comparison

Here is a fuller explanation of the critical differences between the MQTT and HTTP protocols for devices:

- Design orientation
  - MQTT is data-centric. It transfers data content as byte array. It does not care about content.
  - HTTP is document-centric. It supports the MIME standard to define content type, but constrained devices usually do not need this advanced feature.
- Messaging pattern
  - MQTT uses a publish/subscribe messaging pattern that has loose coupling. Clients do not need to be aware of the existence of other devices. They just need to care about the content to be delivered or received.

  - HTTP uses a request/response messaging model. It is a basic and powerful messaging exchange pattern, but the client needs to know the address of all devices to which it connects.
- Complexity of protocol
  - The MQTT specification is short. It has few message types and only the CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, and DISCONNECT types are important for developers.
  - HTTP is a more complex protocol, with a specification that is more than 160 pages long. It uses many return codes and methods (such as POST, PUT, GET, DELETE, HEAD, TRACE, CONNECT, and so on). It works well for hypermedia information systems, but constrained devices typically do not need all of its features.
- Message size
  - MQTT is designed specifically for constrained devices. It includes only the features that are necessary to support them. The message header in MQTT is short, and the smallest packet size for a message is 2 bytes.
  - HTTP uses a text format, not a binary format, which allows for lengthy headers and messages. The text format is readable by humans, which makes the HTTP protocol easy to troubleshoot and has contributed to its popularity. However, this format is more than is needed, or desirable, for constrained devices with limited computational resources in low-bandwidth network environments.
- Quality of Service levels
  - MQTT supports three QoS levels in message publication. Developers do not have to implement additional, complex features to ensure message delivery.
  - HTTP has no retry ability or QoS features. If developers need guaranteed message delivery, they have to implement it themselves.
- Extra libraries
  - MQTT works well on devices with limited memory due, in part, to its small library requirement, only about 30 KB for the C client and 100 KB for the Java client.
  - HTTP does not require any libraries by itself, but additional libraries of parsers for JSON or XML are required if using SOAP- or RESTful-style web services.
- Data distribution
  - MQTT includes a built-in distribution mechanism, supporting the 1 to 0, 1 to 1, and 1 to many distribution models.
  - HTTP is point-to-point and has no built-in distribution feature. Developers must create their own distribution mechanism or adapt common techniques, such as long-polling or Comet.

1.2.5 MQTT and Eclipse Paho

With the rapid expansion of sensors, monitors, and other forms of remote instrumentation, the need to integrate these smart devices into enterprise and web-based systems becomes more important. To date, the typical integration approach has been to combine, to the extent possible, industry-standard communications models with whatever private or non-standard messaging protocols might be resident on the devices or already in use within the enterprise.

However, such integrations often are based on point-to-point standards that require tight coupling between the sending and receiving devices or applications, and the limitations of protocols, such as SOAP or HTTP, make programming requirements quite strict. Additional challenges lie in implementing solutions for constrained devices with minimal storage and calculating capabilities or on unstable, low-performing data networks.

Open source messaging is needed that can exchange data and events between devices in the physical world and the virtual world of the enterprise or web-based systems. It should support the open messaging models that are increasingly prevalent and also cater to the special needs of constrained devices and networks. Open source messaging enables a shift from strict, 1-to-1-style existing protocols to more loosely coupled models and bridges the new middleware architectures with the embedded and wireless device architectures that are associated with machine-to-machine (M2M) communication. See Figure 1-3.

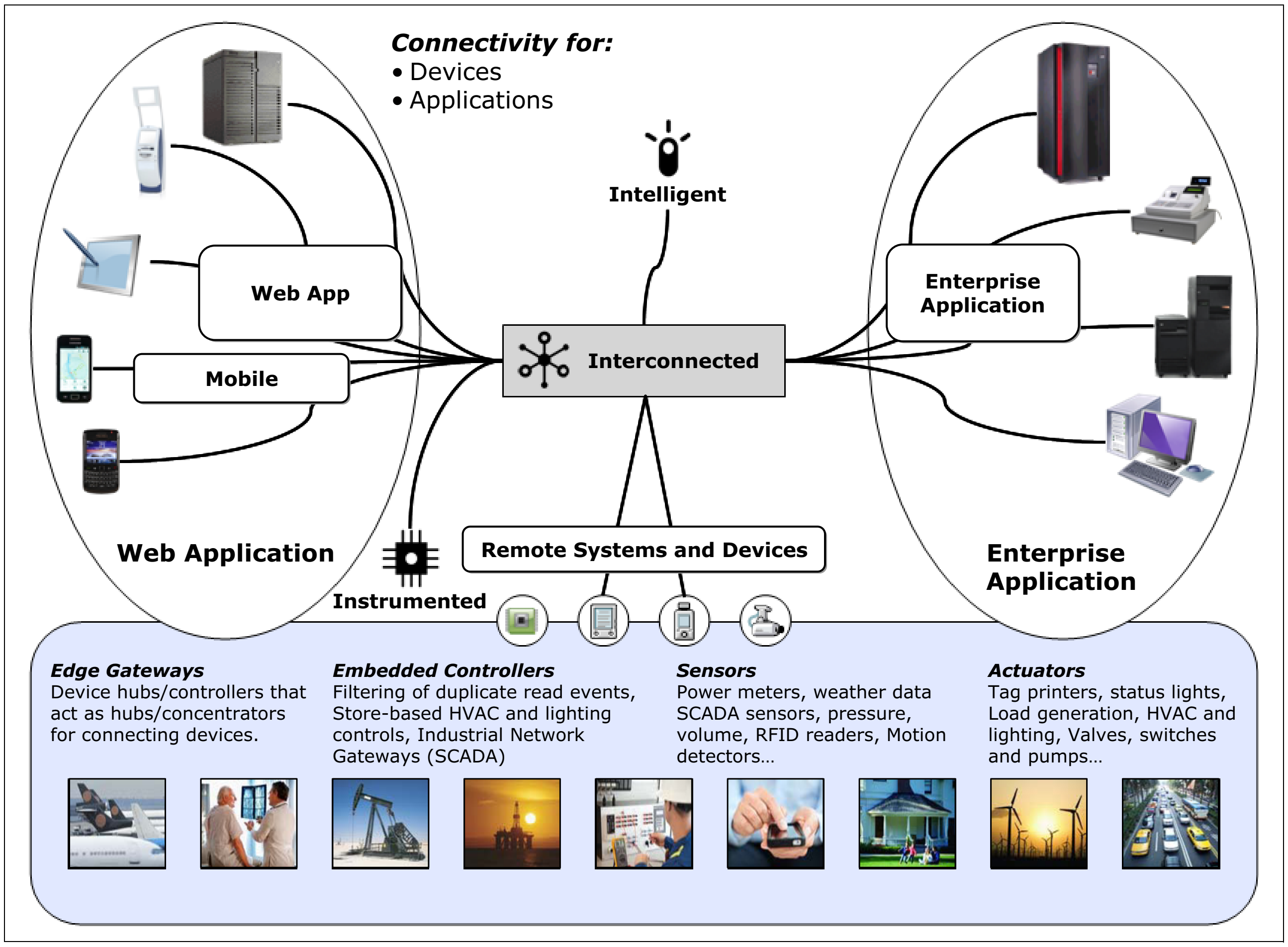

Connectivity for:
• Devices
• Applications
Intelligent
Web App
Mobile
Interconnected
Enterprise Application
Web Application
Instrumented
Remote Systems and Devices
Enterprise Application
Edge Gateways
Device hubs/controllers that act as hubs/concentrators for connecting devices.
Embedded Controllers
Filtering of duplicate read events, Store-based HVAC and lighting controls, Industrial Network Gateways (SCADA)
Sensors
Power meters, weather data SCADA sensors, pressure, volume, RFID readers, Motion detectors…
Actuators
Tag printers, status lights, Load generation, HVAC and lighting, Valves, switches and pumps…

Figure 1-3 Machine-to-machine connectivity

IBM and Eurotech joined Sierra Wireless and the Eclipse Foundation to provide open source tools and protocols to the Eclipse Paho project to simplify development of M2M solutions. The Eclipse Paho project is aimed at developing open source, scalable, and standard messaging protocols of the type needed for improved M2M communication and integration with web, enterprise middleware, and applications. It includes the following major goals:

- Bidirectional messaging
- Determinable delivery of messages
- Loose coupling
- Constrained-platform usability

The scope of the Eclipse Paho project includes client software for use on remote devices along with corresponding server support. The Eclipse Paho project focuses on the framework, best practice samples, and plug-in tools for developers to integrate and test the end-to-end connectivity of messaging components.

IBM contributed Java and C client-side code implementations of the MQTT protocol, and Eurotech contributed the framework implementation and sample applications for developers to use when integrating and testing Paho messaging components. The Java MQTT client libraries run on many variations of Java, including Connected Device Configuration (CDC)/Foundation, Java Platform, Standard Edition (J2SE), and Java Platform, Enterprise Edition (J2EE). The C reference implementation, together with prebuilt native clients for the Windows and Linux operating systems, enables MQTT to be ported to a wide range of devices and platforms.

The Eclipse tools facilitate designing and developing connectivity solutions between devices and applications, thereby enabling and encouraging more innovative M2M integration. Users can write their own API to interface with the MQTT protocol in their preferred programming language on their preferred platform.

To simplify writing MQTT client applications, developers can use the WebSphere MQ Telemetry client libraries and the development software development kit (SDK). The client libraries and the development SDK can be imported into a development environment (for example, WebSphere Eclipse Platform). After relevant applications are developed, the applications and client libraries can then be deployed together to the appropriate system. The SDK includes the WebSphere MQ Telemetry C and Java client libraries that encapsulate the MQTT V3 protocol for a number of platforms.

You can find more information about the Eclipse Paho project at:

http://www.eclipse.org/paho/

Information is also available about the MQTT V3 Java and C clients that IBM contributed to the project, which you can download from the following web pages:

http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.java.git/
http://git.eclipse.org/c/paho/org.eclipse.paho.mqtt.c.git/

1.2.6 MQTT and open source

As an open source protocol, MQTT is an important component in numerous connectivity solutions. Table 1-2 shows the various clients that are provided with servers and brokers.

Table 1-2 MQTT brokers and the clients that come with them

Server/Broker	Client or clients provided	Notes
IBM WebSphere MQ Telemetry	► C client API ► Java client API ► Simple Asynchronous Messaging (SAM) ► WebSphere MQTT client (Perl)	WebSphere MQ Telemetry is a component of WebSphere MQ V7.1 and later. It was included in WebSphere MQ V7.0.1 as an additional installation.
IBM WebSphere Message Broker	► C client (part of SupportPac IA93) ► Java client (part of SupportPac IA92) ► WebSphere-MQTT client (Perl)	The Supervisory Control and Data Acquisition (SCADA) input and output nodes in WebSphere Message Broker V6.0 are replaced by JMS input and output in V7.0.
IBM Lotus Expeditor micro broker	► Java client (including JMS) ► C client ► C# client	Lotus Expeditor micro broker is a small Java implementation of an MQTT server.

Server/Broker	Client or clients provided	Notes
Really Small Message Broker	► C client API	Really Small Message Broker is a small server that uses MQTT (V3 and V3.1).
Mosquitto	► C client library (libmosquitto) ► C++ client library (libmosquittocpp) ► Python client module (python-mosquitto)	Mosquitto is an open source MQTT server that supports MQTT version 3.1.
Eurotech Everyware Device Cloud (EDC)	► Java client ► C client ► C# client ► Other options available	EDC is a cloud-based service that supports MQTT as a communication protocol.
MQTT.js	► JavaScript client API used for client creation	This script is an MQTT server with server/client creation API written in JavaScript.
m2m.io	► C client API ► C# client API ► Java client API ► Ruby client API	This API provides a cloud messaging service.
mqtt4erl	► Erlang client	This client provides an open-source broker/client that supports MQTT.
ELWIX	► C library (libaitmqtt)	A UNIX variant broker that uses MQTT for communication.

Table 1-3 lists common programming languages and the compatible client libraries that are available.

Table 1-3 Programming languages and compatible MQTT client libraries

Language	Compatible client or clients	Notes
C	► liblwmqtt	A lightweight C client.
Delphi	► TMQTTClient	The library is a Delphi client library for MQTT.
Erlang	► erlmqtt ► my-mqtt4erl	erlmqtt is a library for MQTT. my-mqtt4erl is a more recent fork of mqtt4erl
Java	► MeQanTT ► mqtt-client	Mqtt-client is a Fusesource Java MQTT client with a variety of API styles.
JavaScript	► Node.js MQTT Client	A simple MQTT client that runs on node.js.
IBM LotusScript®	► LotusScript Library	Wraps other classes into LotusScript classes for MQTT calls.
Lua	► Lua MQTT Client Library	Implements MQTT V3.1.

Language	Compatible client or clients	Notes
.NET	► MQTTDotNet ► nMQTT	MQTTDotNet is a client implementation of MQTT version 3. nMQTT is an implementation of MQTT v3 for the .Net and Mono platforms.
Perl	► net-mqtt-perl ► anyevent-mqtt-perl	net-mqtt-perl is a Perl module to represent MQTT messages. AnyEvent-mqtt-perl is an AnyEvent module for an MQTT client.
Python	► MQTT-For-Twisted-Python	A Python MQTT protocol client.
REXX	► REXXMQTT	A Regina REXX with an add-on Rxsock library for MQTT clients.
Ruby	► ruby-mqtt ► ruby-em-mqtt	ruby-mqtt is an MQTT client implemented in Ruby gem. ruby-em-mqtt provides MQTT support to EventMachine
Device-specific	► Arduino client for MQTT ► mbed client for MQTT ► Nanode MQTT ► Netduino MQTT	Arduino client for MQTT supports MQTT V3 mbed client for MQTT is a simple MQTT client that runs on the mbed platform Nanode MQTT is an MQTT implementation for Nanode. Netduino MQTT is a simple MQTT client used by Netduino.

1.3 Benefits of using MQTT

Using the MQTT protocol extends WebSphere MQ to tiny sensors and other remote telemetry devices that might otherwise be unable to communicate with a central system or that might be reached only through the use of expensive, dedicated networks. Network limitations can include limited bandwidth, high latency, volume restrictions, fragile connections, or prohibitive costs. Device issues can include limited memory or processing capabilities, or restrictions on the use of third-party communication software. In addition, some devices are battery-powered, which puts additional restrictions on their use for telemetry messaging.

MQTT was designed to overcome these limitations and issues and includes the following underlying principles:

- Simplicity: The protocol was made open so that it can be integrated easily into other solutions.
- Use of a publish/subscribe model: The sender and the receiver are decoupled. Thus, publishers do not need to know who or what is subscribing to messages and vice versa.
- Minimal maintenance: Features, such as automated message storage and retransmission, minimize the need for on-the-fly administration.

- Limited on-the-wire footprint: The protocol keeps data overhead to a minimum on every message.
- Continuous session awareness: By being aware of when sessions have terminated, the protocol can take action accordingly, thanks in part to a *will* feature.
- Local message processing: The protocol assumes that remote devices have limited processing capabilities.
- Message persistence: Through the designation of specific QoS, the publisher can ensure delivery of the most important messages.
- Agnostic regarding data types: The protocol does not require that the content of messages be in any particular format.

Table 1-4 lists potential scenarios where WebSphere MQ and the MQTT protocol might be used to improve communication to and from remote devices or applications.

Table 1-4 Scenarios where WebSphere MQ and MQTT can be used

Scenario	Key industries	Examples
Automated metering	► Chemical and Petroleum ► Energy and Utilities	► A homeowner uses smart metering to obtain a more accurate view of how each household appliance consumes electricity and to make changes. ► A utility company remotely monitors water heaters in customers' homes and turns off the water heaters at times when they are not typically used.
Distribution supply chain and logistics	► Retailers ► Distributors ► Consumer products ► Transportation	► A shipping company gains customer loyalty by providing up-to-the-moment, detailed tracking information for cargo. ► A trucking company cuts costs using remote fleet monitoring, which enables more efficient use of each truck's capacity on every run.
Industrial tracking and visibility	► Automotive ► Industrial manufacturing ► Aerospace ► Defense	► A manufacturing company automates inventory checking to improve management of stock and to optimize production rates. ► An automobile company uses RFID tracking to obtain up-to-the-minute details about the current stage of assembly of each new vehicle as it moves through the assembly line.

Scenario	Key industries	Examples
Healthcare quality and resource tracking	► Pharmaceutical companies ► Medical research ► Hospitals ► Nursing Homes	► A medical clinic remotely tracks the vital signs of at-risk patients to help prevent sudden crises that might arise after a patient goes home. ► A research team monitors chemical reactions in a remote laboratory and alerts the chemist when a particular result or stage of development is achieved.
Location awareness and safety	► Chemical and Petroleum ► Energy and Utilities ► Homeland Defense	► A gas company improves pipeline monitoring by tripling the number of remote control devices along the route from 4,000 to 12,000. ► A government agency improves its early-warning system by placing remote sensors on dams and elsewhere in flood-prone regions.
Executive alerts	► Insurance ► Banking	► A bank retains more customers by monitoring all account-closing inquiries and having a manager call customers who are thinking of leaving. ► An insurance company sends claims adjusters to collect damage reports at disaster sites and collects the data in its central servers.

1.4 Where to use MQTT

The MQTT messaging protocol is designed for devices in constrained environments, such as embedded systems with limited processing ability and memory or systems that are connected to unreliable networks. It provides the robust messaging features that are needed to communicate with remote systems and devices while consuming just a small portion of network bandwidth.

1.4.1 Example implementations of MQTT

Before discussing, generally, the kinds of communication that MQTT can make more efficient, here is a brief look at scenarios where MQTT is already being used successfully.

Healthcare

A medical organization wanted to create an at-home, cardiac pacemaker monitoring solution. The solution needed to address the following aspects of patient care:

- Monitoring cardiac patients after they leave the hospital
- Improving the efficiency of later checkups
- Meeting new industry data-capture standards

The company worked with IBM to create a solution (illustrated in Figure 1-4) in which an MQTT client is embedded in a home monitoring appliance that collects diagnostics whenever the patient is in close proximity to a base unit. The base unit sends the diagnostic data over the Internet to the central messaging server, where it is handed off to an application that analyzes the readings and alerts the medical staff if there are signs the patient might be having difficulty.

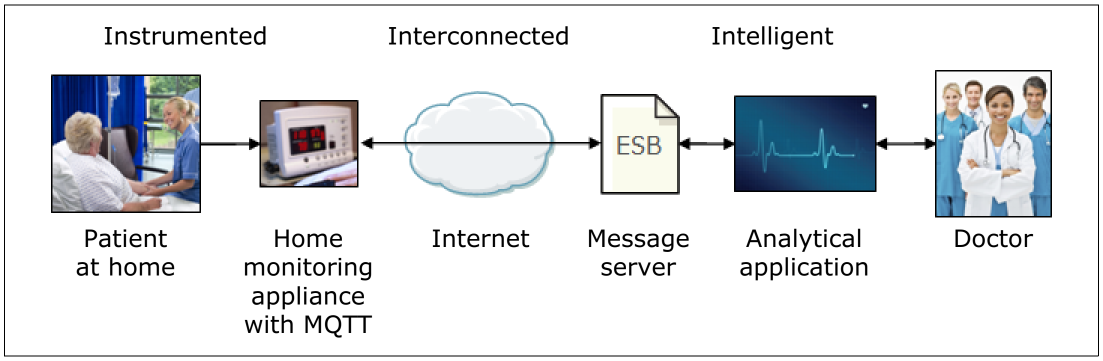


Figure 1-4 Home pacemaker monitoring solution with MQTT

The solution allows the organization to provide a higher level of post-hospital patient care and early diagnosis of followup issues. It also saves money for both the organization and its patients, because there is less need for travel by either party and because patients who are doing well might be allowed to come in for checkups less often.

Energy and utilities

A utility company was faced with both rising costs to produce electricity and increasing demand for power from its customer base, which was unable, generally, to pay ever-increasing rates. So rather than immediately passing on production costs that its customers likely could not pay, the company first sought a solution to reduce overall demand for electricity by placing smart meters in customers' homes to remotely control the use of certain power-consuming devices. However, the solution needed to minimize use of the available data network, for which the company paid according to the volume of data transmitted.

The solution was to create a *virtual power plant* (VPP) that sits between the company's generating sources and its customers. In-home smart meters collect usage data for the various appliances that are used there. Then, home gateway monitors, equipped with an advanced MQTT client, publish the usage data to the VPP at regular intervals over the local mobile telephone network.

As illustrated in Figure 1-5 on page 16, the VPP monitors energy consumption in real time, predicts upcoming consumption needs, and when necessary lowers overall demand by taking control of electricity-using devices in customers' homes. When instructions are sent to electricity-using devices in a home, the commands are pushed to the home gateway box using MQTT.

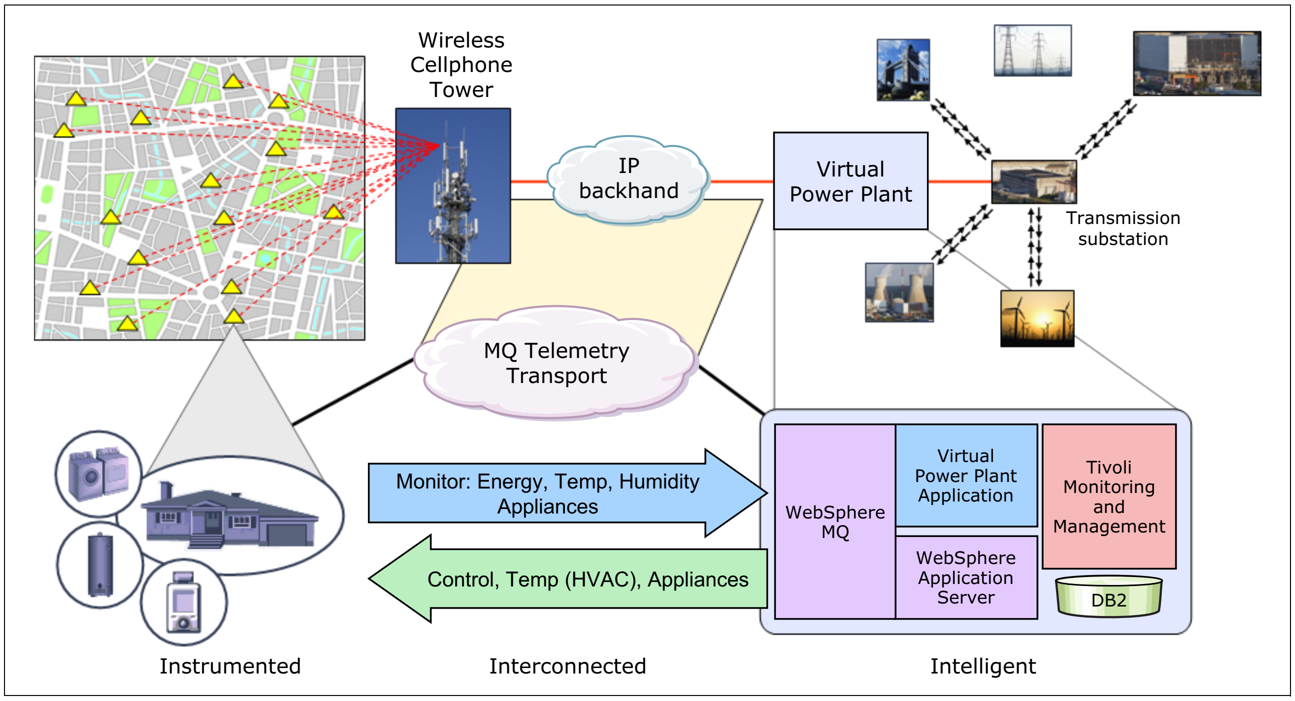

Wireless Cellphone Tower
IP backhand
Virtual Power Plant
Transmission substation
MQ Telemetry Transport
Monitor: Energy, Temp, Humidity Appliances
Control, Temp (HVAC), Appliances
WebSphere MQ
Virtual Power Plant Application
WebSphere Application Server
Tivoli Monitoring and Management
DB2
Instrumented
Interconnected
Intelligent

Figure 1-5 The virtual power plant using MQTT

Social networking

A social networking company experienced latency problems when sending messages. The method the company used to send messages was reliable but slow, and solutions were limited if it continued to use the same or similar technology. A new mechanism was needed that could maintain a persistent connection with the messaging servers without consuming too much battery power, which is critical to users of the company's social networking site because so many of them use the service on battery-powered mobile devices.

The company's developers solved the problem using the MQTT protocol. By maintaining an MQTT connection and routing messages through its chat pipeline, the company was able to achieve message delivery at speeds of just a few hundred milliseconds, rather than multiple seconds.

Other areas where MQTT has been used

MQTT is also applicable across the following additional industries and sectors:

- Point-of-sale solutions that send sales information and receive price updates over slow networks
- Gambling devices, such as slot machines, that are distributed around a casino but must regularly communicate with a central server
- Preventive maintenance of automobiles by means of automated collection of real time diagnostic information from critical vehicle components
- Environmental monitoring, such as through the use of remote sensors along sensitive areas such as riverbanks
- Healthcare and safety solutions that rely on immediate analysis of real-time data that is collected from patients in various locations

1.4.2 Machine-to-machine

When most people think of the Internet, they think of individuals using a web browser to gather information from search engines, to communicate with others using social media, or to connect to viewing devices, such as web cameras. Yet as technology evolves, it is increasingly common for devices to be connected to *each other*. This type of connection has created a need for efficient, machine-to-machine (M2M) communication protocols.

The MQTT protocol is ideal for use in M2M communication. It enables connectivity that extends beyond smart devices to some of the smallest remote devices and sensors, including devices with limited processing or network abilities. This extension makes MQTT a critical component in self-managing M2M networks and a key part of realizing the Smarter Planet vision. In addition, because MQTT is highly scalable, it is possible to create systems that involve hundreds or even thousands of remote sensors or devices.

Figure 1-6 illustrates how the MQTT protocol can improve M2M communication.

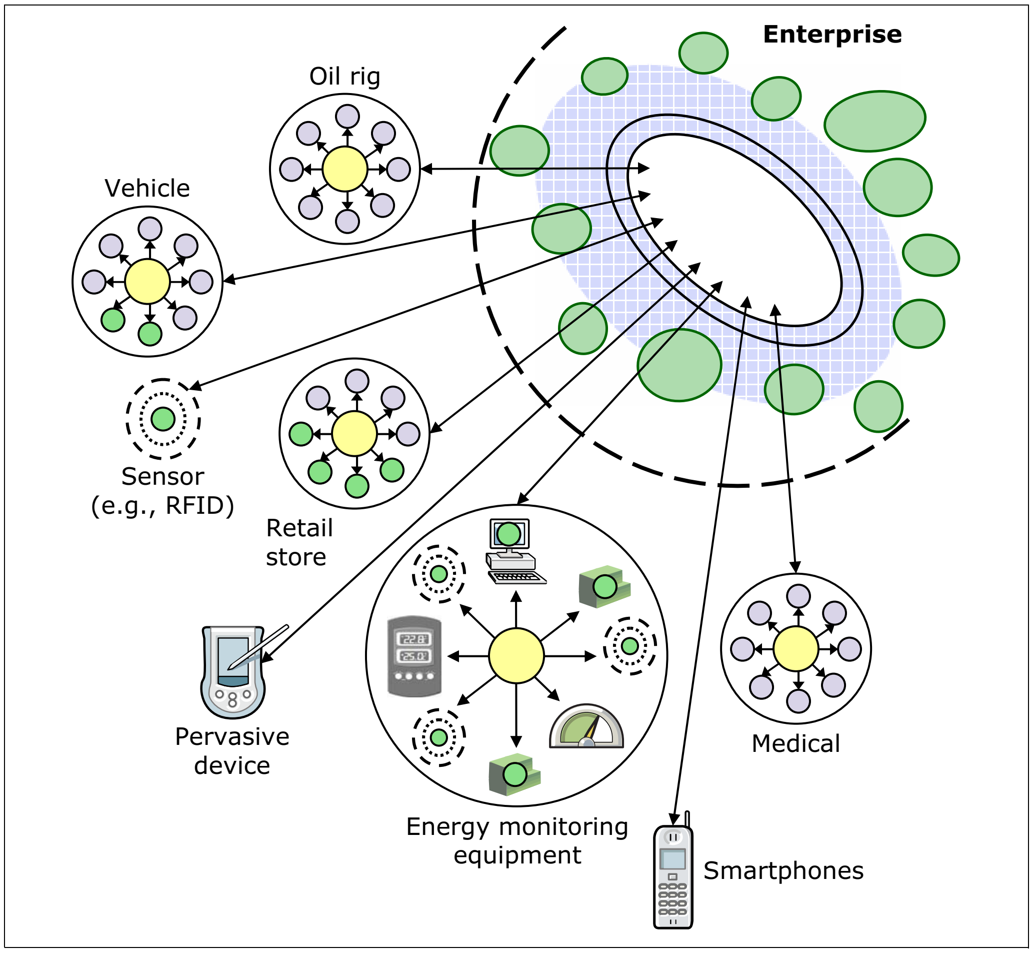


Figure 1-6 Example of M2M communication enabled or improved by the MQTT protocol

The MQTT for Sensors (MQTT-S) protocol enables the inclusion of machines that typically would not be able to use MQTT due to a lack of TCP/IP network abilities. MQTT-S extends the MQTT protocol to low-cost, battery-operated sensor and actuator devices existing on non-TCP/IP networks. It can function on any network that allows bidirectional data transfer.

MQTT-S is ideal for wireless sensor networks (WSN) of spatially distributed autonomous sensors. WSNs are attractive due to their simplicity, low cost and ease of deployment.

MQTT-S clients connect to a gateway that performs the protocol translation between MQTT and MSTT-S. The MQTT-S clients are often used to monitor physical or environmental conditions such as temperature, sound, vibration, pressure, or motion.

1.4.3 MQTT and sensors

The MQTT protocol has special applicability when used with remote sensing devices.

Sensors

Sensors measure or otherwise determine, or sense, particular parameters of a device or system and report it in a manner that is understandable to humans or other devices or systems. The simplest examples might be an old-style clinical thermometer, which senses human body heat and reports it with the rising or falling column of mercury, or a pressure gauge that reports the interior air pressure of a tire through the movement of a needle across a scaled dial. Modern sensors can report their status in more advanced ways, such as on digital panels or, with the help of messaging protocols, such as MQTT, by transmitting their information over a data network.

Sensors of a device or system

One sensor can measure a particular parameter on a particular device or system. Or multiple sensors can be positioned to report on several parameters, providing a complete view of the current operating conditions of the device or system. This view can be critical to a supervisor because it allows him or her to determine, for example, if everything is operating within safety limits.

Consider a system for moving pressurized gas at a chemical factory. The supervisor needs to know, among other things, the operating status of the compressors, the rates of flow at inlet and outlet, the temperature of the gas and the pressure of the gas inside the system. So it is critical that all of these factors be measured by properly calibrated sensors that are placed in the correct locations and are capable of reporting their status reliably and in real time.

Sensors of an enterprise

When applied to an entire enterprise, the nature of an interrelated group of sensors grows even more complicated, and more critical. Take the example above of the chemical factory. There might be other compressors or machinery which are interdependent, such as when compressed gas is fed into a reaction chamber. So in addition to the sensors reporting on the pressurized gas system, other sensors are measuring activity within the reaction chamber. Each individual sensor is reporting its particular parameter, but together, they provide an end-to-end status of the whole operation.

Another aspect of an enterprise is the interface between departments. The production department might be primarily interested in the sensors that report on the compressors and reaction chamber, while the shipping department is mostly interested in how much of the product will need to be shipped out to customers. The amount of the product to be shipped on a given day is dependent on the performance of the compressors and reaction chamber. So, the sensors individually are important, and their ability to communicate current readings is vitally important also.

Actuators

Actuators are a special type of device that take an action based on system behavior. Whereas a sensor reports the status of a particular parameter of a system, an actuator can act to influence that parameter or other parts of the system. A simplified analogy is that of file input-output operations. The work of a sensor can be seen as similar to that of a file reading operation; the work of an actuator can be seen as similar to a file writing operation.

Another example is a flow valve attached to a pipeline, which can alter the rate of flow based on certain parameters that are sensed. Yet another actuator might be set to track a set of parameters and then trigger an alarm if certain thresholds are reached.

Transmitting sensor data with MQTT

Further requirements of a sensor, such as transmitting its readings over a network or raising an alert, are generally not in the scope of the sensor itself. These types of requirements fall outside the scope of the sensor because a sensor is typically a small, specialized piece of hardware that is not designed to contain the computational power that is needed for more advanced functions.

Some sensors come equipped with communications abilities, or additional hardware devices can be added to them to enable this capability. However, these options are generally limited because they still cannot provide the computational power that is required for traditional data transmission protocols.

The MQTT publish/subscribe messaging model can help provide the capabilities to transmit sensor data.

2

Getting started with MQTT

In this chapter we introduce the concepts behind the MQTT protocol. We describe a transportation logistics scenario that will be used throughout the book to demonstrate critical messaging and programming concepts, and provide an example of how to build an initial publish/subscribe program using available MQTT clients and brokers.

This chapter contains the following sections:

- Clients and brokers used in this book
- Scenario used in this book
- MQTT concepts
- MQTT brokers
- Building a sample MQTT application with freely available software

2.1 Clients and brokers used in this book

Several different clients and brokers can be used to build the initial publish/subscribe program that is created later in this chapter. Each is relatively easy to use and can be obtained from sources across the Internet. Additional options exist (refer to 1.2.6, “MQTT and open source” on page 10), but the ones listed here are considered to be among the most mature.

2.1.1 MQTT clients

An MQTT client (also called a client application) collects information from a telemetry device, connects to a messaging server, and uses a topic string to publish the information in a way that allows other clients or applications to retrieve it. An MQTT client also can subscribe to topics, receive publications associated with those topics, and issue commands to control the telemetry device.

Client libraries can simplify the process of writing MQTT client applications. For example, WebSphere MQ Telemetry provides C and Java client libraries that can be used to enable the MQTT V3 protocol for use on a number of platforms. When these libraries are incorporated into MQTT applications, a fully functional MQTT client can be created with just a few lines of code.

WebSphere MQ Telemetry clients

IBM WebSphere MQ ships with a set of fully-supported, precompiled MQTT clients. Use one of these if it is available for your platform.

At the time of writing, MQTT clients are supplied with WebSphere MQ for the following target applications:

- Java applications running on a range of Java Standard Edition (JSE) and Java Micro Edition (JME) virtual machines
- C applications running on Linux operating systems with x8664 and ARM hardware
- C applications running on Windows operating systems with x86 and x8664 hardware

These clients contain mature, high performing implementations of the MQTT V3.1 protocol. They can be found in the WebSphere MQ installation directory or they can be downloaded as a stand-alone package of WebSphere MQ V7.1 clients from this website:

http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24031412

Full details of the supported environments for these precompiled clients are available here:

http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg27023057

Eclipse Paho clients

The Eclipse Paho project provides scalable, open-source messaging models for machine-to-machine (M2M) communication, starting with C- and Java-based client implementations of MQTT. These provide good options for the clients that come with WebSphere MQ Telemetry; developers can compile them on operating systems or hardware on which the WebSphere MQ Telemetry clients are not supported.

The Eclipse Paho MQTT clients can be downloaded from:

http://www.eclipse.org/paho/download.php

2.1.2 MQTT brokers

An MQTT broker is a server that implements the MQTT protocol. It mediates communication between MQTT client applications, such as those running in remote sensors and other devices, and the enterprise integration layer.

WebSphere MQ

WebSphere MQ began including built-in support for MQTT, through the WebSphere MQ Telemetry component, starting with versions 7.0.1 and 7.1. The WebSphere MQ Telemetry component is implemented by the WebSphere MQ Extended Reach (MQXR) service. This MQXR service includes a Java-based broker, which enables delivery of MQTT messages by connecting MQTT clients to MQ queue managers.

Refer to Chapter 3, "Using MQTT with IBM WebSphere MQ Telemetry" on page 55 for an in-depth review of using WebSphere MQ as your message broker. An evaluation version of WebSphere MQ can be found at:

http://www.ibm.com/developerworks/downloads/ws/wmq/

Really Small Message Broker and MQ Telemetry daemon for devices

Really Small Message Broker (RSMB) is a small, freely available C implementation of an MQTT broker. It also is an advanced MQTT V3 client application. Developers usually use it as a hub to store and forward messages from MQTT clients to a backend MQTT broker. RSMB, which was first released in 2003, can be downloaded as a stand-alone client here:

https://www14.software.ibm.com/webapp/iwm/web/reg/download.do?source=AW-0U9&S_PKG=0U9

The MQ Telemetry daemon for devices is the latest version of RSMB. It is provided with WebSphere MQ V7.0.1 and V7.1, in the MQ `/mqxr/SDK/advanced/DeviceDaemon` installation directory.

Mosquitto

Mosquitto is a small, no-cost, open source implementation of an MQTT broker that supports the MQTT V3.1 protocol. Mosquitto replicates the functionality of Really Small Message Broker.

For more information about Mosquitto or to download the Mosquitto broker, refer to this website:

http://mosquitto.org/

Micro broker in IBM Lotus Expeditor integrator

Micro broker is a Java-based MQTT broker that IBM is already using in software products such as IBM Lotus Expeditor integrator. It runs on a wide range of standard devices including PDAs, notebook and desktop systems, and more. Micro broker provides multiple ways to interact with messaging system components, including the following:

- A JMS API for accessing queues and establishing messaging communication
- Micro broker clients (buffering and non-buffering) for sensors and devices using MQTT to communicate with the micro broker itself
- Messaging client plug-ins to communicate with other messaging providers (for example, WebSphere MQ)

The IBM Lotus Expeditor integrator software complements an enterprise service bus by acting as a local concentrator and integration hub for transactional messaging at remote locations. Its object is to provide lightweight, reliable and secure data exchange with the central system while minimizing maintenance. An evaluation version of IBM Lotus Expeditor v6.2.2, including the micro broker, can be downloaded from:

http://www.ibm.com/developerworks/downloads/ls/lxpd/

2.2 Scenario used in this book

The sample MQTT-related programs and applications demonstrated in this book are based, whenever possible, on a scenario involving Transportation Company B, a fictitious logistics company that transports materials to, from and among its customers. The need for the company to regularly monitor the location of its vehicles, and the status of the cargo within each vehicle, makes it ideal for illustrating the kinds of solutions that MQTT-based messaging can provide.

2.2.1 Business challenge

Transportation Company B wants to upgrade its truck monitoring system.

Under the current architecture (refer to Figure 2-1), each of the company's trucks uses wireless technology to regularly inform division headquarters about the load it is carrying, its current position (location, speed, available fuel, and so on), and the status of its HVAC systems (heating, ventilation, and air conditioning). Each division uses this data to monitor its trucks, and sometimes messages containing new travel routes or other instructions are sent back to the trucks. In addition, selected data is sent on to company headquarters for additional analysis.

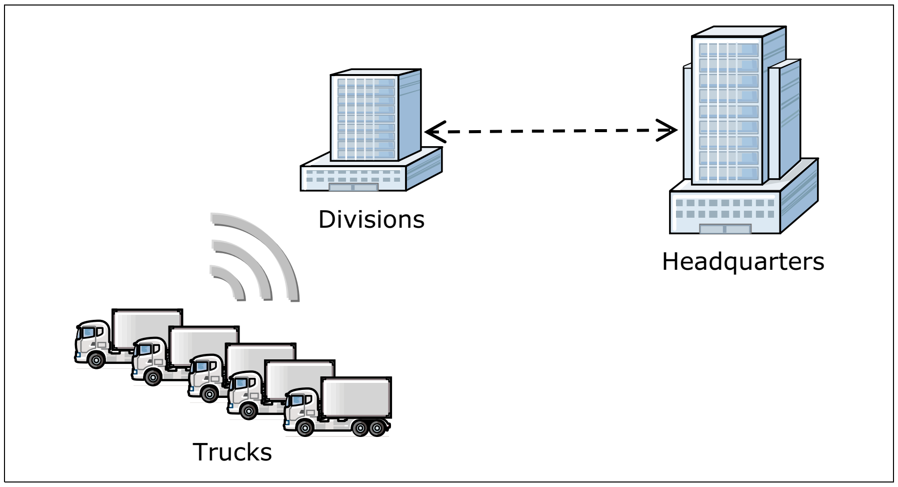


Figure 2-1 Current messaging architecture of Transportation Company B

The current messaging architecture is inadequate to meet the company’s needs for the following reasons:

- The high cost of using multiple, different protocols to communicate with trucks in the fleet, which often have different messaging capabilities and requirements.
- Limited available bandwidth, which makes it essential that every message to or from a truck be as small as possible.
- Frequent network disruptions that cause messages to be lost or garbled in transmission.

- The lack of effective message distribution mechanisms, which limits communication between trucks and forces the use of separate, centralized distribution systems.
- Too wide a variety of on-board telemetry messaging devices, each with different processing capabilities and memory resources.
- The need for customized code to integrate with the messaging requirements of different backend systems used for monitoring, analysis and processing.

2.2.2 MQTT-enabled solution

To eliminate the disadvantages of the current architecture, Transportation Company B plans to use the MQTT protocol and related messaging products to create a more universally compatible system that puts fewer demands on the network and can work with the telemetry devices already present on many of its trucks.

Figure 2-2 shows the new messaging architecture.

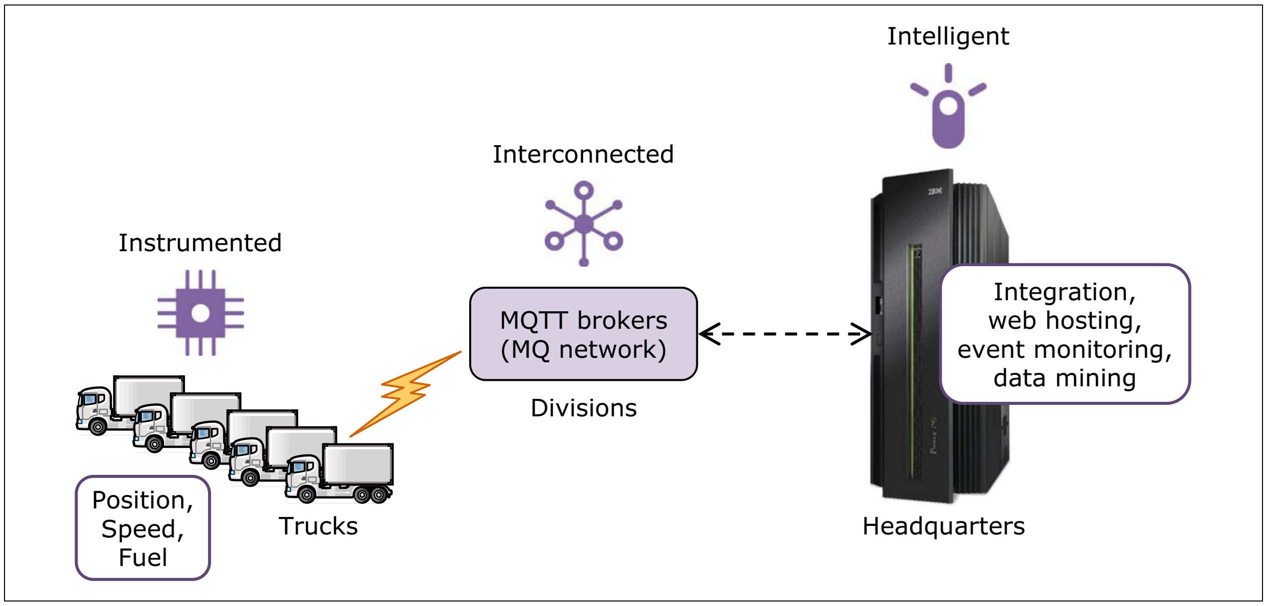


Figure 2-2 Future messaging architecture of Transportation Company B

The new messaging architecture uses Smarter Planet concepts to solve many of the challenges facing Transportation Company B:

- Instrumented

  Information is captured where it exists: in the trucks themselves. Using the lightweight but powerful MQTT protocol helps overcome the fact that many of the devices on the trucks have limited memory or processing power, or connect by means of unreliable networks.

- Interconnected

  Information is sent where it is needed: to the company's division offices and central headquarters, and, when necessary, back to the trucks. By connecting MQTT clients on the trucks to a series of MQTT-enabled message brokers, the critical telemetry information from the trucks is sent where it needs to go quickly and efficiently, with greatly reduced infrastructure costs and architectural challenges.

- Intelligent

  Information is analyzed and decisions are made in real time. All messages are sent to the company's division and central headquarters using the common MQTT format, enabling a single backend system to be used for monitoring, analysis and processing.

2.3 MQTT concepts

This section describes the MQTT publish/subscribe messaging model and introduces the critical concepts involved in MQTT client programming.

2.3.1 MQTT messaging

The popularity of MQTT-based messaging stems from the simple way it allows information to be published or subscribed to, without the need to know who or what is sending or receiving the information. This simplicity allows each message to be very small in size, thereby reducing demands on the network and on the remote monitoring devices from which many MQTT messages emanate.

Publish/subscribe pattern

MQTT uses a publish/subscribe messaging pattern that enables a loose coupling between the information provider, called the *publisher*, and consumers of the information, called *subscribers*. This is achieved by introducing a message broker between the publisher and the subscribers (Figure 2-3).

Compared with the traditional point-to-point pattern, the advantage of the publish/subscribe model is that the publishing device or application does not need to know anything about the subscribing one, and vice versa. The publisher simply sends the message with an identifier that denotes its topic, or subject area. The broker then distributes the message to all applications or devices that have chosen to subscribe to that topic. In this way, the publish/subscribe pattern turns traditional point-to-point messaging into a multicast of content-based communications.

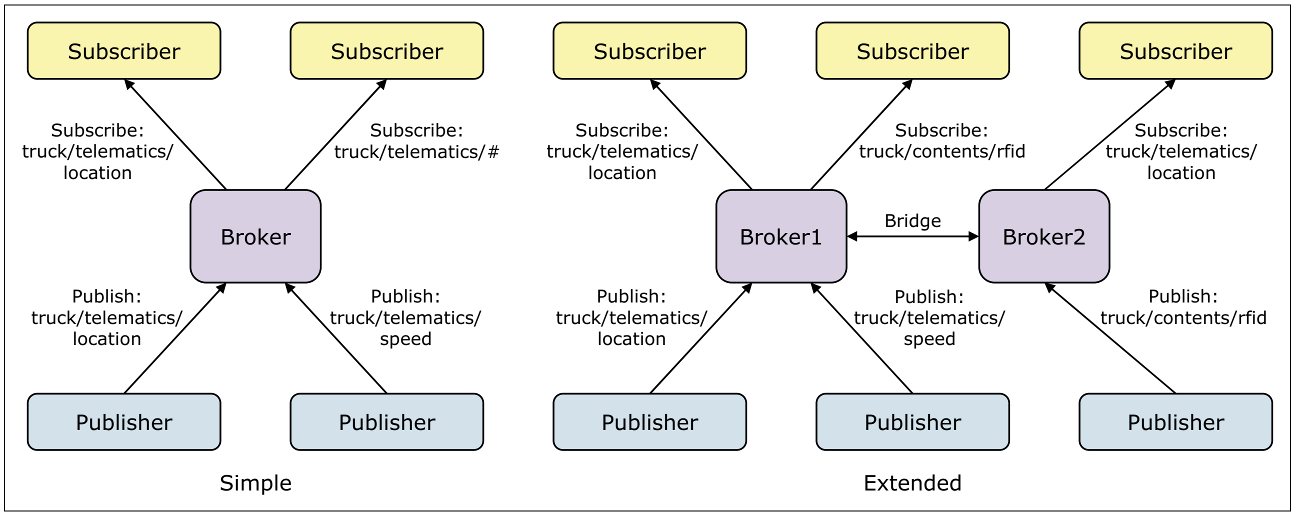

Subscriber
Subscriber
Subscriber
Subscriber
Subscriber
Subscribe: truck/telematics/location
Subscribe: truck/telematics/#
Subscribe: truck/telematics/location
Subscribe: truck/contents/rfid
Subscribe: truck/telematics/location
Broker
Broker1
Bridge
Broker2
Publish: truck/telematics/location
Publish: truck/telematics/speed
Publish: truck/telematics/location
Publish: truck/telematics/speed
Publish: truck/contents/rfid
Publisher
Publisher
Publisher
Publisher
Publisher
Simple
Extended

Figure 2-3 Two examples of publish/subscribe configurations

The left side of Figure 2-3 shows a simple publish/subscribe configuration. Each publisher and subscriber focuses only on sending and receiving the information they care about, while the broker sits between them and routes each message in the proper direction based on its topic designation.

On the right side of the figure, the advanced configuration shows two brokers that exchange information internally by way of a bridge. This allows topic subscribers that are linked to one broker to get messages that are published to that topic through another broker. In this way, the publish/subscribe model simplifies system maintenance and message distribution, which leads to better scalability than the traditional point-to-point pattern. For more information about publish/subscribe topologies and scalability, see Chapter 5, “Typical topologies and patterns of use” on page 159 and Chapter 6, “Scaling a system” on page 181.

Topics, trees, and strings

Message distribution in MQTT-based systems depends on the designation of specific topics, topic trees, and topic strings.

Topics

Publishers are responsible for classifying their message subjects into topics. A *topic* defines the content of a message or the subject area under which the message can be categorized. In our scenario, the simplest example of a topic is the single word truck. Topics are important because while messages in point-to-point systems are sent to specific destination addresses, messages in publish/subscribe systems such as MQTT are distributed based on each subscriber’s choice of topics. By subscribing to a particular topic, the subscriber is signing up to receive every message published to that topic from any publisher.

Topic trees

Typically, topics are organized hierarchically into *topic trees*, using the '/' character to create subtopics in the topic string. In our scenario, an example of a simple topic tree is truck/contents.

Topic strings

A *topic string* is a character string that identifies the topic of a publish/subscribe message. Topic strings can contain either of two wildcard schemes to allow subscribers to match patterns within strings defined by message publishers:

- *Multilevel*: The wildcard character number sign (#) is used to match any number of levels within a topic. For example, subscribers to *truck/contents/#* receive all messages that are designated for the topics truck/contents and truck/contents/rfid.
- *Single level*: The wildcard character plus sign (+) is used to match just one topic level. For example, *truck/+* matches truck/contents but not truck/contents/rfid.

2.3.2 Client programming

Some key concepts for MQTT client programming are provided here. But the information is very general. For additional details, refer to the following websites:

- Java client

  http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60300_.htm

- C client

  http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60600_.htm

Client identifier

The client identifier is a 23-byte string that identifies an MQTT client. Each identifier must be unique to only one connected client at a time. In order to keep the identifier short and unique, developers typically introduce an identifier generation mechanism, such as creating it from the 48-bit device MAC address. If transmission size is not a critical issue, you may use the remaining 17 bytes to make the address easier to administer, such as by inserting some human-readable text in the identifier.

Retained publications

Publications (messages) for a given topic can be retained and delivered when new subscribers sign up for the topic. Publishers must set a retention attribute for each message; a setting of *true* retains the message while a setting of *false* establishes that the message should not be retained for future delivery. When a publication to be retained is created or updated, it should be given a quality of service (QoS) designation of *at least once* (QoS=1) or *exactly once* (QoS=2). If a publication is sent with a QoS setting of *at most once* (QoS=0), a nonpersistent retained publication is automatically created and the publication is not retained if the queue manager stops.

Use retained publications to record the latest value of a measurement. New subscribers to the retained topic immediately receive the most recent value of the measurement. If no new measurements have been taken since the subscriber last subscribed to the publication topic, the subscriber still receives the most recent retained publication on the topic the next time the subscriber connects.

Stateless and stateful sessions (cleanSession)

When you connect to an MQTT client application, the client identifies the connection using the client identifier and the address of the server. The server checks whether session information has been saved from a previous connection to the server. The `cleanSession` parameter in the connection options indicates whether the connection should be stateless or stateful. If a previous session still exists, and `cleanSession=true`, then the previous session information at the client and server is cleared. If `cleanSession=false`, the previous session is resumed. If no previous session exists, a new session is started. The default value of `cleanSession` is true.

For publications, the clean session setting only has effect on publications sent with designations of QoS=1 and QoS=2. Using `cleanSession=true` might result in losing a publication, because it drops all publications that have not been received.

For subscriptions, if `cleanSession=true`, any old subscriptions for the client are removed when the client connects. Any new subscriptions the client makes during the session are removed when it disconnects.

Last will and testament

In case an MQTT client connection ends unexpectedly, you can configure a *last will and testament* publication to a topic. You must predefine the content of the publication, and the topic to send the publication to. The last will and testament is a connection property that must be set before connecting to the client.

2.4 MQTT brokers

This section describes how to install and run two MQTT brokers: Really Small Message Broker and Mosquitto. It provides numerous command line options for publishing and subscribing to MQTT topics, and describes the use of a GUI client, the WMQTT Utility (available in SupportPac IA92), which can make the publishing and subscribing processes even easier.

To download the RSMB and Mosquitto brokers, refer to 2.1, “Clients and brokers used in this book” on page 22. To download the SupportPac IA92 and WMQTT Utility, go to this website:

http://www-01.ibm.com/support/docview.wss?uid=swg24006006

2.4.1 Really Small Message Broker

IBM Really Small Message Broker (RSMB) is an MQTT broker supporting versions 3 and 3.1 of the MQTT protocol. It is available for multiple operating systems, although this book is limited to describing its use under Linux.

Installation

Follow these steps to install RSMB on the 64-bit Linux platform:

1. Extract the downloaded file to a location such as `/home/mqttUser`.
2. Navigate to the `linuxia64` folder. The broker command, which brings up the broker, can be found in this folder.

Starting and stopping the broker

To run the broker unattended, enter this command:

```
nohup ./broker >> /dev/null &
```

To run the broker with the path to a specified library, use the command shown here.

```
LDLIBRARYPATH=/path/to/lib ./broker
```

If this server is to be run on a port other than the default one, 1883, then use a configuration file. The next example shows a sample configuration file named `port2883.cfg`, with the port number set to 2883.

```
#Sample configuration file to start broker on another port, 2883
port 2883
```

To start the broker with the configuration file referenced in the previous example, enter this command.

```
./broker port2883.cfg
```

This configuration file can be used to configure additional parameters. For a list of these additional options, refer to the README file in the RSMB download.

To shut down the RSMB broker, type `Ctrl+C` on the Linux command line, or kill the process that is running the broker.

Publishing to an MQTT topic

With the MQTT server (RSMB) now running, you can test the act of publishing messages to a topic.

To publish a message using RSMB, use the **`stdinpub`** command as shown in the next example. This example assumes that the client and the server are on the same machine. The message is published to the samples/topic01 topic. The actual message to be published would be typed once the connection is established. When the Enter key is pressed, the message is delivered.

```
./stdinpub samples/topic01
```

To publish to a broker on another machine and port (and, optionally, specify a client ID), use the command string shown here:

```
./stdinpub samples/topic01 --host mqtt.host --port 1650 --clientid localGhost
```

Additional options are available to control the publishing process under RSMB. The command shown in Example 2-1 displays a list of these publishing options.

Example 2-1 Publishing options using RSMB

```
nsubrahm@nsubrahm:~$ ./stdinpub
MQTT stdin publisher
Usage: stdinpub topicname <options>, where options are:
  --host <hostname> (default is localhost)
  --port <port> (default is 1883)
  --qos <qos> (default is 0)
  --retained (default is off)
  --delimiter <delim> (default is)
  --clientid <clientid> (default is hostname+timestamp)  --maxdatalen 100
  --username none
  --password none
```

Subscribing to an MQTT topic

To subscribe to a message using RSMB, use the **stdoutsub** command as shown in the next example. The example assumes the client and the server are on the same machine. The subscription is to the topic samples/topic01. The actual message is displayed on the console.

```
./stdoutsub samples/topic01
```

Just as with publishing, RSMB provides additional options related to the subscription process. The command shown in Example 2-2 displays a list of these subscription options.

Example 2-2 Subscription options using RSMB

```
nsubrahm@nsubrahm:$ ./stdoutsub
MQTT stdout subscriber
Usage: stdoutsub topicname <options>, where options are:
--host <hostname> (default is localhost)
--port <port> (default is 1883)
--qos <qos> (default is 2)
--delimiter <delim> (default is
) --clientid <clientid> (default is hostname+timestamp)
--username none
--password none
--showtopics <on or off> (default is on if the topic has a wildcard, else off)
```

2.4.2 Mosquitto

Mosquitto is an open-source MQTT broker that supports the MQTT V3.1 protocol. It is available for multiple operating systems, although this book is limited to describing its use under Linux.

Installation

Download and follow instructions for installation of the Mosquitto binary from this website:

http://mosquitto.org

Starting and stopping the broker

To run the Mosquitto broker, run the **mosquitto** command using the syntax shown here.

```
mosquitto
```

You can also use a configuration file to specify values such as a port number, or a maximum number of connections, as shown in this example:

```
#Sample configuration file to be used when starting up mosquitto
#port number
port 2885
# The maximum number of client connections to allow.
max_connections 20
```

To shut down the Mosquitto broker, type `Ctrl+C` on the Linux command line, or kill the process that is running the broker.

Publishing to an MQTT topic

To publish a message to a topic using Mosquitto, use the **mosquittopub** command as shown in the next example. The example assumes that the client and the server are on the same machine. The message is published to the samples/topic01 topic. The actual message to be published would be typed after the connection is established. The message is then delivered by pressing the Enter key.

```
mosquitto_pub -t samples/topic01 -l
```

To publish to a broker on another machine and port (and, optionally, a different client ID), use the command shown here.

```
mosquitto_pub -h fake.mqtt.server -t samples/topic01 -i localGhost -l
```

As with RSMB, Mosquitto provides additional options to control the publishing process. Example 2-3 displays a list of these publishing options.

Example 2-3 Publishing options using Mosquitto

```
nsubrahm@nsubrahm:~$ mosquitto_pub
mosquitto_pub is a simple mqtt client that will publish a message on a single topic and exit.

Usage: mosquitto_pub [-h host] [-p port] [-q qos] [-r] {-f file | -l | -n | -m message} -t topic
                     [-i id] [-I id_prefix]
                     [-d] [--quiet]
                     [-u username [-P password]]
                     [--will-topic [--will-payload payload] [--will-qos qos] [--will-retain]]

 -d : enable debug messages.
 -f : send the contents of a file as the message.
 -h : mqtt host to connect to. Defaults to localhost.
 -i : id to use for this client. Defaults to mosquitto_pub_ appended with the process id.
 -I : define the client id as id_prefix appended with the process id. Useful for when the
      broker is using the clientid_prefixes option.
 -l : read messages from stdin, sending a separate message for each line.
 -m : message payload to send.
 -n : send a null (zero length) message.
 -p : network port to connect to. Defaults to 1883.
 -q : quality of service level to use for all messages. Defaults to 0.
 -r : message should be retained.
 -s : read message from stdin, sending the entire input as a message.
 -t : mqtt topic to publish to.
 -u : provide a username (requires MQTT 3.1 broker)
 -P : provide a password (requires MQTT 3.1 broker)
 --quiet : don't print error messages.
 --will-payload : payload for the client Will, which is sent by the broker in case of
                  unexpected disconnection. If not given and will-topic is set, a zero
                  length message will be sent.
 --will-qos : QoS level for the client Will.
 --will-retain : if given, make the client Will retained.
 --will-topic : the topic on which to publish the client Will.

See http://mosquitto.org/ for more information.
```

Subscribing to an MQTT topic

To subscribe to a topic, use the **`mosquitto_sub`** command as shown in the next example. The example assumes that the client and the server are on the same machine. The topic being subscribed to is samples/topic01. The actual message appears in the console.

```
mosquitto_sub -t samples/topic01
```

To subscribe to a Mosquitto broker on another machine and port, use the command shown here:

```
mosquitto_sub -h fake.mqtt.server -t samples/topic01
```

As with publishing using Mosquitto, there are additional options related to the subscription process. The command shown in Example 2-4 displays a list of these subscription options.

Example 2-4 Subscription options using Mosquitto

```
nsubrahm@nsubrahm:~$ mosquitto_sub
mosquitto_sub is a simple mqtt client that will subscribe to a single topic and print all
messages it receives.

Usage: mosquitto_sub [-c] [-h host] [-k keepalive] [-p port] [-q qos] [-v] -t topic ...
                     [-i id] [-I id_prefix]
                     [-d] [--quiet]
                     [-u username [-P password]]
                     [--will-topic [--will-payload payload] [--will-qos qos] [--will-retain]]

 -c : disable 'clean session' (store subscription and pending messages when client disconnects).
 -d : enable debug messages.
 -h : mqtt host to connect to. Defaults to localhost.
 -i : id to use for this client. Defaults to mosquitto_sub_ appended with the process id.
 -I : define the client id as id_prefix appended with the process id. Useful for when the
      broker is using the clientid_prefixes option.
 -k : keep alive in seconds for this client. Defaults to 60.
 -p : network port to connect to. Defaults to 1883.
 -q : quality of service level to use for the subscription. Defaults to 0.
 -t : mqtt topic to subscribe to. May be repeated multiple times.
 -u : provide a username (requires MQTT 3.1 broker)
 -v : print published messages verbosely.
 -P : provide a password (requires MQTT 3.1 broker)
 --quiet : don't print error messages.
 --will-payload : payload for the client Will, which is sent by the broker in case of
                  unexpected disconnection. If not given and will-topic is set, a zero
                  length message will be sent.
 --will-qos : QoS level for the client Will.
 --will-retain : if given, make the client Will retained.
 --will-topic : the topic on which to publish the client Will.

See http://mosquitto.org/ for more information.
```

2.4.3 WMQTT Utility

In addition to the command line options listed previously, a GUI client called the WMQTT Utility can be used to control the publishing and subscribing processes. Some developers find using the utility to be a simpler and easier method.

Installation and start-up

Start by downloading SupportPac IA92 as described at the top of this section. SupportPac IA92 is a Java implementation (Java VM 1.3 or above) of the WebSphere MQTT protocol. As this is a Java implementation, the downloaded package is a set of Java JAR files and therefore is independent of the platform. There is no installation procedure except for unzipping the downloaded file.

To start the WMQTT Utility, begin by navigating to the folder where the SupportPac was downloaded. Then run the JAR file by issuing the **`java`** command from the command line, as shown here:

```
java -jar wmqttSample.jar
```

Using the WMQTT Utility

The WMQTT Utility panel has two tabs:

- The WMQTT tab
- The Options tab

The WMQTT tab, shown in Figure 2-4 on page 34, covers the basic publishing and subscribing operations. The numbers here correspond to the numbers shown in the figure.

1. The WMQTT tab includes the remaining items in this list.
2. Broker TCP/IP address: Enter the IP address for the message broker being used or the host name along with the port number (if not using the default port setting 1883).
3. Connection status: The color of the circle indicates the connection status. A red circle indicates a disconnected state, an amber circle indicates a connection is being attempted, and a green circle indicates a connected state.
4. Connect/Disconnect buttons: Click these buttons to initiate or discontinue a connection.

Important: For the Connect and Disconnect buttons to work, the IP address and port number must be entered.

5. Subscribe To Topics: Enter the topic to which you are subscribing. After you enter a topic in the field, the Subscribe button becomes active. Click the Subscribe button to make the subscription take effect.

6. Publish Messages—text display: Enter the topic to which the message will be published here. After you enter the topic in the field, the nearby Publish button becomes active. Click the Publish button to publish the message.
7. Received Topic: After a subscription to a topic is made, the message that is received is displayed in this field.
8. Published Topic: Enter the text of the message that will be published to the designated topic.

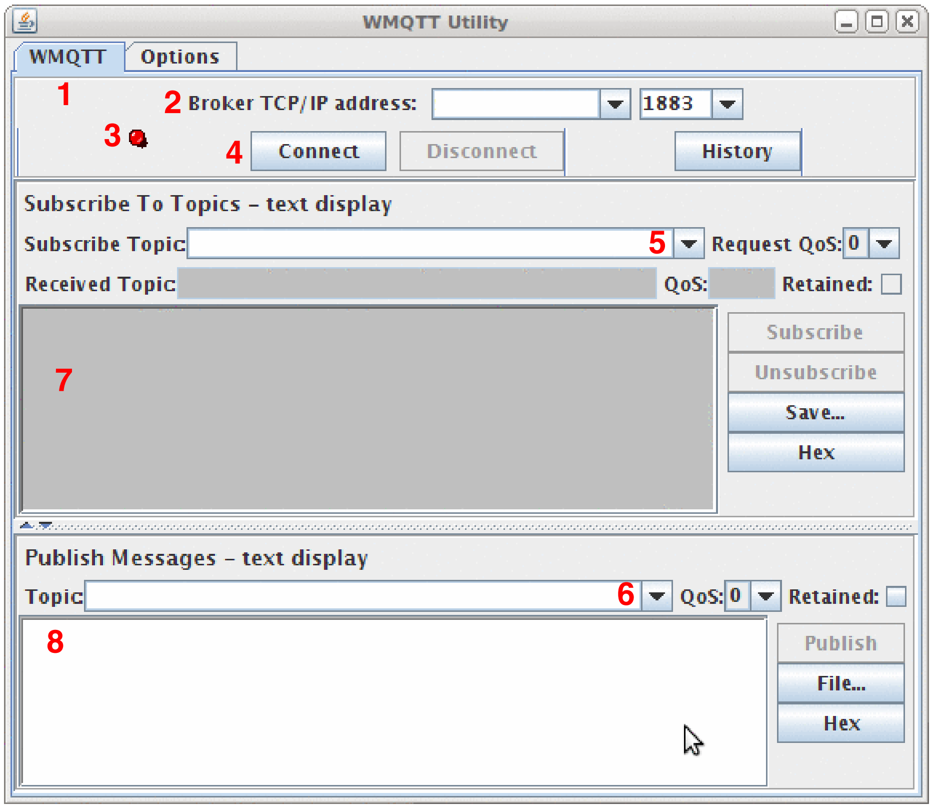
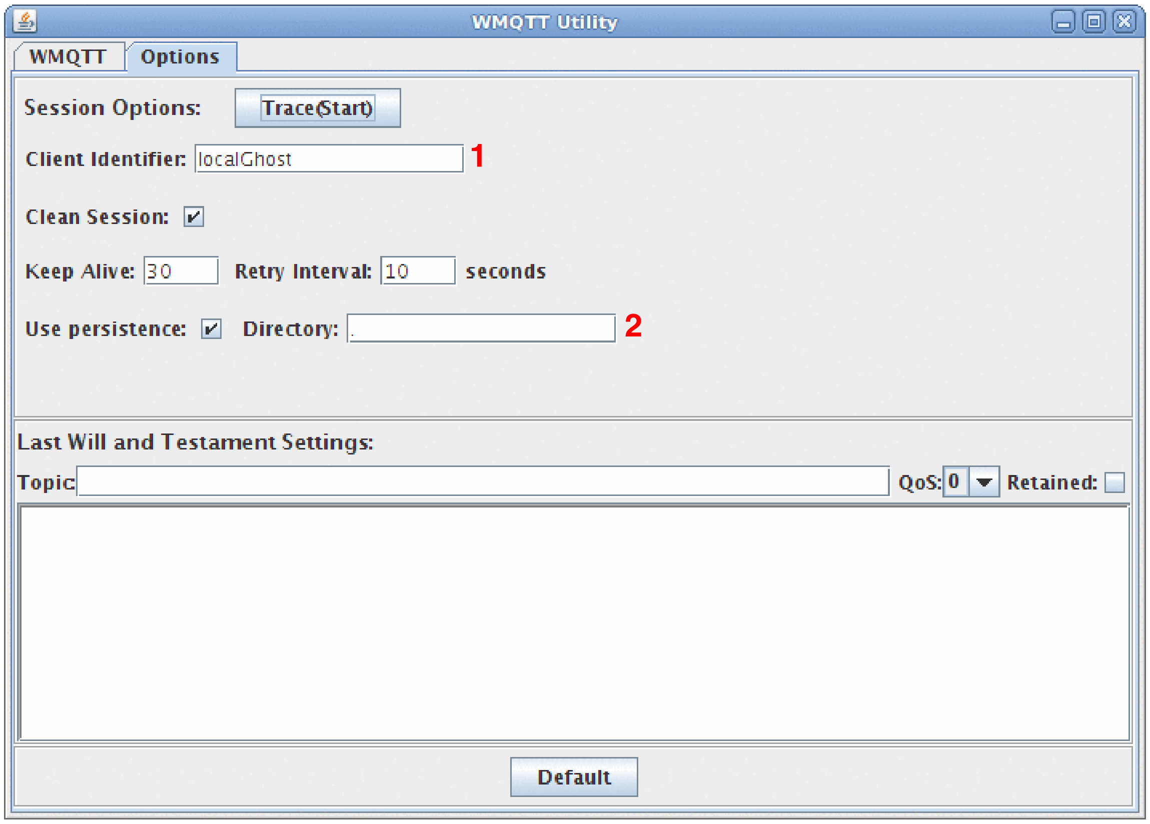


Figure 2-4 The WMQTT tab on the WMQTT Utility panel

The Options tab provides additional settings and opens with the following default settings already selected (see Figure 2-5):

1. Client Identifier field: Enter the client identifier here.
2. User persistence check box and Directory field: Turn on persistence by selecting **Use persistence** and specifying the directory in which persistent messages are stored.



Session Options:
Client Identifier: localGhost
Clean Session: ✓
Keep Alive: 30 Retry Interval: 10 seconds
Use persistence: ✓ Directory: .
Topic
QoS: 0
Retained:

Figure 2-5 The Options tab on the WMQTT Utility panel

2.5 Building a sample MQTT application with freely available software

In this section we demonstrate how to build an initial MQTT-based messaging application. Instructions for both Java- and C-based systems are provided. These examples use the no-cost MQTT clients that have been made available as part of the Eclipse Paho project. For information about downloading these clients, see 2.1, “Clients and brokers used in this book” on page 22.

2.5.1 Preparation

The Java and C clients from the Eclipse Paho project are distributed as source code. Thus, developers need to compile these clients on their own. The Eclipse Paho client source code is in a Git repository, and Git must be installed on the system before the compiling process is begun.

If WebSphere MQ V7.1 is used or if a stand-alone package of WebSphere MQ V7.1 clients has been downloaded, those clients can be used and you can skip to 2.5.2, “Building the sample MQTT application" on page 36. The APIs of the Eclipse Paho clients are basically identical to those in WebSphere MQ.

Preparing the Java client

The build script of the Eclipse Paho Java client requires the Apache Ant automated software build tool:

1. First, download Apache Ant from:

   http://ant.apache.org/

2. Start at the console and enter the following command to get the source code from the Git repository.

   ```
   git clone git://git.eclipse.org/gitroot/paho/org.eclipse.paho.mqtt.java.git
   ```

3. Next, initiate the ant command as shown here.

   ```
   cd org.eclipse.paho.mqtt.java.git
   ant
   ```

After entering the commands just shown, you will see the messages shown in Example 2-5. The location of the compiled `.jar` file is listed here as the `/tmp/out/ship` directory.

Example 2-5   Messages received upon initiating the ant command

```
....
package:
      [jar] Building jar: /tmp/out/ship/org.eclipse.paho.client.mqttv3.jar
clean:
   [delete] Deleting directory /tmp/out/micro-client-v3
full:
BUILD SUCCESSFUL
Total time: 9 seconds
```

Preparing the C client

Building the Eclipse Paho C client requires *make tools*, including the GCC compiler, the Doxygen documentation system, and Graphviz graph visualization software. If you use the Debian or Ubuntu Linux distribution, you can install these make tools with the following command:

```
sudo apt-get install build-essentials doxygen graphviz
```

Next, download the source code from the Git repository, as shown here:

```
git clone git://git.eclipse.org/gitroot/paho/org.eclipse.paho.mqtt.c.git
```

Finally, switch to the `src` folder and build the client and documentation using these commands:

```
cd org.eclipse.paho.mqtt.c.git/src
make -f ../build/Makefile all
doxygen ../doc/DoxyfileV3ClientAPI
```

2.5.2 Building the sample MQTT application

Trucks are the basic transportation units of Transportation Company B and carry goods from one city to another. While making each trip, the trucks send their location information (latitude and longitude) to their division office for monitoring and central control. They also have a channel to receive messages (instructions) from the division office. Division supervisors analyze the location information received from the trucks and then reply with any schedule or route changes that might be needed.

This use case is a classic publish/subscribe application, as illustrated in Figure 2-6.

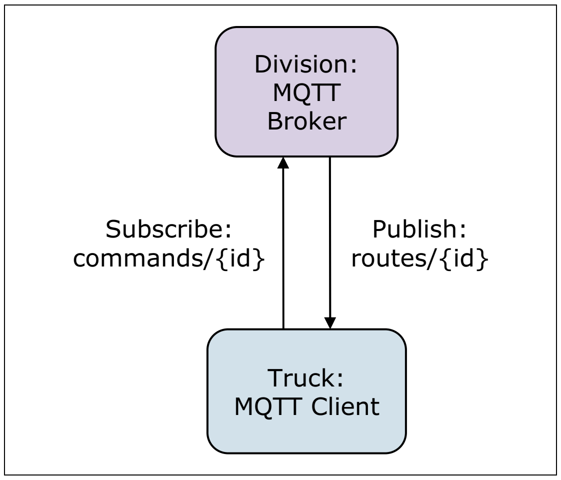


Figure 2-6 Simple publish/subscribe messaging

The next two sections describe how to build an MQTT client application that can publish and subscribe to topics. Examples are provided for Java and C.

The following general steps are performed when creating the applications:

1. Create an instance of an MQTT client.
2. Prepare connection options and connect to the MQTT broker using these options.
3. Publish messages to topics, and subscribe to topics.
4. Disconnect from the MQTT broker (and remember to release resources on platforms without automatic memory management).

2.5.3 Publisher and subscriber in Java

This section describes how to use Java to build an MQTT client application to publish and subscribe to topics. An Eclipse runtime is used to build this client.

Publisher application

This sample publisher application uses the following Java classes:

- `Example.java`: This class defines parameters such as the address of the broker, the message topic (for publishing and subscribing), and other connection settings. It is used by both the publisher and subscriber Java classes. Refer to Example 2-6.

Example 2-6 Example.java with parameters defined

```
package org.mqtt.redbooks.ch2;

public final class Example {
  public static final String TCPAddress =
    System.getProperty("TCPAddress", "tcp://localhost:1883");
  public static String clientId = "Truck01";
  public static final String pubTopicString =
    System.getProperty("pubTopicString", "routes/" + clientId);
  public static final String subTopicString =
    System.getProperty("subTopicString", "commands/" + clientId);
  public static final String publication =
    System.getProperty("publication", System.currentTimeMillis()
      + ",35.918,-78.231,B");
  public static final int sleepTimeout =
    Integer.parseInt(System.getProperty("timeout", "10000"));
  public static final boolean cleanSession =
    Boolean.parseBoolean(System.getProperty("cleanSession", "false"));
  public static final int keepAliveInterval =
    Integer.parseInt(System.getProperty("keepAliveInteral", "20"));
  public static final int QoS =
    Integer.parseInt(System.getProperty("QoS", "1"));
  public static final boolean retained =
    Boolean.parseBoolean(System.getProperty("retained", "false"));
}
```

- `PubSync.java`: This class is used to connect to an MQTT broker, publish a message to a topic, and disconnect from the broker. Refer to Example 2-7.

Example 2-7 PubSync.java

```
package org.mqtt.redbooks.ch2;

import org.eclipse.paho.client.mqttv3.*;

public class PubSync {
  public static void main(String[] args) {
    try {
      //a. Create an instance of MQTT client
      MqttClient client = new MqttClient(Example.TCPAddress, Example.clientId);

      //b. Prepare connection options
      //Use default connection options in this sample.

      //c. Connect to broker with the connection options
      client.connect();

      //d. Publish message to topics
      MqttTopic topic = client.getTopic(Example.pubTopicString);
      MqttMessage message = new MqttMessage(Example.publication.getBytes());
      message.setQos(Example.QoS);
      System.out.println("Waiting for up to " + Example.sleepTimeout / 1000
          + " seconds for publication of \"" + message.toString()
          + "\" with QoS = " + message.getQos());
      System.out.println("On topic \"" + topic.getName()
          + "\" for client instance: \"" + client.getClientId()
          + "\" on address " + client.getServerURI() + "\"");
      MqttDeliveryToken token = topic.publish(message);
      token.waitForCompletion(Example.sleepTimeout);
      System.out.println("Delivery token \"" + token.hashCode()
          + "\" has been received: " + token.isComplete());

      //e. Disconnect to broker
      client.disconnect();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

PubSync.java details

Parts of the `PubSync.java` code require additional explanation.

1. Create a try-catch block, as shown here, to handle any checked exceptions (`MqttException` or its subclasses `MqttPersistenceException` and `MqttSecurityException`) that are thrown by the MQTT client.

   ```
   try { ...
    } catch (Exception e) {
         e.printStackTrace();
    }
   ```

2. Create a new `MqttClient` instance using the following command:

   ```
   MqttClient client = new MqttClient(Example.TCPAddress, Example.clientId)
   ```

   The default WebSphere MQ TCP/IP port for MQTT is 1883. In this example, the default address, Example.TCPAddress, is set to `tcp://localhost:1883`.

   The client identifier, `Example.clientId`, must be unique across all clients connecting to a server. For more information, see the explanation of MQTT client identifier in 2.3, “MQTT concepts” on page 26.

3. Optionally, you can provide an implementation of the `MqttClientPersistence` interface to replace the default implementation. The default implementation stores messages that are awaiting delivery (such as ones with QoS designations of 1 or 2) as files in the current directory. If you want to change the directory where the files are stored, you can create an `MqttDefaultFilePersistence` instance and provide it as third parameter for the constructor `MqttClient`.

   The `MqttDefaultFilePersistence` class needs to be configured as shown here and used while creating the MQTT client instance optionally. It stores messages in the local directory that is specified in the constructor.

   ```
   MqttClientPersistence clientPersistence = new
   MqttDefaultFilePersistence("C:/MQTTPublisher/clientdir");
   client = new MqttClient("tcp://localhost:1883", "SampleClient",
   clientPersistence);
   ```

   A subdirectory prefixed with the client ID is created in the specified directory and is used while sending and receiving QoS 1 and QoS 2 messages.

4. Connect the client to the broker. If done as shown here, the default settings are used: A small message will be sent every 15 seconds to prevent the TCP/IP connection from being closed; the `cleanSession` variable is equal to true; the session is started without checking for the completion of previous publications, and no last will and testament message is created for the connection.

   ```
   client.connect();
   ```

5. Create a topic to which to publish a message. In the code shown here, *Example.pubTopicString* could be replaced with an actual topic string such as routes/Truck01.

   ```
   MqttTopic topic = client.getTopic(Example.pubTopicString);
   ```

6. Create a message for publication, as shown here. The message in this sample, *Example.publication*, will display the current time, the current location information for a particular truck, and its status. For simplicity, the location data shown here is fixed.

   ```
   MqttMessage message = new MqttMessage(Example.publication.getBytes());
   message.setQos(Example.QoS);
   ```

   Since an MQTT message requires a byte array as input, *Example.publication* is converted to a byte array using the getBytes method, and the encoding is set to UTF-8. The QoS designation, which is set to 1 here, determines how reliably the message is transferred to the MQTT client and then between the MQTT client and the broker.

7. Publish the message to the broker, as shown here.

   ```
   MqttDeliveryToken token = topic.publish(message);
   ```

   When the publish method returns, the message has been safely transferred to the MQTT client, but not yet transferred to the broker. If the message has a QoS designation of 1 or 2, then the message is stored locally, in case the client fails before delivery is completed. A delivery token is returned and used to check whether an acknowledgment has been received from the server yet. MQTT delivery tokens are unique to the WebSphere MQ Telemetry client and enable users to monitor delivery of a published message.

8. Wait for an acknowledgment from the server confirming that the message has been delivered. The example provided here shows a timeout, without which the client would wait indefinitely. If using an asynchronous API for publishing, the program does not need to be blocked until it gets the acknowledgment from the server. The callback function will be called after successful message delivery.

   ```
   token.waitForCompletion(Example.timeout);
   ```

9. Disconnect the client from the server, as shown here. The client disconnects from the server and waits for any `MqttCallback` methods that are running to finish. The client then waits for up to 30 seconds to finish any remaining work. You can specify a timeout as an additional parameter.

   ```
   client.disconnect();
   ```

Compilation and execution

Begin by starting the MQTT broker (refer to 2.4, “MQTT brokers” on page 28 for more information). Then compile and run the program:

1. Compile the Java code using the following command.

   ```
   javac -cp jardir\org.eclipse.paho.client.mqttv3.jar
   org.mqtt.redbooks.ch2.PubSync.java
   org.mqtt.redbooks.ch2.Example.java
   ```

2. Run `PubSync`:

   ```
   java -cp jardir\org.eclipse.paho.client.mqttv3.jar
   org.mqtt.redbooks.ch2.PubSync
   ```

You can also run the program through Eclipse. Eclipse does the compilation automatically, after which you right-click the `PubSync.java` file, and then select **Run As → Java Application**.

After starting your `PubSync` application, you should see a message in the console window similar to the messages shown in Example 2-8 on page 41, which means that the message was delivered to the broker successfully.

Example 2-8 Message delivered successfully

```
Waiting for up to 10 seconds for publication of "1332854539607,35.918,-78.231,B"
with QoS = 1
On topic "routes/Truck01" for client instance: "Truck01" on address
tcp://localhost:1883"
Delivery token "368076985" has been received: true
```

Subscriber

This sample subscriber application uses the following Java classes:

- `Example.java`: This class is the same as the one shown for the publisher application in the previous subsection. It defines parameters such as the address of the broker, the message topic (for publishing and subscribing), and other connection settings.
- `Callback.java`: This class implements the `MqttCallback` interface and is set to work with an MQTT client. Refer to Example 2-9. Message arrival acknowledgments are received in a callback function that is defined in the `MqttCallback` interface.

Example 2-9 Callback.java

```
package org.mqtt.redbooks.ch2;

import com.ibm.micro.client.mqttv3.*;

public class Callback implements MqttCallback {
  private String instanceData = "";
  public Callback(String instance) {
    instanceData = instance;
  }

  public void messageArrived(MqttTopic topic, MqttMessage message) {
    try {
      System.out.println("Message arrived: \"" + message.toString()
          + "\" on topic \"" + topic.toString() + "\" for instance \""
          + instanceData + "\"");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void connectionLost(Throwable cause) {
    System.out.println("Connection lost on instance \"" + instanceData
        + "\" with cause \"" + cause.getMessage() + "\" Reason code "
        + ((MqttException)cause).getReasonCode() + "\" Cause \""
        + ((MqttException)cause).getCause() +  "\"");
    cause.printStackTrace();
  }

  public void deliveryComplete(MqttDeliveryToken token) {
    try {
      System.out.println("Delivery token \"" + token.hashCode()
          + "\" received by instance \"" + instanceData + "\"");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

- `Subscribe.java`: This class creates the subscription and waits for matching publications. Refer to Example 2-10. The handling of the actual message is delegated to the Callback class, which is defined in Callback.java. `Subscribe.java` uses an asynchronous programming model, which means it does not need to wait for a message to arrive and block the execution of the program.

Example 2-10 Subscribe.java

```
package org.mqtt.redbooks.ch2;

import java.util.Scanner;

import com.ibm.micro.client.mqttv3.*;

public class Subscribe {
  public static void main(String[] args) {
    try {
      //a. Create an instance of MQTT client
      MqttClient client = new MqttClient(Example.TCPAddress, Example.clientId);

      //b. Prepare connection options
      Callback callback = new Callback(Example.clientId);
      client.setCallback(callback);
      MqttConnectOptions conOptions = new MqttConnectOptions();
      conOptions.setCleanSession(Example.cleanSession);
      conOptions.setKeepAliveInterval(Example.keepAliveInterval);

      //c. Connect to broker with the connection options
      client.connect(conOptions);
      System.out.println("Subscribing to topic \"" + Example.subTopicString
          + "\" for client instance \"" + client.getClientId()
          + "\" using QoS " + Example.QoS + ". Clean session is "
          + Example.cleanSession);

      //d. Subscribe interested topics.
      client.subscribe(Example.subTopicString, Example.QoS);
      System.out.format("Subscribing to topic %s\nfor client %s using
QoS%d\n\n"
              + "Press Q<Enter> to quit\n\n", Example.subTopicString,
              Example.clientId, Example.QoS);
      Scanner scanner = new Scanner(System.in);
      for(String input=""; !input.equalsIgnoreCase("q");
        input = scanner.nextLine());

      //e. Disconnect to broker
      client.disconnect();
      System.out.println("Finished");
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Subscribe.java details

Parts of the `Subscribe.java` code require additional explanation.

- Create an instance of Callback class as shown here and set it to the MQTT client that was declared in the publishing application. The Callback class implements the `MqttCallback` interface. One callback instance per client identifier is required. In this example, the constructor passes the client identifier to save as instance data.

  ```
  Callback callback = new Callback(Example.clientId);
  client.setCallback(callback);
  ```

  The `Callback` interface includes the following methods:

  – `public void messageArrived(MqttTopic topic, MqttMessage message)`: Receives a publication that has been subscribed to.

  – `public void connectionLost(Throwable cause)`: Called when the connection is lost.

  – `public void deliveryComplete(MqttDeliveryToken token)`: Called when a delivery token is received for a published message carrying a QoS designation of 1 or 2

  The Callback class in `Callback.java` implements these three methods. When a message is delivered to the topic to which the program subscribes, it writes the message to the standard output device (typically the console).

- Create an `MqttConnectOptions` object to configure the connection properties of the MQTT client. In this example, the cleanSession and `keepAliveInterval` attributes are set.

  ```
  MqttConnectOptions conOptions = new MqttConnectOptions();
  conOptions.setCleanSession(Example.cleanSession);
  conOptions.setKeepAliveInterval(Example.keepAliveInterval);
  ```

  If you use the default `MqttConnectOptions`, or set `MqttConnectOptions.cleanSession` to `true` before connecting the client, any old subscriptions for the client are removed when the client connects. Any new subscriptions the client makes during the session are removed when it disconnects.

  If you set `MqttConnectOptions.cleanSession` to `false` before connecting, any subscriptions the client creates are added to all the subscriptions that existed for the client before it connected. All the subscriptions remain active when the client disconnects.

  Set the `cleanSession` mode before connecting. The mode lasts for the whole session. To change this setting, disconnect and reconnect the client. If you change modes from using `cleanSession=false` to `cleanSession=true`, all previous subscriptions for the client, and any publications that have not been received are discarded.

  Additional connection options are available, including MQTT connect timeout. With this option, when the client connects to the server it can set a connection timeout. This is important because in certain networks and operating systems, when a connection is attempted on a socket that is not reachable, the ensuing errors are not returned immediately. Using the method `setConnectionTimeout()` allows the developer to specify how long the client can wait before it decides that the server is not reachable.

- Create a subscription and set its QoS level. The example shown here uses an MqttClient.subscribe method that passes one topic string with a QoS option:

  ```
  client.subscribe(Example.subTopicString, Example.QoS);
  ```

  Each time you run Subscribe.java, it creates a subscription. So unless Example.topicString is changed each time, the same subscription will be recreated over and over again. In this example, the subscription topic is `commands/Truck01` by default.

- Wait for publications to arrive or, if you do not wish to wait, type `q` or `Q` on the command line to disconnect the client and exit the application, as shown here:

  ```
  Scanner scanner = new Scanner(System.in);
  for(String input=""; !input.equalsIgnoreCase("q");
  input = scanner.nextLine());
  ```

Compilation and execution

Begin by starting the MQTT broker (refer to 2.4, "MQTT brokers" on page 28 for more information). Then compile and run the program:

1. Compile the Java code using the command shown here:

   ```
   javac -cp jardir\org.eclipse.paho.client.mqttv3.jar
   org.mqtt.redbooks.ch2.Subscribe.java
   org.mqtt.redbooks.ch2.Example.java
   ```

2. Run `Subscribe.java`:

   ```
   java -cp jardir\org.eclipse.paho.client.mqttv3.jar
   org.mqtt.redbooks.ch2.Subscribe
   ```

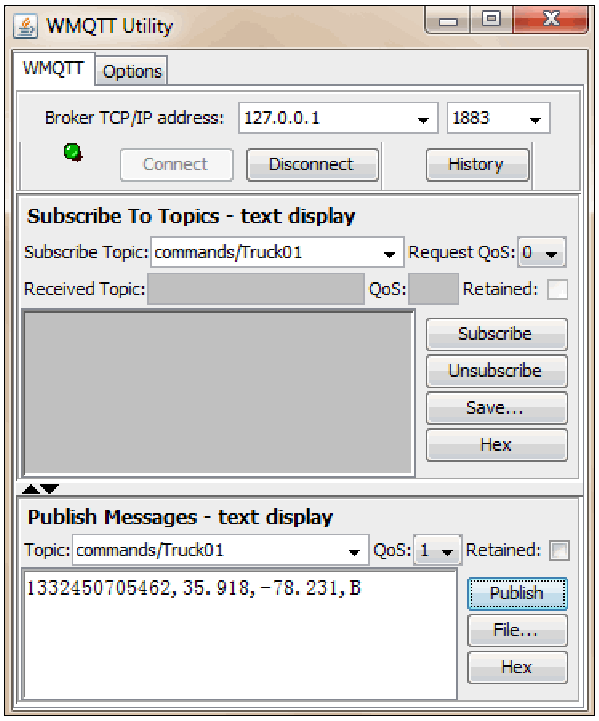
You can also run the program through Eclipse. Eclipse does the compilation automatically, after which you right-click the `Subscribe.java` file, and then select **Run As** → **Java Application**.

After starting your Subscribe application, you should see a logging message in the console window similar to the message shown in Example 2-11, which means that the application has started successfully.

Example 2-11 Application started successfully

```
Subscribing to topic "commands/Truck01" for client instance "Truck01" using QoS 1.
Clean session is false
Subscribing to topic commands/Truck01
for client Truck01 using QoS1

Press Q<Enter> to quit
```

Now that the Subscribe application is started, a message can be published. To do this, start the WMQTT Utility (the GUI interface supplied in SupportPac IBM IA-92) and publish a message to the `commands/Truck01` topic, as shown in Figure 2-7.



```
1332450705462,35.918,-78.231,B
```

Figure 2-7 Publishing a message to the commands/Truck01 topic

After publishing a message with the WMQTT Utility, the Subscribe application subscribes to the topic commands/Truck01. The application receives the message and displays it on the console.

2.5.4 Publisher and subscriber in C

In this section we describe how to use the C programming language to build an MQTT client application similar to the Java one built in the previous section.

The biggest difference between the Java and C clients is that for the C client, you have to handle some memory management tasks manually, because C does not have an automatic memory management runtime. So here, you need to release memory resources and then dynamically allocate them in the program. Otherwise, it leads to a memory leak. Resources that need to be released include the MQTT client instance and message instance.

Publisher

The sample C publisher, pubsync.c, is shown in Example 2-12. It is built in synchronous mode, both for the sake of simplicity and because for this demonstration there is no need to track the status of delivered messages.

Example 2-12 Sample C publishing program, pubsync.c

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "MQTTClient.h"
```

```
#define ADDRESS     "tcp://localhost:1883"
#define CLIENTID    "Truck01"
#define TOPIC       "routes/Truck01"
#define PAYLOAD     "1332450705462,35.918,-78.231,B"
#define QOS         1
#define TIMEOUT     10000L

int main(int argc, char* argv[])
{
    MQTTClient client;
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    MQTTClient_message pubmsg = MQTTClient_message_initializer;
    MQTTClient_deliveryToken token;
    int rc;
    //a. Create an instance of MQTT client
    MQTTClient_create(&client, ADDRESS, CLIENTID, MQTTCLIENT_PERSISTENCE_NONE,
NULL);
    //b. Prepare connection options
    conn_opts.keepAliveInterval = 20;
    conn_opts.cleansession = 1;
    //c. Connect to broker with the connection options
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to connect, return code %d\n", rc);
        exit(-1);
    }
    //d. Publish message to topics
    pubmsg.payload = PAYLOAD;
    pubmsg.payloadlen = strlen(PAYLOAD);
    pubmsg.qos = QOS;
    pubmsg.retained = 0;
    MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token);
    printf("Waiting for up to %d seconds for publication of %s\n"
            "on topic %s for client with ClientID: %s\n",
            (int)(TIMEOUT/1000), PAYLOAD, TOPIC, CLIENTID);
    rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
    printf("Message with delivery token %d delivered\n", token);
    //e. Disconnect to broker
    MQTTClient_disconnect(client, 10000);
    //f. Release resources
    MQTTClient_destroy(&client);
    return rc;
}
```

The pubsync.c program details

Each piece of the pubsync.c program is critical to its success. Each part of the code performs the following steps:

1. Include the necessary header files, as shown here. MQTTClient.h has the definitions for MQTT functions and constants.

   ```
   #include "stdio.h"
   #include "stdlib.h"
   #include "string.h"
   #include "MQTTClient.h"
   ```

2. Define the following additional constants:

```
#define ADDRESS     "tcp://localhost:1883"
#define CLIENTID    "Truck01"
#define TOPIC       "routes/Truck01"
#define PAYLOAD     "1332450705462,35.918,-78.231,B"
#define QOS         1
#define TIMEOUT     10000L
```

The following additional constants are defined for this sample program:

`ADDRESS`	The address and port of the message broker.
`CLIENTID`	The unique identifier for the truck.
`TOPIC`	The subject of the message, used for publishing and subscribing. In this case, it defines the message as conveying route information for the truck identified as 01.
`PAYLOAD`	The message being transmitted. In this case, it includes a time stamp (presented in milliseconds since a fixed date in the past), the current longitude and latitude of the truck (presented as decimal values), and the status of the truck (presented as letters representing (B)egin, (T)ransit, (P)ark and (E)nd). The message is shown in text format as a way of keeping this sample code simple and easy to debug.
`QOS`	The QoS designation for the message. In this case, it is set to 1, meaning the message will be sent at least once.
`TIMEOUT`	A built-in delay to await any responses from the server before the program assumes the network is down.

3. Define the necessary local variables, as shown here. `MQTTClientconnectOptionsinitializer` and `MQTTClientmessageinitializer` are macros which are defined in MQTTClient.h and can help a developer avoid having to memorize the meaning of every option.

```
MQTTClient client;
MQTTClientconnectOptions connopts = MQTTClientconnectOptionsinitializer;
MQTTClientmessage pubmsg = MQTTClientmessageinitializer;
MQTTClientdeliveryToken token;
```

4. Create a client instance, as shown in the following example:

```
MQTTClientcreate(&client, ADDRESS, CLIENTID, MQTTCLIENTPERSISTENCENONE, NULL);
connopts.keepAliveInterval = 20;
connopts.cleansession = 1;
```

Client creation involves these parameters:

`&client`	A pointer to a handle for the newly created client. When this function returns with a 0 return code, it contains a handle to the new client.
`ADDRESS`	The URI of the MQTT port that the broker for incoming client connection requests.
`CLIENTID`	The name used to identify the client to the broker. Each active client must have a unique name. If you duplicate a client identifier in two running clients, an exception is thrown in both clients, and one client terminates.
`MQTTCLIENT_PERSISTENCE_NONE`	A specification that the client state is held in memory and will therefore be lost if a system failure occurs. The setting in this case is NULL. One option here would be `MQTTCLIENT_PERSISTENCE_DEFAULT`, which

specifies file system-based persistence and provides some protection against failures. For more specialized applications, `MQTTCLIENT_PERSISTENCE_USER` provides an interface for you to implement your own persistence mechanism. Additional details about persistence for C clients is provided in the API documentation for `MQTTClientPersistence.h`.

5. Configure the connection options that are required by the `MQTTClient_connect` function. In the code shown here, the `connopts` variable is already assigned a value through use of the `MQTTClient_connectOptions_initializer` macro.

```
conn_opts.keepAliveInterval = 20;
conn_opts.cleansession = 1;
```

6. Connect the client to the broker as shown here. After the application connects, you can start publishing and subscribing to topics.

```
if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS) {
     printf("Failed to connect, return code %d\n", rc);
     exit(-1);
}
```

   Certain aspects of the broker connection process deserve further explanation, as follows:

   – The `MQTTClientconnect` function is called, passing the client handle and a pointer to the connection options as arguments.

   – The return code from the `MQTTClientconnect` call is tested to make sure that the connection request is successful. If the `MQTTClientconnect` call fails, the program ends with an error code of -1.

   – A small keep-alive message is sent every 20 seconds to prevent the TCP/IP connection from being closed. This option is set by `conn_opts.keepAliveInterval`.

   – The session is started without checking for the completion of any inflight messages remaining from a previous connection. This is because `conn_opts.cleansession` is set to true. For more details, see 2.3.2, “Client programming” on page 27.

   – No last will and testament message is created for the connection.

7. Populate the `MQTTClientmessage` structure, including the data necessary to define the message payload and its attributes. The example shown here uses a string payload but MQTT payloads are byte arrays. The string length is required to specify the payload size. The example publishes a message with a QoS designation of 1. The retained attribute is set to false (0) because the message is not to be retained.

```
pubmsg.payload = PAYLOAD;
pubmsg.payloadlen = strlen(PAYLOAD);
pubmsg.qos = QOS;
pubmsg.retained = 0;
```

8. Publish the message to the broker, as shown here. Upon completing this step, the message is safely transferred to the MQTT client, but is not yet transferred to the daemon. If the message has a QoS designation of 1 or 2, then the message is stored locally in case the client fails before delivery is completed.

```
MQTTClientpublishMessage(client, TOPIC, &pubmsg, &token);
```

   Portions of the message publishing process warrant further explanation, as follows:

   – The publish function specifies the client, the topic, and the payload to be sent to the broker.

   – The function is also passed a pointer to an MQTT client delivery token. When the function returns, the pointer is populated with a token representing the message.

   - This function returns an error code that you can test for correct completion in production code.

9. Wait for an acknowledgment from the server, as shown here. Because synchronous mode is used to publish here, for messages with QoS designations of 1 and 2, the program will be blocked until it gets the returned information from the broker.

   ```
   rc = MQTTClientwaitForCompletion(client, token, TIMEOUT);
   ```

10. Disconnect the client from the broker. The second argument shown here specifies a timeout in milliseconds; the program waits for up to 10 seconds to finish any other work it must do before disconnecting.

    ```
    MQTTClientdisconnect(client, 10000);
    ```

11. Free up the memory used by the client and end the program, as shown here:

    ```
    MQTTClientdestroy(&client);
    ```

Compiling and verifying pubsync.c

To compile and verify the pubsync.c program, perform these steps:

1. Compile the program in `gcc` using the command syntax shown here. You need to provide the source code directory of the Eclipse Paho C client and the library to the command.

   ```
   gcc -I<source code directory of C client> -L< library directory of C client>
   pubsync.c -lmqttv3c -pthread -o pubsync
   ```

   For example, if the Paho project is in the directory `org.eclipse.paho.mqtt.c`, the following command would initiate the compilation.

   ```
   gcc -Iorg.eclipse.paho.mqtt.c/src -Lorg.eclipse.paho.mqtt.c/build pubsync.c
   -lmqttv3c -pthread -o pubsync
   ```

2. Start the MQTT message broker as described in 2.4, “MQTT brokers” on page 28, and then use the following command to publish the message:

   ```
   LDLIBRARYPATH=org.eclipse.paho.mqtt.c/build ./pubsync
   ```

   After the message is published, a statement such as the following displays, indicating that the message was delivered to the broker successfully:

   ```
   Waiting for up to 10 seconds for publication of 1332450705462,35.918,-78.231,B
   on topic routes/Truck01 for client with ClientID: Truck01
   Message with delivery token 1 delivered
   ```

   If you check the output message from the broker, a message similar to the following displays. This method is another way to verify that the message has arrived at the broker.

   ```
   20120323 051433.570 CWNAN0033I Connection attempt to listener 1883 received
   from client Truck01 on address 127.0.0.1:54267
   20120323 051433.671 CWNAN0038I Disconnection request received from client
   Truck01
   ```

Subscriber

Example 2-13 shows the sample C subscriber, `subasync.c`. It subscribes to the `commands/Truck01` topic and receives any messages that are published to that topic, such as when new instructions are issued from the trucking company’s central headquarters. The message handling logic is contained in the `callback.h` file.

Example 2-13 Sample C subscriber program, subasync.c

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
```

```
#include "MQTTClient.h"
#include "callback.h

#define ADDRESS     "tcp://localhost:1883"
#define CLIENTID    "Truck01"
#define TOPIC       "commands/Truck01"
#define QOS         1
int main(int argc, char* argv[])
{
    MQTTClient client;
    MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
    int rc;
    int ch;
    //a. Create an instance of MQTT client
    MQTTClient_create(&client, ADDRESS, CLIENTID,
        MQTTCLIENT_PERSISTENCE_NONE, NULL);
    //b. Prepare connection options
    conn_opts.keepAliveInterval = 20;
    conn_opts.cleansession = 1;
    MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, delivered);
    //c. Connect to broker with the connection options
    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to connect, return code %d\n", rc);
        exit(-1);
    }
    //d. Subscribe interested topics.
    printf("Subscribing to topic %s\nfor client %s using QoS%d\n\n"
           "Press Q<Enter> to quit\n\n", TOPIC, CLIENTID, QOS);

    MQTTClient_subscribe(client, TOPIC, QOS);

    do
    {
        ch = getchar();
    } while(ch!='Q' && ch != 'q');
    //e. Disconnect to broker
    MQTTClient_disconnect(client, 10000);
    //f. Release resources
    MQTTClient_destroy(&client);
    return rc;
}
```

The subasync.c program details

The subscriber program uses the asynchronous programming model, so it does not need to block execution while waiting for messages to arrive. Message arrival acknowledgments are received in a callback function, which follows the `MQTTClientmessageArrived` template.

Each part of the code performs the following steps. If a particular piece of code has already been explained as part of `pubsync.c`, no additional details are provided here.

1. Include the necessary header files, as shown here. `MQTTClient.h` has the definitions for MQTT functions and constants.

   ```
   #include "stdio.h"
   #include "stdlib.h"
   ```

```
#include "string.h"
#include "MQTTClient.h"
#include "callback.h"
```

2. Define certain additional constants, as shown here. Note that the topic is `commands/Truck01`, which means the subscription is intended only to obtain any new commands that are published for the truck identified as 01.

```
#define ADDRESS     "tcp://localhost:1883"
#define CLIENTID    "Truck01"
#define TOPIC       "commands/Truck01"
#define QOS         1
```

3. Immediately prior to the `MQTTClientconnect` function call, set the callback methods for the client, as shown here. The second argument here would allow you to pass contextual information to the callback functions, but since this function is not applicable to this sample program, it is set to `NULL`.]

```
MQTTClientsetCallbacks(client, NULL, connlost, msgarrvd, delivered);
```

The `connlost`, `msgarrd`, and `delivered` functions implement the callback functions. They are defined in the `callback.h` header file.

– `connlost` is a pointer of type `MQTTClientconnectionLost` and is called when the client loses its connection to the server.

– `msgarrvd` is a pointer of type `MQTTClientmessageArrived` and is called when a message is sent to the client with a matching subscription. It returns a value of true when the message has been successfully received by the client application. A return value of false indicates to the client that the application had a problem receiving the message.

– `delivered` is a pointer of type `MQTTClientdeliveryComplete`. It is for publishing messages only and is called for messages with QoS designations of 1 and 2. The MQTT client must be disconnected when this function is called.

4. Subscribe to the desired topic, as shown here. Use the `MQTTClientsubscribe` function to subscribe the client application to the selected topic.

```
MQTTClientsubscribe(client, TOPIC, QOS);
```

5. Wait in a loop until the user instructs the program to quit (by typing `q` or `Q`). Until the quit command is issued, the program waits for messages to arrive. Message handling takes place in the callback function `MQTTClientmessageArrived`.

```
do
    {
        ch = getchar();
} while(ch!='Q' && ch != 'q');Receiving messages
```

Example 2-14 shows the `callback.h` header file. It defines all callback functions. The file also implements the three callback functions described previously.

Example 2-14 Sample header file, callback.h

```
#include "MQTTClient.h"

volatile MQTTClient_deliveryToken deliveredtoken;

void delivered(void *context, MQTTClient_deliveryToken dt) {
    printf("Message with token value %d delivery confirmed\n", dt);
    deliveredtoken = dt;
}
```

```
int msgarrvd(void *context, char *topicName, int topicLen, MQTTClient_message
*message) {
    int i;
    char* payloadptr;

    printf("Message arrived\n");
    printf("     topic: %s\n", topicName);
    printf("   message: ");

    payloadptr = message->payload;
    for(i=0; i<message->payloadlen; i++) {
        putchar(*payloadptr++);
        }
    putchar('\n');
    MQTTClient_freeMessage(&message);
    free(topicName);
    return 1;
}
void connlost(void *context, char *cause) {
    printf("\nConnection lost\n");
    printf("     cause: %s\n", cause);
}
```

Procedure for msgarrvd in callback.h

There are three callback functions defined in `callback.h`. The `delivered` and `connlost` functions contain only logging messages; a line of logging information is written to the console if a message arrives at the broker or if the connection is lost. When messages arrive from the server, the `msgarrvd` function is started.

To use the `msgarrvd` function, follow these steps:

1. Start the definition of the callback function, as shown here. This definition must match the `MQTTClientmessageArrived` function template.

   ```
   int msgarrvd(void context, char topicName, int topicLen, MQTTClientmessage
   message) {
   ```

   where:

   – `context` provides access to the context passed to the client library when the `MQTTClientsetCallbacks` function was called. This function is not used in this example program.

   – `topicName` is a pointer to the topic to which the received message is published. If you have subscribed using wildcard characters, this parameter identifies the specific topic used for the message.

   – `topicLen` is the length of the topic string. This option is provided for clients who must embed `NULL` characters in topic strings.

   – `message` is a pointer to the `MQTTClientmessage` structure containing the message payload and attributes.

2. Define the local variables to be used, as shown here. These variables are used in the example program to print out the message payload by iterating over it.

   ```
   int i;
   char* payloadptr;\
   ```

3. Print out a message as shown here, displaying the topic and the payload of the message. The sample program simplifies the received payload, which is a sequence of printable characters. Otherwise, it involves simply parsing code to interpret the meaning of a string with the format `time,latitude,longitude,tripStatus`.

```
printf("Message arrived\n");
printf("     topic: %s\n",topicName);
printf("   message: ");
payloadptr = message->payload;
for(i=0; i<message->payloadlen; i++){
    putchar(*payloadptr++);
    }
putchar('\n');\
```

4. Free the memory used to store the message, as shown here. As mentioned previously, when processing is complete, the main program must free the memory used by the message.

```
MQTTClientfreeMessage(&message);
free(topicName);
```

   When performing this step, keep these things in mind:

   – Ensure that the callback functions are short, and return control to its calling thread as soon as possible.
   – The message pointer is passed for handling in the main part of the program.
   – `MQTTClientfreeMessage()` is a convenience function that returns the two memory blocks used to hold the `MQTTClientmessage` structure and the message payload back to the system.
   – The memory allocated to the `topicName` must be freed separately as shown.

5. Return a true value, as shown here, when the callback has successfully handled the message

```
return 1;
}
```

   When performing this step, keep these things in mind:

   – Returning a true value indicates that the client library can treat the message as successfully delivered.
   – If the callback function cannot properly process the message, a false value is returned. For example, if the callback is putting messages onto a queue for the main program to process and the queue is full, returning false would be appropriate.
   – For QoS 1 and 2 messages, returning a false value indicates that the message was not delivered and further attempts to deliver it are made.

Compiling and verifying subasync.c

To compile and verify the `subasync.c` program, perform these steps:

1. Compile the program in `gcc` using the command syntax shown here. You need to provide the source code directory of the Eclipse Paho C client and the library to the command.

```
gcc -I<source code directory of C client> -L< library directory of C client>
subasync -lmqttv3c -pthread -o subasync
```

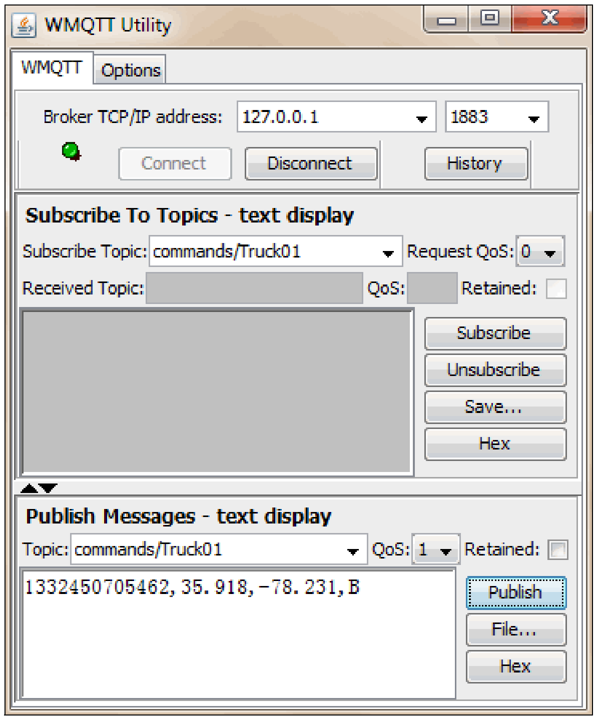
   For example, if the Paho project is in the directory `org.eclipse.paho.mqtt.c`, then the following command would initiate the compilation of *subasync.c*.

```
gcc -Iorg.eclipse.paho.mqtt.c/src -Lorg.eclipse.paho.mqtt.c/build subasync.c
-lmqttv3c -pthread -o subasync
```

2. Start the MQTT message broker as described in 2.4, "MQTT brokers" on page 28, then use the command shown here to publish the message.

   ```
   LDLIBRARYPATH=org.eclipse.paho.mqtt.c/build ./subasync
   ```

3. Start the WMQTT Utility (available in SupportPac IA-92) and publish a message to the `command/Truck01` topic string. Refer to Figure 2-8.



Broker TCP/IP address: 127.0.0.1 1883

Subscribe Topic: commands/Truck01 Request QoS: 0

Received Topic: QoS: Retained:

Topic: commands/Truck01 QoS: 1 Retained:

```
1332450705462,35.918,-78.231,B
```

Figure 2-8 Publishing a message with the WMQTT Utility

After publishing the message, you should receive a command console notification such as the message shown in Example 2-15, which means that you successfully subscribed to the `commands/Truck01` topic and received the test message that you just published.

Example 2-15 Successfully subscribed

```
Subscribing to topic commands/Truck01
for client Truck01 using QoS1
Press Q<Enter> to quit

Message arrived
     topic: commands/Truck01
message: 1332450705462,35.918,-78.231,B
```

3

Using MQTT with IBM WebSphere MQ Telemetry

WebSphere MQ Telemetry provides reliable, secure messaging features that can be scaled to include thousands of clients and that can interface with other technology within the enterprise, such as WebSphere Message Broker and WebSphere Application Server. WebSphere MQ Telemetry extends the reach of WebSphere MQ to connect the smallest and most remote devices and sensors to the enterprise messaging network.

This chapter describes a simple MQTT solution using WebSphere MQ and the interoperability capabilities of MQTT with JMS messaging clients. For the examples shown in this chapter and throughout this book (unless otherwise noted), the WebSphere MQ software is installed and configured on a Linux 64-bit system.

This chapter contains the following sections:

- Installing, configuring, and managing WebSphere MQ Telemetry
- Building a simple solution using WebSphere MQ Telemetry
- WebSphere MQ classes for Java Message Service
- WebSphere MQ Telemetry daemon for devices
- Troubleshooting WebSphere MQ Telemetry

3.1 Installing, configuring, and managing WebSphere MQ Telemetry

At the time of writing, IBM offers both WebSphere MQ V7.0.1, in which MQTT protocol support is provided as an additional installation, and WebSphere MQ V7.1, in which WebSphere MQ Telemetry support is included as part of the product installation. For the purposes of this analysis, in this section we explain how to install and verify, configure, and manage MQ Telemetry as part of WebSphere MQ V7.1.

3.1.1 Before you begin

With WebSphere MQ V7.1, the MQ Telemetry component is available either as a server or a client feature.

Tip: You can download a no-cost WebSphere MQ client as SupportPac MQC71 from the MQ SupportPacs website at:

http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24031412

SupportPac MQC71 includes the telemetry `MQSeriesXRClients-7.1.0-0.x8664.rpm` component for Linux or `mqm.xr.client` component for IBM AIX®. Java and C MQTT client samples are located in the `<`*`MQ Installation path`*`>\mqxr\SDK` directory.

Prerequisites: Before you install the WebSphere MQ server, review the hardware and software requirements located at:

https://www-304.ibm.com/support/docview.wss?rs=171&uid=swg27006467

Prior to installing WebSphere MQ, consider the following recommended first steps:

- Create an mqm group and an mqm ID. Be sure the primary group of the mqm ID is the mqm group that was created. Do not add the mqm ID to other groups such as staff or users.
- Create a file system for the product code (example: `/opt/mqm/` on Linux) and the working data (example: `/var/mqm`).
- Set system kernel tuning parameters as outlined in the WebSphere MQ Quick Beginnings Guide at:

  http://publib.boulder.ibm.com/infocenter/wmqv7/v7r0/topic/com.ibm.mq.amq1ac.doc/lq10120_.htm

  The `mqconfig` script can be downloaded and run on the system to verify that the kernel tuning parameters are set to meet the minimum recommendation for WebSphere MQ. Find it at:

  http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg21271236
- On a Windows operating system, the WebSphere MQ Telemetry components will be installed unless you choose the option to not install the WebSphere MQ Telemetry libraries, provided you have the prerequisites on your system. If you do not have the prerequisites, the components cannot be installed. On Linux, the MQ Telemetry feature is installed by specifying the telemetry component specifically or choosing to install all MQ components.

  Information about the prerequisites for installing WebSphere MQ Telemetry is available at:

  http://www-01.ibm.com/software/integration/wmqfamily/telemetry/requirements/

Important: At the time of writing, the WebSphere MQ software for a 32-bit Linux system does not contain the `MQSeriesXRService` and `MQSeriesXRClients` components for MQTT.

3.1.2 Installing WebSphere MQ

The steps outlined here are for WebSphere MQ V7.1 on a 64-bit Linux system. Details and installation instructions for other platforms are available in the WebSphere MQ V7.1 Information Center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/zi00010_.htm

To install WebSphere MQ server software with the MQ Telemetry feature:

1. Log in as root, or switch to the superuser by using the **`su`** command.
2. Set your current directory to the location of the installation file. The location might be the mount point of the server CD, a network share, or a local file system directory.
3. Accept the software license:
   a. To view and accept the license, run the script `./mqlicense.sh`.
   b. To just view the license, run `./mqlicense.sh -textonly`.
   c. To accept the license without it being displayed, run `./mqlicense.sh -accept`.
4. Install WebSphere MQ. For the purposes of this project, the defaults are installed, including the telemetry service and telemetry clients. The components and package names are listed in the WebSphere MQ Information Center at:

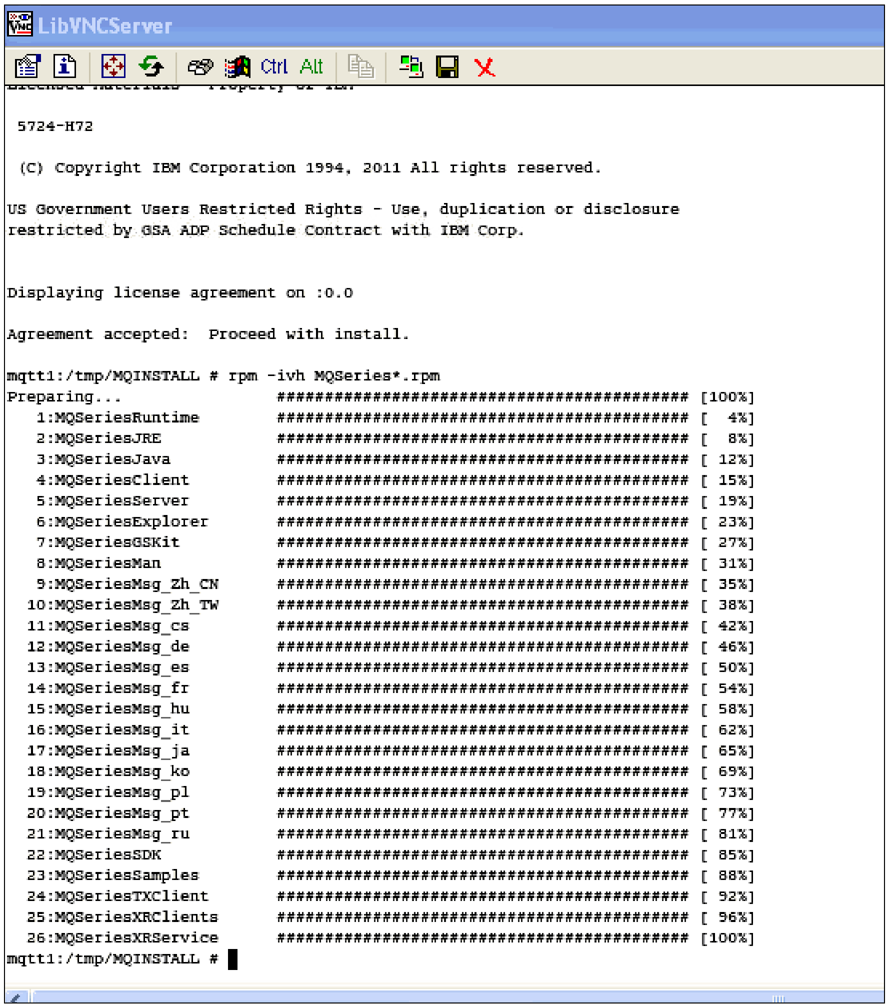
   http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/zi00920_.htm

   To install selected components to the default location, `/opt/mqm`, enter the following command and specify the components you want to install:

   ```
   rpm -ivh
   ```

   To install all components to the default location, as shown in Figure 3-1 on page 58, enter the following command:

   ```
   rpm -ivh MQSeries.rpm
   ```

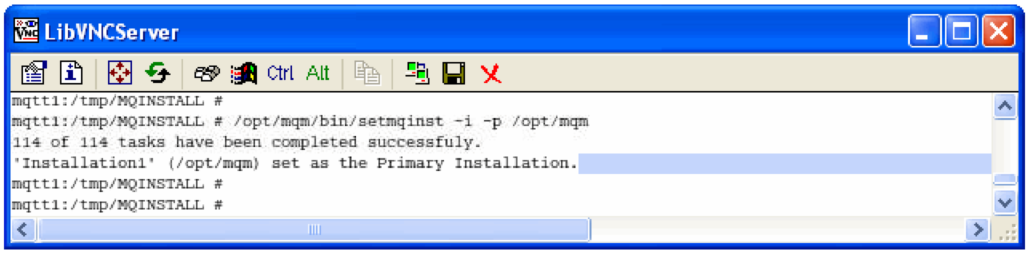


```
Licensed Materials - Property of IBM.

 5724-H72

 (C) Copyright IBM Corporation 1994, 2011 All rights reserved.

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


Displaying license agreement on :0.0

Agreement accepted:  Proceed with install.

mqtt1:/tmp/MQINSTALL # rpm -ivh MQSeries*.rpm
Preparing...                ########################################### [100%]
   1:MQSeriesRuntime        ########################################### [  4%]
   2:MQSeriesJRE            ########################################### [  8%]
   3:MQSeriesJava           ########################################### [ 12%]
   4:MQSeriesClient         ########################################### [ 15%]
   5:MQSeriesServer         ########################################### [ 19%]
   6:MQSeriesExplorer       ########################################### [ 23%]
   7:MQSeriesGSKit          ########################################### [ 27%]
   8:MQSeriesMan            ########################################### [ 31%]
   9:MQSeriesMsg_Zh_CN      ########################################### [ 35%]
  10:MQSeriesMsg_Zh_TW      ########################################### [ 38%]
  11:MQSeriesMsg_cs         ########################################### [ 42%]
  12:MQSeriesMsg_de         ########################################### [ 46%]
  13:MQSeriesMsg_es         ########################################### [ 50%]
  14:MQSeriesMsg_fr         ########################################### [ 54%]
  15:MQSeriesMsg_hu         ########################################### [ 58%]
  16:MQSeriesMsg_it         ########################################### [ 62%]
  17:MQSeriesMsg_ja         ########################################### [ 65%]
  18:MQSeriesMsg_ko         ########################################### [ 69%]
  19:MQSeriesMsg_pl         ########################################### [ 73%]
  20:MQSeriesMsg_pt         ########################################### [ 77%]
  21:MQSeriesMsg_ru         ########################################### [ 81%]
  22:MQSeriesSDK            ########################################### [ 85%]
  23:MQSeriesSamples        ########################################### [ 88%]
  24:MQSeriesTXClient       ########################################### [ 92%]
  25:MQSeriesXRClients      ########################################### [ 96%]
  26:MQSeriesXRService      ########################################### [100%]
mqtt1:/tmp/MQINSTALL #
```

Figure 3-1   Installing WebSphere MQ components

5. If you have chosen this installation to be the primary installation on the system, declare it to be the primary installation, as shown in Figure 3-2, by entering the following command at the command prompt:

   ```
   MQINSTALLATIONPATH/bin/setmqinst -i -p MQINSTALLATIONPATH
   ```



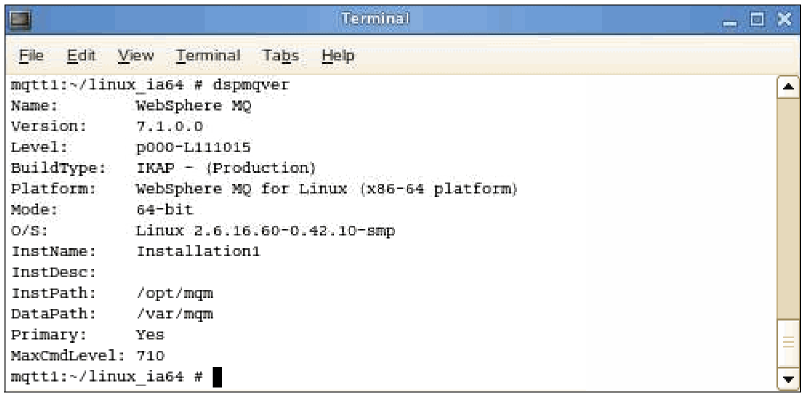
```
mqtt1:/tmp/MQINSTALL #
mqtt1:/tmp/MQINSTALL # /opt/mqm/bin/setmqinst -i -p /opt/mqm
114 of 114 tasks have been completed successfuly.
'Installation1' (/opt/mqm) set as the Primary Installation.
mqtt1:/tmp/MQINSTALL #
mqtt1:/tmp/MQINSTALL #
```

Figure 3-2   Setting MQ instance as primary

Remember: You can have only one primary installation on a system. If there is already a primary installation on the system, you must unset it before you can set another installation as the primary installation. Use the **`setmqinst`** command to set or unset an installation as the primary one.

If an installation of WebSphere MQ V7.1 or later is coexisting with an installation of WebSphere MQ V7.0.1, the WebSphere MQ V7.0.1 installation must be the primary. It is flagged as primary when the WebSphere MQ V7.1 or later version is installed, and the WebSphere MQ V7.1 or later installation cannot be made primary.

6. After installation, you can use the **`dspmqver`** command to show the WebSphere MQ version, as depicted in Figure 3-3, which shows WebSphere MQ Version 7.1.



```
mqtt1:~/linux_ia64 # dspmqver
Name:        WebSphere MQ
Version:     7.1.0.0
Level:       p000-L111015
BuildType:   IKAP - (Production)
Platform:    WebSphere MQ for Linux (x86-64 platform)
Mode:        64-bit
O/S:         Linux 2.6.16.60-0.42.10-smp
InstName:    Installation1
InstDesc:
InstPath:    /opt/mqm
DataPath:    /var/mqm
Primary:     Yes
MaxCmdLevel: 710
mqtt1:~/linux_ia64 #
```

Figure 3-3 Displaying MQ version

3.1.3 Verifying the installation of WebSphere MQ Telemetry

After you install the WebSphere MQ software, you can verify the installation of WebSphere MQ Telemetry using the define sample configuration wizard and the MQTT client utility in WebSphere MQ Explorer. WebSphere MQ Explorer is a GUI interface that is included with WebSphere MQ. It can also be downloaded at no cost in SupportPac MS0T at:

`http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg24021041`

For this example, we are creating a queue manager called MQTTVerification to test the configuration of WebSphere MQ Telemetry. WebSphere MQ also includes a sample script (`SampleMQM.sh`) that can be used to manually create an MQTT-enabled queue manager for verification. For more information about a queue manager using the sample script, refer to 3.1.4, “Configuring a queue manager for WebSphere MQ Telemetry” on page 65.

Using WebSphere MQ Explorer

To verify the installation of WebSphere MQ Telemetry using WebSphere MQ Explorer:

1. Start WebSphere MQ Explorer (illustrated in Figure 3-4). On Windows and Linux operating systems, you can start WebSphere MQ Explorer using the system menu, the MQExplorer executable file, the **`mqexplorer`** command, or the **`strmqcfg`** command.

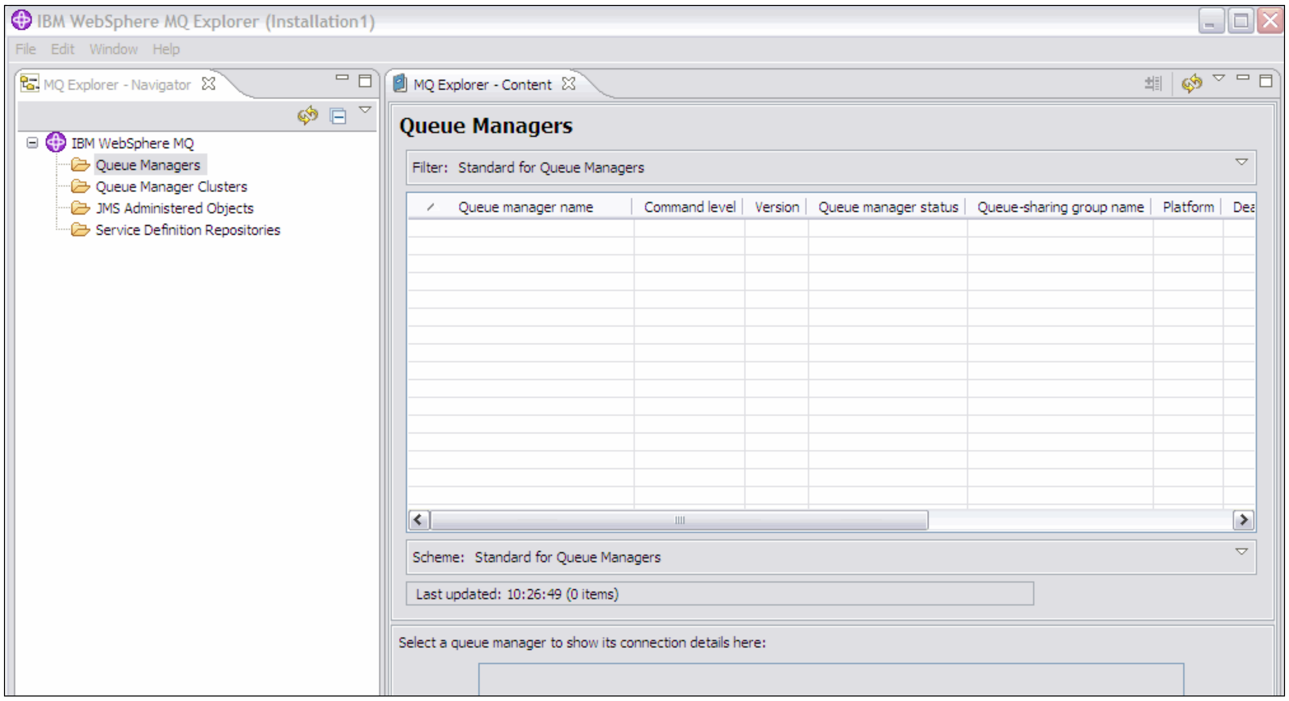
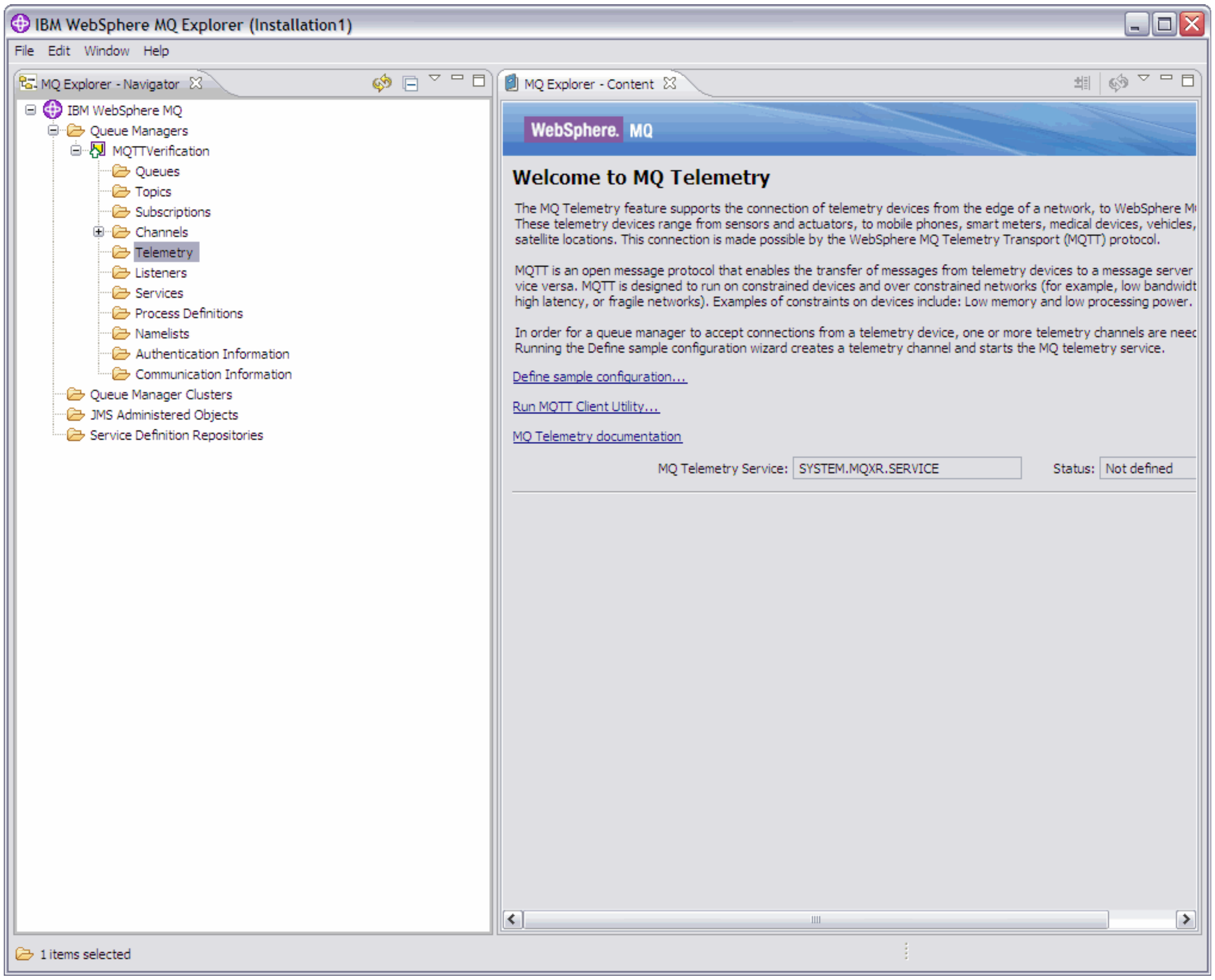


Figure 3-4 The MQ Explorer GUI

2. Create a new queue manager:
   a. Right-click **Queue Managers** in MQ Explorer Navigator, and select **New** → **Queue Manager**.
   b. Type `MQTTVerification` as the Queue manager name.
   c. Click **Next** → **Next** → **Next**.
   d. Confirm the port that is listed in the “listen on port number” field. If the port listed there is already in use by another queue manager, you will need to change it.
   e. Click **Finish**.
3. Configure a new queue manager:
   a. In MQ Explorer Navigator select **IBM WebSphere MQ** → **Queue Managers** → **MQTTVerification**.
   b. Open the `Telemetry` folder to open the Welcome to MQ Telemetry content window (as shown in Figure 3-5 on page 61). The queue manager must be running for the Telemetry folder to be viewable.



Welcome to MQ Telemetry

The MQ Telemetry feature supports the connection of telemetry devices from the edge of a network, to WebSphere M
These telemetry devices range from sensors and actuators, to mobile phones, smart meters, medical devices, vehicles,
satellite locations. This connection is made possible by the WebSphere MQ Telemetry Transport (MQTT) protocol.

MQTT is an open message protocol that enables the transfer of messages from telemetry devices to a message server
vice versa. MQTT is designed to run on constrained devices and over constrained networks (for example, low bandwidt
high latency, or fragile networks). Examples of constraints on devices include: Low memory and low processing power.

In order for a queue manager to accept connections from a telemetry device, one or more telemetry channels are neec
Running the Define sample configuration wizard creates a telemetry channel and starts the MQ telemetry service.

Define sample configuration...

Run MQTT Client Utility...

MQ Telemetry documentation

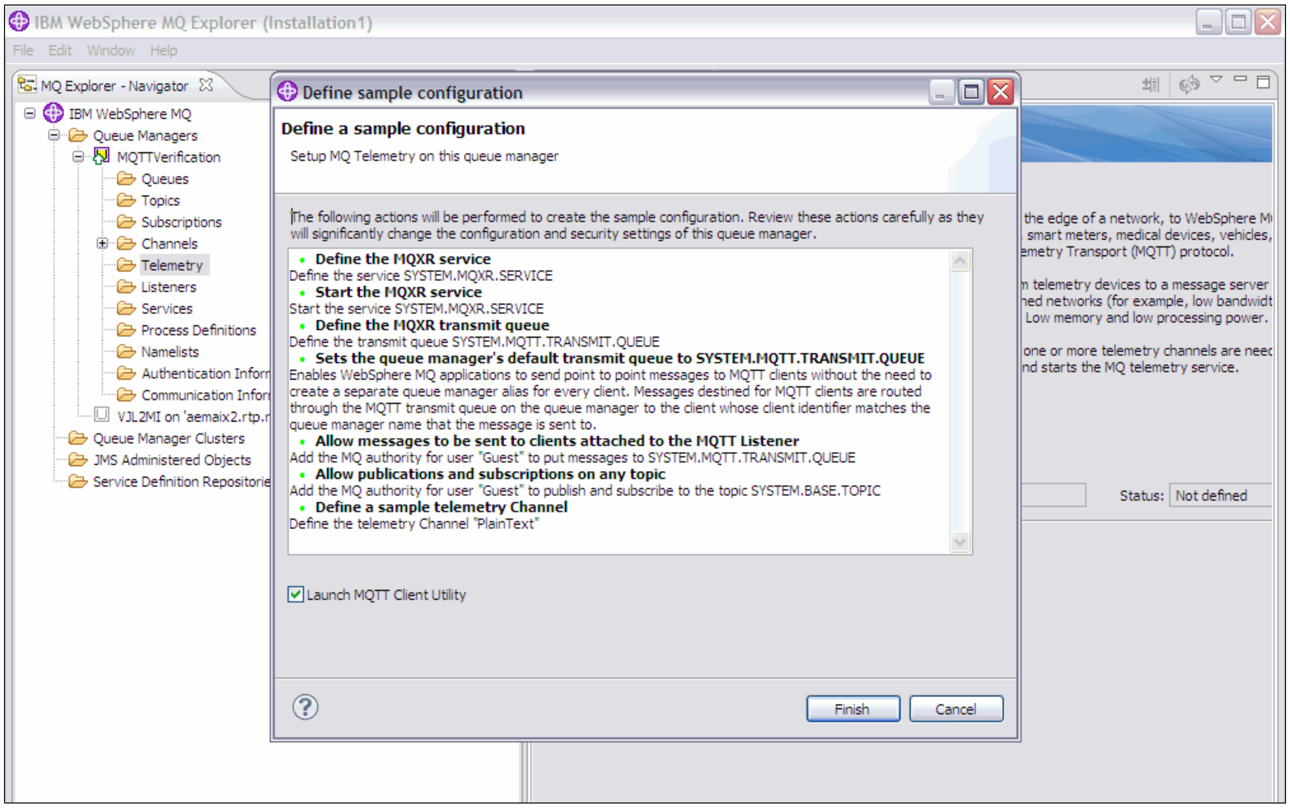
MQ Telemetry Service: SYSTEM.MQXR.SERVICE Status: Not defined

Figure 3-5 MQ Telemetry Welcome panel

c. Click **Define sample configuration** to open a page that lists steps that will be performed automatically as part of the sample configuration (illustrated in Figure 3-6 on page 62). If you do not want to accept these default steps, you can use the sample `SampleMQM.sh` configuration file.

For more information about manual configuration, see 3.1.4, “Configuring a queue manager for WebSphere MQ Telemetry” on page 65.

If the Define sample configuration link is not present and if instead you see the message `the sample configuration has been set up for this queue manager`, telemetry is already configured. If this is the case, you can launch the MQ Client Utility panel from the Welcome panel and continue to step 4.



Define a sample configuration

Setup MQ Telemetry on this queue manager

The following actions will be performed to create the sample configuration. Review these actions carefully as they will significantly change the configuration and security settings of this queue manager.

- **Define the MQXR service**
  Define the service SYSTEM.MQXR.SERVICE
- **Start the MQXR service**
  Start the service SYSTEM.MQXR.SERVICE
- **Define the MQXR transmit queue**
  Define the transmit queue SYSTEM.MQTT.TRANSMIT.QUEUE
- **Sets the queue manager's default transmit queue to SYSTEM.MQTT.TRANSMIT.QUEUE**
  Enables WebSphere MQ applications to send point to point messages to MQTT clients without the need to create a separate queue manager alias for every client. Messages destined for MQTT clients are routed through the MQTT transmit queue on the queue manager to the client whose client identifier matches the queue manager name that the message is sent to.
- **Allow messages to be sent to clients attached to the MQTT Listener**
  Add the MQ authority for user "Guest" to put messages to SYSTEM.MQTT.TRANSMIT.QUEUE
- **Allow publications and subscriptions on any topic**
  Add the MQ authority for user "Guest" to publish and subscribe to the topic SYSTEM.BASE.TOPIC
- **Define a sample telemetry Channel**
  Define the telemetry Channel "PlainText"

[✓] Launch MQTT Client Utility

Figure 3-6 Using WebSphere MQ Explorer to define a sample configuration

Tip: The default authorization setting on Linux for full access to MQ resources (using either the sample telemetry script or the sample telemetry setup wizard in MQ Explorer) is `nobody`. By giving access to user `nobody`, you are effectively giving all users the same authority.

Important: The queue manager being created here is just to verify that telemetry is working. To eventually test telemetry, you will need to create a different queue manager that does not carry the default authorization setting of `nobody`.

You can edit the `SampleMQM.sh` script manually to remove or alter the **`setmqaut`** commands. This sample script can be found in the `/opt/mqm/mqxr/samples` directory. More information about security is provided in 3.1.5, "Authorizing MQTT clients to access WebSphere MQ" on page 72 and in 3.1.6, "Enabling the security features of WebSphere MQ for telemetry channels" on page 74.

d. Confirm whether you want to automatically launch the MQTT Client Utility panel when you are finished setting up the configuration. The MQTT Client Utility panel can be used to subscribe and publish test messages. The Launch MQTT Client Utility check box is selected by default. If you clear it but later want to open the panel, you can do so using a direct link located on the Welcome to MQ Telemetry panel (as shown in Figure 3-5 on page 61).

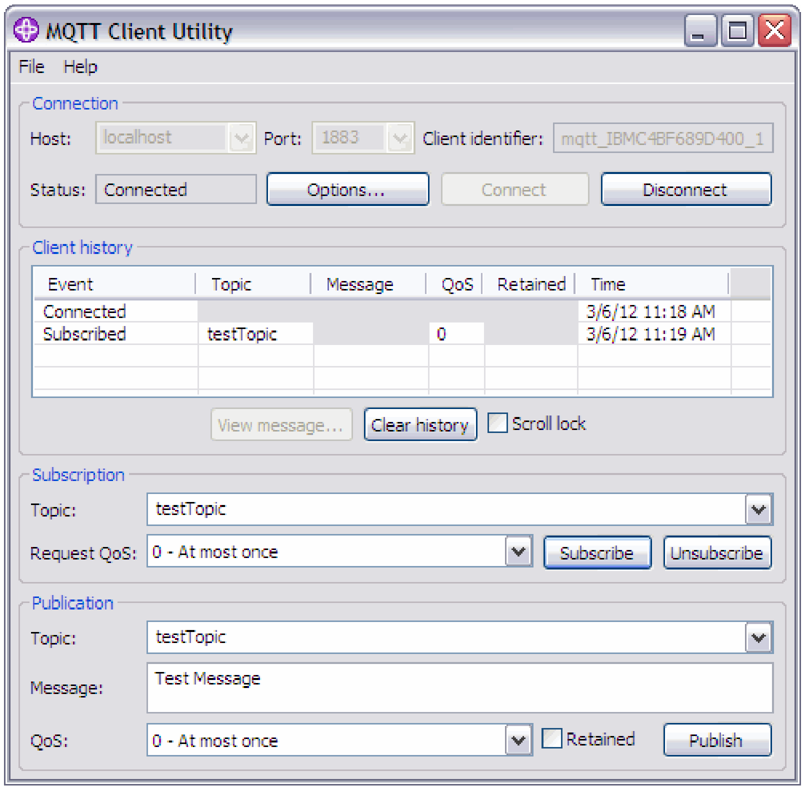
e. Click **Finish**. This completes the configuration of WebSphere MQ Telemetry.

4. Verify that WebSphere MQ Telemetry is installed using the MQTT Client Utility to perform a test:

   a. Confirm that the host and port names shown on the MQTT Client Utility panel are correct.

   b. Click **Connect**. A connected event displays under client history.

   c. Click **Subscribe**. A subscribed displays under client history (see Figure 3-7).



MQTT Client Utility

File Help

Connection

Host: localhost Port: 1883 Client identifier: mqtt_IBMC4BF689D400_1

Status: Connected

Client history

Event	Topic	Message	QoS	Retained	Time
Connected					3/6/12 11:18 AM
Subscribed	testTopic		0		3/6/12 11:19 AM

Subscription

Topic: testTopic

Request QoS: 0 - At most once

Publication

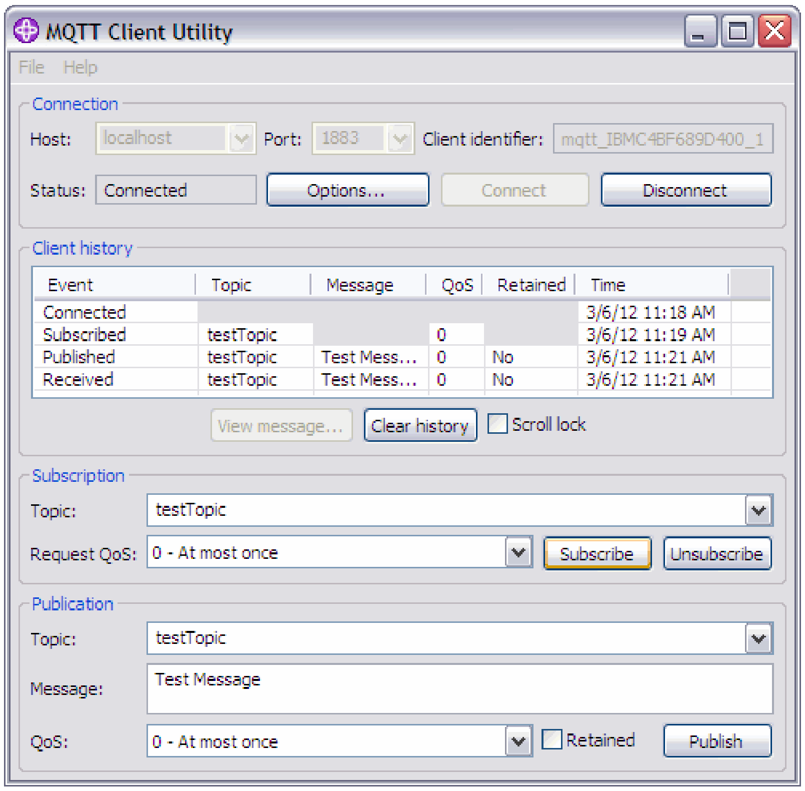
Topic: testTopic

Message: Test Message

QoS: 0 - At most once

Figure 3-7 Testing a subscription using the MQTT Client Utility

d. Click **Publish**. Both a published and a received event displays under client history (see Figure 3-8).



Host: localhost
Port: 1883
Client identifier: mqtt_IBMC4BF689D400_1
Status: Connected

Event	Topic	Message	QoS	Retained	Time
Connected					3/6/12 11:18 AM
Subscribed	testTopic		0		3/6/12 11:19 AM
Published	testTopic	Test Mess...	0	No	3/6/12 11:21 AM
Received	testTopic	Test Mess...	0	No	3/6/12 11:21 AM

Subscription
Topic: testTopic
Request QoS: 0 - At most once

Publication
Topic: testTopic
Message: Test Message
QoS: 0 - At most once
Retained: ☐

Figure 3-8 Testing publication using the MQTT Client Utility

The WebSphere MQ Telemetry installation is verified if the publish/subscribe processes complete successfully and the client history portion of the MQTT Client Utility panel shows that the test publication message was received.

If you encounter problems during the installation process, you can view the error log at the following locations:

- On a Windows operating system, `<`*`WebSphere MQ data directory`*`>\qmgrs\`*`qMgrName`*`\mqxr`
- On AIX and Linux systems, `/var/mqm/qmgrs/`*`qMgrName`*`/mqxr/`

Using the WebSphere MQ Telemetry client samples

WebSphere MQ Telemetry client samples can be used for additional testing or application design guidance.

Telemetry client code is provided in the `mqxr\SDK\clients` folder in the WebSphere MQ installation. The C and Java client implementations are based on the MQTT V3.1 protocol and can be used to develop applications with any server that supports V3.1 of the MQTT specification.

The Javadoc information for the WebSphere MQ Telemetry Java client is available in `mqxr\SDK\clients\java\doc\javadoc`. The `index.html` file contains some basic examples of how to use Java classes to design application code for connecting to a server that implements the MQTT V3 protocol. You can find more details about how to develop applications that use WebSphere MQ Telemetry at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/index.jsp?topic=%2Fcom.ibm.mq.doc%2Ftt60000_.htm

3.1.4 Configuring a queue manager for WebSphere MQ Telemetry

In this section we describe how to manually configure a queue manager to run WebSphere MQ Telemetry. Configuring a queue manager manually allows you to customize settings for testing and production scenarios.

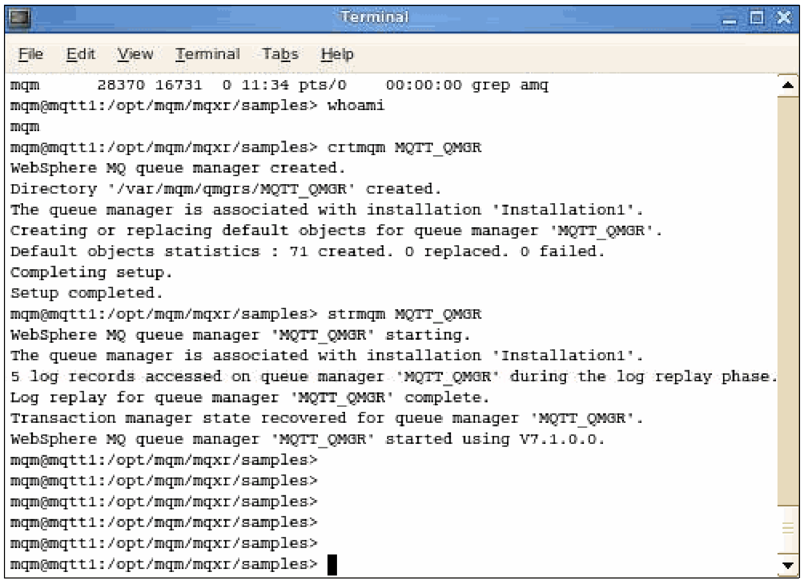
You can also use the sample configuration wizard in WebSphere MQ Explorer, as described in 3.1.3, “Verifying the installation of WebSphere MQ Telemetry” on page 59, or the sample `SampleMQM.sh` script that is provided with WebSphere MQ. The sample script is located in the `/opt/mqm/mqxr/samples` directory on a Linux server or the `C:\Program Files\IBM\WebSphere MQ\mqxr\samples` directory in a Windows operating system. You can edit the script to alter the queue manager name, channel parameters, and authority settings.

Create and start the queue manager

To begin the manual configuration process, create and start a queue manager using the **`crtmqm`** and **`strmqm`** commands. The queue manager name here is `MQTTQMGR`. The following commands are how they are exactly entered:

```
crtmqm MQTTQMGR
strmqm MQTTQMGR
```

Figure 3-9 shows the full command sequence and responses.



```
mqm      28370 16731  0 11:34 pts/0    00:00:00 grep amq
mqm@mqtt1:/opt/mqm/mqxr/samples> whoami
mqm
mqm@mqtt1:/opt/mqm/mqxr/samples> crtmqm MQTT_QMGR
WebSphere MQ queue manager created.
Directory '/var/mqm/qmgrs/MQTT_QMGR' created.
The queue manager is associated with installation 'Installation1'.
Creating or replacing default objects for queue manager 'MQTT_QMGR'.
Default objects statistics : 71 created. 0 replaced. 0 failed.
Completing setup.
Setup completed.
mqm@mqtt1:/opt/mqm/mqxr/samples> strmqm MQTT_QMGR
WebSphere MQ queue manager 'MQTT_QMGR' starting.
The queue manager is associated with installation 'Installation1'.
5 log records accessed on queue manager 'MQTT_QMGR' during the log replay phase.
Log replay for queue manager 'MQTT_QMGR' complete.
Transaction manager state recovered for queue manager 'MQTT_QMGR'.
WebSphere MQ queue manager 'MQTT_QMGR' started using V7.1.0.0.
mqm@mqtt1:/opt/mqm/mqxr/samples>
mqm@mqtt1:/opt/mqm/mqxr/samples>
mqm@mqtt1:/opt/mqm/mqxr/samples>
mqm@mqtt1:/opt/mqm/mqxr/samples>
mqm@mqtt1:/opt/mqm/mqxr/samples>
mqm@mqtt1:/opt/mqm/mqxr/samples>
```

Figure 3-9 Creating and starting a WebSphere MQ Telemetry queue manager

Configure the queue manager

After you create and start the queue manager, you can manually configure it to run WebSphere MQ Telemetry support using **`runmqsc`** commands to define the objects.

Important: The **runmqsc** command is a scripting language for WebSphere MQ. It accepts commands from the command-line interface from which it is executed. To start an interactive session against a queue manager, use the following syntax:

```
runmqsc QueueManagerName
```

When finished configuring the queue manager, type `end` to exit the **runmqsc** session. If a set of MQSC commands has been saved in a file, the contents of this file can be passed into the **runmqsc** program using the following syntax:

```
runmqsc QueueManagerName <filename>
```

To configure the queue manager using **runmqsc**, perform these steps:

1. Open a command window at the telemetry `samples` directory. The telemetry `samples` directory on Linux is `/opt/mqm/mqxr/samples`.
2. Create the telemetry transmission queue using the following **runmqsc** command.

   ```
   DEFINE QLOCAL('SYSTEM.MQTT.TRANSMIT.QUEUE') USAGE(XMITQ) MAXDEPTH(100000)
   ```

   You receive the following confirmation message:

   ```
   AMQ8006: WebSphere MQ queue created
   ```

3. Set the default transmission queue to the one you just created by using the **runmqsc** command:

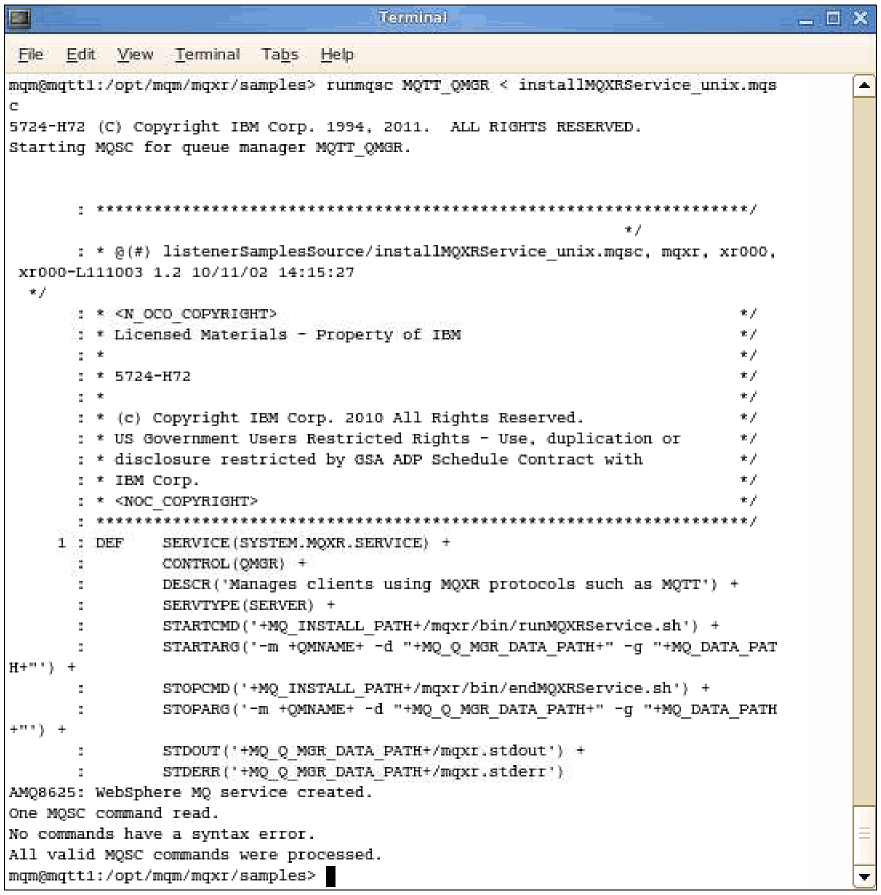
   ```
   ALTER QMGR DEFXMITQ('SYSTEM.MQTT.TRANSMIT.QUEUE')
   ```

   Remember: Altering the default transmission queue might interfere with your existing configuration.

   The reason for altering the queue manager to set the default transmission queue to `SYSTEM.MQTT.TRANSMIT.QUEUE` is to make sending messages directly to MQTT clients easier. Unless the default transmission queue is changed, you will have to add a remote queue for every client that receives WebSphere MQ messages. For more information about sending a message to a client directly using remote queues, see:

   http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt40150_.htm

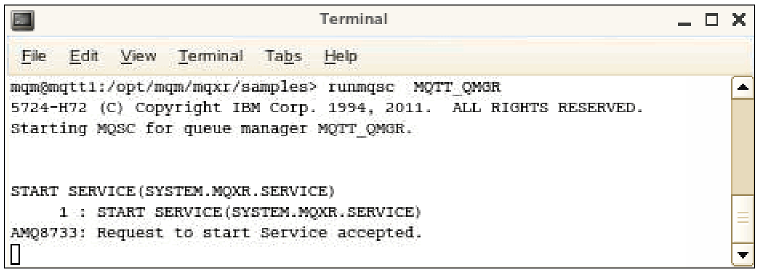
4. Install the telemetry service by reading the `installMQXRServiceunix.mqsc` script into **`runmqsc`**. To do this, enter the **`runmqsc < installMQXRServiceunix.mqsc`** command to read the commands in the `installMQXRServiceunix.mqsc` file into the **`runmqsc`** program to define the telemetry service (`SYSTEM.MQXR.SERVICE`), as shown in Figure 3-10.



```
mqm@mqtt1:/opt/mqm/mqxr/samples> runmqsc MQTT_QMGR < installMQXRService_unix.mqs
c
5724-H72 (C) Copyright IBM Corp. 1994, 2011.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager MQTT_QMGR.


       : ******************************************************************/
                                                                      */
       : * @(#) listenerSamplesSource/installMQXRService_unix.mqsc, mqxr, xr000,
 xr000-L111003 1.2 10/11/02 14:15:27
  */
       : * <N_OCO_COPYRIGHT>                                              */
       : * Licensed Materials - Property of IBM                           */
       : *                                                                */
       : * 5724-H72                                                       */
       : *                                                                */
       : * (c) Copyright IBM Corp. 2010 All Rights Reserved.              */
       : * US Government Users Restricted Rights - Use, duplication or    */
       : * disclosure restricted by GSA ADP Schedule Contract with        */
       : * IBM Corp.                                                      */
       : * <NOC_COPYRIGHT>                                                */
       : ******************************************************************/
     1 : DEF    SERVICE(SYSTEM.MQXR.SERVICE) +
       :        CONTROL(QMGR) +
       :        DESCR('Manages clients using MQXR protocols such as MQTT') +
       :        SERVTYPE(SERVER) +
       :        STARTCMD('+MQ_INSTALL_PATH+/mqxr/bin/runMQXRService.sh') +
       :        STARTARG('-m +QMNAME+ -d "+MQ_Q_MGR_DATA_PATH+" -g "+MQ_DATA_PAT
H+"') +
       :        STOPCMD('+MQ_INSTALL_PATH+/mqxr/bin/endMQXRService.sh') +
       :        STOPARG('-m +QMNAME+ -d "+MQ_Q_MGR_DATA_PATH+" -g "+MQ_DATA_PATH
+"') +
       :        STDOUT('+MQ_Q_MGR_DATA_PATH+/mqxr.stdout') +
       :        STDERR('+MQ_Q_MGR_DATA_PATH+/mqxr.stderr')
AMQ8625: WebSphere MQ service created.
One MQSC command read.
No commands have a syntax error.
All valid MQSC commands were processed.
mqm@mqtt1:/opt/mqm/mqxr/samples>
```

Figure 3-10 Creating the telemetry service using the `installMQXRService` script

5. Start the service using the START SERVICE(SYSTEM.MQXR.SERVICE) **`runmqsc`** command as shown in Figure 3-11.



```
mqm@mqtt1:/opt/mqm/mqxr/samples> runmqsc  MQTT_QMGR
5724-H72 (C) Copyright IBM Corp. 1994, 2011.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager MQTT_QMGR.


START SERVICE(SYSTEM.MQXR.SERVICE)
     1 : START SERVICE(SYSTEM.MQXR.SERVICE)
AMQ8733: Request to start Service accepted.
```

Figure 3-11 Starting the telemetry service

Configure the telemetry channels

After you configure the queue manager with the WebSphere MQ Telemetry service, you can use the WebSphere MQ Explorer GUI to configure its telemetry channels. These channels are used by the MQTT clients to connect to the queue manager. In this example, the channel is configured with the mqttuser ID, which requires the mqttuser ID to have been previously created on the server by the administrator. For purposes of this example, the mqttuser ID was created as part of a user group called *mqtt*.

To create the WebSphere MQ Telemetry channels using WebSphere MQ Explorer:

1. Open the `Telemetry` folder in MQ Explorer Navigator and right-click **Channel**. Then select **New** → **Telemetry Channel** as shown in Figure 3-12.

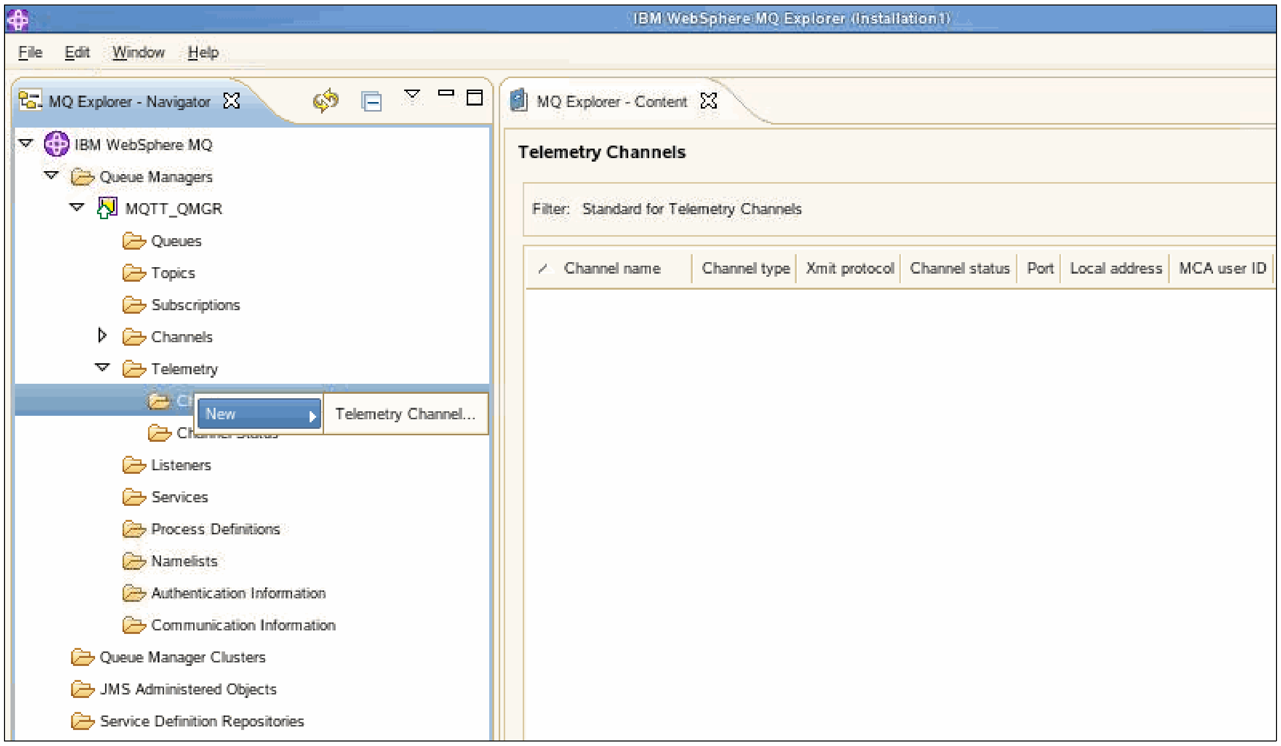
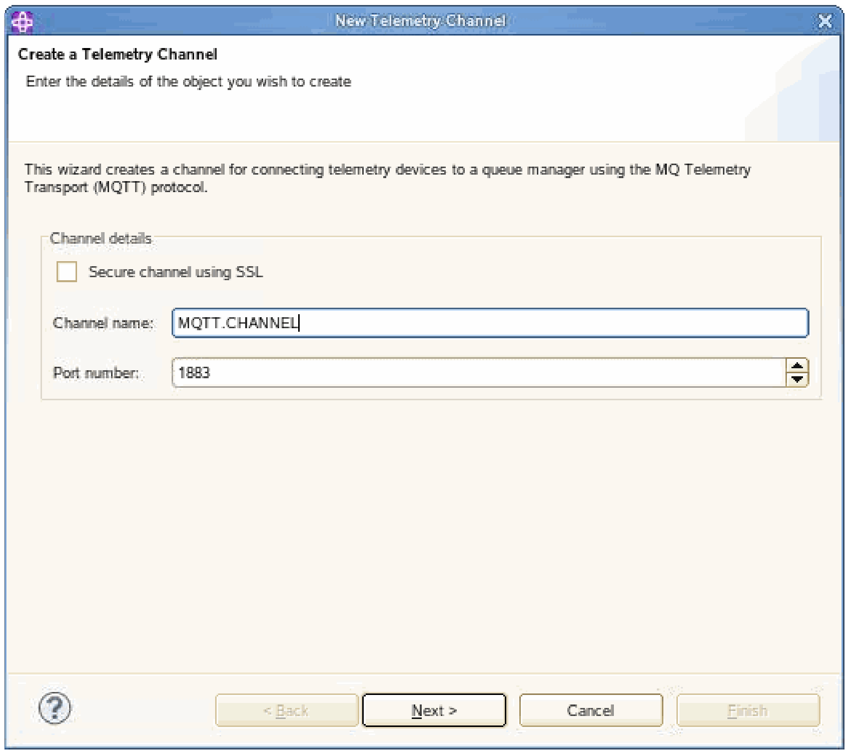


Figure 3-12 Creating a new telemetry channel

2. Next, specify the channel name and select the port number, and then click **Next** as shown in Figure 3-13.



Create a Telemetry Channel

Enter the details of the object you wish to create

This wizard creates a channel for connecting telemetry devices to a queue manager using the MQ Telemetry Transport (MQTT) protocol.

Channel details

☐ Secure channel using SSL

Channel name: MQTT.CHANNEL

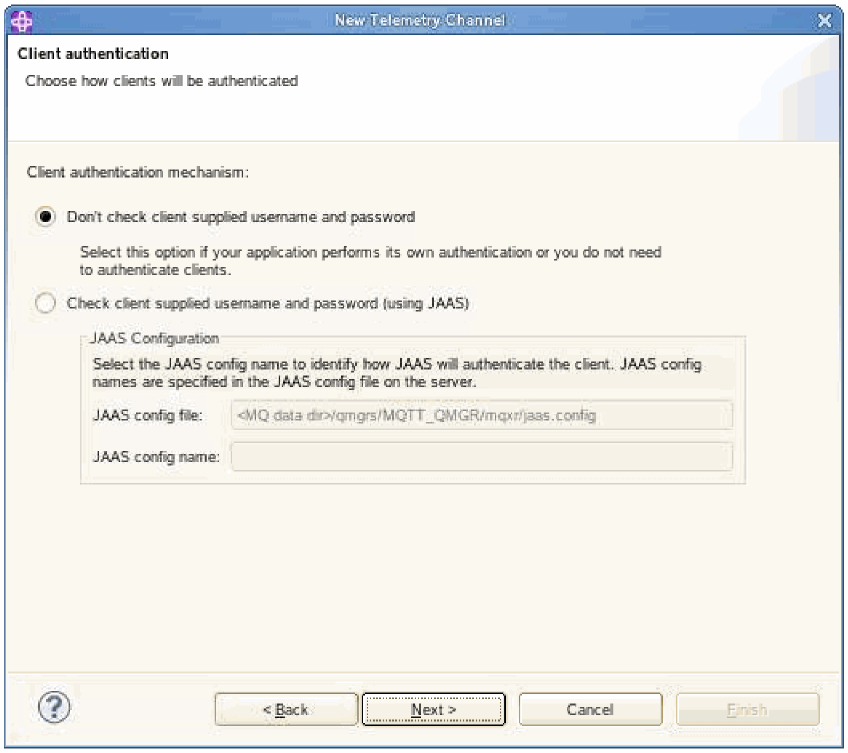
Port number: 1883

Figure 3-13 Specifying the name of a telemetry channel

Note: Port 1883 is the default for WebSphere MQ Telemetry channels, but you have the option to change it. More secure options and configurations are possible. For more information, see the information center at:

`http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt40200_.htm`

3. Declare if you will be using the Java Authentication and Authorization Service (JAAS) to authenticate client-supplied user names and passwords, and then click **Next**, as shown in Figure 3-14.



Client authentication

Choose how clients will be authenticated

Client authentication mechanism:

(● Don't check client supplied username and password)

Select this option if your application performs its own authentication or you do not need to authenticate clients.

○ Check client supplied username and password (using JAAS)

JAAS Configuration

Select the JAAS config name to identify how JAAS will authenticate the client. JAAS config names are specified in the JAAS config file on the server.

JAAS config file: <MQ data dir>/qmgrs/MQTT_QMGR/mqxr/jaas.config

JAAS config name:

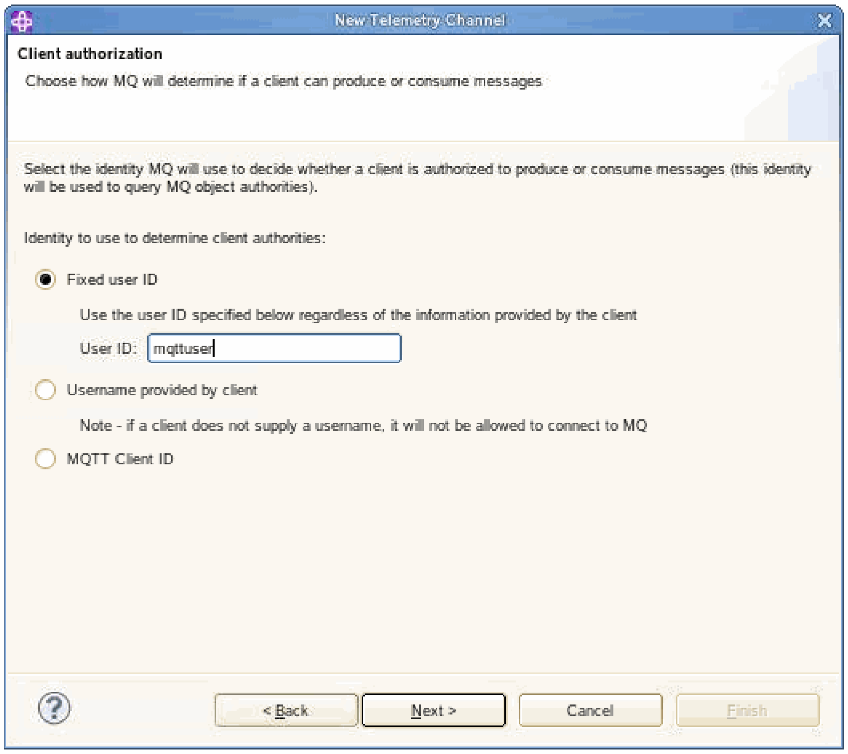
Figure 3-14 Choosing the authentication method for a channel

4. Use the Client authorization panel to declare how WebSphere MQ will authorize clients to send or receive messages, and then click **Next**. Select one of the following choices, as shown in Figure 3-15 on page 71:

   – Fixed user ID: With this option, you designate a User ID that you want to be assigned to all clients connecting to this channel. This option creates the channel with the specified ID as the `MCAUSER`.

   – Username provided by client: If this option is selected, the client must supply a user name, or it will not be allowed to connect to WebSphere MQ.

   – MQTT Client ID: If this option is selected, connection authentication is performed against the actual ID of each individual client.


New Telemetry Channel

Client authorization

Choose how MQ will determine if a client can produce or consume messages

Select the identity MQ will use to decide whether a client is authorized to produce or consume messages (this identity will be used to query MQ object authorities).

Identity to use to determine client authorities:

● Fixed user ID

Use the user ID specified below regardless of the information provided by the client

User ID: mqttuser

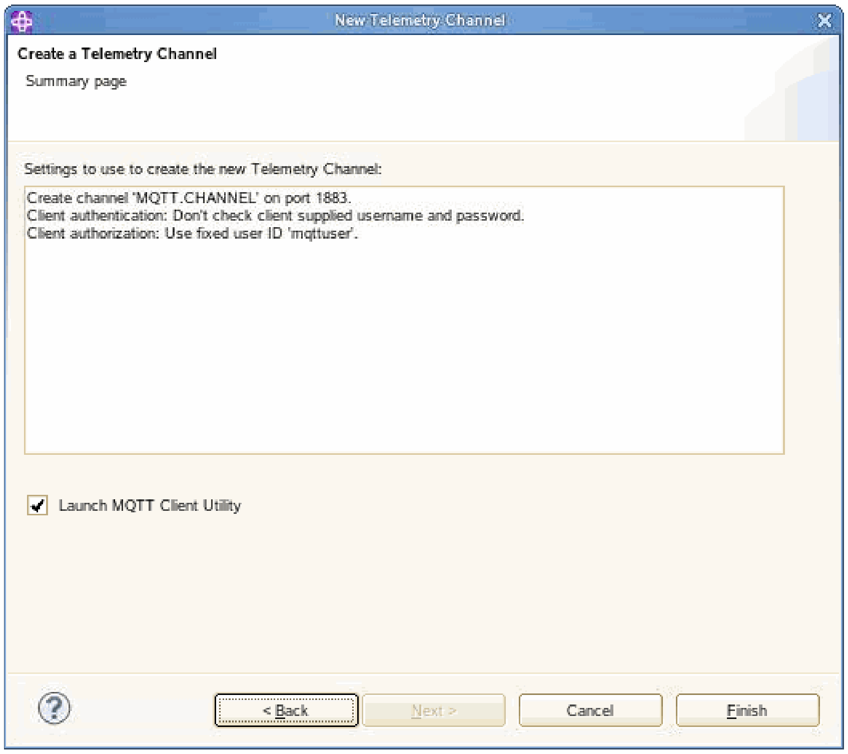
○ Username provided by client

Note - if a client does not supply a username, it will not be allowed to connect to MQ

○ MQTT Client ID

Figure 3-15 Specifying the User ID for a telemetry channel

5. Select whether you want to launch the MQTT Client Utility, and then click **Finish** to complete creation of the telemetry channel, as shown in Figure 3-16.


New Telemetry Channel

Create a Telemetry Channel

Summary page

Settings to use to create the new Telemetry Channel:

Create channel 'MQTT.CHANNEL' on port 1883.
Client authentication: Don't check client supplied username and password.
Client authorization: Use fixed user ID 'mqttuser'.

[✓] Launch MQTT Client Utility

Figure 3-16 The final step in creating a telemetry channel

6. You can use the MQTT Client Utility to test connectivity by publishing and subscribing to test messages, as described in 3.1.3, “Verifying the installation of WebSphere MQ Telemetry” on page 59.

Restart to apply the changes: When you edit the attributes of a telemetry channel, you must restart the channel to apply the changes.

3.1.5 Authorizing MQTT clients to access WebSphere MQ

WebSphere MQ allows you to grant access to MQTT V3 clients using an authorization service called the *object authority manager* (OAM). But before describing this in detail, a quick review of client identities and methods of authorization is required.

MQTT clients are recognized and given access based on an identity that is presented when they connect to a telemetry channel, as described in 3.1.4, “Configuring a queue manager for WebSphere MQ Telemetry” on page 65. The administrator determines how this access will be done when configuring the channel and has the following options:

- The administrator can designate a specific user ID in the channel configuration so that every client that connects to the channel gets that ID. The ID acts as an *mcauser* attribute, and every client that connects to the channel gets the level of authority that is associated with that ID.
- The administrator can require the client to pass a user name for authentication when it connects. The user name is an attribute of the MqttConnectOptions class and must be set by the MQTT client application before it connects to WebSphere MQ. This option can simplify administration, because multiple clients can be assigned the same user name and still get controlled access to the server.
- A final option is to have WebSphere MQ check the ClientID for authorization. One benefit of this approach is the ability to grant fine-grained levels of authority to specific clients, because ClientIDs can be generated based on things such as a device’s serial number or other uniquely identifying characteristics.

If one of the final two options is chosen, the user ID or ClientID that is presented must be defined to WebSphere MQ as a *principal*. A principal is a user ID that can belong to one or more user groups on the server.

The OAM learns the authorities that are granted to each ID by validating the ID and groups against the local operating system. Thus, the ID must be defined to the local operating system in order for the authorization to succeed. Although the authorization can be set at either the principal ID or the group ID level, when WebSphere MQ grants authorization to a principal, that authority is applied to all users in the same group.

Role-based access control

Rather than managing tens of thousands of IDs unique to each client, many administrators employ *role-based access control*. In a role-based model, IDs and groups are created for each security role. Individual clients are then mapped to the appropriate role by selecting which channel to use based on the channel's ID configuration. By grouping client connections to a particular channel that has the identity configured for a given role, each of the clients gets access based on the single ID that is associated with the channel. More complex role and authorization mapping can be done, but that discussion is outside the scope of this book.

The OAM allows you to combine MQTT clients into the following common identities:

- Define multiple telemetry channels and align them each with a specific user ID. Then, give every client that connects to a given channel the specific user ID that is aligned with that channel. In this way, every user that connects to a channel will have the same authority, based on the common user ID that is assigned to them upon connecting.
- Define a single telemetry channel but have each client submit a predefined user name and password that is assigned to it, or pick from a group of predefined user names and passwords that were given to the client when it was provisioned. This design requires authentication by the OAM of the different user names that are submitted.

Full details about OAM are available in the WebSphere MQ Information Center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/fa13160_.htm

Access control patterns

You can choose from among the approaches described in the following sections to tailor access control patterns to your specific requirements. In these examples, the identity used by clients connecting to the telemetry channel is *mqttuser*. If collections of clients need different identities, use multiple user IDs, one for each group.

No access control

In this pattern, MQTT clients are given WebSphere MQ administrative authority and therefore can perform any action on any object. An ID such as mqttuser is added to the mqm group, thereby giving it authority to access all MQ objects. The mqm ID and all users in the mqm group are automatically privileged users with full administrative authority for WebSphere MQ.

Coarse-grained access control

With coarse-grained access control, MQTT clients have authority to publish and subscribe, and to send messages to MQTT clients. They do not have authority to perform other actions, or to access other objects. Granting course-grained access involves using **`setmqaut`** commands to grant authority to the group containing the User ID of the MQTT clients (the mqttuser ID is a member of the mqtt group).

The following commands grant authority to publish and subscribe to all topics and to send publications to MQTT clients:

```
setmqaut -m MQTTQMGR -t topic -n SYSTEM.BASE.TOPIC -g mqtt -all pub sub
setmqaut -m MQTTQMGR -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -g mqtt -all put
```

Medium-grained access control

With medium-grained access control, MQTT clients are divided into different groups to publish and subscribe to different sets of topics and to send messages to MQTT clients. For this level of access, multiple user IDs must be created and segregated into different groups (such as mqtt1, mqtt2, and so on). Multiple administrative topics in the publish/subscribe topic tree are also defined, and the different user IDs are authorized to different topics.

The following examples can be used for medium-grained access control:

```
setmqaut -m MQTTQMGR -t topic -n topic1 -g mqtt1 -all pub sub
setmqaut -m MQTTQMGR -t topic -n topic2 -g mqtt2 -all pub sub
```

These commands allow IDs in the mqtt1 group to have publish/subscribe access to topic1 but not to topic2. Similarly, users in the mqtt2 group get publish/subscribe access to topic2 but not to topic1.

The system administrator creates a user group called mqtt, and adds the various mqttuser IDs to the group. All users in the mqtt group can then be authorized to send messages to MQTT clients by granting put authority to the transmit queue. You can use this command:

```
setmqaut -m MQTTQMGR -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -g mqtt -all put
```

Fine-grained access control

In this pattern, MQTT clients are incorporated into an existing system of access control that authorizes groups to perform actions on objects. This model is used when WebSphere MQ applications are publishing and subscribing to the same topic as MQTT clients.

Here, a user ID is assigned to one or more operating system groups depending on the authorizations it requires. The groups in this example are PublishX, SubscribeY, and mqtt, and carry the following authorizations:

- PublishX: Members of the PublishX group can publish to topicX.
- SubscribeY: Members of the SubscribeY group can subscribe to topicY.
- mqtt: Members of the mqtt group can send publications to MQTT clients.

To implement fine-grained access control, follow these steps:

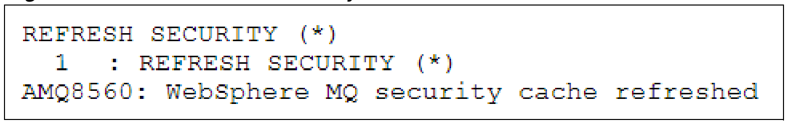
1. Create the groups PublishX and SubscribeY and allocate them to multiple administrative topics in the publish/subscribe topic tree.
2. Create the group mqtt.
3. Create multiple user IDs (mqttusr1, mqttusr2, and so on), and add the users to the groups based on what they must be authorized to do.
4. Authorize different PublishX and SubscribeX groups to different topics, and authorize the mqtt group to send messages to MQTT clients. The following commands can be used:

   ```
   setmqaut -m MQTTQMGR -t topic -n topic1 -g PublishX -all pub
   setmqaut -m MQTTQMGR -t topic -n topic1 -g SubscribeX -all pub sub
   setmqaut -m MQTTQMGR -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -g mqtt -all put
   ```

Refreshing the security if changes are made

WebSphere MQ queries OS group memberships upon startup and builds an internal cache. If the group membership of an ID changes, the queue manager does not recognize the change until the **`refresh security`** command is issued, as shown in Figure 3-17. A queue manager recycle refreshes the security, or if necessary you can issue a **`runmqsc`** command. The **`refresh security`** command dumps the cache, and is required only after the group membership of an ID changes.

Figure 3-17 The refresh security command



```
REFRESH SECURITY (*)
  1   : REFRESH SECURITY (*)
AMQ8560: WebSphere MQ security cache refreshed
```

The WebSphere MQ Information Center has detailed information about managing security using the Explorer GUI:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.explorer.doc/o_oam.htm

3.1.6 Enabling the security features of WebSphere MQ for telemetry channels

Security is a broad subject and involves much more than can be presented in this book. However, some particular methods of security involving MQTT and WebSphere MQ deserve special mention here, each of which can be introduced by the administrator at the telemetry channel level.

One option is to check the ClientID when the channel connection is made, but this option has limitations. A more secure option is to use the Java Authentication and Authorization Service (JAAS) to authenticate a client's user name and password. In addition, you can strengthen the security of both of these options by using an encrypted TLS/SSL certificate to pass the client's identifying information and authorization back and forth with the server.

The administrator can require a client to present its identity using the following methods:

- User ID: All connections to a particular channel are designated (during channel configuration) as having the same user ID and, therefore, the same level of access, regardless of the clients' individual identities.
- Username: When connecting to a channel, the client passes a predefined user name and password ID for verification and gets the level of access that is designated for that user name.
- ClientID: Each connection is authenticated based on a unique identifier, or ClientID, that is associated with each particular client, with access granted accordingly.

Authorization using ClientID

One of the options for WebSphere MQ channel configuration is to perform client authorization during channel connection using the ClientID, as explained in 3.1.5, "Authorizing MQTT clients to access WebSphere MQ" on page 72. However, the use of ClientID to resolve an identity is not true authentication. The ClientID that is presented is accepted at face value, with no additional confirming passwords or other information required. However, this method can still be useful, particularly in the case of an application with many thousands or even millions of clients. To speed things up, in this situation, the client application typically uses some globally unique element of the device, such as its MAC address, as the ClientID.

Authentication using a password

Another method is to authenticate a client user name and password using JAAS. This is particularly useful when the client application is driven by a human user, such as is the case with a cell phone, where the application can present a dialog to collect the password from the user. Typically in such systems, the application also provides methods for the user to update their password on demand, for server-side password management and for ID revocation. JAAS authentication can also be used without user interaction in situations where the credentials can be stored in a configuration file or elsewhere on the device.

Authentication using TLS/SSL

TLS/SSL provides encryption but can also provide authentication if the client is provided with a private, signed digital certificate (X.509). The encryption features of TLS/SSL are normally used to protect credentials, such as a user ID and password. However, a client that has a proper certificate can present it for the server to authenticate and map to a particular server-side identity. For added security, certificate-based authentication can be combined with the other forms of authorization listed in this section.

Additional details about implementing security for MQTT clients are available in the WebSphere MQ Information Center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r0/topic/com.ibm.mq.amqtat.doc/tt60312_.htm

3.1.7 Configuring WebSphere MQ to send messages to MQTT clients

A WebSphere MQ application can send a message to an MQTT V3 client using one of the following methods:

- Publish the message to a topic (the publish/subscribe model)
- Send the message to the client directly (the point-to-point model)

Whichever method is used, the message is placed by the queue manager onto `SYSTEM.MQTT.TRANSMIT.QUEUE` and sent to the client by the WebSphere MQ telemetry service.

Publishing to a topic

One way to understand publishing a message to a topic is to discuss the subscription process first. When a telemetry client creates a subscription to a topic, the WebSphere MQ telemetry service creates a proxy subscription on the queue manager on behalf of the MQTT client. With the subscription in place, the client becomes an automatic destination for any messages that are published to that topic.

To route messages to clients with subscriptions, the queue manager puts the messages onto `SYSTEM.MQTT.TRANSMIT.QUEUE`. It then uses the ClientID of the subscribing clients as the key to getting each topic-specific message to the correct client subscribers.

To enable MQTT client subscribers and publishers to connect using different queue managers, the clustering features of WebSphere MQ can be implemented as shown in Figure 3-18. When an administrative topic is created in a cluster with the attribute CLUSTER (*`clusterName`*), any application in the cluster can publish to the client. The subscribers and publishers can be part of the same cluster or can be connected by a publish/subscribe hierarchy.

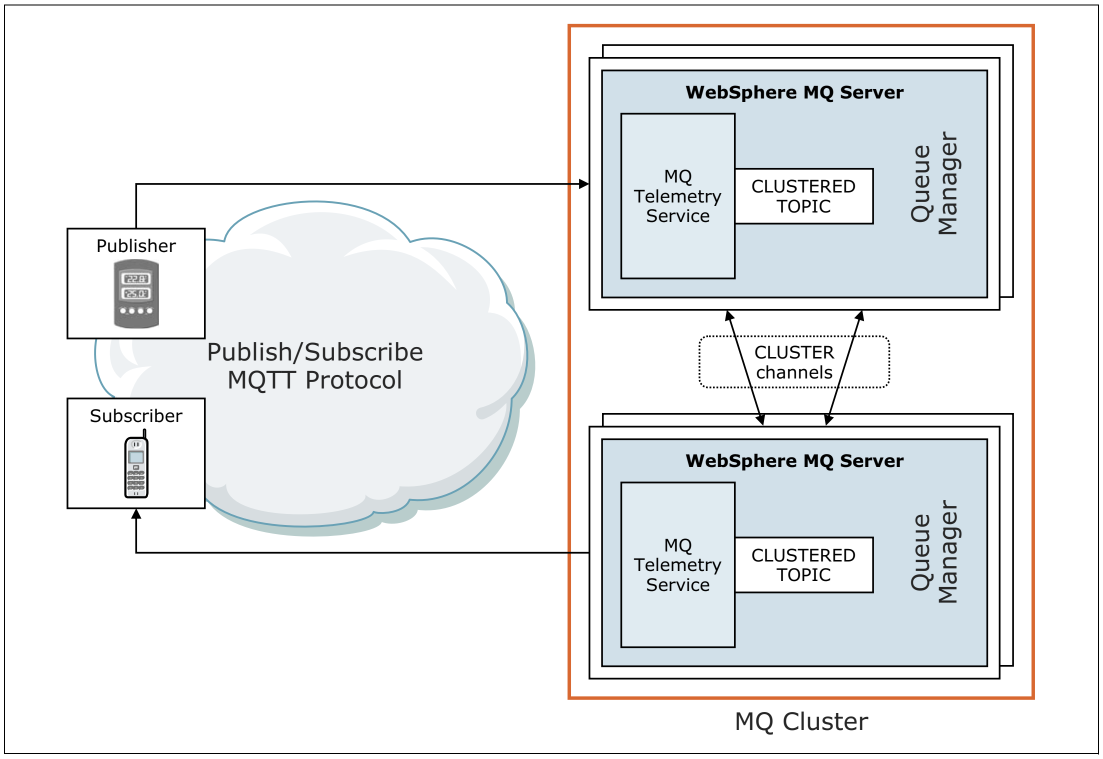


Figure 3-18 Publishers and subscribers connecting to queue managers within a cluster

Sending a message to a client directly

As an alternative to the publish/subscribe approach, a WebSphere MQ application has the capability to send an unsolicited message to an MQTT V3 client directly. This method can be accomplished by putting the message onto the `SYSTEM.MQTT.TRANSMIT.QUEUE`, specifying the ClientID as the queue manager and designating the topic as a queue. The message goes directly to the client because the ClientID (ClientIdentifier) is specified in the code through the creation of an object descriptor, with the client as the `ObjectQmgrName` in the WebSphere MQ application.

The following syntax is an example of that process:

```
MQOD.ObjectQmgrName = ClientIdentifier;
MQOD.ObjectName = name;
```

If the application is written using Java Messaging Service (JMS), a point-to-point destination must be created to specify the ClientID and designate the topic name as a queue. The following syntax exemplifies the process:

```
javax.jms.Destination jmsDestination =
                        (javax.jms.Destination)jmsFactory.createQueue
                        ("queue://ClientIdentifier/name");
```

Another method of sending a message directly to a client is to use an MQPUT command to place the message to a remote queue, which is basically a pointer to the transmit queue. The definition of the remote queue must have the remote queue manager name attribute (`RQMNAME`) set as the ClientID of the client for which the message is intended. This method allows the WebSphere MQ telemetry service to route the messages that were put to the remote queue to the appropriate MQTT client. The transmit queue must resolve to `SYSTEM.MQTT.TRANSMIT.QUEUE`. The remote queue name (`RNAME`) attribute can be anything because it is sent to the client as a topic string.

Regardless of which approach is used, the message goes directly to the MQTT client, provided it is connected. If the client is not connected, the queue manager can continue to receive publications on its behalf so long as the client created a stateful session with the queue manager. The messages are stored in `SYSTEM.MQTT.TRANSMIT.QUEUE`, then forwarded to the client when it reconnects. See 2.3, "MQTT concepts" on page 26 for more information about stateful sessions.

If WebSphere MQ sends a persistent message directly to a client, it is sent by default with a quality of service (QoS) designation of 2 (exactly once). If WebSphere MQ sends a non-persistent message, it is sent with a QoS designation of 0 (deliver once, at most). The client might not be able to support all qualities of service. In such cases, the client can set the maximum QoS it can support by making a subscription to the topic `DEFAULT.QoS` to lower the QoS it accepts for messages sent directly.

For more information about how to configure a queue manager to send messages to MQTT clients visit the WebSphere MQ Information Center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt40150_.htm

3.1.8 Performance considerations for WebSphere MQ Telemetry

WebSphere MQ Telemetry enables up to hundreds of thousands of MQTT V3 clients to connect to WebSphere MQ queue managers. However, when managing so many clients, several factors must be considered to ensure adequate messaging performance and make scalability manageable.

Minimize the cost of connections

There are numerous costs involved with client-queue manager connections:

- Processor usage and set-up time to create the connection
- Network usage and maintenance to enable messages to flow
- Memory used when keeping a connection open but not using it

There is an extra load incurred when clients stay connected, both in terms of frequent checks to confirm that the network is active and memory used on the server for each open connection. So if you are sending messages at a rate greater than one per minute, keep your connection open to avoid the cost of initiating a new connection each time. However, if you are sending messages at intervals of less than one every 10-15 minutes, consider dropping the connection to avoid the cost of keeping it open between messages. TLS/SSL connections might be kept open, but idle, for longer periods because of the extra expense of establishing secure connections.

Additionally, consider the capability of the client and the purpose of your application. If there is a store-and-forward facility on the client, you can batch up and send multiple messages at the same time, then drop the connection until it is needed again. However, if the client is disconnected, it is not possible for it to receive any potential replies, such as error messages, from the server.

Maximize throughput

If your system has one client sending many messages (for example, file transfers), do not wait for a server response to every message. Instead, send all of the messages and check at the end to confirm they were all received. This verification can also be accomplished by giving the messages specific QoS levels, ranging from 0 for unimportant messages to 2 for critically important messages. Overall message throughput can be around twice as high with a QoS designation of 0 compared to a designation of 2.

The length of topic names affects the number of bytes that flow across the network. So another option to limit throughput is to limit the number of characters in each topic name. Because small message size is one of the advantages of MQTT, it is best not to append a large ClientID and topic string to each message.

Improve scalability

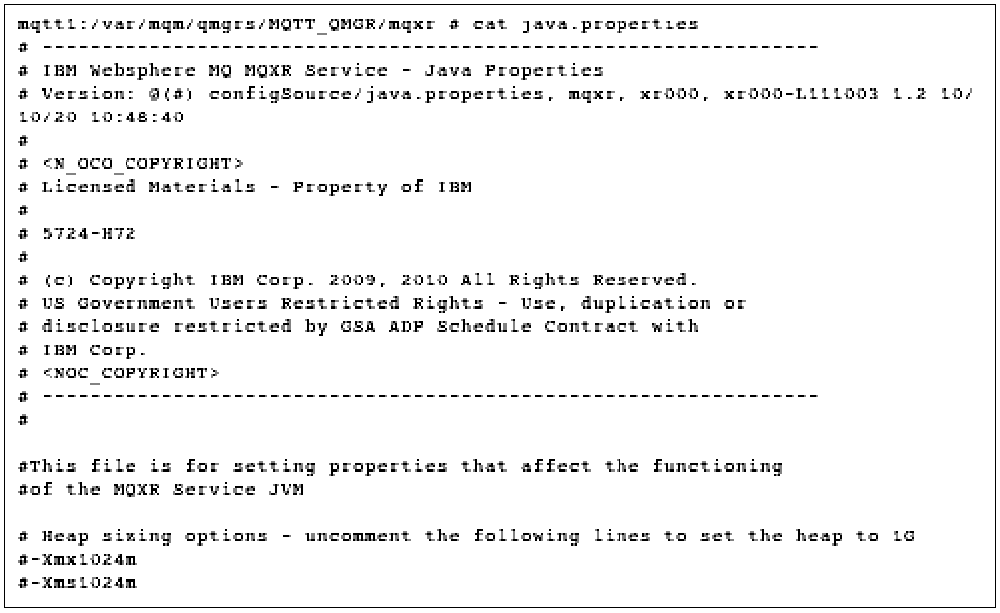
When an MQTT client subscribes to a topic, a proxy subscription is created by the telemetry service in WebSphere MQ. Publications for the client are then routed onto `SYSTEM.MQTT.TRANSMIT.QUEUE`. The remote queue manager name in the transmission header of each publication is set to the ClientIdentifier of the MQTT client that made the subscription. If there are many clients, each making their own subscriptions, this results in many proxy subscriptions being maintained throughout the WebSphere MQ publish/subscribe cluster or hierarchy.

For situations requiring distribution of messages to many clients (airline schedule updates, weather alerts, and so on), it might be best to use subscriptions. However, if possible, use point-to-point messaging for individual clients, such as actuators, that typically only receive information pertaining to them specifically. For information about sending messages to clients directly, see 3.1.6, “Enabling the security features of WebSphere MQ for telemetry channels” on page 74.

Manage large numbers of clients

To support many concurrently connected clients, increase the memory available for the telemetry service by setting the Java virtual machine (JVM) parameters `-Xms` and `-Xmx` by updating the `java.properties` file in the telemetry service configuration directory

(`/var/mqm/qmgrs/`*`qMgrName`*`/mqxr` on Linux). The file includes directions to uncomment certain lines of code to set the heap to 1 gigabyte (as shown in Figure 3-19). Other settings, as well, can be added to the file to pass to the JVM running the telemetry service.



```
mqtt1:/var/mqm/qmgrs/MQTT_QMGR/mqxr # cat java.properties
# ------------------------------------------------------------------
# IBM Websphere MQ MQXR Service - Java Properties
# Version: @(#) configSource/java.properties, mqxr, xr000, xr000-L111003 1.2 10/
10/20 10:48:40
#
# <N_OCO_COPYRIGHT>
# Licensed Materials - Property of IBM
#
# 5724-H72
#
# (c) Copyright IBM Corp. 2009, 2010 All Rights Reserved.
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with
# IBM Corp.
# <NOC_COPYRIGHT>
# ------------------------------------------------------------------
#

#This file is for setting properties that affect the functioning
#of the MQXR Service JVM

# Heap sizing options - uncomment the following lines to set the heap to 1G
#-Xmx1024m
#-Xms1024m
```

Figure 3-19 Example of the java.properties file

The system administrator also needs to configure the operating system where the queue manager resides to enable it to handle a large number of sockets. Each socket requires one file descriptor. The telemetry service requires some additional file descriptors, so this number must be larger than the number of open sockets required.

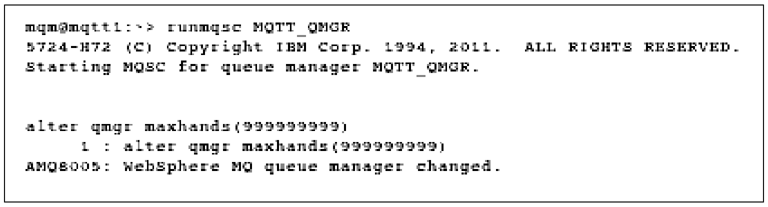
To increase the number of open file descriptors allowed, add the following lines to the `/etc/security/limits.conf` file:

```
@mqm soft nofile 65000
@mqm hard nofile 65000
```

For these settings to take effect, the user must log out and log back in to the server.

If there are a large number of client connections, similar settings for the ClientID must be added to the `/etc/security/limits.conf` file.

The queue manager uses an object handle for each nondurable subscription. To support many active, nondurable subscriptions, increase the maximum number of active handles in the queue manager by using the runmqsc command ALTER QMGR MAXHANDS(999999999) as shown in Figure 3-20.



```
mqm@mqtt1:~> runmqsc MQTT_QMGR
5724-H72 (C) Copyright IBM Corp. 1994, 2011.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager MQTT_QMGR.


alter qmgr maxhands(999999999)
     1 : alter qmgr maxhands(999999999)
AMQ8005: WebSphere MQ queue manager changed.
```

Figure 3-20 Example of altering the maximum number of handles

As the number of clients increases, another consideration is the time required to restart the system. The planned downtime for this might have implications in the number of messages that queue up to be processed. So configure the system so that the messages can be successfully processed in an acceptable time. In particular, review the disk storage, memory, and processing capacity.

With some client applications, it might be possible to discard stored messages when the client reconnects and thereby reduce overhead associated with delivery of outdated or unnecessary messages. To discard messages, set `MqttConnectOptions.cleanSession` in the client connection parameters. Alternatively, publish and subscribe using a QoS level of 0 (deliver once, at most). If the information is not mission-critical, consider using non-persistent messages when sending messages from WebSphere MQ. Non-persistent messages are not recovered when the queue manager restarts.

Performance reports

For information about performance reports for WebSphere MQ Telemetry, review the WebSphere MQ Telemetry Performance Evaluations at:

http://www-01.ibm.com/support/docview.wss?rs=171&uid=swg27007150

At the time of this publication, the available Telemetry Performance Evaluation is for WebSphere MQ V7.0.1 (SupportPac MP0A), but the website is expected to be updated as more evaluations become available.

3.2 Building a simple solution using WebSphere MQ Telemetry

WebSphere MQ Telemetry, a component of WebSphere MQ, can be used to build a simple solution for the transportation logistics scenario being used throughout this book.

3.2.1 Solution overview

The example solutions presented here assume that the number of trucks in Transportation Company B's fleet is not too large, and that the sensor and communication hardware in the trucks can support embedded third-party software. Based on this, an appropriate messaging system topology becomes clear. The trucks will connect to company headquarters directly with no edge server in between. This approach will keep things simple and flexible.

The flow of data in the scenario is as follows:

- Each truck regularly sends to headquarters a series of reports, or messages, containing its logistical data (facts such as current position, speed and fuel consumption) and the current condition of its cargo and cargo area (things such as interior temperature and ventilation). The example solution being created here will generate a message that transmits the temperature of the truck's cargo area.
- The system at company headquarters receives the data from each truck, analyzes it, and if necessary sends messages containing commands back to the truck or to the fleet as a whole.

With WebSphere MQ Telemetry, MQTT client libraries can be embedded into the sensor and communications hardware on the trucks, messages can be transmitted to and from the trucks using the lightweight MQTT protocol, and the message traffic can be managed using an MQTT-based message broker on the enterprise server at company headquarters.

Here is how the pieces fit together:

- WebSphere MQ Telemetry: A lightweight implementation of a publish/subscribe messaging model for communicating with devices at the edge of a network. In this example, it will act as the server to which the devices on the trucks will connect.
- MQTT client libraries: Used for building client messaging applications on remote devices. WebSphere MQ Telemetry provides reference implementations of MQTT client libraries in C and Java.
- WebSphere MQ: A universal messaging backbone. In this example, the server will use WebSphere MQ to interact with the devices on the trucks.

3.2.2 Solution implementation

Implementation of these sample solutions starts with designing the message topic hierarchy and message format. Then the C and Java MQTT client libraries are used to develop client messaging applications that will run on the devices on the trucks. Then WebSphere MQ Telemetry is configured as an MQTT message server, and finally the necessary server-side applications are built to manage the incoming and outgoing messages and commands.

Topic hierarchy

The example solutions use the following topics:

- *temperature*, to which the trucks will publish their reports on the temperature of the interior cargo space
- *command*, to which the system at company headquarters will publish any instructions for the trucks

In turn, the headquarters system will subscribe to the temperature topic, and the trucks will subscribe to the command topic.

To provide as much flexibility as possible, MQTT supports hierarchical topic names that enable application designers to organize topics in a way that simplifies their management. Levels in the hierarchy are delimited by the forward slash (/) character, as in the example SENSOR/1/HUMIDITY.

In this example solution, the first and second level topic is reserved by the system:

- The first level is defined for vehicle type. So all trucks can subscribe to the topic Truck to receive messages from headquarters. In addition, the applications in headquarters can publish messages to this topic so all trucks can receive them.
- The second level identifies a specific vehicle. So the truck identified as number 00001 subscribes to the topic Truck/00001 to receive messages intended just for it, and the applications at headquarters can send commands specific to the truck by publishing a message to the same topic.
- Additional topic levels can be used for specific business purposes. For example, a topic for collecting the temperature data of the truck's cargo area can be Truck/00001/Temp.

Because the topic string is transmitted along with the message (also called the payload), developers of topic strings are advised to use abbreviations where possible to cut down network overhead. So in the sample solutions presented here, temperature is shortened to temp, command is shortened to comm, and so on.

Message format

The message format in the sample solutions presented here will follow specific standards, as shown in Table 3-1.

Table 3-1 Topic hierarchy and message formats

Hierarchy	Topic string	Structure	Message format	Note
1st	Truck	Truck	From truck to headquarters: (Reserved) From headquarters to truck: String (Broadcast)	The type of vehicle, such as a truck or car.
2nd	xxxxx	Truck\xxxxx	From truck to headquarters: String From headquarters to truck: String	A unique identifier for each truck.
3rd	Temp	Truck\xxxxx\Temp	From truck to headquarters: temperature values From headquarters to truck: (Reserved)	The specific subject area of the message, in this case, the temperature of the interior cargo space.
3rd	Comm	Truck\xxxxx\Comm	From truck to headquarters: (Reserved) From headquarters to truck: String	A command sent from company headquarters to a truck or trucks.

Client development

The MQTT client messaging applications on the trucks are developed using the WebSphere MQTT Java and C client libraries. These are explained in 2.1, “Clients and brokers used in this book” on page 22.

Begin by copying the following libraries to an appropriate project folder:

- WebSphere MQTT C libraries
  - `MQTTClient.h`
  - `MQTTClientPersistence.h`
  - `libmqttv3c.so`
- WebSphere MQTT Java libraries
  - `com.ibm.micro.client.mqttv3.jar`

The steps to develop the application differ based on whether the Java client or the C client is chosen. Tools are available to help with the development process; this sample application was built using Eclipse.

Java application

To create the sample MQTT client application in Java:

1. Create a new project in Eclipse.
2. Configure the WebSphere MQTT Java libraries as the dependency libraries in the project’s Java build path.
3. Develop the publish/subscribe application, `PubSubAsync.java`, as shown in Example 3-1 on page 83. This sample application is asynchronous, in that it does not wait for a reply

message before moving on to another task. Instead it invokes the MqttCallback callback method to await any replies or new messages.

Example 3-1 PubSubAsync.java

```
package com.ibm.redbooks.mqtt.ch3.client.PubSub;

import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import java.util.Random;

import com.ibm.micro.client.mqttv3.MqttCallback;
import com.ibm.micro.client.mqttv3.MqttClient;
import com.ibm.micro.client.mqttv3.MqttDeliveryToken;
import com.ibm.micro.client.mqttv3.MqttException;
import com.ibm.micro.client.mqttv3.MqttMessage;
import com.ibm.micro.client.mqttv3.MqttTopic;

public class PubSubAsync implements MqttCallback {

   private MqttClient client = null;
   private String mqttURI = null;
   String ip = "";
   String def_ip = "localhost";
   String port = "";
   String message = "";
   String topic = "";
   String lis_topic = "";

   public static String clientId = "00001";
   public static String user = "mqttuser";
   public static final String subTopicString = System.getProperty(
         "subTopicString", "Truck/" + clientId + "/Comm");
   public static final String pubTopicString = System.getProperty(
            "pubTopicString", "Truck/" + clientId + "/Temp");
   public static final String broadcastTopicString = System.getProperty(
         "broadcastTopicString", "Truck");

   class Truck extends Thread {

      String topic = "";
      MqttClient client;

      public Truck(MqttClient client, String stopic)
      {
         this.client = client;
         this.topic = stopic;
      }

      public void run() {
         try {
            while(true)
            {
               Random random = new Random(System.currentTimeMillis());
               int temperature = random.nextInt(35);
               MqttMessage message = new MqttMessage();
               message.setQos(2);
               message.setPayload(String.valueOf(temperature).getBytes());
               MqttTopic topic = this.client.getTopic(this.topic);
               MqttDeliveryToken token = topic.publish(message);
               token.waitForCompletion();
```

```
                Thread.sleep(60000);
             }

          } catch (MqttException e) {
             // TODO Auto-generated catch block
             e.printStackTrace();
          } catch (InterruptedException e) {
                // TODO Auto-generated catch block
             e.printStackTrace();
          }
       }
    }

    public PubSubAsync(String mqttURI) {
       this.mqttURI = mqttURI;
       this.lis_topic = subTopicString;
       this.topic = subTopicString;
    }

    public PubSubAsync(String sip, String sport, String stopic, String l_topic) {
       this.ip = sip;
       this.port = sport;
       this.topic = subTopicString;
       this.lis_topic = subTopicString;
    }

    public void start() throws MqttException {
       if(mqttURI == null || mqttURI.equalsIgnoreCase(""))
          {
          mqttURI = "tcp://"+this.ip+":"+this.port;
       }
       client = new MqttClient(mqttURI, user);
       client.setCallback(this);
       client.connect();
       if(this.lis_topic != null && !this.lis_topic.equalsIgnoreCase(""))
       {
          client.subscribe(this.lis_topic);
       }else
       {
          client.subscribe(subTopicString, 2);
       }

       Truck truck = new Truck(client,pubTopicString);
       truck.start();

       System.out.println("Connected and subscribed to "+mqttURI);
    }

    public void stop() throws MqttException {
       if (client != null) {
          client.disconnect();
       }
       System.out.println("Disconnect from "+mqttURI);
    }

    public void connectionLost(Throwable arg0) {
       // In a production system, we would decide what we want to do in the event the
       // connection drops. In our case we're expecting that not to happen so simply
       // write a message to standard error.
```

```
        System.err.println("The connection to WMQ was lost. Closing down the 
connection.");
    }

    public void deliveryComplete(MqttDeliveryToken arg0) {
        // We're just subscribing in this example.
    }

    public void messageArrived(MqttTopic topic, MqttMessage message)
            throws Exception {
        // Message received, simply write it onto the console.
        String tex = "";
        ByteArrayInputStream bais = new ByteArrayInputStream(message.getPayload());
        DataInputStream dis = new DataInputStream(bais);
        // Use the Java input streams to do the type conversion for us.
        tex = dis.readLine();
        dis.close();
        bais.close();
        System.out.println("Message " + tex);
    }

        public static void main(String[] args) {
        PubSubAsync sub = null;

        if(args != null && args.length == 1)
        {
            sub = new PubSubAsync(args[0]);
        }else if(args != null && args.length == 4)
        {
            sub = new PubSubAsync(args[0],args[1],args[2],args[3]);
        }

        try {
            sub.start();
        } catch (MqttException ex) {
            System.err.println("Could not connect to WMQ: "+ex);
        }
    }

}
```

C application

Follow these steps to create the sample MQTT client applications in C:

1. Create a new project in Eclipse.
2. Configure the WebSphere MQTT C libraries as the dependency libraries in the project's GCC C linker path.
3. Develop the publisher application, `Pubasync.c`, shown in Example 3-2, and the subscriber application, `Subasync.c`, shown in Example 3-3 on page 87. Like the Java application presented in the previous section, these sample C applications are asynchronous. `Pubasync.c` uses the void delivered (void context, MQTTClientdeliveryToken dt method) to confirm that messages with QoS designations of 1 or 2 have been delivered and acknowledged by the server.

Example 3-2 Pubasync.c

```
#include "stdio.h"
#include "stdlib.h"
```

```
#include "string.h"
#include "MQTTClient.h"

#define ADDRESS     "tcp://192.168.0.11:1883"
#define CLIENTID    "Truck_00001_Pub"
#define TOPIC_Command       "Truck/00001/Comm"
#define TOPIC_Temperature       "Truck/00001/Temp"
#define QOS         2
#define TIMEOUT     10000L

volatile MQTTClient_deliveryToken deliveredtoken;

void delivered(void *context, MQTTClient_deliveryToken dt)
{
    printf("Message with token value %d delivery confirmed\n", dt);
    deliveredtoken = dt;
}

void connlost(void *context, char *cause)
{
    printf("\nConnection lost\n");
    printf("     cause: %s\n", cause);
   }

int main(int argc, char* argv[])
 {

   MQTTClient client;
   MQTTClient_connectOptions conn_opts;
   MQTTClient_message pubmsg;
   MQTTClient_deliveryToken token;
   int rc;

   MQTTClient_create(&client, ADDRESS, CLIENTID,
         MQTTCLIENT_PERSISTENCE_NONE, NULL);
   memset(&conn_opts, '\0', sizeof(conn_opts));
   conn_opts.keepAliveInterval = 200;
   conn_opts.cleansession = 1;

   MQTTClient_setCallbacks(client, NULL, connlost, NULL, delivered);

   if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS) {
   printf("Failed to connect, return code %d\n", rc);
      exit(-1);
   }

   pubmsg.payload = argv[1];
   pubmsg.payloadlen = strlen(argv[1]);
   pubmsg.qos = QOS;
   pubmsg.retained = 0;
   deliveredtoken = 0;
   printf("Waiting for publication of %s on topic %s for client with ClientID:
%s\n",
    argv[1], TOPIC_Temperature, CLIENTID);
   MQTTClient_publishMessage(client, TOPIC_Temperature, &pubmsg, &token);
   while(deliveredtoken != token);
   MQTTClient_disconnect(client, 10000);
   MQTTClient_destroy(&client);
   return -1;
}
```

Example 3-3 Subasync.c

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "MQTTClient.h"

#define ADDRESS     "tcp://192.168.0.11:1883"
#define CLIENTID    "Truck_00001"
#define TOPIC_Command       "Truck/00001/Comm"
#define TOPIC_Temperature       "Truck/00001/Temp"
#define QOS         2
#define TIMEOUT     10000L

volatile MQTTClient_deliveryToken deliveredtoken;

void delivered(void *context, MQTTClient_deliveryToken dt)
{
    printf("Message with token value %d delivery confirmed\n", dt);
    deliveredtoken = dt;
}

int msgarrvd(void *context, char *topicName, int topicLen, MQTTClient_message *message)
{
    int i;
    char* payloadptr;
    int temp;

    printf("Message arrived\n");
    printf("   Topic: %s", topicName);
    printf("   Command message: %s ",  message->payload);

    putchar('\n');
    MQTTClient_freeMessage(&message);
    return 1;
}

void connlost(void *context, char *cause)
{
    printf("\nConnection lost\n");
    printf("     cause: %s\n", cause);
}

int main(int argc, char* argv[])
{
    MQTTClient client;
    MQTTClient_connectOptions conn_opts;
    int rc;
    int ch;

    MQTTClient_create(&client, ADDRESS, CLIENTID,
        MQTTCLIENT_PERSISTENCE_NONE, NULL);
    memset(&conn_opts, '\0', sizeof(conn_opts));
    conn_opts.keepAliveInterval = 20;
    conn_opts.cleansession = 1;

    MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, delivered);

    if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
    {
        printf("Failed to connect, return code %d\n", rc);
        exit(-1);
    }
    printf("Subscribing to topic %s\nfor client %s using QoS%d\n\n"
           "Press Q<Enter> to quit\n\n", TOPIC_Command, CLIENTID, QOS);
    MQTTClient_subscribe(client, TOPIC_Command, QOS);

    do
    {
        ch = getchar();
    } while(ch!='Q' && ch != 'q');

    MQTTClient_disconnect(client, 10000);
    MQTTClient_destroy(&client);
}
```

Server setup and application development

On the server side, the data from the trucks can be received using the following methods:

- Through a publish/subscribe model in which the server subscribes to a particular truck-related topic and receives any messages sent with that topic
- Through a point-to-point model in which the data is received in a message from a queue

To set up the server to receive messages from the MQTT-enabled clients on the trucks:

1. Configure WebSphere MQ Telemetry. For details about this process and the other setup procedures listed here, see 3.1, “Installing, configuring, and managing WebSphere MQ Telemetry” on page 56.
2. Start a queue manager and its telemetry service.
3. Create the telemetry channel by using the following example. Make sure that the TCP/IP address and port that are defined for it match the values that are established in the MQTT client application.

   ```
   DEFINE CHANNEL('Truckchannel') CHLTYPE(MQTT) PORT(1885) MCAUSER('mqttuser')
   ```

4. Create a subscription topic and run it using the **runmqsc** command.

   The syntax differs depending on whether you are using a publish/subscribe or point-to-point message model.

   – For a point-to-point model, use the following command:

     ```
     DEFINE TOPIC('Temperatertopic') TOPICSTR('Truck/#/Temp') REPLACE
     ```

     For point-to-point models only: Configure the channel and local queue. In the following code, the queue TemperatureQueue is defined to store all temperature messages from the truck. Then, a subscription to the temperature topic is created, and its destination is set to the queue that was just defined. Finally, the channel TEST is created so the Message Queue Interface (MQI) program can connect to it.

     ```
     DEFINE QLOCAL('TemperatureQueue') REPLACE
     DEFINE SUB('Temperatersub') DEST('TemperatureQueue')
     TOPICOBJ('Temperatertopic') REPLACE
     ```

```
DEFINE CHANNEL('TEST') CHLTYPE(SVRCONN) REPLACE
```

– For a publish/subscribe model, use the following command:

```
DEFINE TOPIC('Temptopic00001') TOPICSTR('Truck/00001/Temp') REPLACE
```

For publish/subscribe models only: Create a publishing topic for sending commands to the trucks using the following code.

```
DEFINE TOPIC('Truck00001Command') TOPICSTR('Truck/00001/Comm') REPLACE
```

The server-side application monitors the temperature data that it receives and, when necessary, replies to the truck with new commands.

For this scenario, the truck’s cargo space is supposed to be maintained at 20 degrees Celsius. So if a truck sends a temperature reading below this level, a command is sent to lower the amount of cool air that is pushed into the cargo space. If a temperature reading above this level is received, the command is to increase the amount of cooling. If a reading of exactly 20 degrees C is received, no command is sent. Temperature readings above 30 degrees C indicate a problem, and a command is sent that orders the truck to transfer its cargo to another vehicle.

The server-side application can be built in using the MQTT Java libraries or using the WebSphere MQ Message Queue Interface (MQI).

Example 3-4 shows a Java application, `ServerHandler.java`, that is built using the MQTT Java libraries. It subscribes to the appropriate topic, receives any messages from the truck, analyzes the temperature data and, when necessary, calls out to a second library-built Java application, `MsgSender.java` (see Example 3-5 on page 93), to send commands back to the truck.

Example 3-4 ServerHandler.java

```
package com.ibm.redbooks.mqtt.ch3;

import java.io.ByteArrayInputStream;
import java.io.DataInputStream;

import com.ibm.micro.client.mqttv3.MqttCallback;
import com.ibm.micro.client.mqttv3.MqttClient;
import com.ibm.micro.client.mqttv3.MqttDeliveryToken;
import com.ibm.micro.client.mqttv3.MqttException;
import com.ibm.micro.client.mqttv3.MqttMessage;
import com.ibm.micro.client.mqttv3.MqttTopic;


public class ServerHandler implements MqttCallback {

   private MqttClient client = null;
   private String mqttURI = null;
   String ip = "";

   String def_ip = "localhost";
   String port = "";
   String message = "";
   String topic = "";
   String lis_topic = "";

   public static String clientId = "00001";
   public static final String subTopicString = System.getProperty(
         "subTopicString", "Truck/" + clientId + "/Temp");
   public static final String pubTopicString = System.getProperty(
         "pubTopicString", "Truck/" + clientId + "/Comm");
   public static final String broadcastTopicString = System.getProperty(
         "broadcastTopicString", "Truck");
   public static final String publication = System.getProperty("publication",
         System.currentTimeMillis() + ",35.918,-78.231" + ",21C");

   private static final String MQTT_TRACKSIDE_TOPIC = "Test";

   String mesage1 = "Warning! The temperature is above 20'C! Please cooling! ";
   String mesage2 = "Warning! The temperature is below 20'C! Please heating up!";
   String mesage3 = "Info ! The temperature is 20'C!";
   String mesage4 = "Alarm! The temperature is above 30'C! Find the closest division to transfer 
the goods!";
   String mesage5 = "Alarm! The temperature is below 10'C! Find the closest division to transfer 
the goods!";

   public ServerHandler(String mqttURI) {
      this.mqttURI = mqttURI;
   }

   public ServerHandler(String sip, String sport, String stopic, String l_topic) {
      this.ip = sip;
      this.port = sport;
      this.topic = subTopicString;
      this.lis_topic = subTopicString;
   }


   public void start() throws MqttException {
      if(mqttURI == null || mqttURI.equalsIgnoreCase(""))
      {
         mqttURI = "tcp://"+this.ip+":"+this.port;
      }
      client = new MqttClient(mqttURI, "ServerHandler1");
      client.setCallback(this);
      client.connect();
      if(this.lis_topic != null && !this.lis_topic.equalsIgnoreCase(""))
      {
         client.subscribe(this.lis_topic);
      }else
      {
         client.subscribe(MQTT_TRACKSIDE_TOPIC, 2);
      }

      System.out.println("Connected and subscribed to "+mqttURI);
   }


   public void stop() throws MqttException {
      if (client != null) {
         client.disconnect();
      }
      System.out.println("Disconnect from "+mqttURI);
   }


   public void connectionLost(Throwable arg0) {
      // In a production system, we would decide what we want to do in the event the
      // connection drops. In our case we're expecting that not to happen so simply
      // write a message to standard error.
      System.err.println("The connection to WMQ was lost. Closing down the connection.");
   }


   public void deliveryComplete(MqttDeliveryToken arg0) {
      // We're just subscribing in this example.
   }


   public void messageArrived(MqttTopic topic, MqttMessage message)
         throws Exception {

      // Message received, simply write it onto the console.

      String tex = "";
      ByteArrayInputStream bais = new ByteArrayInputStream(message.getPayload());
      DataInputStream dis = new DataInputStream(bais);

      // Use the Java input streams to do the type conversion for us.
      tex = dis.readLine();

      dis.close();
      bais.close();

      System.out.println("Message " + tex);
      //String Truck_no = getTruckNo(topic.getName()); // The Truck number could be used to
construct the topic
      String tep = getTemperature(tex);

      int temperature = Integer.valueOf(tep).intValue();

      MsgSender sender = new MsgSender(this.ip,this.port,pubTopicString);

      if(temperature > 30)
      {
         sender.send(this.mesage4);
      }else if(temperature > 20)
      {
         sender.send(this.mesage1);
      }else if (temperature < 10)
      {
         sender.send(this.mesage5);
      }else if (temperature < 20)
      {
         sender.send(this.mesage2);
      }if (temperature == 20)
      {
            //sender.send(this.mesage3);
        }


    }

    public String getTruckNo(String topic)
    {
        if (topic == null) return "00001";
        String[] tps = topic.split("/");
        if(tps.length > 1)
        {
            return tps[1];
        }
        {
            return "";
        }
    }

    public String getTemperature(String message)
    {
        if (message == null) return "15";
        String[] tps = message.split(",");
        if(tps.length > 2)
        {
            String temp = tps[3].substring(0, tps[3].length()-1);
            return temp;
        }
        {
            return "0";
        }
    }


    public static void main(String[] args) {
        ServerHandler sub = null;

        if(args != null && args.length == 1)
        {
            sub = new ServerHandler(args[0]);
        }else if(args != null && args.length == 4)
        {
            sub = new ServerHandler(args[0],args[1],args[2],args[3]);
        }


        try {
            sub.start();
        } catch (MqttException ex) {
            System.err.println("Could not connect to WMQ: "+ex);
        }
    }

}
```

If the `ServerHandler.java` needs to send commands back to the truck, it calls out to a second library-built Java application, `MsgSender.java` (see Example 3-5).

Example 3-5 MsgSender.java

```
package com.ibm.redbooks.mqtt.ch3;

import java.util.Random;

import com.ibm.micro.client.mqttv3.MqttClient;
import com.ibm.micro.client.mqttv3.MqttConnectOptions;
import com.ibm.micro.client.mqttv3.MqttDeliveryToken;
import com.ibm.micro.client.mqttv3.MqttException;
import com.ibm.micro.client.mqttv3.MqttMessage;
import com.ibm.micro.client.mqttv3.MqttTopic;

public class MsgSender {
   String ip = "";
   String port = "";
   String message = "";
   String topic = "";

   public MsgSender(String sip, String sport, String stopic)
   {
      this.ip = sip;
      this.port = sport;
      this.topic = stopic;
   }

   public MsgSender(String sip, String sport, String smessage ,String stopic)
   {
      this.ip = sip;
      this.port = sport;
      this.message = smessage;
      this.topic = stopic;
   }

   public String getMessage() {
      return message;
   }

   public void setMessage(String message) {
      this.message = message;
   }

   public void send(String msg)
   {
      try {
         Random random = new Random(System.currentTimeMillis());
         int address = random.nextInt(1000);
         String clientid = "MQTTClient" + address;
         String iddead = clientid + " is drop!";
         MqttClient mqttClient = new MqttClient("tcp://"+ip+":"+port,clientid);
          MqttTopic iotopic = mqttClient.getTopic("Dead");

         MqttConnectOptions connectOptions = new MqttConnectOptions();
```

```
            connectOptions.setCleanSession(true);
            connectOptions.setWill(iotopic, iddead.getBytes(), 2, true);
            mqttClient.connect(connectOptions);

            MqttMessage message = new MqttMessage();
            message.setQos(2);
            message.setPayload(msg.getBytes());

            MqttTopic topic = mqttClient.getTopic(this.topic);
            if(msg == null || msg.equalsIgnoreCase(""))
            {
               msg = this.message;
            }
            MqttDeliveryToken token = topic.publish(message);

            token.waitForCompletion();

            mqttClient.disconnect();

         } catch (MqttException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
         }
      }
      public static void main(String[] args)
      {


      }

   }
```

Example 3-6 shows a different Java application, `MQIServerHandler.java`. It uses the Message Queue Interface (MQI) of WebSphere MQ to connect to the queue manager. As with the library-built application described previously, this one also calls out to a second application, `MQIPubSender.java` (see Example 3-7 on page 97), to handle any commands that need to be sent back out to the truck.

Example 3-6 MQIServerHandler.java

```
package com.ibm.redbooks.mqtt.ch3.server.mqi;

import java.io.IOException;

import com.ibm.mq.MQC;
import com.ibm.mq.MQEnvironment;
import com.ibm.mq.MQException;
import com.ibm.mq.MQGetMessageOptions;
import com.ibm.mq.MQMessage;
import com.ibm.mq.MQQueue;
import com.ibm.mq.MQQueueManager;
import com.ibm.mq.constants.MQConstants;
import com.ibm.mq.headers.MQDataException;
import com.ibm.mq.headers.MQHeader;
import com.ibm.mq.headers.MQHeaderIterator;
```

```
public class MQIServerHandler implements Runnable {
   private static final String MQ_MANAGER = "mqtt_qm";
   private static final String MQ_HOST_NAME = "localhost";
   private static final String MQ_CHANNEL = "TEST";
   private static final String MQ_QUEUE_NAME = "TemperatureQueue";
   private static final int MQ_PROT = 1414;
   private static final String fix_test_topic = "Truck_00001_Command";
   String mesage1 = "Warning! The temperature is above 20'C! Please cooling! ";
   String mesage2 = "Warning! The temperature is below 20'C! Please heating up!";
   String mesage3 = "Info ! The temperature is 20'C!";

   String mesage4 = "Alarm! The temperature is above 30'C! Find the closest division to transfer
the goods!";
   String mesage5 = "Alarm! The temperature is below 10'C! Find the closest division to transfer
the goods!";

   public void run() {
      MQQueueManager mqQueueManager = null;
      MQQueue mqQueue = null;
      try {
         MQEnvironment.addConnectionPoolToken();
         MQEnvironment.hostname = MQ_HOST_NAME;
         MQEnvironment.channel = MQ_CHANNEL;
         MQEnvironment.port = MQ_PROT;
         // MQEnvironment.userID = "GUEST";
         MQEnvironment.properties.put(MQC.TRANSPORT_PROPERTY,
               MQC.TRANSPORT_MQSERIES);
         // MQEnvironment.CCSID = MQ_CCSID;
         mqQueueManager = new MQQueueManager(MQ_MANAGER);
         int receiveOptions = MQC.MQOO_INPUT_SHARED
               | MQC.MQOO_FAIL_IF_QUIESCING;

         int openOptions = MQConstants.MQOO_INPUT_AS_Q_DEF
               | MQConstants.MQOO_OUTPUT;
         mqQueue = mqQueueManager.accessQueue(MQ_QUEUE_NAME, openOptions);
         MQGetMessageOptions mqGetMessageOptions = new MQGetMessageOptions();
         mqGetMessageOptions.options = mqGetMessageOptions.options
               + MQC.MQGMO_SYNCPOINT + MQC.MQGMO_WAIT
               + MQC.MQGMO_FAIL_IF_QUIESCING;
         mqGetMessageOptions.waitInterval = MQC.MQWI_UNLIMITED; //
mqGetMessageOptions.waitInterval
                                                   // = 1000 * 10;
         while (true) {

            MQMessage mqMessage = new MQMessage();
            mqQueue.get(mqMessage, mqGetMessageOptions);
            // String topic = mqMessage.getStringProperty("mqps.Top");//
            // Could be used to analyze the client id, and rebuild the topic
            // string
            String msg = getMQMessage(mqMessage);
            int temperature = Integer.valueOf(getTemperature(msg))
                  .intValue();
            MQIPubSender sender = new MQIPubSender(mqQueueManager,
                  fix_test_topic);
            if (temperature > 30) {
               sender.send(this.mesage4);
            } else if (temperature > 20) {
               sender.send(this.mesage1);
            } else if (temperature < 10) {
               sender.send(this.mesage5);
            } else if (temperature < 20) {
               sender.send(this.mesage2);
            }
            if (temperature == 20) {
               //sender.send(this.mesage3);
            }
            mqQueueManager.commit();
         }
      } catch (MQException e) {
         e.printStackTrace();
      } catch (Exception e1) {
         e1.printStackTrace();
      } finally {
         if (mqQueue != null) {
            try {
               mqQueue.close();
            } catch (MQException e) {
               e.printStackTrace();
            }
         }
         if (mqQueueManager != null) {
            try {
               mqQueueManager.close();
            } catch (MQException e) {
               e.printStackTrace();
            }
         }
      }
   }

   private String getMQMessage(MQMessage mqMsg) throws MQDataException,
         IOException {

      System.out.println("Retrieved message header:\n");
      MQHeaderIterator headerIterator = new MQHeaderIterator(mqMsg);
      while (headerIterator.hasNext()) {
         MQHeader header = headerIterator.nextHeader();
         System.out.println("Header " + header.type() + " : " + header);
      }
      String msgText = mqMsg.readLine();
      System.out.println("Retrieved message body: \n" + msgText);
      return msgText;
   }

   public String getTruckNo(String topic) {
      if (topic == null)
         return "00001";
      String[] tps = topic.split("/");
      if (tps.length > 1) {
         return tps[1];
      }
      {
         return "";
      }
   }

   public String getTemperature(String message) {
      if (message == null)
         return "15";
      String[] tps = message.split(",");
      if (tps.length > 2) {
         String temp = tps[3].substring(0, tps[3].length() - 1);
         return temp;
      }
      {
         return "0";
      }
   }

   public static void main(String arg[])

   {
      MQIServerHandler client = new MQIServerHandler();
      Thread mqClientThread = new Thread(client);
      mqClientThread.start();

   }
}
```

To handle any commands that need to be sent back out to the truck, `MQIServerHandler.java` calls out to a second application, `MQIPubSender.java` (Example 3-7).

Example 3-7 MQIPubSender.java

```
package com.ibm.redbooks.mqtt.ch3.server.mqi;
import java.io.IOException;

import com.ibm.mq.MQException;
import com.ibm.mq.MQMessage;
import com.ibm.mq.MQQueueManager;
import com.ibm.mq.constants.CMQC;

public class MQIPubSender {

  private String topicObject;
  MQQueueManager queueManager;

  MQIPubSender(MQQueueManager queueManager, String topicString) {
    topicObject = System.getProperty("com.ibm.mq.pubSubSample.topicObject", topicString);
    this.queueManager = queueManager;
    return;
  }

  public static void main(String[] arg) {
    return;
  }
  public void send(String msg) throws MQException, IOException {
    int destinationType = CMQC.MQOT_TOPIC;
    /* Do the publish */
    MQMessage messageForPut = new MQMessage();
    System.out.println("***Publishing ***");
    messageForPut.writeString(msg);
    queueManager.put(destinationType, topicObject, messageForPut);
    return;
  }

}
```

3.3 WebSphere MQ classes for Java Message Service

WebSphere MQ classes for Java Message Service (JMS) is a JMS provider that is supplied with WebSphere MQ. This section focuses on understanding the interoperability between MQTT and JMS clients, including how a message sent from an MQTT client can be consumed by a JMS client and vice-versa.

First, it is important to understand the differences between the MQTT protocol and the JMS Application Programming Interface (API):

- JMS is a Java API, whereas MQTT is a wire-level protocol.
- JMS is an API that is used by application developers to write programs, whereas MQTT is a protocol used by the client and server to communicate.
- The JMS API is implemented by a JMS provider such as WebSphere MQ, which can use a proprietary protocol internally to communicate with the messaging server. The MQTT protocol enables an MQTT-enabled client to communicate with an MQTT-enabled server.

Although JMS and MQTT are fundamentally different, they must be able to work together, such as when messages from an MQTT client need to be received by a JMS client or the other way around.

For example, a temperature sensor might use MQTT to send its readings to the WebSphere MQ server, but the temperature monitoring application that ultimately receives and interprets those readings might use the JMS API. In such a scenario, the WebSphere MQ queue manager would convert the MQTT messages to a type that can be parsed by the JMS client program. The messages from the MQTT client to the WebSphere MQ queue manager would use an RFH header internally to identify certain properties of the MQTT message. The JMS client would then request the message, which would be sent with the RFH header. The JMS client would then parse the properties and convert the message to a JMSBytesMessage.

For more details about JMS, refer to these websites:

- Java Message Service Tutorial:

  http://docs.oracle.com/javaee/1.3/jms/tutorial/
- J2EE pathfinder: Enterprise messaging with JMS:

  http://www.ibm.com/developerworks/java/library/j-pj2ee5/

3.3.1 Sample applications

Many implementations of the JMS API are available. The sample applications presented here use the one provided with WebSphere MQ.

The first sample here is a publish/subscribe application using an MQTT-based client as the publisher and a JMS client using WebSphere MQ Classes for JMS as the subscriber. A different sample application using JMS as the publisher and a WebSphere MQ Telemetry client as the subscriber is provided later in this section.

MQTT publisher to JMS subscriber

To build the MQTT-to-JMS application, you will need the following items set up:

- A WebSphere MQ Telemetry listener running at port 1883
- The Eclipse development environment to develop applications

MQTT publisher

For the publisher, follow the instructions provided in 2.5.3, "Publisher and subscriber in Java" on page 37 to import the sample MQTT publisher into your workspace.

JMS subscriber

Follow these steps to build an asynchronous subscriber application that uses WebSphere MQ Classes for JMS:

1. In Eclipse, create a new Java project, such as `JMSSubscriber`.
2. In the `src` folder, create the class JMSSubscriber with the package name `test`.
3. Add the jars `<MQINSTALLPATH>\java\lib\com.ibm.mqjms.jar` and `<MQINSTALLPATH>\java\lib\jms.jar` to the build path.
4. Copy the code from Example 3-8 into `JMSSubscriber.java`. The sample code was developed for a WebSphere MQ V7.0.1.7 client and Java V6.

Example 3-8 JMSSubscriber.java

```
package test;

import javax.jms.Connection;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageListener;
import javax.jms.Session;

import com.ibm.mq.jms.MQConnectionFactory;

public class JMSSubscriber implements MessageListener {

        private Connection conn;
        private Session sess;
        private MessageConsumer consumer;

        public static void main(String\[\] args) {
                JMSSubscriber jmssub = new JMSSubscriber();
                jmssub.createSubscriber();
                try {
                        Thread.sleep(1000*60*20); // Run for 20 minutes
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                jmssub.closeSubscriber();
        }

        private void closeSubscriber() {
                try {
                        conn.stop();
                        consumer.close();
                        sess.close();
                        conn.close();
                        System.out.println("JMS Subscriber closed");
                } catch (JMSException e) {
                        e.printStackTrace();
                        if(e.getLinkedException() \!= null) {
                                e.getLinkedException().printStackTrace();
                        }
                }

        }

        private void createSubscriber() {
                MQConnectionFactory connFact = new MQConnectionFactory();
                try {
                        //We are connecting to queue manager using client mode
                        connFact.setQueueManager("MQQM");
                        connFact.setHostName("127.0.0.1");
                        connFact.setPort(1414);

                        conn = connFact.createConnection();
                        sess = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
                        consumer = sess.createConsumer(sess.createTopic("routes/Truck01"));
                        consumer.setMessageListener(this);
                        System.out.println("JMS Subscriber started and subscribed to topic 
routes/Truck01");
                        //Now we are ready to start receiving the messages. Hence start the 
connection
                        conn.start();
                } catch (JMSException e) {
                        e.printStackTrace();
                        if(e.getLinkedException() \!= null) {
                                e.getLinkedException().printStackTrace();
                        }
                }

        }

        @Override
        public void onMessage(Message message) {
                System.out.println(message);

        }
}
```

JMS subscriber output

Run the JMSSubscriber program, and then start the MQTT publisher program. Be sure that the WebSphere MQ queue manager is configured for telemetry, as described in 3.1.4, “Configuring a queue manager for WebSphere MQ Telemetry” on page 65.

The output message is shown in Example 3-9.

Example 3-9 JMSSubscriber output

```
JMSMessage class: jms_bytes
  JMSType:         null
  JMSDeliveryMode:  1
  JMSExpiration:    0
  JMSPriority:      0
  JMSMessageID:     ID:414d51204d51514d2020202020202020207e98614f20002601
  JMSTimestamp:     1331796336760
  JMSCorrelationID: ID:414d51204d51514d2020202020202020207e98614f20002504
  JMSDestination:   topic://routes/Truck01
  JMSReplyTo:       null
  JMSRedelivered:   false
    JMSXAppID: MQQM
    JMSXDeliveryCount: 1
    JMSXUserID: neekrish
    JMS_IBM_Encoding: 0
    JMS_IBM_Format:
    JMS_IBM_MsgType: 8
    JMS_IBM_PutApplType: 26
    JMS_IBM_PutDate: 20120315
    JMS_IBM_PutTime: 07253676
48656c6c6f20776f726c64
```

The message received by the client will be a JMSBytesMessage. Example 3-10 shows the code that is required to read the JMSBytesMessage and display it as a string. This code might be useful in cases when the message sent is a text message and needs to be used for further processing as a string rather than bytes.

Example 3-10 Displaying the JMSBytesMessage as a string

```
public void onMessage(Message message) {
        System.out.println(message);
        BytesMessage bytesMessage = (BytesMessage)message;
        try {
                long bodyLength = bytesMessage.getBodyLength();
                if (bodyLength > Integer.MAX_VALUE) {
                        // array is too large
                        return;
                }
                byte[] msgBytes = new byte[(int)bodyLength];
                String messageData = new String();

                while(bytesMessage.readBytes(msgBytes) != -1) {
                        String tmpStr = new String(msgBytes);
                        messageData += tmpStr;
                }

                System.out.println("The message received is " + messageData);
        } catch (JMSException e) {
                e.printStackTrace();
                if(e.getLinkedException() != null) {
                        e.getLinkedException().printStackTrace();
                }
        }
}
```

The code in Example 3-10 shows that the length of the message body is determined using the `BytesMessage.getBodyLength()` method, after which space is allocated for the message in the byte array. Then the `BytesMessage.readBytes()` method is used to read the bytes. The string constructor is then used to convert the message from a byte array to a string.

JMS publisher to WebSphere MQ Telemetry subscriber

Another message path is from a JMS client publisher to a WebSphere MQ Telemetry subscriber.

WebSphere MQ Telemetry subscriber

See 2.5.3, "Publisher and subscriber in Java" on page 37 for a WebSphere MQ Telemetry subscriber program.

JMS publisher

Then create a JMS publisher program as shown in Example 3-11.

Example 3-11 JMSPublisher.java

```
package test;

import javax.jms.Connection;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;

import com.ibm.mq.jms.MQConnectionFactory;

public class JMSPublisher {
        private Connection conn;
        private Session sess;
        private MessageProducer publisher;

        public static void main(String\[\] args) {
                JMSPublisher jmspub = new JMSPublisher();
                jmspub.createPublisher();
                jmspub.closePublisher();
        }

        private void closePublisher() {
                try {
                        publisher.close();
                        sess.close();
                        conn.close();
                        System.out.println("JMS Publisher closed");
                } catch (JMSException e) {
                        e.printStackTrace();
                        if(e.getLinkedException() \!= null) {
                                e.getLinkedException().printStackTrace();
                        }
                }

        }

        private void createPublisher() {
                MQConnectionFactory connFact = new MQConnectionFactory();
                try {
                        //We are connecting to queue manager using client mode
                        connFact.setQueueManager("MQQM");
                        connFact.setHostName("127.0.0.1");
                        connFact.setPort(1414);

                        conn = connFact.createConnection();
                        sess = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
                        publisher = sess.createProducer(sess.createTopic("routes/Truck01"));
                        publisher.send(sess.createTextMessage("Hello World"));
                        System.out.println("JMS Publisher started and published message to 
topic routes/Truck01");

                } catch (JMSException e) {
                        e.printStackTrace();
                        if(e.getLinkedException() \!= null) {
                                e.getLinkedException().printStackTrace();
                        }
                }

        }
}
```

Certain fields of the JMS message can be set by the MQTT client. The JMS delivery mode field is used by JMS clients to mark a message as persistent or nonpersistent. A persistent message will not be lost on transit to the destination. So messages with an MQTT QoS level of 0 are equivalent to JMS nonpersistent messages, whereas MQTT QoS 1 and 2 messages are equivalent to JMS persistent messages.

Point-to-point JMS applications and MQTT

MQTT is a publish/subscribe-based protocol. But there might be situations where a JMS client is designed to be a point-to-point application, such as when it intends to send a message to only one client.

Example 3-12 is a program that will send a message to the same MQTT subscriber built earlier in this section.

Example 3-12 JMSSender.java

```
package test;

import javax.jms.Connection;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;

import com.ibm.mq.jms.MQConnectionFactory;
```

```
public class JMSSender {
	private Connection conn;
	private Session sess;
	private MessageProducer sender;

	public static void main(String[] args) {
		JMSSender jmsSender = new JMSSender();
		jmsSender.createSender();
		jmsSender.closeSender();
	}

	private void closeSender() {
		try {
			sender.close();
			sess.close();
			conn.close();
			System.out.println("JMS Sender closed");
		} catch (JMSException e) {
			e.printStackTrace();
			if(e.getLinkedException() != null) {
				e.getLinkedException().printStackTrace();
			}
		}

	}

	private void createSender() {
		MQConnectionFactory connFact = new MQConnectionFactory();
		try {
			//We are connecting to queue manager using client mode
			connFact.setQueueManager("MQQM");
			connFact.setHostName("127.0.0.1");
			connFact.setPort(1414);

			conn = connFact.createConnection();
			sess = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
			sender = 
sess.createProducer(sess.createQueue("queue://Truck01/routes/Truck01"));
			sender.send(sess.createTextMessage("Hello World"));
			System.out.println("JMS Sender started and published message to topic 
routes/Truck01");

		} catch (JMSException e) {
			e.printStackTrace();
			if(e.getLinkedException() != null) {
				e.getLinkedException().printStackTrace();
			}
		}

	}
}
```

3.3.2 Troubleshooting

If the exception shown in Example 3-13 is thrown by the JMS client, ensure that the queue manager is started and the listener is running.

Example 3-13 Example of exception thrown by JMS client

```
com.ibm.msg.client.jms.DetailedIllegalStateException: JMSWMQ0018: Failed to
connect to queue manager 'MQQM' with connection mode 'Bindings' and host name
'127.0.0.1(1414)'. Check the queue manager is started and if running in client
mode, check there is a listener running. Please see the linked exception for more
information. com.ibm.mq.MQException: JMSCMQ0001: WebSphere MQ call failed with
compcode '2' ('MQCC_FAILED') reason '2059' ('MQRC_Q_MGR_NOT_AVAILABLE').
```

If the following shown in Example 3-14 is returned by the MQTT client, ensure that the queue manager is running and the MQTT listener is created and running on the queue manager.

Example 3-14 Example of exception thrown by MQTT client

```
Unable to connect to server (32103)
at
com.ibm.micro.client.mqttv3.internal.ExceptionHelper.createMqttException(Exception
Helper.java:33)
at
com.ibm.micro.client.mqttv3.internal.TCPNetworkModule.start(TCPNetworkModule.java:
64)
at com.ibm.micro.client.mqttv3.internal.ClientComms.connect(ClientComms.java:140)
at com.ibm.micro.client.mqttv3.MqttClient.connect(MqttClient.java:344)
at com.ibm.micro.client.mqttv3.MqttClient.connect(MqttClient.java:313)
at test.MQTTPubTest.main(MQTTPubTest.java:14)
```

3.4 WebSphere MQ Telemetry daemon for devices

The WebSphere MQ Telemetry daemon for devices is an advanced MQTT V3 client that can act as a concentrator to connect telemetry channels to a queue manager, shown in Figure 3-21 on page 106. This connection makes it possible to minimize the number of concurrent channel connections on a WebSphere MQ queue manager.

The daemon can also be used to store and forward messages from other MQTT clients. It connects to WebSphere MQ like an MQTT client but can also have other MQTT clients connected to it. You can even connect it to other telemetry daemons to create a complex network of remote devices.

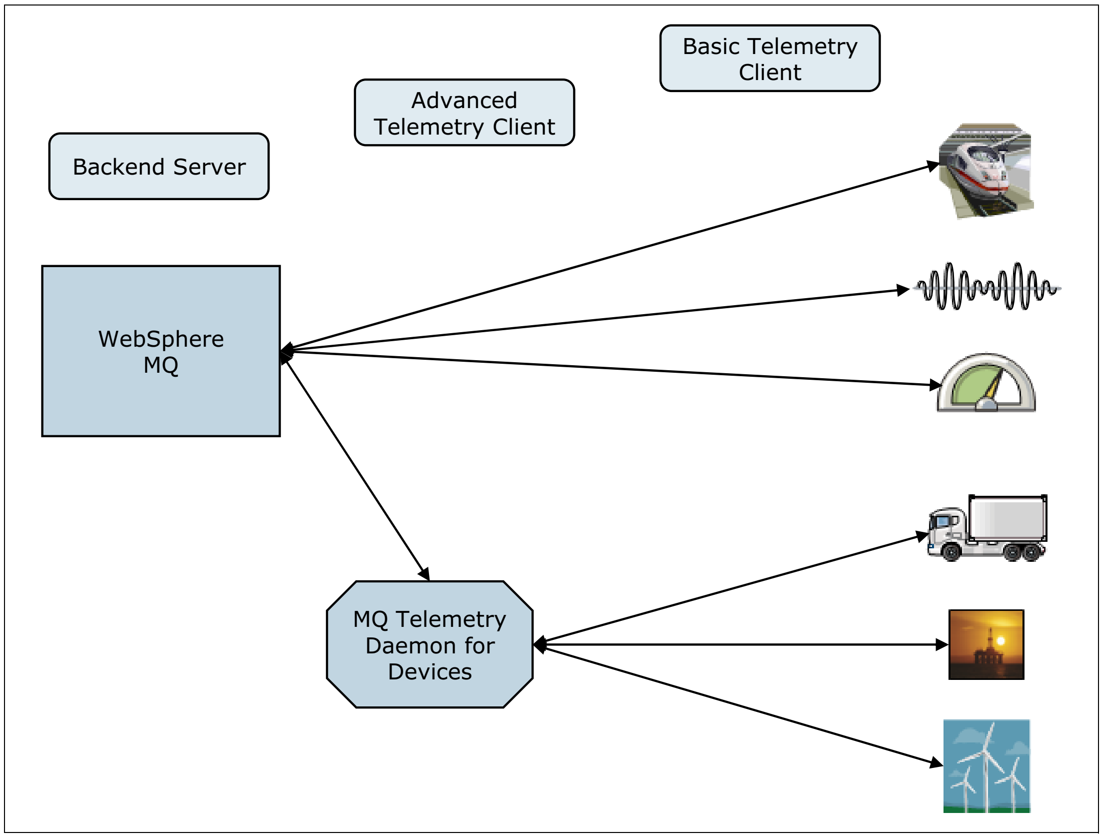


Figure 3-21 Typical system architecture with the WebSphere MQ Telemetry daemon for devices

The WebSphere MQ Telemetry daemon for devices has the following primary functions:

- It connects MQTT clients in a publish/subscribe messaging network. For example, the sensor and actuator of a device can be connected to the daemon as separate MQTT clients. The sensor publishes its readings while the actuator subscribes to the readings and modifies its behavior based on their values.
- It manages which subscriptions and messages are published to the queue manager and to the device. For example, the daemon can forward to the queue manager a subscription request created by the actuator on a remote client.
- It combines multiple MQTT client data streams onto a single TCP/IP telemetry channel connection. Rather than each device connecting separately to the telemetry server, the daemon transmits publications and subscriptions by way of a single TCP/IP connection. This reduces the number of TCP/IP connections by making just one connection to the queue manager on behalf of all the clients that need to communicate with it.
- It stores messages from devices and forwards them to the queue manager. This helps to insulate the messaging system from short-term connection failures or other issues by making it possible to store messages in a buffer for later delivery. For example, some devices can publish a message only once, regardless of whether it is ever received. This makes messaging unreliable if the connection to the queue manager is only available intermittently, or is subject to sudden disruption. However, by attaching the device to the daemon using a local connection that is always available, the daemon, when necessary, can buffer any messages received when the network is down and send them on to the queue manager when it comes back up.

Important: The daemon can buffer the messages that flow to and from the queue manager in its memory. However, it does not have persistent storage for in-flight messages interrupted during transmission.

3.4.1 Installing and configuring the WebSphere MQ Telemetry daemon for devices

Installation, configuration, and control of the WebSphere MQ Telemetry daemon for devices is file-based. The daemon is installed by copying the Software Development Kit (SDK) to the device where you are going to run the daemon. On Linux (64-bit), the directory path containing the daemon is `/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linuxia64`.

To start the daemon with its default configuration, execute the `amqtdd` script. The default configuration file is named `amqtdd.cfg` and is located in the same directory as the daemon executable program. Type the following syntax to start the daemon with its default configuration:

```
./amqtdd
```

Operation of the daemon can be modified with entries in its configuration file. The configuration file can specify listener ports, topics, retained persistence and other parameters. For details, refer to the WebSphere MQ Information Center telemetry reference section at this location:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60912_.htm

To start the daemon with a different configuration file, pass the new path and file name as a single parameter, using the configuration file name `testdaemon.cfg`. To execute this process type the following syntax:

```
./amqtdd testdaemon.cfg
```

When started in this manner, the daemon checks for the existence of the designated configuration file. If the file does not exist, the daemon runs with its default settings.

You can also change some of the configuration options while the daemon is running. To control a running daemon, create entries in a new temporary daemon command file named `amqtdd.upd`. The daemon reads this file every five seconds, runs the commands, and then deletes the file.

3.4.2 WebSphere MQ Telemetry daemon for devices bridges

The WebSphere MQ Telemetry daemon for devices supports one or more simultaneous connections to other brokers through the use of *bridges*, illustrated in Figure 3-22 on page 108. These bridges are defined by setting bridge parameters within the daemon configuration file.

A bridge connects two publish/subscribe brokers that are using the MQTT protocol. It propagates publications from one broker to the other, in either direction, by designating *in*, *out*, or *both* in the configuration file.

Typically, the local broker is the WebSphere MQ Telemetry daemon for devices. The remote broker can be any publish/subscribe broker that supports the MQTT V3 protocol, such as another instance of the daemon or WebSphere MQ. MQTT V3 support is required because the daemon itself is an MQTT V3 client.

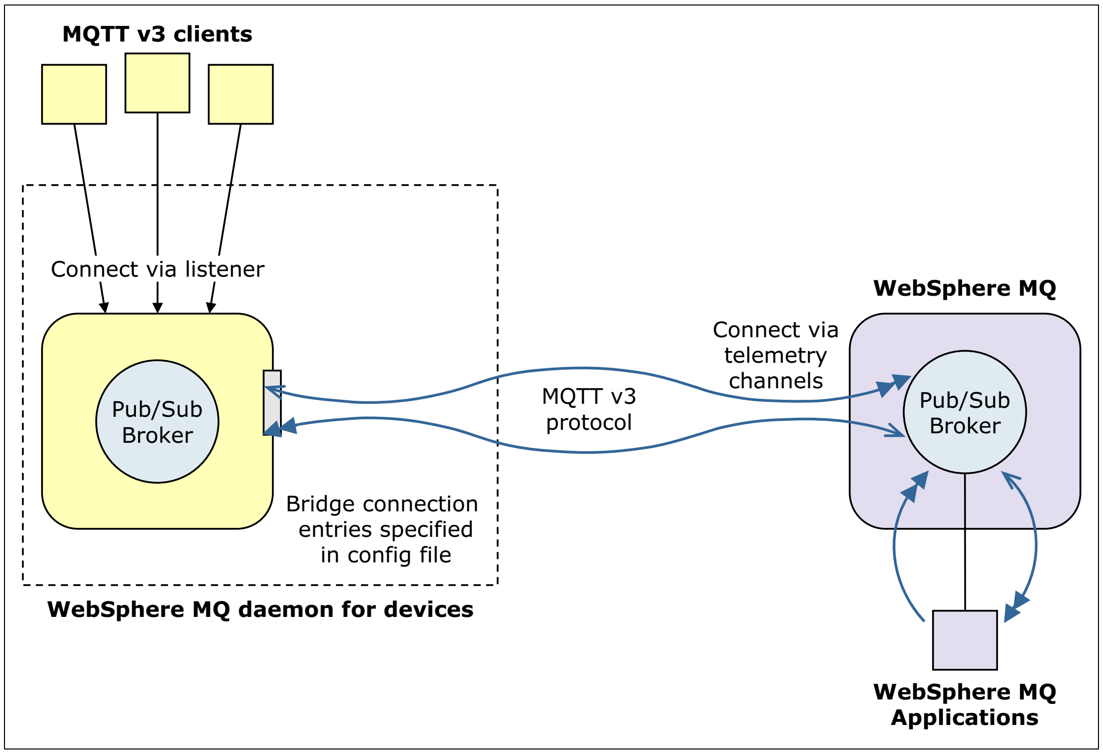

MQTT v3 clients
Connect via listener
Pub/Sub Broker
Bridge connection entries specified in config file
WebSphere MQ daemon for devices
MQTT v3 protocol
Connect via telemetry channels
WebSphere MQ
Pub/Sub Broker
WebSphere MQ Applications

Figure 3-22 Example of using a bridge to connect the daemon to WebSphere MQ

Multiple bridges can be configured to connect a network of daemons that can exchange messages, as shown in 3.4.3, “Sample implementations of WebSphere MQ Telemetry daemon for devices” on page 109. Each bridge can route messages to topics at its local daemon, to topics at another daemon, to a WebSphere MQ publish/subscribe broker, or to any other MQTT V3 broker to which it is connected.

Advantages of using bridges

The following list is composed of several reasons a developer might choose to bridge daemons to WebSphere MQ:

- Fewer MQTT client connections to the WebSphere MQ queue manager. Using a hierarchy of daemons, you can effectively connect more clients to WebSphere MQ than can be connected using a single queue manager because the clients come together at the daemon and are then connected to the queue manager by way of a single channel.
- Messages can be stored and forwarded between MQTT clients and WebSphere MQ. This can help avoid the need to maintain continuous network connections for devices that do not have their own storage.
- It enables filtering of publications exchanged between MQTT clients and WebSphere MQ. A common example of filtering at the daemon level is to divide publications into messages that are processed locally (such as control flows between sensors and actuators) and messages that involve other applications (such as requests for readings, status, and configuration commands).
- Topic spacing can be modified to avoid situations where strings from clients attached to different listener ports collide with each other. An example is using the daemon to label the readings coming from different trucks. For more details about using the topic parameter to

configure the bridge, refer to the bridges section of the WebSphere MQ Information Center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60916_.htm

Security for daemons and bridges

For this publication, sample bridge daemons were run with a unique ID, *mqttuser00*. For the client to have the necessary authorizations to publish data to the queue manager, this ID had to be defined on the Linux server where the queue manager existed. So the mqttuser00 ID was added to the mqtt user group and granted publishing authority using the **`setmqaut`** command. This is shown in the following syntax and explained in greater detail in 3.1.5, "Authorizing MQTT clients to access WebSphere MQ" on page 72.

```
setmqaut -m MQTTQMGR -t topic -n SYSTEM.BASE.TOPIC -g mqtt -all +pub +sub
```

Additionally, bridge connection security can be controlled by setting a username and password for each bridge connection. This is done by entering a `username` and `password` as bridge connection parameters in the daemon configuration file, or by creating a password file as a global parameter.

For more information about this security for daemons and bridges, refer to the product information center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60524_.htm

3.4.3 Sample implementations of WebSphere MQ Telemetry daemon for devices

In this section we provide sample implementations of the WebSphere MQ Telemetry daemon for devices. First, multiple daemons are connected to each other using a bridge. Then, a daemon is connected to a WebSphere MQ queue manager, also using a bridge. The bridge feature can also be used to connect a daemon, which is an MQTT V3 client, to any MQTT V3-compliant broker.

The sample code shown here extends the transportation logistics scenario used throughout this book and builds upon the example code provided in earlier chapters.

Assumptions

Transportation Company B divides its fleet into two broad categories, North and South, based on the general destination of each truck. Within these categories, some trucks travel weekly but others travel daily, and all trucks must undergo routine maintenance at scheduled intervals.

Each truck publishes information using the MQTT protocol, so each truck can be considered an individual MQTT client. And although trucks can publish a wide variety of information, these examples relate only to transmitting the truck's current location, presented in terms of a time stamp and coordinates for longitude and latitude obtained by an on-board GPS device, and the status of the trip. The status reading is obtained from an on-board unit which the driver uses to designate the current status of the trip, either B(egin), T(ransit), P(ark), or E(nd).

Topic strings

The topic string for publication of each message is based on each truck's trip frequency and direction. Thus, for northbound daily trucks, the topic would be routes/north/daily/Truck01, where 01 is the ID assigned to a specific truck. Table 3-2 shows the topic strings that are used in these examples, with *nn* representing the two-digit truck ID numbers. For maximum clarity, the range of IDs for trucks in each category is provided.

Table 3-2 Topic strings

Topic String	Trucks	IDs
routes/north/daily/Truck*nn*	North-bound, daily	01 to 09
routes/north/weekly/Truck*nn*	North-bound, weekly	10 to 19
routes/south/daily/Truck*nn*	South-bound, daily	20 to 29
routes/south/weekly/Truck*nn*	South-bound, weekly	30 to 39

Message frequency and format

The trucks are expected to publish messages once every two minutes to a topic that is specific for each truck. The messages must be in comma-delimited format, as in `ddmmyyyyhhmmss,latitude,longitude,tripStatus`.

Daemon topologies

Three typical daemon topologies, ranging from simple to complex, will be addressed here. The first topology simply connects clients to daemons. The second connects daemons to other daemons using bridges. The third uses bridges to connect daemons to WebSphere MQ queue managers.

Connecting clients to daemons

To handle the publications for all trucks, one daemon is assigned for each of the truck categories and its associated topic string. Figure 3-23 shows a simple messaging topology with daemons and MQTT clients (trucks) only.

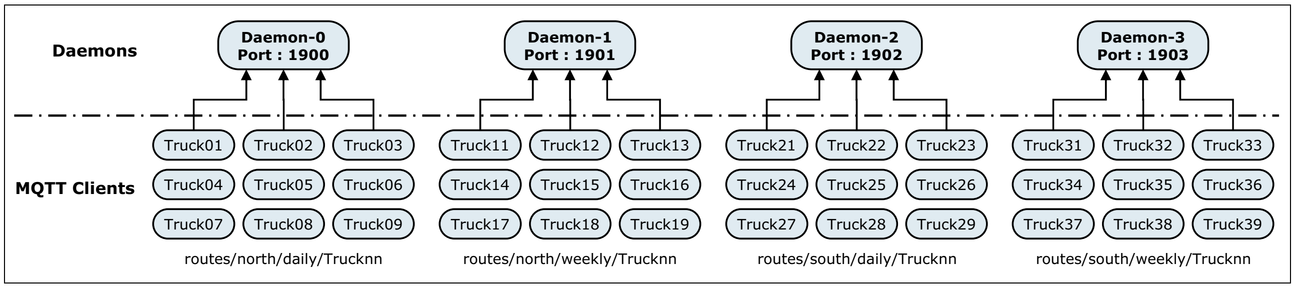

Daemons
Daemon-0 Port : 1900
Daemon-1 Port : 1901
Daemon-2 Port : 1902
Daemon-3 Port : 1903
MQTT Clients
Truck01 Truck02 Truck03 Truck04 Truck05 Truck06 Truck07 Truck08 Truck09
Truck11 Truck12 Truck13 Truck14 Truck15 Truck16 Truck17 Truck18 Truck19
Truck21 Truck22 Truck23 Truck24 Truck25 Truck26 Truck27 Truck28 Truck29
Truck31 Truck32 Truck33 Truck34 Truck35 Truck36 Truck37 Truck38 Truck39
routes/north/daily/Trucknn
routes/north/weekly/Trucknn
routes/south/daily/Trucknn
routes/south/weekly/Trucknn

Figure 3-23 Correlation of daemons with topic strings in a simple messaging topology

The daemon configuration file names, port numbers and corresponding topic strings are shown in Table 3-3.

Table 3-3 Daemon configuration information for simple messaging topology

Topic String	Daemon Configuration file	Port number
routes/north/daily/Truck*nn*	Daemon-0.cfg	1900
routes/north/weekly/Truck*nn*	Daemon-1.cfg	1901
routes/south/daily/Truck*nn*	Daemon-2.cfg	1902
routes/south/weekly/Truck*nn*	Daemon-3.cfg	1903

For example, Truck25 would publish its information to the topic routes/south/daily/Truck25 on the daemon running at port 1902, and so on. Truck25 is forced to publish to the daemon running on port 1902 using the `clientidprefixes` global parameter in the configuration file. Example 3-15 shows the daemon configuration file for connecting to port 1902.

Example 3-15 Daemon configuration file for Daemon-2 on port 1902

```
# Configuration file for Daemon-2
#
# Only trucks with client ID from Truck20 to Truck29 will publish to this daemon.
#
# Topic : routes/south/daily/Truck2n
#
# Global parameters only
#
port 1902
clientid_prefixes Truck2 BridgeDaemon
```

More information about global parameters in the daemon configuration file is available at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60914_.htm

Connecting daemons using bridges

You create bridges between daemons so that messages can be transferred not only among the clients and brokers attached to one daemon, but with clients and brokers attached to other daemons, too. This expands the potential communications network significantly. In addition, it has implications for the messaging system being developed by Transportation Company B.

Figure 3-23 on page 110 shows clients split among daemons for daily northbound trucks, daily southbound trucks, weekly northbound trucks and weekly southbound trucks. However, what if the company wanted to process messages from all northbound trucks, regardless of whether they travel daily or weekly?

The solution is to add a bridge between the daemons, as shown in Figure 3-24 on page 112. On one end will be Daemon-0 publishing to the topic routes/north/daily/# and Daemon-1 publishing to the topic routes/north/weekly/#. On the other end will be another daemon, BridgeDaemon-0, which will make publications for both of these topics available to other clients and brokers that are not directly connected to either Daemon-0 or Daemon-1.

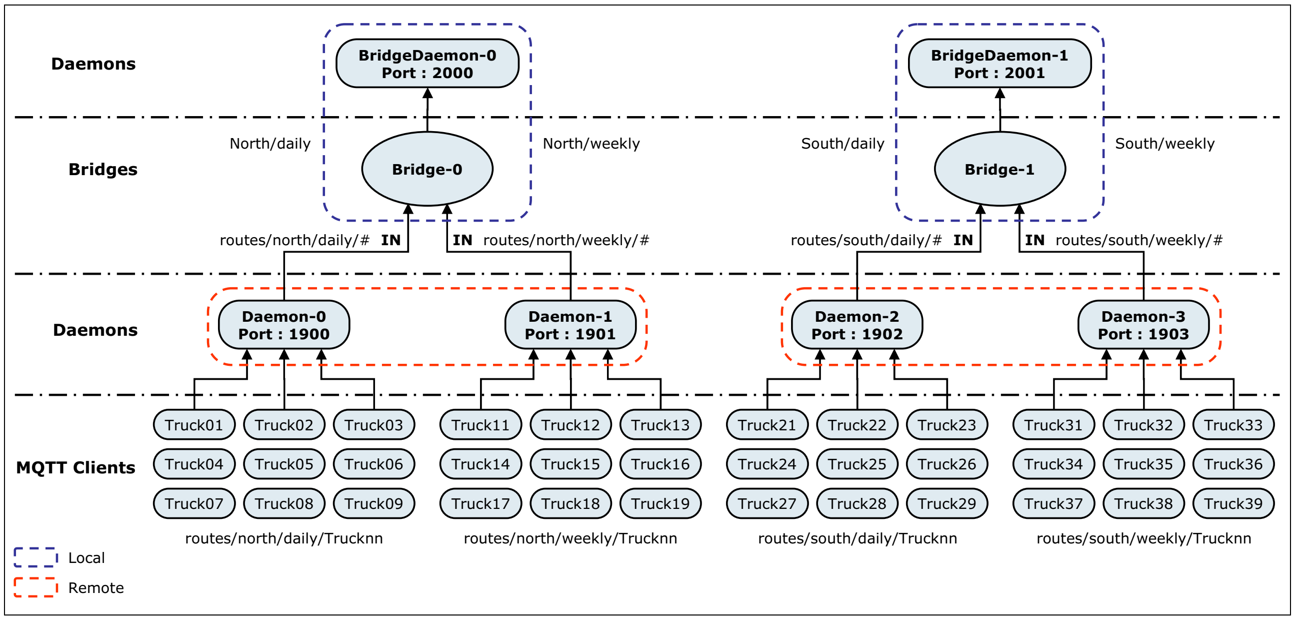

Daemons
BridgeDaemon-0 Port : 2000
BridgeDaemon-1 Port : 2001
Bridges
North/daily
North/weekly
South/daily
South/weekly
Bridge-0
Bridge-1
routes/north/daily/# IN
IN routes/north/weekly/#
routes/south/daily/# IN
IN routes/south/weekly/#
Daemons
Daemon-0 Port : 1900
Daemon-1 Port : 1901
Daemon-2 Port : 1902
Daemon-3 Port : 1903
MQTT Clients
Truck01 Truck02 Truck03 Truck04 Truck05 Truck06 Truck07 Truck08 Truck09
Truck11 Truck12 Truck13 Truck14 Truck15 Truck16 Truck17 Truck18 Truck19
Truck21 Truck22 Truck23 Truck24 Truck25 Truck26 Truck27 Truck28 Truck29
Truck31 Truck32 Truck33 Truck34 Truck35 Truck36 Truck37 Truck38 Truck39
routes/north/daily/Trucknn
routes/north/weekly/Trucknn
routes/south/daily/Trucknn
routes/south/weekly/Trucknn
Local
Remote

Figure 3-24 Correlation of daemons with topic strings in a topology using bridges

The bridge daemon configuration file names, port numbers and corresponding topic strings are shown in Table 3-4.

Table 3-4 Daemon configuration information for topology using bridges

Topic String	Daemon Configuration file	Port number
North/daily/#	Bridge-Daemon-0.cfg	2000
North/weekly/#	Bridge-Daemon-0.cfg	2000
South/daily/#	Bridge-Daemon-1.cfg	2001
South/weekly/#	Bridge-Daemon-1.cfg	2001

To start BridgeDaemon-0 on port 2000, including the bridges to Daemon-0 and Daemon-1, prepare the configuration file as shown in Example 3-16 on page 113.

Here is a breakdown of what the code in Example 3-16 on page 113 accomplishes:

1. First bridge configuration:
   a. With the connection name set as NorthDailyTrucks, the BridgeDaemon-0 connects to Daemon-0 as denoted by the address parameter.
   b. The bridge considers BridgeDaemon-0 as local.
   c. The bridge considers Daemon-0 running on port number 1900 as remote.
   d. The publications of Daemon-0 are considered as inputs to the bridge.
   e. For incoming publications, the bridge:
      i. Checks if the topic strings of publications match routes/north/daily/#
      ii. Delivers the publications to the topic North/daily/#
2. Second bridge configuration:
   a. With the connection name set as NorthWeeklyTrucks, BridgeDaemon-0 connects to Daemon-1 as denoted by the address parameter.

b. The bridge considers BridgeDaemon-0 as local. Note the blue boundary in the figure marking it as local.
c. The bridge considers Daemon-1 running on port 1901 as remote. Note the red boundary in the figure marking it as remote.
d. The publications of Daemon-1 are considered as inputs to the bridge.
e. For the incoming publications, the bridge
   i. Checks if the topics of publications match routes/north/weekly/#.
   ii. Delivers the publications to a topic string North/weekly/#.

Example 3-16 Daemon configuration file for BridgeDaemon-0 on port 2000

```
# Configuration file for BridgeDaemon-0
#
# Collect North daily and weekly truck messages into North daily and weekly
topics
#
# Topics (From Remote)
#      routes/north/daily/Truck0n
#      routes/north/weekly/Truck1n
#
# Topics (To Local)
#      North/daily
#      North/weekly
#
# Global parameters
#
port 2000
#
# Bridge parameters
#
connection NorthDailyTrucks
clientid BridgeDaemon01
address 9.42.170.179:1900
topic daily/# in North/ routes/north/
#
# Bridge parameters
#
connection NorthWeeklyTrucks
clientid BridgeDaemon02
address 9.42.170.179:1901
topic weekly/# in North/ routes/north/
```

Connecting daemons to WebSphere MQ queue managers

Figure 3-24 on page 112 showed a bridge between different daemons. But what if the enterprise applications at company headquarters need to get messages from all of the clients attached to all of the daemons?

One solution is to connect the daemons to WebSphere MQ queue managers, which can then act as the publication source for any applications that require data from the MQTT clients on the trucks, as shown in Figure 3-25 on page 114.

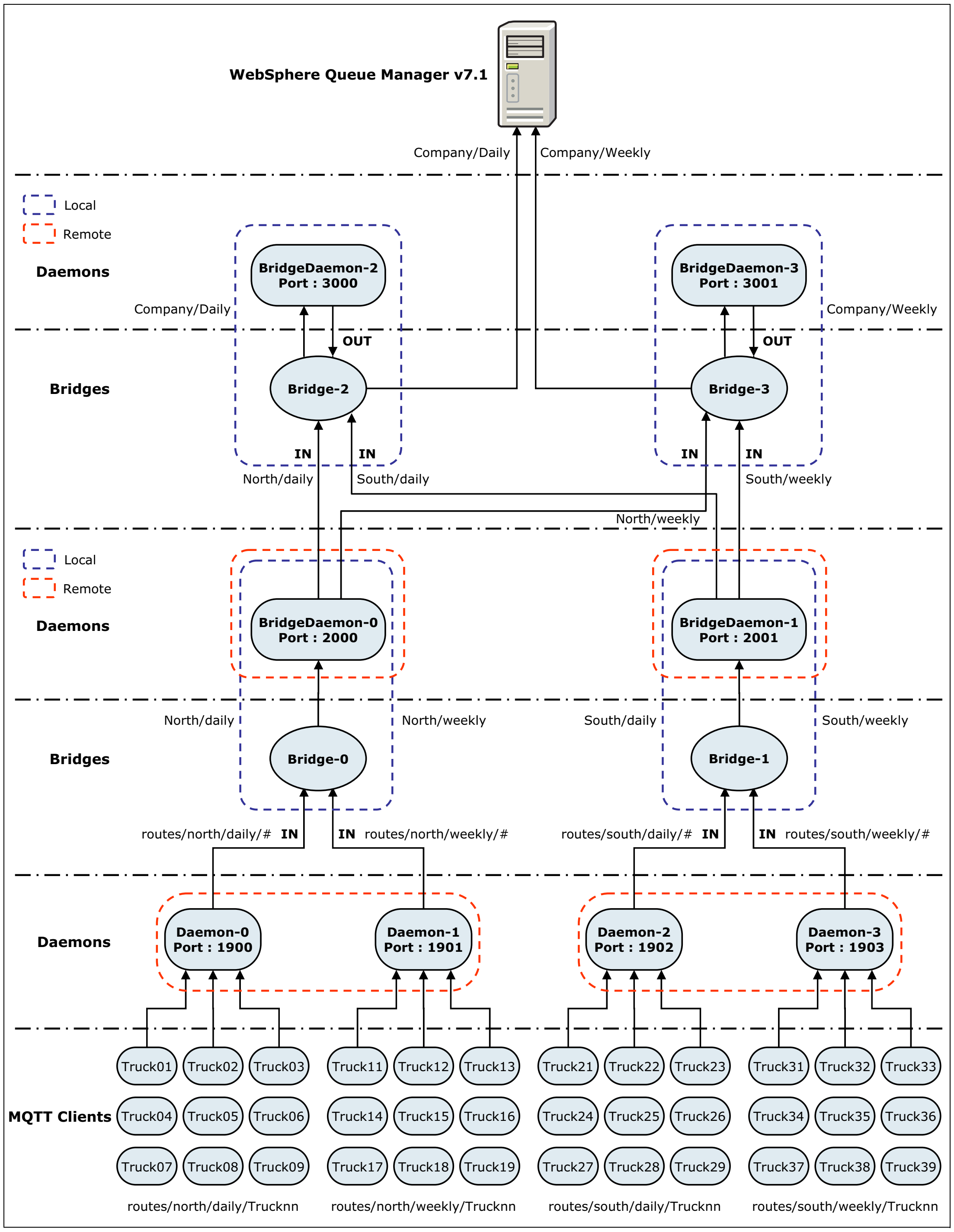

WebSphere Queue Manager v7.1
Company/Daily
Company/Weekly
Local
Remote
Daemons
BridgeDaemon-2 Port : 3000
BridgeDaemon-3 Port : 3001
Company/Daily
Company/Weekly
OUT
OUT
Bridges
Bridge-2
Bridge-3
IN
IN
IN
IN
North/daily
South/daily
South/weekly
North/weekly
Local
Remote
Daemons
BridgeDaemon-0 Port : 2000
BridgeDaemon-1 Port : 2001
North/daily
North/weekly
South/daily
South/weekly
Bridges
Bridge-0
Bridge-1
routes/north/daily/# IN
IN routes/north/weekly/#
routes/south/daily/# IN
IN routes/south/weekly/#
Daemons
Daemon-0 Port : 1900
Daemon-1 Port : 1901
Daemon-2 Port : 1902
Daemon-3 Port : 1903
MQTT Clients
Truck01 Truck02 Truck03 Truck04 Truck05 Truck06 Truck07 Truck08 Truck09
Truck11 Truck12 Truck13 Truck14 Truck15 Truck16 Truck17 Truck18 Truck19
Truck21 Truck22 Truck23 Truck24 Truck25 Truck26 Truck27 Truck28 Truck29
Truck31 Truck32 Truck33 Truck34 Truck35 Truck36 Truck37 Truck38 Truck39
routes/north/daily/Trucknn
routes/north/weekly/Trucknn
routes/south/daily/Trucknn
routes/south/weekly/Trucknn

Figure 3-25 Correlation of daemons with topic strings in a topology connecting to WebSphere MQ queue managers

Table 3-5 on page 115 shows the daily and weekly bridge daemons and their port numbers.

Table 3-5 Daemon configuration information for connecting to WebSphere MQ queue managers

Topic String	**Daemon Configuration file**	**Port number**
Company/daily/#	Bridge-Daemon-2.cfg	3000
Company/weekly/#}{color	Bridge-Daemon-3.cfg}{color	3001

To start BridgeDaemon-2 on port 3000, including the bridges to BridgeDaemon-0, BridgeDaemon-1 and the WebSphere MQ queue manager, prepare the configuration file as shown in Example 3-17.

Here is a breakdown of what the code in Example 3-17 on page 116 accomplishes:

1. First bridge configuration:
   a. With the connection name as NorthDaily, the bridge connects to BridgeDaemon-0 as denoted by the address parameter.
   b. The bridge considers BridgeDaemon-2 as local. Note the blue boundary in the figure marking it as local.
   c. The bridge considers BridgeDaemon-0 on port number 2000 as remote. Note the red boundary in the figure marking it as remote.
   d. The publications of BridgeDaemon-0 are considered inputs to the bridge.
   e. For incoming publications, the bridge
      i. Checks if the topics of publications match North/daily/#.
      ii. Delivers the publications to Company/daily/#.
   f. With the connection name as SouthDaily, the bridge connects to BridgeDaemon-1 as denoted by the address parameter.
   g. The bridge considers BridgeDaemon-2 as local. Note the blue boundary in the figure marking it as local.
   h. The bridge considers BridgeDaemon-1 on port 2001 as remote. Note the red boundary in the figure marking it as remote.
2. Second bridge configuration:
   a. The publications of BridgeDaemon-1 are considered inputs to the bridge.
   b. For incoming publications, the bridge
      i. Checks if the topics of publications match South/daily/#.
      ii. Delivers the publications to Company/ daily/#.
   c. With the connection name as Daily, the bridge connects to the queue manager as denoted by the address parameter.
3. Third bridge configuration:
   a. The bridge considers BridgeDaemon-2 as local.
   b. The bridge considers the queue manager on port 1883 as remote.
   c. The publications of BridgeDaemon-2 are considered outputs to the bridge.
   d. For incoming subscriptions, the bridge
      i. Checks if the topics of subscriptions match Company/daily/#.
      ii. Delivers the subscriptions to Company/daily/#.

Example 3-17 Daemon configuration file for BridgeDaemon-2 on port 3000

```
# Configuration file for BridgeDaemon-2
#
# Collect North and South daily messages into Daily topic
#
# Topics (From Remote)
#       North/daily
#       South/daily
#
# Topics (To Local)
#       Company/daily
#
# Global parameters
#
port 3000
#
# Bridge parameters
#
connection NorthDaily
clientid BridgeDaemon21
address 9.42.170.179:2000
topic daily/# in Company/ North/
#
# Bridge parameters
#
connection SouthDaily
clientid BridgeDaemon22
address 9.42.170.179:2001
topic daily/# in Company/ South/
#
# Bridge parameters
#
connection Daily
clientid BridgeDaemon23
address 9.42.170.179:1883
topic Company/daily/# out
```

Sample implementations

The previous subsection described three possible messaging topologies using daemons and bridges, and included numerous sample configuration files. The next step is to use the most complex topology, the one extending all the way to a WebSphere MQ queue manager, to show how everything works together.

There are three parts to this process:

1. Starting the daemons
2. Publishing and subscribing to test messages
3. Creating the necessary WebSphere MQ objects

Starting the daemons

To start the daemons, run the **amqtdd** command for each daemon in the chain. The configuration files for the daemons, for which some examples were provided in the previous subsection, must be in place before the command to start the daemons is issued.

For more information about starting the daemons, refer to 3.4.2, “WebSphere MQ Telemetry daemon for devices bridges” on page 107.

The output of the daemon startup process, including the process IDs of the running daemons, can be redirected to the `/tmp/daemons/daemonname.out` file, as shown in Example 3-18.

Example 3-18 The daemonname.out file

```
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Daemon-0.cfg >
/tmp/daemons/Daemon-0.out &
[1] 20335
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Daemon-1.cfg >
/tmp/daemons/Daemon-1.out &
[2] 20336
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Daemon-2.cfg >
/tmp/daemons/Daemon-2.out &
[3] 20355
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Daemon-3.cfg >
/tmp/daemons/Daemon-3.out &
[4] 20357
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Bridge-Daemon-0.cfg >
/tmp/daemons/Bridge-Daemon-0.out &
[5] 20359
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Bridge-Daemon-1.cfg >
/tmp/daemons/Bridge-Daemon-1.out &
[6] 20380
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Bridge-Daemon-2.cfg >
/tmp/daemons/Bridge-Daemon-2.out &
[7] 20382
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ./amqtdd Bridge-Daemon-3.cfg >
/tmp/daemons/Bridge-Daemon-3.out &
[8] 20383
mqtt1:/opt/mqm/mqxr/SDK/advanced/DeviceDaemon/linux_ia64 # ps -ef | grep amqtdd
root     20335  2691  0 09:41 pts/0    00:00:00 ./amqtdd Daemon-0.cfg
root     20336  2691  0 09:41 pts/0    00:00:00 ./amqtdd Daemon-1.cfg
root     20355  2691  0 09:41 pts/0    00:00:00 ./amqtdd Daemon-2.cfg
root     20357  2691  0 09:41 pts/0    00:00:00 ./amqtdd Daemon-3.cfg
root     20359  2691  0 09:41 pts/0    00:00:00 ./amqtdd Bridge-Daemon-0.cfg
root     20380  2691  0 09:41 pts/0    00:00:00 ./amqtdd Bridge-Daemon-1.cfg
root     20382  2691  0 09:41 pts/0    00:00:00 ./amqtdd Bridge-Daemon-2.cfg
root     20383  2691  0 09:42 pts/0    00:00:00 ./amqtdd Bridge-Daemon-3.cfg
root     20390  2691  0 09:42 pts/0    00:00:00 grep amqtdd
```

Example 3-19 shows the `Daemon-0.out` file with the details of the running daemon. The final line shows that a client with the ID BridgeDaemon01 is trying to connect to the daemon.

Example 3-19 Output of the running daemon Daemon-0

```
mqtt1:/tmp/daemons # more Daemon-0.out
20120322 094106.431 CWNAN9999I MQ Telemetry Daemon for Devices
20120322 094106.431 CWNAN9997I Licensed Materials - Property of IBM
20120322 094106.431 CWNAN9996I Copyright IBM Corp. 2007, 2011 All Rights Reserved
20120322 094106.431 CWNAN9995I US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
20120322 094106.431 CWNAN0049I Configuration file name is Daemon-0.cfg
20120322 094106.432 CWNAN0053I Version 1.2.0.5, Oct 10 2011 22:48:55
```

```
20120322 094106.432 CWNAN0054I Features included: bridge MQTTMP
20120322 094106.432 CWNAN0014I MQTT protocol starting, listening on port 1900
20120322 094145.893 CWNAN0033I Connection attempt to listener 1900 received from client
BridgeDaemon01 on address 9.42.170.179:45070
```

Example 3-20 shows the configuration file for BridgeDaemon-0 on port 2000. It shows that the clientID is set to BridgeDaemon02.

Example 3-20 Configuration file for BridgeDaemon-0 on port 2000

```
# Configuration file for BridgeDaemon-0
#
# Collect North daily and weekly truck messages into North daily and weekly topics
#
# Topics (From Remote)
#       routes/north/daily/Truck0n
#       routes/north/weekly/Truck1n
#
# Topics (To Local)
#       North/daily
#       North/weekly
#
# Global parameters
#
port 2000
#
# Bridge parameters
#
connection NorthDailyTrucks
clientid BridgeDaemon01
address 9.42.170.179:1900
topic daily/# in North/ routes/north/
#
# Bridge parameters
#
connection NorthWeeklyTrucks
clientid BridgeDaemon02
address 9.42.170.179:1901
topic weekly/# in North/ routes/north
```

Example 3-21 shows the output of the running daemon Daemon-1. The final line shows that a client with the ID BridgeDaemon02 is trying to connect to the daemon.

Example 3-21 Output of the running daemon Daemon-1

```
mqtt1:/tmp/daemons # more Daemon-1.out
20120322 094111.397 CWNAN9999I MQ Telemetry Daemon for Devices
20120322 094111.397 CWNAN9997I Licensed Materials - Property of IBM
20120322 094111.397 CWNAN9996I Copyright IBM Corp. 2007, 2011 All Rights Reserved
20120322 094111.397 CWNAN9995I US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
20120322 094111.397 CWNAN0049I Configuration file name is Daemon-1.cfg
20120322 094111.397 CWNAN0053I Version 1.2.0.5, Oct 10 2011 22:48:55
20120322 094111.397 CWNAN0054I Features included: bridge MQTTMP
20120322 094111.397 CWNAN0014I MQTT protocol starting, listening on port 1901
```

```
20120322 094145.893 CWNAN0033I Connection attempt to listener 1901 received from client
BridgeDaemon02 on address 9.42.170.179:59802
```

Example 3-22 shows the output of the running daemon that is connected to the WebSphere MQ queue manager. When the bridge tries to establish a connection, it checks whether the connection at the other end is also a WebSphere MQ Telemetry daemon for devices. If not, the socket error appears. This error does not cause a problem, but it does add the extra line in the log. This extra line can be avoided by using the `tryprivate` false parameter in the bridge configuration. A list of bridge configuration parameters is available at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt60918_.htm

Important: When connecting to a WebSphere MQ queue manager, the client ID must conform to WebSphere MQ naming rules. The information center has more details available at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/index.jsp?topic=/com.ibm.mq.doc/fa10390_.htm

Example 3-22 Output of the running daemon connected to the WebSphere MQ queue manager

```
mqtt1:/tmp/daemons # more Bridge-Daemon-2.out
20120322 094154.901 CWNAN9999I MQ Telemetry Daemon for Devices
20120322 094154.901 CWNAN9997I Licensed Materials - Property of IBM
20120322 094154.901 CWNAN9996I Copyright IBM Corp. 2007, 2011 All Rights Reserved
20120322 094154.901 CWNAN9995I US Government Users Restricted Rights - Use, duplication or
disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
20120322 094154.901 CWNAN0049I Configuration file name is Bridge-Daemon-2.cfg
20120322 094154.901 CWNAN0053I Version 1.2.0.5, Oct 10 2011 22:48:55
20120322 094154.901 CWNAN0054I Features included: bridge MQTTMP
20120322 094154.902 CWNAN0014I MQTT protocol starting, listening on port 3000
20120322 094155.901 CWNAN0124I Starting bridge connection NorthDaily
20120322 094155.901 CWNAN0124I Starting bridge connection SouthDaily
20120322 094155.901 CWNAN0124I Starting bridge connection Daily
20120322 094156.902 CWNAN0133I Bridge connection NorthDaily to 9.42.170.179:2000 now established
20120322 094156.902 CWNAN0133I Bridge connection SouthDaily to 9.42.170.179:2001 now established
20120322 094156.902 CWNAN0018W Socket error for client identifier BridgeDaemon23, socket 6, peer
address 9.42.170.179:1883; ending connection
20120322 094156.902 CWNAN0099I Trying bridge connection Daily address 9.42.170.179:1883 again,
without private protocol
20120322 094156.961 CWNAN0133I Bridge connection Daily to 9.42.170.179:1883 now established
```

Publishing and subscribing to test messages

With the network of daemons running, the act of subscribing to topics and publishing messages to those topics can be tested. The procedures shown here are effectively manual simulations of the activities that occur automatically in a fully implemented publish/subscribe messaging system.

Start by publishing a message or messages to which you will later subscribe. To simulate trucks in all daemon categories, messages are to be published concerning Truck05, Truck12, Truck29, and Truck38.

To publish a message, you can use any of the following methods:

- Mosquitto (**`mosquitto_pub`** command)

  ```
  mosquittopub -h ipAddressDaeomon0 -p 1900 -t routes/north/daily/Truck05 -i
  Truck05 -l
  ```

  The command establishes the connection to the daemon, after which it waits for you to input the text of the message. In this scenario, the message must follow the format `ddmmyyyyhhmmss,latitude,longitude,tripStatus`.

  The following syntax is an example:

  ```
  †21032012173800,35.5999,-78.40001,T
  ```

- Really Small Message Broker (**`stdinpub`** command)

  ```
  LDLIBRARYPATH=. ./stdinpub routes/north/weekly/Truck12 --host ipAddressDaeomon1
  --port 1901 --clientid Truck12
  ```

  The command establishes the connection to the daemon, after which it waits for you to input the text of the message.

- WMQTT Utility (GUI interface)

  On the WMQTT tab, enter the topic string for publication (in this case routes/south/daily) and the TCP/IP address value to connect to the daemon on port 1902. On the Options tab, provide the Client Identifier as Truck29. Note that the Client Identifier prefixes parameter in the configuration file requires the Client Identifier to begin with Truck2.

Next, set up subscriptions to BridgeDaemon-0, BridgeDaemon-1, BridgeDaemon-2, and BridgeDaemon-3. These subscriptions must end with a `\#` so that, regardless of the truck ID, the messages are received. So, when connecting to BridgeDaemon-1, set up the subscription to say `South/daily/#`.

Next, create a subscription to a topic into which all the messages you just published will flow, in this case `South/daily/#`.

To create a subscription any of the following methods can be used:

- Mosquito

  ```
  mosquittosub -h ipAddressBridgeDaeomon0 -p 2000 -t South/daily/# -v
  ```

- Really Small Message Broker

  ```
  LDLIBRARYPATH=. ./stdoutsub South/daily --host ipAddressBridgeDaeomon1 --port
  2001
  ```

- WMQTT Utility

Tip: The command line options for testing subscriptions are typically the better ones to choose because the WMQTT Utility shows only the most recently received message, not a list of all recent messages.

After the subscription is made, the message arrives.

Subscription results from Bridge-Daemon-0:

```
mosquittosub -h 9.42.170.179 -p 2000 -t North/daily/# -t North/weekly/# -v
North/daily/Truck05 21032012173800,35.5999,-78.40001,T
North/weekly/Truck12 21032012173800,35.5999,-78.40001,T
```

Subscription results from Bridge-Daemon-1:

```
mosquittosub -h 9.42.170.179 -p 2001 -t South/daily/# -t South/weekly/# -v
South/daily/Truck29 21032012173800,35.5999,-78.40001,T
South/weekly/Truck38 21032012173800,35.5999,-78.40001,T
```

Subscription results from Bridge-Daemon-2:

```
mosquittosub -h 9.42.170.179 -p 3000 -t Company/daily/# -v
Company/daily/Truck05 21032012173800,35.5999,-78.40001,T
Company/daily/Truck29 21032012173800,35.5999,-78.40001,T
```

Subscription results from Bridge-Daemon-3:

```
mosquittosub -h 9.42.170.179 -p 3001 -t Company/weekly/# -v
Company/weekly/Truck12 21032012173800,35.5999,-78.40001,T
Company/weekly/Truck38 21032012173800,35.5999,-78.40001,T
```

Creating the necessary WebSphere MQ objects

For the final two bridges (Bridge-Daemon-2.cfg and Bridge-Daemon-3.cfg) to connect to the WebSphere MQ queue manager, objects must be created on the queue manager.

Start by defining two subscriptions, SUBS.FLEET.DAILY (see Example 3-23) and SUBS.FLEET.WEEKLY (see Example 3-24), with corresponding topic strings Company/daily and Company/weekly. These subscriptions must have the local queue as their destination.

Example 3-23 Subscription definition for daily trucks

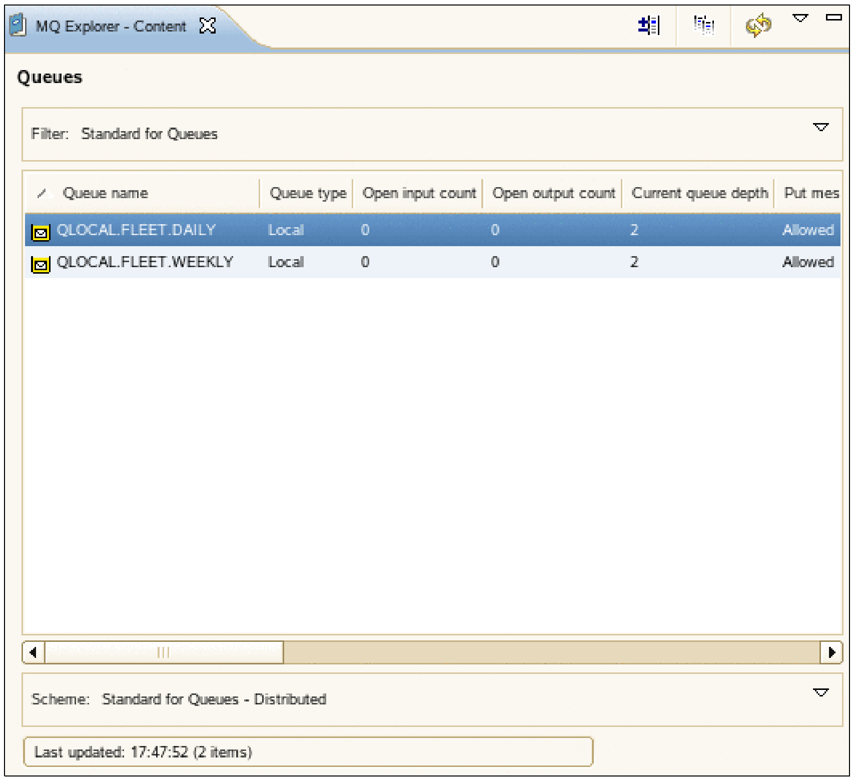
```
DEFINE SUB(SUBS.FLEET.DAILY) TOPICSTR(Company/daily/#) +
       DEST(QLOCAL.FLEET.DAILY) +
       DESTQMGR(MQTT_QMGR) +
       PUBACCT(0130000000000000000000000000000000000000000000000000000000000006) +
       DESTCORL(414D51204D5154545F514D4752202020D8675E4F0E0B1220) +
       DESTCLAS(PROVIDED) EXPIRY(UNLIMITED) PSPROP(MSGPROP) +
       PUBPRTY(ASPUB) REQONLY(NO) SUBSCOPE(ALL) SUBLEVEL(1) +
       VARUSER(ANY) WSCHEMA(TOPIC) SUBUSER('root')
```

Example 3-24 Subscription information for weekly trucks

```
DEFINE SUB(SUBS.FLEET.WEEKLY) TOPICSTR(Company/weekly/#) +
       DEST(QLOCAL.FLEET.WEEKLY) +
       DESTQMGR(MQTT_QMGR) +
       PUBACCT(0130000000000000000000000000000000000000000000000000000000000006) +
       DESTCORL(414D51204D5154545F514D4752202020D8675E4F7A0B1220) +
       DESTCLAS(PROVIDED) EXPIRY(UNLIMITED) PSPROP(MSGPROP) +
       PUBPRTY(ASPUB) REQONLY(NO) SUBSCOPE(ALL) SUBLEVEL(1) +
       VARUSER(ANY) WSCHEMA(TOPIC) SUBUSER('root')
```

Next, define two local queues, QLOCAL.FLEET.DAILY and QLOCAL.FLEET.WEEKLY. These queues are set as destinations for the subscriptions created in the previous two examples.

The local queues are populated as shown by their queue depths in the MQ Explorer view of the queues, illustrated in Figure 3-26 on page 122.

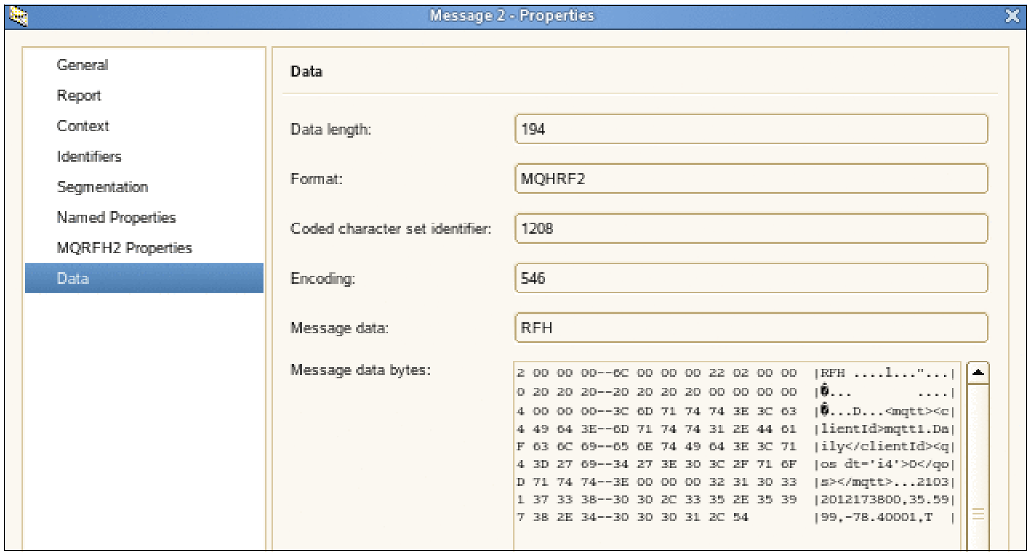

MQ Explorer - Content
Queues
Filter: Standard for Queues

Queue name	Queue type	Open input count	Open output count	Current queue depth	Put mes
QLOCAL.FLEET.DAILY	Local	0	0	2	Allowed
QLOCAL.FLEET.WEEKLY	Local	0	0	2	Allowed

Scheme: Standard for Queues - Distributed
Last updated: 17:47:52 (2 items)

Figure 3-26 Queue depth of the local queues receiving the publications

The data of one of the queues is shown in the MQ Explorer view of the queues, and depicted in Figure 3-27.



Data length:	194
Format:	MQHRF2
Coded character set identifier:	1208
Encoding:	546
Message data:	RFH

Message data bytes:

```
2 00 00 00--6C 00 00 00 22 02 00 00   |RFH ....l..."...|
0 20 20 20--20 20 20 20 00 00 00 00   |Ô...        ....|
4 00 00 00--3C 6D 71 74 74 3E 3C 63   |Ô...D...<mqtt><c|
4 49 64 3E--6D 71 74 74 31 2E 44 61   |lientId>mqtt1.Da|
F 63 6C 69--65 6E 74 49 64 3E 3C 71   |ily</clientId><q|
4 3D 27 69--34 27 3E 30 3C 2F 71 6F   |os dt='i4'>0</qo|
D 71 74 74--3E 00 00 00 32 31 30 33   |s></mqtt>...2103|
1 37 33 38--30 30 2C 33 35 2E 35 39   |2012173800,35.59|
7 38 2E 34--30 30 30 31 2C 54         |99,-78.40001,T  |
```

Figure 3-27 Message in the local queue

3.5 Troubleshooting WebSphere MQ Telemetry

In this section we provide guidance on locating WebSphere MQ Telemetry logs and error files, and tracing.

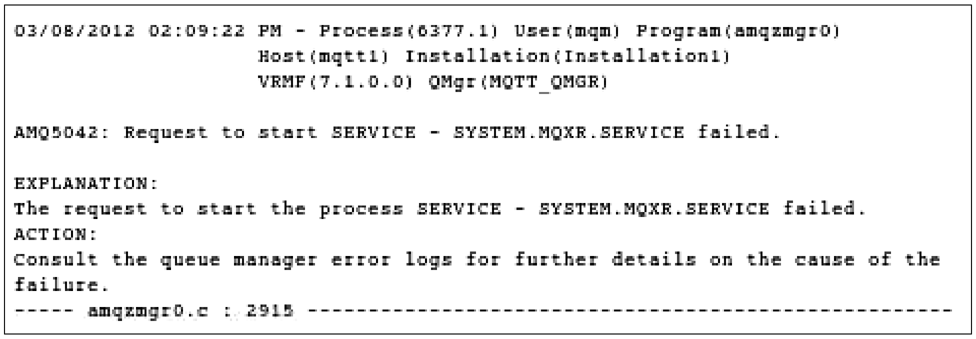
3.5.1 Location of logs and error files

The WebSphere MQ Telemetry service writes informational and error messages to the WebSphere MQ queue manager error log, and creates IBM FFST error files (ending in. FDC) in the WebSphere MQ error directory.

The logs and FDC error files are located in different areas based on the operating system. The following information provides the location of the files based on a specific system:

- On a Windows operating system
  - *WMQ data directory*`\Qmgrs\`*qMgrName*`\errors\AMQERR01.LOG`
  - *WMQ data directory*`\errors\AMQnnn.n.FDC`
- On Linux:
  - `/var/mqm/qmgrs/`*QMGRNAME*`/errors/AMQERR01.LOG`
  - `/var/mqm/qmgrs/`*QMGRNAME*`/errors/AMQnnn.n.FDC`

Figure 3-28 shows a typical WebSphere MQ Telemetry error log entry.



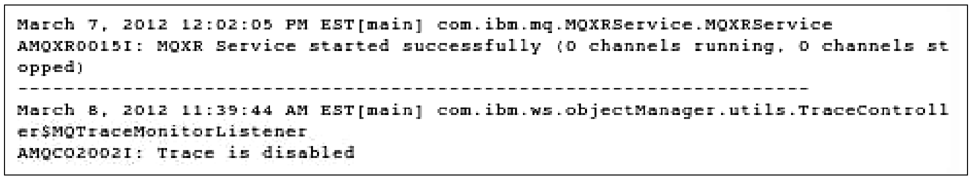
```
03/08/2012 02:09:22 PM - Process(6377.1) User(mqm) Program(amqzmgr0)
                    Host(mqtt1) Installation(Installation1)
                    VRMF(7.1.0.0) QMgr(MQTT_QMGR)

AMQ5042: Request to start SERVICE - SYSTEM.MQXR.SERVICE failed.

EXPLANATION:
The request to start the process SERVICE - SYSTEM.MQXR.SERVICE failed.
ACTION:
Consult the queue manager error logs for further details on the cause of the
failure.
----- amqzmgr0.c : 2915 ---------------------------------------------------
```

Figure 3-28 WebSphere MQ Telemetry error log

The WebSphere MQ telemetry service also writes to a log that displays the properties that were invoked when the telemetry service was started, plus any errors it might have found while acting as a proxy for WebSphere MQ. The following list represents the log paths based on the operating system:

- On a Windows operating system: *`WMQ data directory`*`\Qmgrs\`*`qMgrName`*`\errors\mqxr.log`
- On Lynx: `/var/mqm/qmgrs/`*`QMGRNAME`*`/errors/mqxr.log`

Figure 3-29 shows some typical error messages written to the `mqxr` log file.



```
March 7, 2012 12:02:05 PM EST[main] com.ibm.mq.MQXRService.MQXRService
AMQXR0015I: MQXR Service started successfully (0 channels running, 0 channels st
opped)
-------------------------------------------------------------------
March 8, 2012 11:39:44 AM EST[main] com.ibm.ws.objectManager.utils.TraceControll
er$MQTraceMonitorListener
AMQCO2002I: Trace is disabled
```

Figure 3-29 WebSphere MQ Telemetry mqxr log messages

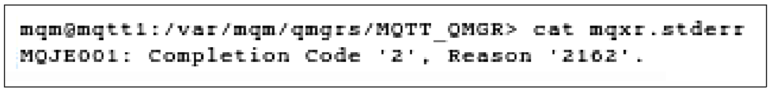
When WebSphere MQ Explorer is used to create the WebSphere MQ Telemetry sample configuration, it starts the telemetry service using the **`runMQXRService`** command from the WebSphere MQ Telemetry install directory `\bin` and writes messages to the following log files:

- On a Windows operating system:
  - *WMQ data directory*`\Qmgrs\`*qMgrName*`\mqxr.stdout`

  – *WMQ data directory*`\Qmgrs\`*qMgrName*`\mqxr.stderr`

- On Linux:
  – `/var/mqm/qmgrs/`*QMGRNAME*`/mqxr.stdout`
  – `/var/mqm/qmgrs/`*QMGRNAME*`/mqxr.stderr`

Figure 3-30 provides examples of error messages written to the `mqxr.stderr` file when the telemetry service is started.



```
mqm@mqtt1:/var/mqm/qmgrs/MQTT_QMGR> cat mqxr.stderr
MQJE001: Completion Code '2', Reason '2162'.
```

Figure 3-30 Errors written to the mqxr.stderr file

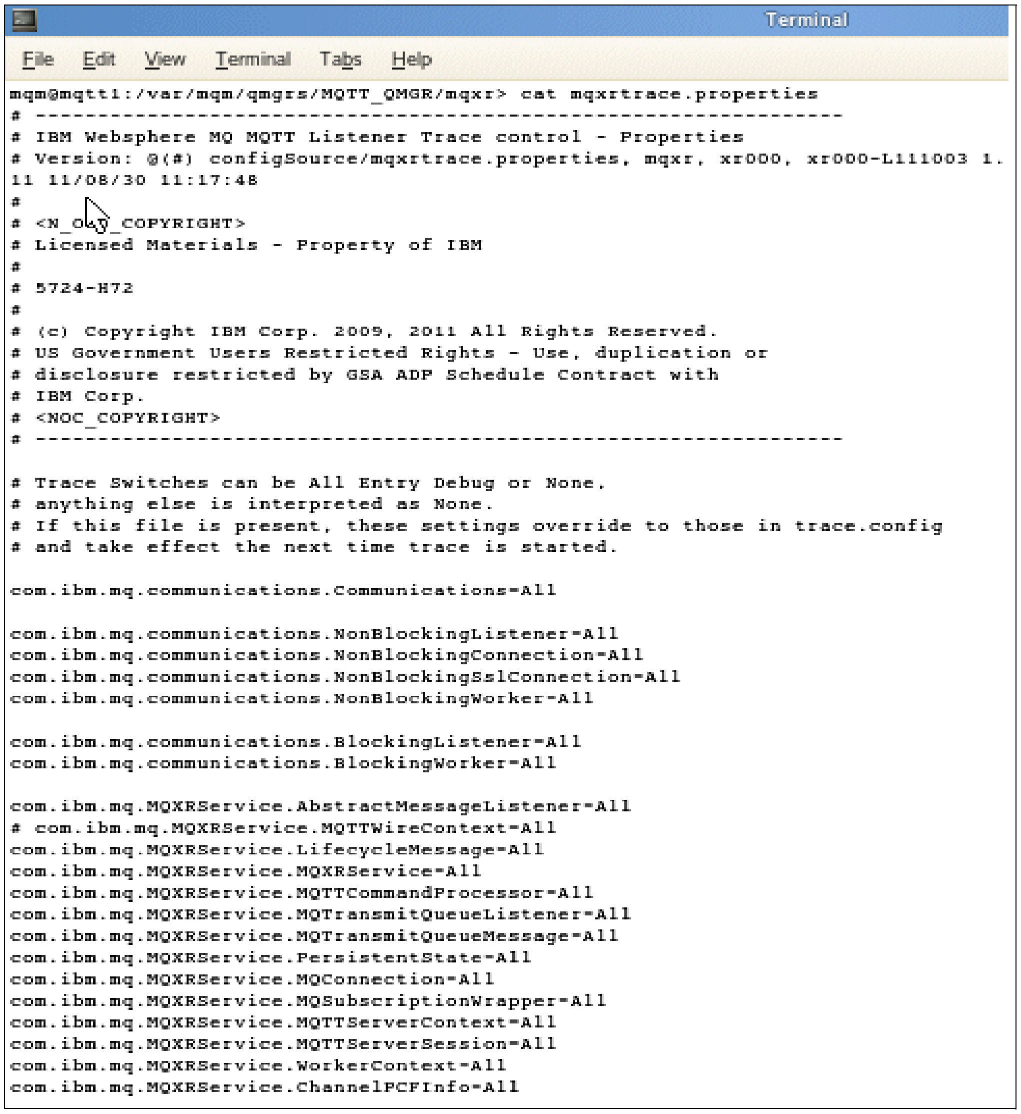
3.5.2 Tracing the telemetry service

WebSphere MQ Telemetry traces can be captured in multiple ways. Traces can be useful for troubleshooting if you are experiencing failures or unexpected results.

To start and stop a full WebSphere MQ trace, including telemetry, use the standard WebSphere MQ trace commands **`strmqtrc`** and **`endmqtrc`**. Or you can use the `controlMQXRChannel` script to initiate WebSphere MQ telemetry tracing only. Use of this script is detailed later in this section.

When using the **`strmqtrc`** command, there is a brief delay (up to 2-3 seconds) before telemetry service tracing begins.

To control the amount of detail and size of the trace, you can set trace options in the `mqxrtrace.properties` and `trace.config` files located in the `/qmgrs/QMGRNAME/mqxr` subdirectory of the WebSphere MQ directory. Illustrated in Figure 3-31 on page 125 are examples of the properties users can configure in the `mqxrtrace.properties` file, which can apply to traces started either by command or by script.



```
mqm@mqtt1:/var/mqm/qmgrs/MQTT_QMGR/mqxr> cat mqxrtrace.properties
# ----------------------------------------------------------------
# IBM Websphere MQ MQTT Listener Trace control - Properties
# Version: @(#) configSource/mqxrtrace.properties, mqxr, xr000, xr000-L111003 1.
11 11/08/30 11:17:48
#
# <N_OCO_COPYRIGHT>
# Licensed Materials - Property of IBM
#
# 5724-H72
#
# (c) Copyright IBM Corp. 2009, 2011 All Rights Reserved.
# US Government Users Restricted Rights - Use, duplication or
# disclosure restricted by GSA ADP Schedule Contract with
# IBM Corp.
# <NOC_COPYRIGHT>
# ----------------------------------------------------------------

# Trace Switches can be All Entry Debug or None,
# anything else is interpreted as None.
# If this file is present, these settings override to those in trace.config
# and take effect the next time trace is started.

com.ibm.mq.communications.Communications=All

com.ibm.mq.communications.NonBlockingListener=All
com.ibm.mq.communications.NonBlockingConnection=All
com.ibm.mq.communications.NonBlockingSslConnection=All
com.ibm.mq.communications.NonBlockingWorker=All

com.ibm.mq.communications.BlockingListener=All
com.ibm.mq.communications.BlockingWorker=All

com.ibm.mq.MQXRService.AbstractMessageListener=All
# com.ibm.mq.MQXRService.MQTTWireContext=All
com.ibm.mq.MQXRService.LifecycleMessage=All
com.ibm.mq.MQXRService.MQXRService=All
com.ibm.mq.MQXRService.MQTTCommandProcessor=All
com.ibm.mq.MQXRService.MQTransmitQueueListener=All
com.ibm.mq.MQXRService.MQTransmitQueueMessage=All
com.ibm.mq.MQXRService.PersistentState=All
com.ibm.mq.MQXRService.MQConnection=All
com.ibm.mq.MQXRService.MQSubscriptionWrapper=All
com.ibm.mq.MQXRService.MQTTServerContext=All
com.ibm.mq.MQXRService.MQTTServerSession=All
com.ibm.mq.MQXRService.WorkerContext=All
com.ibm.mq.MQXRService.ChannelPCFInfo=All
```

Figure 3-31 The mqxrtrace.properties file

To start MQ Telemetry tracing without using the standard commands, run the script `controlMQXRChannel.sh` to start a SYSTEM.MQXR.SERVICE trace. The generic syntax for doing this is shown in Figure 3-32.

```
>>-+-./controlMQXRChannel.sh-+-- -qmgr=--qMgrName-- -mode=--+-starttrace-+-->
   '-controlMQXRChannel.bat--'                               '-stoptrace--'

>--+-------------------------------+--------------------------><
   '- -clientid=--ClientIdentifier-'
```

Figure 3-32 Generic syntax for initiating tracing using controlMQXRChannel.sh

When using `controlMQXRChannel.sh`, two *mandatory* tracing parameters must be specified:

- `qmgr=qmgrName`: Sets qmgrName to the queue manager name.
- `mode=starttrace\|stoptrace`: Sets starttrace to begin tracing or stoptrace to end tracing.

The following parameter is an *optional* tracing parameter:

- `clientid=ClientIdentifier`: Sets ClientIdentifier to the ClientID of a single client. This reduces the trace to just the single client, so if this parameter is set, the trace command must be run multiple times to trace multiple clients.

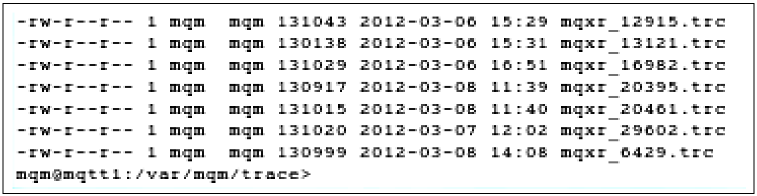
The specific commands that follow are for starting and stopping a trace using `controlMQXRChannel.sh`:

```
/opt/mqm/mqxr/bin/controlMQXRChannel.sh -qmgr=QM1 -mode=starttrace
/opt/mqm/mqxr/bin/controlMQXRChannel.sh -qmgr=QM1 -mode=stoptrace
```

To view the trace output, go to the following directory:

- On a Windows operating system: `\trace` (in the WebSphere MQ data directory)
- On Linux: `/var/mqm/trace`

You will find trace files named using the format `mqxr####.trc`, where `####` is the process ID, as shown in Figure 3-33.



```
-rw-r--r-- 1 mqm  mqm 131043 2012-03-06 15:29 mqxr_12915.trc
-rw-r--r-- 1 mqm  mqm 130138 2012-03-06 15:31 mqxr_13121.trc
-rw-r--r-- 1 mqm  mqm 131029 2012-03-06 16:51 mqxr_16982.trc
-rw-r--r-- 1 mqm  mqm 130917 2012-03-08 11:39 mqxr_20395.trc
-rw-r--r-- 1 mqm  mqm 131015 2012-03-08 11:40 mqxr_20461.trc
-rw-r--r-- 1 mqm  mqm 131020 2012-03-07 12:02 mqxr_29602.trc
-rw-r--r-- 1 mqm  mqm 130999 2012-03-08 14:08 mqxr_6429.trc
mqm@mqtt1:/var/mqm/trace>
```

Figure 3-33 Example of WebSphere MQ Trace files

For more information about troubleshooting WebSphere MQ Telemetry, visit the WebSphere MQ Information Center at:

http://publib.boulder.ibm.com/infocenter/wmqv7/v7r1/topic/com.ibm.mq.doc/tt80000_.htm

4

Integrating MQTT

In this chapter we describe the integration of IBM products with applications and devices that communicate using the MQTT protocol. The examples are based on the transportation logistics scenario first outlined in 2.2, “Scenario used in this book” on page 24.

This chapter contains the following sections:

- Integrate MQTT with IBM WebSphere Message Broker
- Integrate MQTT with IBM WebSphere Application Server
- Integrate MQTT with IBM WebSphere Operational Decision Management
- Integrate MQTT with IBM Intelligent Operations Center
- Integrate MQTT with IBM Lotus Expeditor integrator

4.1 Integrate MQTT with IBM WebSphere Message Broker

In this section we address the integration of MQTT-based applications and devices with IBM WebSphere Message Broker, an enterprise service bus (ESB) that uses IBM WebSphere MQ as the backbone for message routing.

4.1.1 IBM WebSphere Message Broker versions

The specifics of integration differ based on which version of WebSphere Message Broker is being used.

WebSphere Message Broker V6.1

WebSphere Message Broker V6.1 was the first version to allow connections with telemetry devices, which it accomplished using the supervisory control and data acquisition (SCADA) nodes:

- The SCADAInput node was used to receive MQTT messages from clients. After an MQTT client published a message, the SCADAInput node initiated the flow in which the message was transformed and consumed by other clients or applications.
- The SCADAOutput node was used to deliver MQTT messages to clients. It ended the message flow, and the transformed message was delivered to the consuming clients or applications.

For more information about designing telemetry applications with WebSphere Message Broker V6.1, including the use of the SCADA nodes, see:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r1m0/topic/com.ibm.etools.mft.doc/ac23500_.htm

WebSphere Message Broker V7.0

Beginning with WebSphere Message Broker V7.0, the SCADA nodes are no longer available. Instead, support for the MQTT protocol is included in WebSphere MQ V7.0 with the introduction of *telemetry channels*, in which messages from MQTT clients were either made available using JMS topic destinations or routed to standard WebSphere MQ message queues.

Due to this change, starting with WebSphere Message Broker V7.0, communication using MQTT is accomplished using the MQInput, Publication, JMSInput, and JMSOutput nodes of WebSphere Message Broker.

For more information about WebSphere Message Broker V7.0, see:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v7r0m0/index.jsp

WebSphere Message Broker V8.0

WebSphere Message Broker V8.0 continues to process MQTT messages using the MQInput, Publication, JMSInput, and JMSOutput nodes. However, WebSphere MQ V7.1 is not compatible with WebSphere Message Broker V8.0. Thus, to use WebSphere Message Broker V8.0, you must have WebSphere MQ V7.0. For more information about WebSphere Message Broker V8.0, see:

http://publib.boulder.ibm.com/infocenter/wmbhelp/v8r0m0/index.jsp

The following sections include sample applications that show MQTT integration with WebSphere Message Broker.

4.1.2 Example 1: MQInput node

This is a sample WebSphere Message Broker V8.0 application that communicates with the MQTT clients (trucks) in the transportation logistics scenario.

The company wants to track the location of its trucks based on human-readable addresses rather than just the latitude and longitude values reported by the trucks' MQTT-enabled devices. So, this sample application subscribes to the raw location information that is published by each truck, converts those values to human-readable addresses using a *reverse geo-coding* API, and then publishes the formatted address information. Personnel or applications at company headquarters can then subscribe to these messages to obtain the current location of each truck in the new, easy-to-understand format.

This sample illustrates one method of implementing the solution using the MQInput node of WebSphere Message Broker to parse and process the incoming publication and using WebSphere MQ to route the publications to a local queue using a subscription object.

As mentioned previously, starting with WebSphere Message Broker V7.0, support for MQTT is through WebSphere MQ. So in this sample, all publications sent by the MQTT clients (the devices on the trucks) are directed to message queues using a WebSphere MQ subscription object. This subscription object ensures that the message queue receives all messages pertaining to truck locations by using a wildcard character in the topic string to which it subscribes. The message in the message queue is parsed in the message flow using an MQInput. Similarly, when publishing the formatted address from the message flow, this sample uses a publication node.

MQInput node: When using the MQInput node, the original topic string as published by the MQTT client (the device on a truck) is no longer available when the message arrives at the node, because the subscription object routes the published message only to the local queue with the topic string discarded. This concept is an important point for this sample, because the WebSphere MQ subscription object uses a wildcard character in the topic string to qualify all trucks.

This sample solution is ideal for situations where Java Message Service (JMS) is not widely used. A sample solution using JMS is presented in 4.1.3, "Example 2: JMSInput node" on page 133.

The sample presented here subscribes to the `Company/daily/#` topic. It takes the latitude and longitude values in the message and invokes a reverse geo-coding API. It retrieves the response (an XML document), parses it for the formatted address, and publishes it to another topic (truck or position). Another MQTT client then subscribes to this topic to track the vehicle based on the formatted address.

Reverse geo-coding API

To help design the message flow, first consider how the invocation of the GeoNames API works outside of WebSphere Message Broker. The following information can be obtained from the API documentation:

- Invocation API URL

  `http://api.geonames.org/findNearbyPlaceName`

  This URL returns a response in XML.

- Query parameters

`lat`	A latitude value in decimal format (not in degrees, minutes, and seconds) separated by a comma. Positive values indicate northern latitudes and negative values indicate southern latitudes.
`lng`	A longitude value, also in decimal format, separated by a comma. Eastern longitudes are given positive values and western longitudes are given negative ones.
`username`	A user name registered on the GeoNames website.

- Query response

  An XML document containing the place name (not necessarily an established city) that is closest to the provided coordinates, and it is a measurement of the distance between the named place and the precise coordinates that were provided.

To learn more about the GeoNames API, visit the company website at:

`http://www.geonames.org/export/web-services.html`

Invoking the API

Example 4-1 is a sample invocation of the API.

Example 4-1 An API invocation

```
http://api.geonames.org/findNearbyPlaceName?lat=12.966666667&lng=77.566666667&user
name=teamitso
```

The URL asks the web service to locate the populated area closest to the supplied latitude and longitude values (the `lat` and `long` query parameters, respectively). The eastern longitudes and northern latitudes are provided as positive real numbers. The western longitudes and southern longitudes are provided as negative real numbers. The username parameter is required by the GeoNames website so that only registered users can use the API.

API response

Example 4-2 is a sample of the API response in XML.

Example 4-2 API response in XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<geonames>
<geoname>
<toponymName>Binny Mills</toponymName>
<name>Binny Mills</name>
<lat>12.96954</lat>
<lng>77.56695</lng>
<geonameId>6692887</geonameId>
<countryCode>IN</countryCode>
<countryName>India</countryName>
<fcl>P</fcl>
<fcode>PPL</fcode>
<distance>0.32049</distance>
</geoname>
</geonames>
```

The response comes in the form of an XML document that lists a populated location that is closest to the provided latitude and longitude values. Among the items returned are the name of the location (in this case, Binny Mills), its latitude and longitude values, its country code, the name of the country, and the distance (in kilometers) from the named location to the precise spot indicated by the latitude and longitude values that were submitted originally.

Message flow

The previous section described how to invoke the web service and then parse its response. The next step is to do the same thing within a WebSphere Message Broker message flow. It has already been established that publications from the clients (the devices on the trucks) are routed to local message queues using a WebSphere MQ subscription object. So the following steps describe how to implement the solution from the point of the MQInput node onward.

The message flow, starting with the MQInput node, works as follows:

1. Parse the incoming comma-delimited message (from the MQTT-enabled device on a truck) in the MQInput node.
2. Parse the latitude and longitude values in the Compute node.
3. Use the latitude and longitude values to create the HTTP URL with the query parameters in the Compute node.
4. Invoke the API in the HTTPRequest node
5. Parse the API response, also in the HTTPRequest node.
6. Extract the formatted address from the XML document in another Compute node.
7. Route the formatted address information to an MQTT topic using the Publication node.

Sample application

Publications from the trucks (the MQTT clients) are routed to local message queues for reverse geocoding, which provide a formatted address instead of the original latitude and longitude values. The message queues serve as an input to the message flow.

WebSphere MQ

The following steps are required to set up WebSphere MQ objects to run the WebSphere Message Broker message flow:

1. Define a queue manager as `MQTT.QMGR00`. This name must match the queue manager name in the MQInput node of the message flow.
2. Define a local queue as `QLOCAL.FLEET.DAILY`. This name must match the queue name in the MQInput node of the message flow.
3. Define a subscription to the `Company/daily/#` topic pointing to the local queue defined in step 1.

Local queue definition

Define a local queue where the publications from the MQTT clients (the devices on the trucks) are routed using the subscription object. Example 4-3 shows the process of defining a local queue.

Example 4-3 Defining a local queue

```
DEFINE QL(QLOCAL.FLEET.DAILY) +
DESCR('GPS co-ordinates of trucks') +
DISTL(NO) GET(ENABLED) MAXDEPTH(500) MAXMSGL(100) MONQ(QMGR) +
TRIGMPRI(0) TRIGTYPE(FIRST) USAGE(NORMAL)
```

Subscription definition

A defined subscription object that routes the publications from the MQTT clients to the local queue was shown in Example 4-3. Example 4-4 shows the process of defining that subscription object.

Example 4-4 Defining a subscription object

```
TOPICSTR('Company/daily/#') +
DEST(QLOCAL.FLEET.DAILY) +
DESTQMGR(MQTT.QMGR00) +
```

WebSphere Message Broker

The following steps describe the installation of the message flow project:

1. Open the `GeoNamesSample.zip` file supplied in the additional materials of this book.
2. Import the file into the WebSphere Message Broker Toolkit.
3. Ensure that the queue name in MQInput node is set to `QLOCAL.FLEET.DAILY`.
4. Build and create a `geoNamesSample.bar` file. Then, deploy the file to an execution group.

WMQTT Utility setup

The following steps simulate the publisher (the devices on the trucks) and the subscriber (applications or personnel at company headquarters). WMQTT Utility instances are designed as the publisher and subscriber.

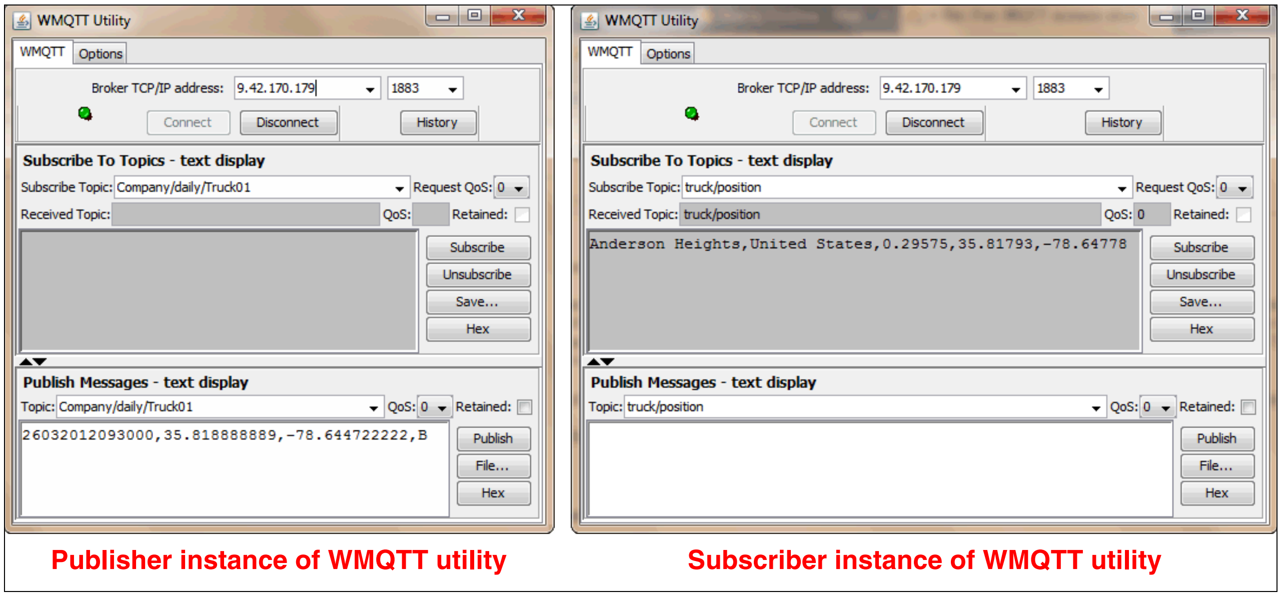
1. Start an instance of the WMQTT Utility. This instance will be the publisher.
2. Enter the IP address for the queue manager (`MQTT.QMGR00`).
3. Enter the client ID, such as `Truck01`.
4. Click **Connect**. The publisher is now connected to the broker.
5. Enter the topic for publication, such as `Company/daily/Truck01.` Do not publish yet, because the subscriber instance is not yet set up.
6. Start another instance of the WMQTT Utility. This instance will be the subscriber.
7. Enter the IP address for the queue manager (`MQTT.QMGR00`).
8. Enter the client ID as `TruckPosition`.
9. Click **Connect**. The subscriber is now connected to the broker.
10. Enter the topic to which you are subscribing, as in `truck/position`.
11. Click **Subscribe**. You are now subscribing to the truck/position topic.

Running the sample

To run the application to view the formatted address of the trucks, use the following steps.

1. Enter a message similar to `26032012093000,35.818888889,-78.644722222,B` in the Truck01 instance of the WMQTT utility.
2. Click **Publish** to send the message.
3. Check the TruckPosition instance of WMQTT Utility. The formatted address is visible at this point.

Figure 4-1 on page 133 illustrates the outcome of this process.



WMQTT Utility (publisher)

Broker TCP/IP address: 9.42.170.179 1883

Subscribe To Topics - text display

Subscribe Topic: Company/daily/Truck01 Request QoS: 0

Received Topic: QoS: Retained:

Publish Messages - text display

Topic: Company/daily/Truck01 QoS: 0 Retained:

```
26032012093000,35.818888889,-78.644722222,B
```

WMQTT Utility (subscriber)

Broker TCP/IP address: 9.42.170.179 1883

Subscribe To Topics - text display

Subscribe Topic: truck/position Request QoS: 0

Received Topic: truck/position QoS: 0 Retained:

```
Anderson Heights,United States,0.29575,35.81793,-78.64778
```

Publish Messages - text display

Topic: truck/position QoS: 0 Retained:

Figure 4-1 The publisher (a truck) sends its GPS coordinates; the subscriber (at company headquarters) sees the corresponding formatted address

4.1.3 Example 2: JMSInput node

This example is similar to the previous example, except that it uses the JMSInput node. With the JMSInput node, the topics of publications are received as JMS topic destinations.

The benefit of this approach is the ability to identify the exact topic string from where the message originated. The topic string has the Truck ID in it, so using the JMSInput node allows the company to identify the specific truck that sent a message. This is a clear advantage over the MQInput node sample, where it was not possible to determine the exact topic string.

In this sample, the subscription is to the topic `Company/daily/#`. The sample takes the latitude and longitude values in the message and invokes a reverse geo-coding API to convert them into a more easily understood street address. The response (an XML document) is retrieved, parsed for the formatted address and then published to another topic, truck/position, to which other clients or applications can subscribe. After they have subscribed, they can track the vehicle based on the formatted address.

Reverse geo-coding API

To help design the message flow, first consider how the invocation of the Geonames API works outside of WebSphere Message Broker. The following information can be learned from the API documentation:

- Invocation API at:

  `http://api.geonames.org/findNearbyPlaceName`

  This address returns a response in XML.

- Query parameters

`lat`	Latitude value in decimal format (not in degrees, minutes, and seconds) separated by a comma. Positive values indicate northern latitudes and negative values indicate southern latitudes.

`lng`	Longitude value, also in decimal format, separated by a comma. Eastern longitudes are given positive values and western longitudes are given negative ones.
`username`	A user name registered on the GeoNames website.

- Query response

  XML document. The API returns the place name (not necessarily an established city) that is closest to the provided coordinates, and it is a measurement of how far it is from those coordinates.

To learn more about the GeoNames API, visit the company website at:

http://www.geonames.org/export/web-services.html

Invoking the API

Example 4-5 is a sample invocation of the API.

Example 4-5 API invocation

```
http://api.geonames.org/findNearbyPlaceName?lat=12.966666667&lng=77.566666667&user
name=teamitso
```

This asks the web service to locate the populated area closest to the supplied latitude and longitude values. These values are provided by the `lat` and `long` query parameters, respectively. The eastern longitudes and northern latitudes are provided as positive real numbers; the western longitudes and southern longitudes are provided as negative real numbers. The username parameter is required by the Geonames website so that only registered users can use the API.

API response

Example 4-6 is a sample of the API response in XML.

Example 4-6 API response in XML

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<geonames>
<geoname>
<toponymName>Binny Mills</toponymName>
<name>Binny Mills</name>
<lat>12.96954</lat>
<lng>77.56695</lng>
<geonameId>6692887</geonameId>
<countryCode>IN</countryCode>
<countryName>India</countryName>
<fcl>P</fcl>
<fcode>PPL</fcode>
<distance>0.32049</distance>
</geoname>
</geonames>
```

The response comes in the form of an XML document that lists a populated location that is closest to the provided latitude and longitude values. Among the items returned are the name of the location (in this case Binny Mills), its latitude and longitude values, its country code, the name of the country, and the distance (in kilometers) from the named location to the precise spot indicated by the latitude and longitude values that were submitted originally.

Message flow

The next step is to do the same thing within a WebSphere Message Broker message flow. It has already been established that publications from the clients (the devices on the trucks) will be routed to local message queues using a WebSphere MQ subscription object. So the following steps describe how to implement the solution from the point of the JMSInput node onward.

The message flow, starting with the JMSInput node, works as follows:

1. Parse the incoming comma-delimited message (from the MQTT client or the truck) in the JMSInput node.
2. Parse the latitude and longitude values in the Compute node.
3. Use the latitude and longitude values to create the HTTP address with the query parameters in the Compute node.
4. Invoke the API in the HTTPRequest node.
5. Parse the API response also in the HTTPRequest node.
6. Extract the formatted address from the XML document in another Compute node.
7. Route the formatted address information to an MQTT topic using the JMSOutput node.

Sample application

As discussed earlier, publications from the trucks (the MQTT clients) are routed to local message queues for reverse geocoding, which will provide a formatted address instead of the original latitude and longitude values. The message queues serve as an input to the message flow.

WebSphere MQ JMS objects

The following JMS object definitions are required for the sample:

- A topic connection factory.
- A topic name, `DAILY.TRUCKS`, with a topic string of `Company/daily/#`. This topic name will be provided for subscription in the JMSInput node.
- A topic name, `DAILY.TRUCKS.POSITION`, with topic string of truck/address. This topic name will be provided for subscription in JMSOutput node.

WebSphere Message Broker

The following steps describe the installation of the message flow project:

1. Open the `GeoNamesJMSSample.zip` file in the additional materials supplied with this book.
2. Import the file into the WebSphere Message Broker Toolkit.
3. Ensure that the JMS objects match the JMS input and output nodes.
4. Build and create a `geoNamesJMSSample.bar` file. Deploy the file to an execution group.

WMQTT Utility setup

The following steps simulate the publisher (the devices on the trucks) and the subscriber (applications or personnel at company headquarters):

1. Start an instance of the WMQTT Utility. This instance will be the publisher.
2. Enter the IP address for the queue manager (`MQTT.QMGR00`).
3. Enter a client ID, in this case, `Truck13`.
4. Click **Connect**. The publisher is now connected to the broker.

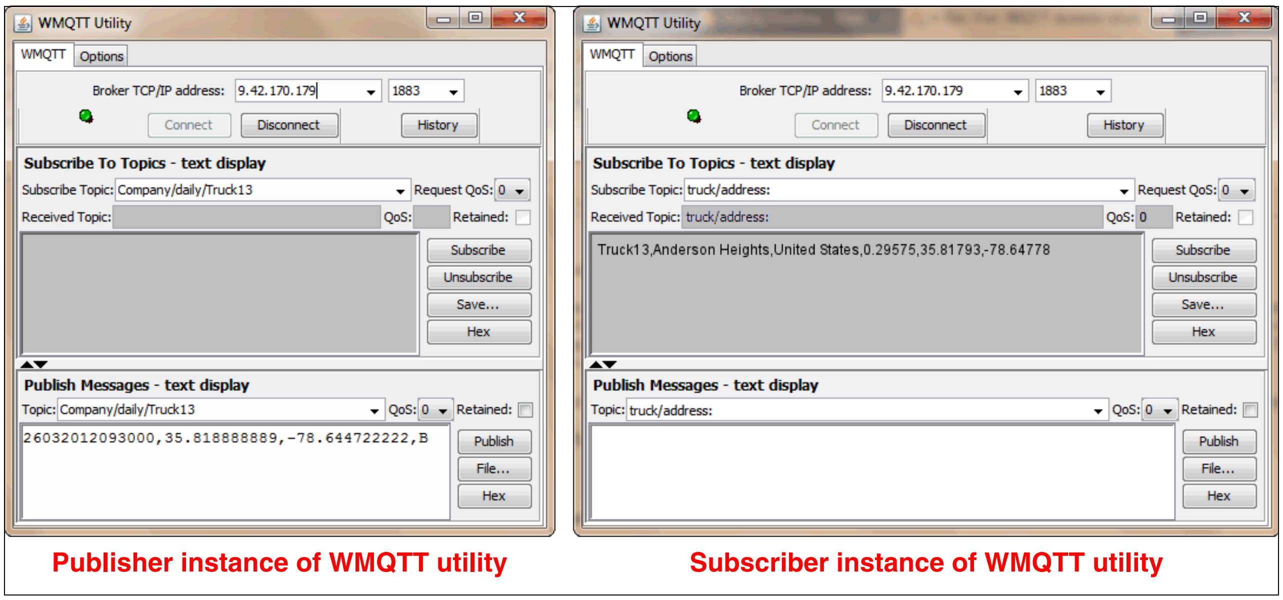
5. Enter a topic for publication, such as `Company/daily/Truck13`. Do not publish yet, because the subscriber instance is not yet set up.
6. Start another instance of the WMQTT Utility. This instance will be the subscriber.
7. Enter the IP address for the queue manager (`MQTT.QMGR00`).
8. Enter a client ID, in this case, `TruckPosition`.
9. Click **Connect**. The subscriber is now connected to the broker.
10. Enter the topic to which you are subscribing, as in `Truck/address`.
11. Click **Subscribe**. You are now subscribing to the truck/position topic.

Running the sample

To run the application to view the formatted address of the trucks, use the following steps.

1. Enter a message similar to `26032012093000,35.818888889,-78.644722222,B` in the Truck13 instance of WMQTT utility.
2. Click **Publish** to send the message.
3. Check the TruckPosition instance of WMQTT Utility.
4. The formatted address for Truck13 is visible.

Figure 4-2 shows what the subscriber sees returned from the publisher.


WMQTT Utility
WMQTT
Options
Connect
Disconnect
History
Subscribe To Topics - text display
Subscribe
Unsubscribe
Save...
Hex
Publish Messages - text display
Publish
File...
Hex
Publisher instance of WMQTT utility
Subscriber instance of WMQTT utility

Publisher instance:
Broker TCP/IP address: 9.42.170.179 1883
Subscribe Topic: Company/daily/Truck13 Request QoS: 0
Received Topic: QoS: Retained:
Topic: Company/daily/Truck13 QoS: 0 Retained:

```
26032012093000,35.818888889,-78.644722222,B
```

Subscriber instance:
Broker TCP/IP address: 9.42.170.179 1883
Subscribe Topic: truck/address: Request QoS: 0
Received Topic: truck/address: QoS: 0 Retained:

```
Truck13,Anderson Heights,United States,0.29575,35.81793,-78.64778
```

Topic: truck/address: QoS: 0 Retained:

Figure 4-2 The publisher (a truck) sends its GPS coordinates and the subscriber (at company headquarters) sees the corresponding formatted address

4.2 Integrate MQTT with IBM WebSphere Application Server

In this section we describe how to integrate MQTT-based applications with WebSphere Application Server. This makes it possible to publish messages to topics in WebSphere MQ and have them be consumed by applications running in WebSphere Application Server, and vice versa.

The WebSphere MQ JMS Resource Adapter is used for the interaction between WebSphere Application Server and WebSphere MQ. This resource adapter allows JMS applications and message-driven beans that are running in the application server to access the resources of a WebSphere MQ queue manager. The resource adapter supports both point-to-point and publish/subscribe messaging.

The transportation logistics scenario used throughout this book is extended now to demonstrate the following applications:

- Developing a simple application to store the trucks' current location data in a database
- Developing a web application for sending information to the trucks about their next destination

4.2.1 Assumptions

As described previously, the logistics scenario involves the following roles:

- Individual trucks with attached smart devices that monitor each vehicle's current location and use MQTT messages to send the data to the enterprise back end at company headquarters.
- Applications (or people) at company headquarters that receive the information from each truck, analyze it, and send updates to the truck, such as when a destination needs to be changed.

Two applications are required:

- The smart device on the truck that uses MQTT to send messages to headquarters
- A browser-based application to send destination changes to the trucks

To implement this combined solution, the following components are needed:

- On the devices on the trucks:
  - The Java MQTT client library from the WebSphere MQ Telemetry software
  - An MQTT-based messaging application developed using the Java API
- At headquarters:
  - A messaging server, in this case WebSphere MQ 7.1 with WebSphere MQ Telemetry
  - An application server, in this case WebSphere Application Server V8.0, located on the same machine as WebSphere MQ
  - A message-driven bean (MDB), hosted on the application server, to process the incoming messages and update a database

    MDB note: An MDB is a consumer of messages from a JMS provider and is invoked upon arrival of a message at the destination or endpoint that the MDB services.

  - A web application, hosted on the application server, that can be used to fetch the database records and to send messages

4.2.2 Integration

In this solution, shown in Figure 4-3 on page 138, logistical information from a truck is collected by an on-board sensor and sent by a smart device as an MQTT message to a specific topic on the WebSphere MQ Telemetry server at company headquarters. There, an analytical application residing on WebSphere Application Server subscribes to the topic to obtain the truck's latest logistical information. After the information is analyzed by the application, the truck's next destination is determined and the information is published to a different topic on the WebSphere MQ Telemetry server. The smart device on the truck, which is a subscriber to the topic, then receives the message containing the new destination.

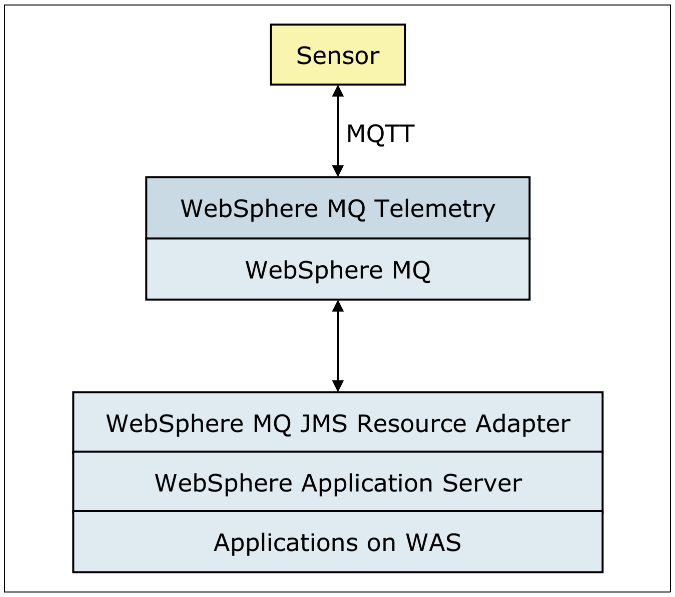


Figure 4-3 High-level topology of the sample solution

The WebSphere MQ Telemetry server in this scenario is a telemetry service hosted on a WebSphere MQ queue manager. WebSphere Application Server connects to the WebSphere MQ queue manager using the WebSphere MQ JMS resource adapter. A message-driven bean running in the application server picks up the messages and saves the data as records in a database. The MDB will be invoked after the message is available on the topic destination.

To begin the process of integrating with WebSphere Application Server, perform the following first steps:

- Install WebSphere MQ V7.1 as the messaging server.
- Install WebSphere Application Server V8.0.0.2 (the PM40967 Fix is required).
- Install IBM Rational® Application Developer 8.0.4.

When connecting from WebSphere Application Server to WebSphere MQ, WebSphere MQ classes for JMS will be used. WebSphere MQ classes for JMS is the JMS provider that is supplied with WebSphere MQ. The WebSphere MQ resource adapter containing implementation of WebSphere MQ classes for JMS, which adheres to J2EE connector architecture, is packaged with WebSphere Application Server as a resource adapter.

There are two ways a WebSphere MQ client can connect to WebSphere Application Server. In the *client* mode, WebSphere MQ classes for JMS connects to the queue manager using TCP/IP. In the *bindings* mode, it uses the Java Native Interface (JNI) to call directly into the queue manager API rather than communicating through the network. We use the client mode in this example. The WebSphere MQ and WebSphere Application Server instances can be on different machines.

Setting up WebSphere MQ

The following steps set up the WebSphere MQ queue manager and the telemetry service:

1. Create and configure a queue manager in WebSphere MQ using the steps given in Chapter 3, “Using MQTT with IBM WebSphere MQ Telemetry” on page 55, although here we name it MQQM.

2. Create a TCP/IP listener on port 1414 that enables WebSphere Application Server to connect using the client mode.

Setting up outbound communication from WebSphere Application Server

Outbound communication from WebSphere Application Server is required to send messages from the web application to WebSphere MQ queues and topics. You need to create the following JMS resources using the WebSphere Application Server administrative console (or the Integrated Solutions Console).

First, create the required connection factories as follows:

1. Under the Resources/JMS section, click **Connection Factories**. Then, click **New** to create a new connection factory.
2. In the next window, ensure that **WebSphere MQ Messaging Provider** is selected.
3. In the ensuing tabs, designate the following properties:

Name:	`cf`
JNDI name:	`jms/cf`
Queue manager:	`MQQM`
Transport:	`Client`
Hostname:	`127.0.0.1` (or the machine IP address where the WebSphere MQ queue manager MQQM was created and is running)
Port:	`1414`

Next, create the required topic as follows:

1. Under the Resources/JMS section, click **Topics**. Then, click **New** to create a new topic reference.
2. In the next window, ensure that **WebSphere MQ Messaging Provider** is selected.
3. In the ensuing tabs, designate the following properties:

Name:	`truckinfo`
JNDI name:	`jms/truckinfo`
WebSphere MQ topic name:	`truck/` (serves as a wildcard)

Developing applications using Rational Application Developer

You need to develop two applications using Rational Application Developer:

- A web application that accepts input from a browser (in this case, the next destination for the truck) and sends that information to a remote device
- A message-driven bean (MDB) that receives the current location as a message and saves the information in a database

The code of the browser-based application is provided in the application material supplied with this book. It can be imported into the Rational Application Developer workspace and run from there.

The other part of the application is the outbound communication that is used to send the messages to the truck.

Inbound communication

The first part of the application is the inbound communication that contains the data sent from the trucks. This is done using an MDB that watches for messages that arrive with the topic `truck/+` with the plus sign (+) referring to the unique client identifier of the truck. The message content is the location of the truck.

To keep the example simple, the message received is displayed only on the console and is not stored in a database. In a real scenario, the database is updated. The updated data can be read by a servlet or a JSP and can be displayed on a browser.

Outbound communication

A JSP is used to interact with the user. The input is processed by a servlet and invokes a session bean that publishes the message to a topic. The message is the next location the truck has to go to. The message is published to the *destination/truckname topic*, where *truckname* is the unique client identifier of the truck. The truck subscribed to this topic will receive the message.

Example 4-7 shows the session bean code that publishes the message to the topic.

Example 4-7 Session bean code that publishes the message to the topic

```
public void sendLocationData(String location, String truckid) {
try {
Connection conn = cf.createConnection();
Session sess = conn.createSession(false, Session.AUTOACKNOWLEDGE);
MessageProducer prod = sess.createProducer(sess.createTopic("truck/" + truckid));
prod.send(sess.createTextMessage(location));
prod.close();
sess.close();
conn.close();
} catch (JMSException e) {
e.printStackTrace();
}
}
```

The topic string is dynamically created. The message is published as a JMS text message. This message will be automatically converted to an MQTT message and sent to a subscriber.

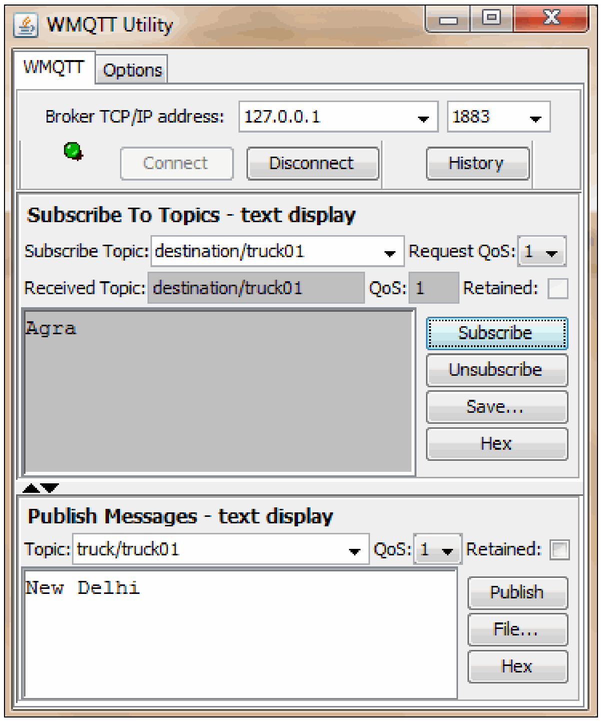
Deploying the sample

The attached sample can be deployed easily. It is provided in the form of an EAR file that can be installed on WebSphere Application Server. You can find deployment instructions at:

http://publib.boulder.ibm.com/infocenter/wasinfo/v8r0/topic/com.ibm.websphere.base.doc/info/aes/ae/welc_howdoi_tapp.html

Running the sample

After the sample is deployed into the WebSphere Application Server instance, the WMQTT Utility (from SupportPac IA92) can be used as a client to send and receive messages (Figure 4-4 on page 141).



Broker TCP/IP address: 127.0.0.1 1883

Subscribe To Topics - text display

Subscribe Topic: destination/truck01 Request QoS: 1

Received Topic: destination/truck01 QoS: 1 Retained: ☐

```
Agra
```

Publish Messages - text display

Topic: truck/truck01 QoS: 1 Retained: ☐

```
New Delhi
```

Figure 4-4 Using the WMQTT Utility

4.3 Integrate MQTT with IBM WebSphere Operational Decision Management

Integrating MQTT-based messaging systems with WebSphere Operational Decision Management, a business events processing engine, can turn simple status updates from remote devices into alarms that focus immediate attention on the things being monitored. WebSphere Operational Decision Management can be used to define and apply business rules to incoming events, so it is an ideal partner with MQTT messaging, which enables communication with devices at the farthest reaches of a network.

The scenario involving Transportation Company B also illustrates how to apply business rules to events that are communicated through MQTT messages. For example, the company, in addition to monitoring the specific location of each of its trucks, might also need to track whether the trucks are staying on the projected travel routes to ensure that goods are delivered on time. The company can use WebSphere Operational Decision Management to define the projected route of each truck and an area of acceptable variance around that route where the truck is allowed to travel.

Frequent updates from MQTT-enabled remote devices give WebSphere Operational Decision Management the current location of each truck, as shown in Figure 4-5 on page 142. This information is analyzed against the projected route (and the acceptable area of variance) to generate an alarm if the truck is driven off course.

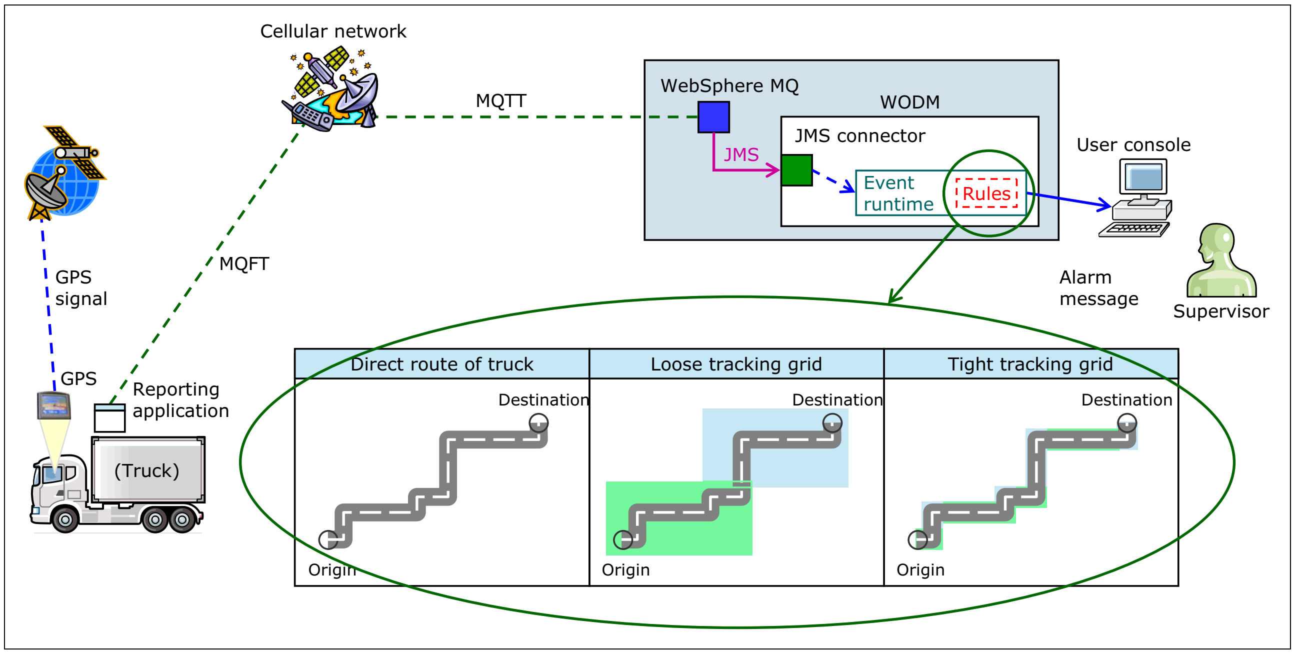


Figure 4-5 Using MQTT and WebSphere Operational Decision Management to generate an alarm when a truck goes too far from its intended travel route

To implement this solution, the developer must define the events, rules, and actions using the WebSphere Operational Decision Management Event Designer and then publish the assets to the WebSphere Operational Decision Management Event Runtime.

For more information about the components of a WebSphere Operational Decision Management installation, see:

http://publib.boulder.ibm.com/infocenter/dmanager/v7r5/index.jsp

4.3.1 Using Event Designer

Event Designer is an Eclipse-based application used to define events in WebSphere Operational Decision Management, along with the rules for handling each event and the actions, if any, that are executed in response to each event.

Events

Events represent an occurrence outside the monitoring system. They generally take the form of an incoming message that is presented in a format that makes the information being conveyed understandable to the system. In the scenario outlined in this section, the arrival of each message containing the location information for a given truck is considered an event. Events hold relevant information in *event objects*. In a simple system, such as that described here, it is sufficient to use fields for the truck identification number and its current position (using latitude and longitude coordinates), along with a time stamp.

Connectors

Connectors are the way data enters or exits the system. A connector is responsible for handling the technical aspects of each type of communication. There are specific connectors for handling HTTP posts, JMS messages, or file system operations.

For using MQTT-generated data as an event source, a JMS connector is defined for the event; it is the easiest way to receive the message from the queue manager without any intermediary applications. In this case the original MQTT message from the device on the truck should be published as a JMS text message with a predefined topic. The connector must be configured to subscribe to that predefined topic to allow the message to enter the Events Runtime for processing.

Although four different message formats are available for communicating events, this discussion focuses on the connector packet and the custom XSL transformation formats.

Using a connector packet

A *connector packet* is an XML file that is defined using the Decision Server Events packet schema, which is available at:

```
http://publib.boulder.ibm.com/infocenter/dmanager/v7r5/topic/com.ibm.wbe.reference
.doc/doc/developingapplications-ituser.html
```

If this option is chosen, the developer must ensure that the incoming messages match the connector packet format. Depending on the scenario, there are different possibilities for which component will be responsible for composing the message.

The easiest way is to compose the message in the specified format directly on the device, and then publish without further need for transformation. This approach is an added burden for the messaging application on the device. Not all message applications can handle the specified XML format. In addition, this method can waste network bandwidth, because the message length can be smaller if only the needed data is sent without any additional formatting.

A second option is to add a formatting component between the incoming MQTT message from the device and the outgoing JMS message to WebSphere Operational Decision Management. This component can read the message directly from the queue manager or can subscribe to the appropriate JMS topic and receive the message and then transform it to the connector packet format. This method can be done using IBM WebSphere Message Broker or a custom-made application to intercept the messages, re-format them to meet the connector packet specification, and republish the messages to the final destination topic. This approach saves bandwidth and allows the application on the remote device to remain relatively simple.

Using a custom XSL transformation

The *custom XSL transformation* approach allows the developer to use any message format as long as an XSL file is provided to transform the original XML format to what is required for the internal Decision Server Events packet format. This approach is most useful when there are no extra components between the message reception and WebSphere Operational Decision Management, but the construction and debugging of the XSL file can be time consuming.

Business objects

Business objects are the way data flows inside the WebSphere Operational Decision Management Event Runtime engine. They typically include some event object information but also add some extra fields needed to implement the functionality. For example, business objects can be created from the event object, with an additional field added to indicate the date and time when an alarm is raised.

Actions

Actions are the output for the Event Runtime engine. They are executed when a condition is evaluated, possibly initially triggered by an incoming event. For the scenario being defined here, only one action needs to be defined, to be invoked when a truck is out of its planned route.

Just like events, actions can hold a connector that determines the nature of the action taken place. For our sample scenario, a user console connector is enough to notify a supervisor at company headquarters that a truck is out of route. If the supervisor works in a remote location or travels to different locations, a JMS connector can be used to send the notification using the MQTT protocol, with the same message formatting considerations used for events.

Event rules

The final step in the development process is creating the *event rules* that determine which actions are invoked when a event arrives under defined circumstances.

For the scenario described in this paper, the developer must define rules to determine if the truck's location coordinates (latitude and longitude) are within the acceptable route. If the coordinates are ever not within the acceptable route, an alarm action is raised containing all the information included in the Business Object Definition (Event Object Data plus the date and time of the message).

4.3.2 Deploying the solution

For initial testing purposes, the solution created in Event Designer can be deployed into an Event Runtime environment. You need to define a runtime connection that includes the host, port, user, and password in the Event Runtime server.

For information about using the Business Space widgets to test the solution, see:

http://publib.boulder.ibm.com/infocenter/dmanager/v7r5/index.jsp

4.4 Integrate MQTT with IBM Intelligent Operations Center

You can integrate devices or applications that use the MQTT protocol with the IBM Intelligent Operations Center, an event management system that can help municipalities monitor operations and predict and respond to changing situations. For example, integrating MQTT makes it possible to extend the reach of the IBM Intelligent Operations Center to traveling road maintenance vehicles or to remote devices monitoring rising water or approaching storms.

Figure 4-6 provides a high-level view of the IBM Intelligent Operations Center. When MQTT devices are introduced, they connect to the internal event management engine, through which events and other updates are processed.

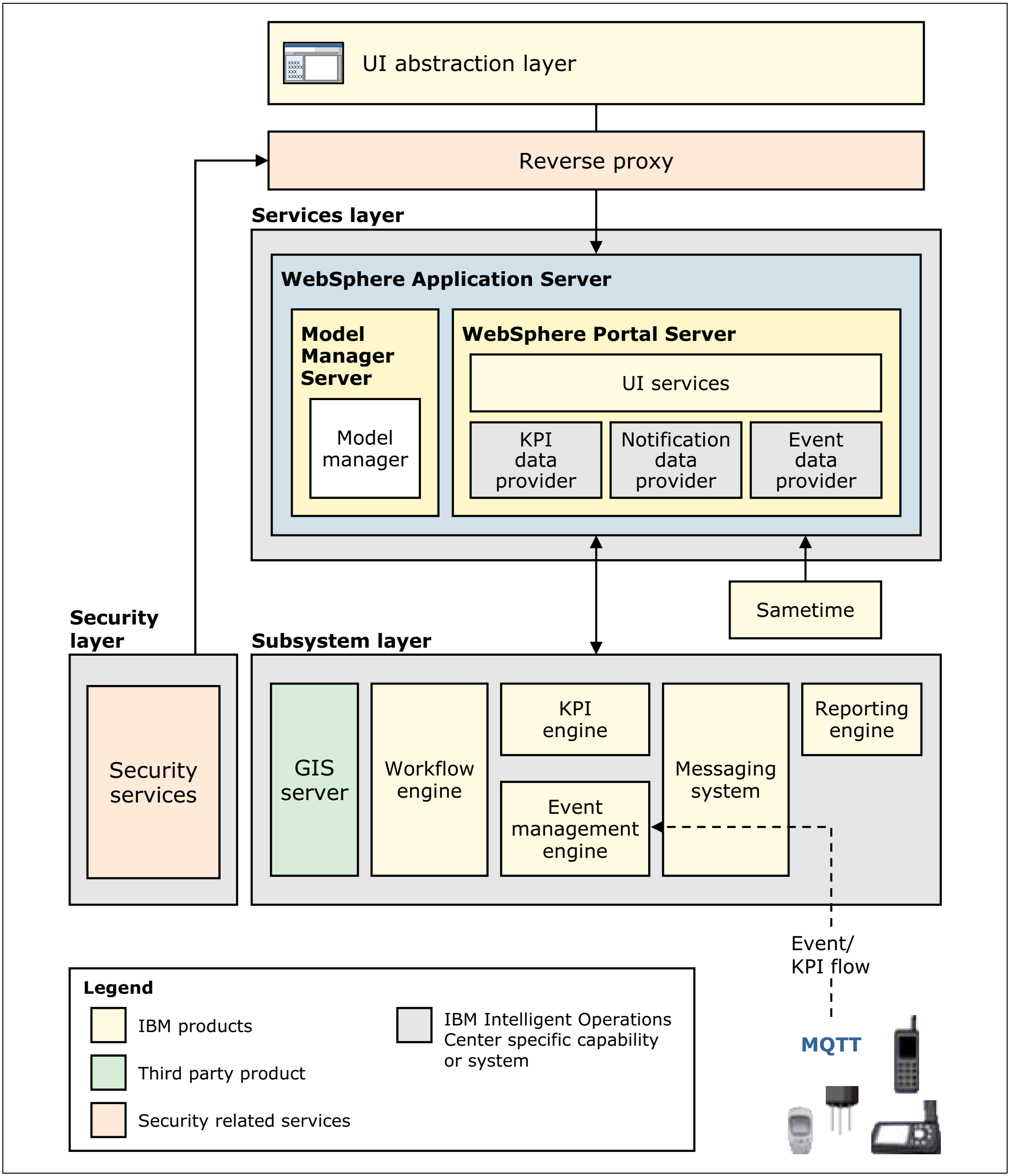


Figure 4-6 IBM Intelligent Operations Center architecture as integrated with devices

For more information about event management using the IBM Intelligent Operations Center, see:

http://pic.dhe.ibm.com/infocenter/cities/v1r0m0/index.jsp?topic=/com.ibm.iicoc.doc/admin_evtprocov.html

For more information about components and subsystems, see:

http://pic.dhe.ibm.com/infocenter/cities/v1r0m0/index.jsp?topic=/com.ibm.iicoc.doc/ov_products.html

4.4.1 The Common Alert Protocol format

The Common Alerting Protocol (CAP) format is an OASIS standard. It is a general format for exchanging emergency alerts and public warnings over various types of networks. CAP allows a consistent warning message to be disseminated simultaneously over many different warning systems. It also facilitates the detection of emerging patterns in local warnings of various kinds, which can indicate undetected hazards or hostile acts.

Additional information: For more information and the complete CAP specification, visit the OASIS website at:

http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html

The IBM Intelligent Operations Center supports messages in the CAP format and uses it to exchange event information with external systems. Products and services can be integrated with the IBM Intelligent Operations Center using events. Events communicated to the IBM Intelligent Operations Center must be in the CAP format. These events can relate to key performance indicators (KPI) monitored by the IBM Intelligent Operations Center or might be unrelated to KPIs.

Example 4-8 shows a sample CAP message that describes a car accident.

Example 4-8 Example of a CAP message describing a car accident

```
<?xml version="1.0" encoding="UTF-8"?>
<cap:alert xmlns:cap="urn:oasis:names:tc:emergency:cap:1.2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:oasis:names:tc:emergency:cap:1.2 CAP-v1.2-os.xsd ">
<cap:identifier>1112</cap:identifier>
<cap:sender>Transportation</cap:sender>
<cap:sent>2011-02-17T15:00:00-05:00</cap:sent>
<cap:status>Actual</cap:status>
<cap:msgType>Alert</cap:msgType>
<cap:scope>Public</cap:scope>
<cap:code>KPI</cap:code>
<cap:info>
<cap:category>Transport</cap:category>
<cap:event>TrafficAccident</cap:event>
<cap:urgency>Unknown</cap:urgency>
<cap:severity>Extreme</cap:severity>
<cap:certainty>Unknown</cap:certainty>
<cap:eventCode>
<cap:valueName>OwningOrg</cap:valueName>
<cap:value>Police</cap:value>
</cap:eventCode>
<cap:onset>2011-02-17T15:00:00-05:00</cap:onset>
<cap:senderName>Transportation</cap:senderName>
<cap:description>Single car crash</cap:description>
<cap:parameter>
<cap:valueName>accident number</cap:valueName>
<cap:value>1112</cap:value>
</cap:parameter>
</cap:info>
</cap:alert>
```

For more information about integrating IBM Intelligent Operations Center with the CAP format, see:

http://publib.boulder.ibm.com/infocenter/cities/v1r0m0/topic/com.ibm.iicoc.doc/int_cap.html

CAP format messages can be published from a device or from an intermediate application:

- A device generates the complete CAP standard-compliant XML document and sends it over MQTT. However, this option is generally not expected of devices because they tend to have little memory. Where possible, the CAP format message can be published directly to the inbound message queue in WebSphere Message Broker.
- A device can also publish its raw data, such as temperature, water level, and other data, to an application server or IBM WebSphere Message Broker. The application server or IBM WebSphere Message Broker then transforms the incoming message to CAP format and forwards it to the inbound message queue in WebSphere Message Broker. Alternatively, the raw message can be published to a topic on the queue manager (MQTT broker) in the IBM Intelligent Operations Center, and a separate message broker instance can transform it into CAP format for further processing.

4.4.2 Integration points of the IBM Intelligent Operations Center

A complete installation of the IBM Intelligent Operations Center involves an instance of WebSphere Message Broker V8.0 and WebSphere MQ V7.0. In both cases, the format of the message is an XML document that is compliant with the CAP format.

The WebSphere Message Broker instance has an inbound queue that is defined to consume messages that are published in CAP format. This queue name is also exposed as a JMS topic destination. For details about using the inbound queue, see:

http://pic.dhe.ibm.com/infocenter/cities/v1r0m0/topic/com.ibm.iicoc.doc/int_eventinboundqueue.html

WebSphere MQ V7.0 uses telemetry channels. After a WebSphere MQ V7.0 queue manager with telemetry support is established, the queue manager (or the MQTT broker, specifically) in the IBM Intelligent Operations Center can accept the connections from devices (using the MQTT protocol) directly. The publications are received as topics, which are then consumed by WebSphere Message Broker applications either using a JMS topic or a local queue.

4.5 Integrate MQTT with IBM Lotus Expeditor integrator

IBM Lotus Expeditor integrator is an edge server used for n-tier messaging topologies. It is an OSGi event and message-driven Java application that consists of Eclipse plug-ins that allow for simple, configurable data exchange over different transport protocols to integrate remote locations (and remote devices and appliances) with powerful back-end service buses. The communication channel between the back-end messaging system and Lotus Expeditor integrator is based on JMS messaging.

Using IBM Lotus Expeditor integrator as an edge server includes the following typical industry examples:

- Transporting data to the remote endpoint
- Retrieving data from the remote endpoint
- Managing data exchange between remote resources without back-end interaction
- Retrieving information about remote endpoint resources

- Controlling the Expeditor integrator run time and resources

More information about IBM Lotus Expeditor integrator is available at:

http://www.ibm.com/software/lotus/products/expeditor/integrator.html

You can find additional technical details about integrating applications with IBM Lotus Expeditor integrator at:

http://www.lotus.com/ldd/lewiki.nsf/dx/integrator02.html

To achieve an independent remote runtime environment for an edge server that includes a messaging provider, Expeditor integrator comes with the Eclipse/Equinox application runtime environment plus additional Java EE-like services, including IBM micro broker as a messaging provider for MQTT and JMS clients, monitoring services for end-to-end monitoring, and an adapter framework for integration of devices with traditional protocols and remote management facilities.

Lotus Expeditor integrator uses the included IBM micro broker as the MQTT messaging provider and an IBM micro broker bridge (see Figure 4-7). The IBM micro broker and broker bridge is used for transparent connectivity to other JMS-compliant messaging back ends (such as IBM WebSphere MQ). The core component of the Lotus Expeditor integrator application is the Expeditor integrator Application Control Service (ACS). The Lotus Expeditor integrator ACS is a central processing component for all incoming and outgoing messages. It interprets the messages and triggers further actions to orchestrate a use case within a transactional unit of work. For example, the ACS provides file transmission capabilities for retail environments and messaging processing and forwarding functions for SmartGrid environments.

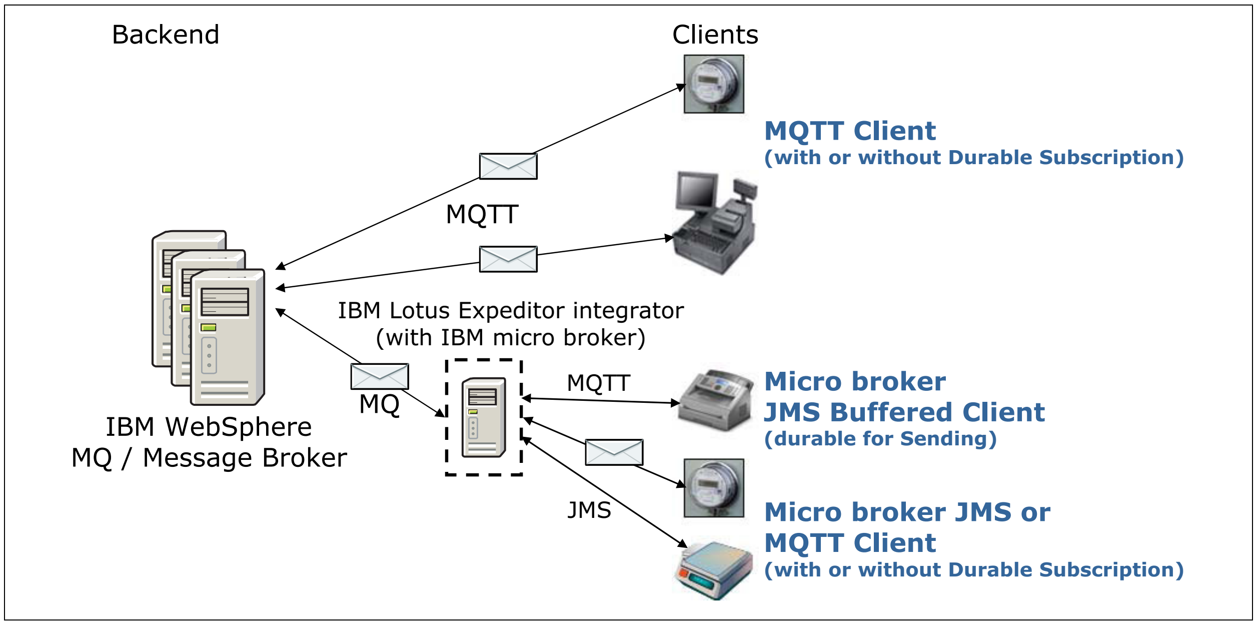


Figure 4-7 Lotus Expeditor integrator extends IBM micro broker with application intelligence

4.5.1 Messaging in Expeditor integrator

Lotus Expeditor integrator is based on IBM micro broker as the MQTT messaging provider and is an event- and message-driven integration hub for exchanging data using MQTT and other data transport protocols.

There are two communication channels for bridging MQTT messages from the client side (the device or appliance) to the back-end service bus:

- JMS over WebSphere MQ
- HTTP (inbound: REST, outbound: based on the Apache HTTP client)

Custom header parameters (message properties) control the executed Application Control Service actions. They define the message's purpose, such as sending/retrieving files, messages or other data; executing simple data processing such as data evaluation and filtering; and controlling the Lotus Expeditor integrator application through remote control, remote management, and remote monitoring.

Figure 4-8 on page 150 shows an example for retrieving MQTT messages from devices at the decentralized integration point (the edge server). The received MQTT message can be further processed there or transformed into a message for the target back-end messaging system (for example, WebSphere MQ). The message header of the message publisher can contain specific message properties (custom message headers) that trigger an assigned ACS flow in Expeditor integrator.

There are many default flows for predefined standard use cases already available that can be kicked off by the correct message properties. Figure 4-8 on page 150 illustrates setting the property MessagePurpose to `MessageForward` and TransportType to `MESSAGE`. The ResourceType can be used as an alias through which further configuration data can be retrieved during flow execution, for example, the DestinationName (the target queue/topic name that is bridged to a specific queue/topic in the back-end messaging system). The JMS Destination Adapter monitors the topic or queue (CustomerOutQ/CustomerTopic) to which the initial message is sent and triggers the correct ACS flow based on the provided message properties. The triggered ACS flow gets the received message, retrieves further configuration information, such as the target queue name, and transfers the message payload to the provided back-end queue.

ACS flow execution status messages are sent to the monitoring queues at the beginning and the end of the flow. Messages (for example MQTT) can be sent to the Expeditor integrator application to transport data and trigger actions.

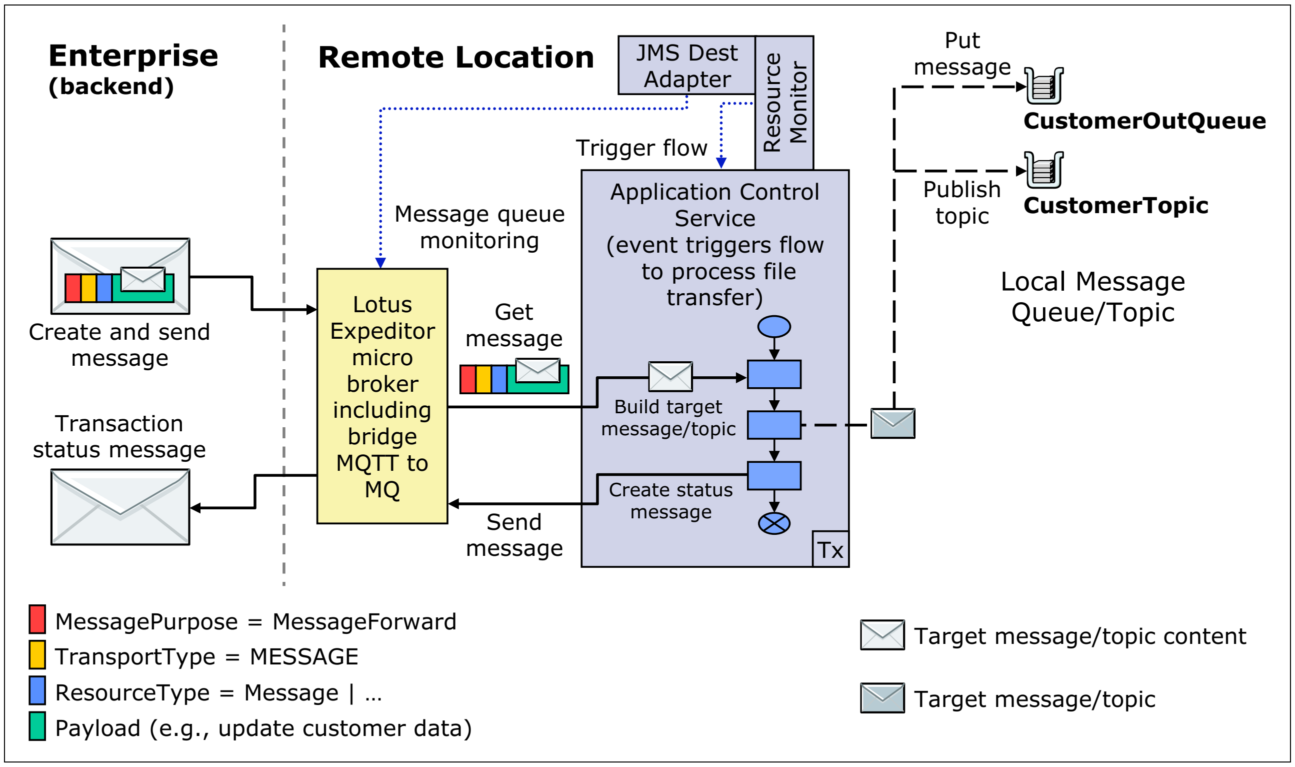

Enterprise
(backend)
Remote Location
JMS Dest Adapter
Resource Monitor
Put message
CustomerOutQueue
Publish topic
CustomerTopic
Trigger flow
Message queue monitoring
Application Control Service (event triggers flow to process file transfer)
Local Message Queue/Topic
Create and send message
Lotus Expeditor micro broker including bridge MQTT to MQ
Get message
Build target message/topic
Transaction status message
Create status message
Send message
Tx
MessagePurpose = MessageForward
TransportType = MESSAGE
ResourceType = Message | …
Payload (e.g., update customer data)
Target message/topic content
Target message/topic

Figure 4-8 Principle of receiving messages from devices and forwarding them to back-end integration layer

Predefined values for the message header property MessagePurpose are defined in Expeditor integrator for specific actions and use cases for which corresponding ACS default flows are included.

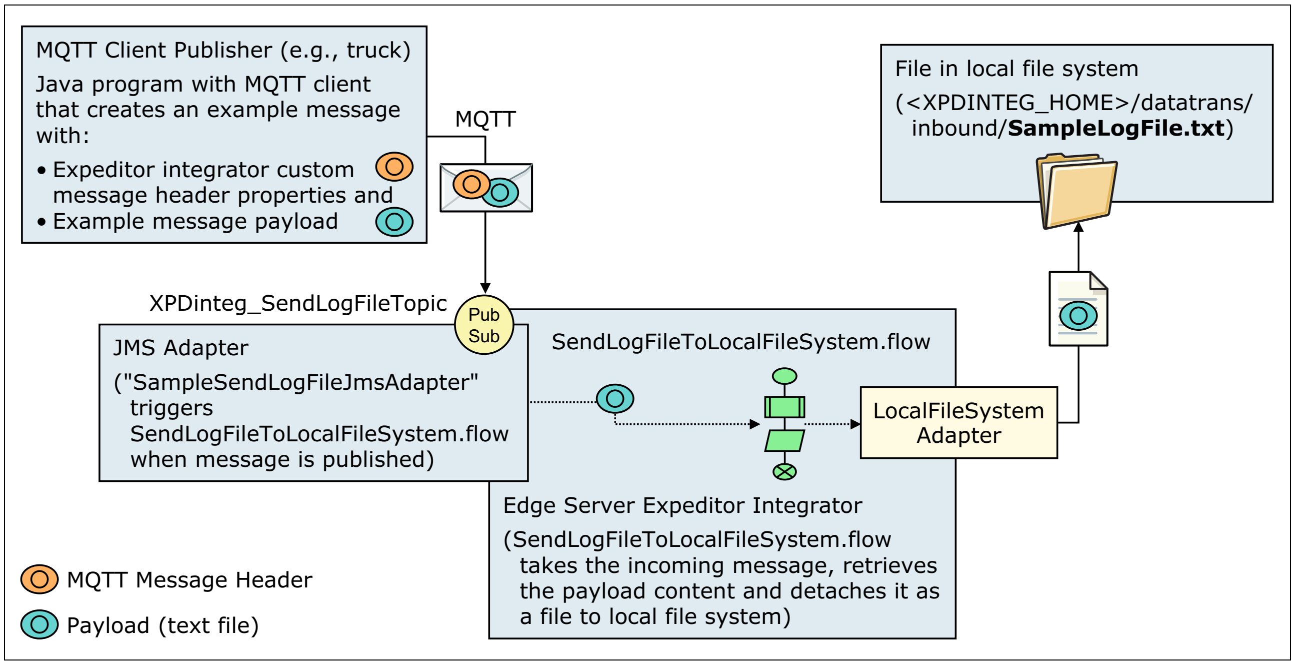
4.5.2 Example 1: Local integration of established devices and protocols through messaging

This example demonstrates how data from a remote MQTT-enabled instance is sent to Lotus Expeditor integrator as a decentralized integration hub. For example, the retrieved data from a delivery truck can position log data that is required for instant truck dispatching and rescheduling. The payload data is then retrieved from the message and is detached locally, for example as a file in a local file system where other systems use it as a data import source. There can also be requirements for the other direction, sending local data from traditional applications that is available in a file format to be connected to messaging-enabled devices.

Use case flow

This use case uses the following flow (Figure 4-9 on page 151):

1. MQTT client application (Publisher) sends the MQTT message containing example text data (for example, the truck position log) to the edge server at company headquarters.
2. Lotus Expeditor integrator monitors for published messages from trucks at a defined MQTT topic. If a message is received, the message content is then detached to the local file system as a `SampleLogFile.txt` file (using the `SendLogFileToLocalFileSystem.flow` file).
3. When the received truck position log data is detached as a local file, this file can be easily used for data input into traditional applications (for example, a truck scheduling system).



MQTT Client Publisher (e.g., truck)
Java program with MQTT client that creates an example message with:
• Expeditor integrator custom message header properties and
• Example message payload
MQTT
File in local file system
(<XPDINTEG_HOME>/datatrans/inbound/SampleLogFile.txt)
XPDinteg_SendLogFileTopic
Pub Sub
JMS Adapter
("SampleSendLogFileJmsAdapter" triggers SendLogFileToLocalFileSystem.flow when message is published)
SendLogFileToLocalFileSystem.flow
LocalFileSystem Adapter
Edge Server Expeditor Integrator
(SendLogFileToLocalFileSystem.flow takes the incoming message, retrieves the payload content and detaches it as a file to local file system)
MQTT Message Header
Payload (text file)

Figure 4-9 Local integration example of writing content of received MQTT messages to a local file

Prerequisites

This example requires the following prerequisites:

- ► Installed Lotus Expeditor integrator run time

  Make note of the `=<XPDINTEGHOME>` installation directory
- ► The Lotus Expeditor `XPDinteg.xml` integrator configuration file, which is available in the `<XPDINTEGHOME>/config` directory
- ► A simple text file, `myFile.txt`, that contains sample position data and that can be copied into the text message body of the MQTT client application

Configuration steps

Complete the following to configure this example.

1. Configure the SampleSendLogFileJmsAdapter JMS adapter in the `XPDinteg.xml` configuration file, as shown in Example 4-9.

Example 4-9 Step 1 for the JMSDESTINATIONADAPTER to receive log messages

```
<adapter type="XPDINTEGJMSDESTINATIONADAPTER" name="SampleSendLogFileJmsAdapter">
<listener>
<topic>com/ibm/integrator/flowtriggerevent/FileTransfer/LocalFileSystemFile/Sample
SendLogFileJmsAdapter</topic>
</listener>
<configuration>
<param name="JndiConnectionFactoryKey" value="jms/XPDintegConnectionFactory"/>
<param name="JndiDestinationKey" value="jms/XPDintegSendLogFileTopic"/>
<param name="JndiDeadLetterKey" value="jms/XPDintegServerDeadletterQ"/>
<param name="ValidateLocationId" value="OFF"/>
</configuration>
</adapter>
```

2. Configure the publisher/subscriber topic under which the message carrying the log file will be published, as shown in Example 4-10.

   Example 4-10 Step 2 in JMSDESTINATIONADAPTER configuration

   ```
   <topics> ...
   <topic purpose="SendLogFileTopic">
   <topic-name>XPDintegSendLogFileTopic</topic-name>
   <jndi-key>jms/XPDintegSendLogFileTopic</jndi-key>
   </topic>
   ...
   ```

3. Create the ACS flow for picking up the message and writing the payload into a local file. Locate examples of ACS flow definition files (under `<XPDINTEGHOME>/samples/flowdefs`). Make sure a unique process name and the EventAdmin Service trigger topic of the configured SampleSendLogFileJmsAdapter are used. Create a new flow file `SendLogFileToLocalFileSystem.flow` in directory `<XPDINTEGHOME>/flowdefs` with the following content and these settings:
   – DestinationPath = `datatrans/inbound`
   – DestinationName = `SampleLogFile.txt`
   – DestinationCreationMode = `append`

Example 4-11 shows the ACS flow definition for detaching the payload of the received log messages to a local file (`SampleLogFile.txt`).

Example 4-11 ACS flow definition

```
<?xml version="1.0" encoding="UTF-8"?>
<Process Name="SendLogFileToLocalFileSystem\Process"
Trigger="com/ibm/integrator/flowtriggerevent/FileTransfer/LocalFileSystemFile/Samp
leSendLogFileJmsAdapter">
<XPDintegActivity
Name="SendLogFileToLocalFileSystem\MessageReadActity"
ActivityName="XPDINTEGMESSAGEREAD"
PropertyKey="DATAREFERENCE"
JndiConnectionFactoryKey="jms/XPDintegConnectionFactory"
/>
<XPDintegActivity
Name="SendLogFileToLocalFileSystem\ChangeCustomHeader"
ActivityName="XPDINTEGCHANGECUSTOMHEADER"
AddDefaultHeaders="FALSE"
Header1="OVERWRITE::DestinationName::SampleLogFile.txt"
Header2="OVERWRITE::DestinationPath::datatrans/inbound"
Header3="OVERWRITE::DestinationCreationMode::APPEND"
/>
<XPDintegActivity
Name="SendLogFileToLocalFileSystem\MessageToFile"
ActivityName="XPDINTEGMESSAGETOFILE"
JndiConnectionFactoryKey="jms/XPDintegConnectionFactory"
/>
<XPDintegActivity
Name="SendLogFileToLocalFileSystem\FileWriteToFileSystem"
ActivityName="XPDINTEGFILEWRITETOFILESYSTEM"
AdapterName="XPDINTEGFILESYSTEMADAPTER"
/>
</Process>
```

Deploy and activate the configuration

To deploy and activate the configuration, follow these steps:

1. Stop the Lotus Expeditor integrator run time (click **Close** or **Exit** in the OSGi console).
2. Reset the Lotus Expeditor integrator run time (run `<XPDINTEGHOME>resetscript/XPDintegReset.bat` or `.sh`).
3. Start the Lotus Expeditor integrator run time (run `<XPDINTEGHOME>/XPDintegStart.bat` or `.sh`).
4. Wait until the Lotus Expeditor integrator run time is available (check by pointing your browser to the admin GUI). You can use a local host (`http://localhost:8777/ui`). Under Linux, the run time starts in the background the first time it is started after a reset. Check for the expeditor process and end/kill it. Then rerun `XPDintegStart.sh`.

MQTT client application (Publisher)

Example 4-12 shows the syntax that creates the MQTT client application (Publisher) to send the log message to topic `XPDintegSendLogFileTopic`.

Example 4-12 MQTT client application sending message to a topic

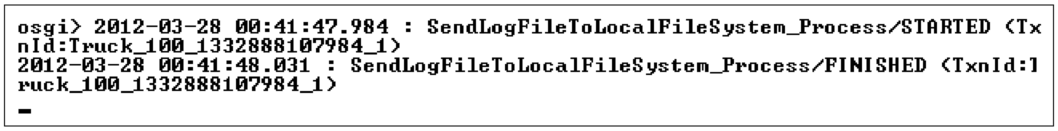
```
public static final String pubTopicString = System.getProperty("pubTopicString",
"XPDintegSendLogFileTopic");
```

Run the example

To run the example, follow these steps:

1. Start Lotus Expeditor integrator.
2. Switch on ACS flow logging by entering `toggleFlowLog`.
3. Use your MQTT client application to publish a log message with some log sample text payload (for example, content from `myFile.txt`).
4. The message is retrieved and its payload appended to `SampleLogFile.txt` in `<XPDINTEGHOME>/datatrans/inbound`.
5. The execution of your ACS flow SendLogFileToLocalFileSystemProcess is then displayed in the OSGi console.
6. Note that the file `datatrans/inbound/SampleLogFile.txt` is created and the content of the message payload is appended.
7. You can repeat sending messages. The message payload is appended to the target `SampleLogFile.txt`.

See Figure 4-10 for a successful example of OSGi console output when SendLogToLocalFileSystem.flow is executed properly.



```
osgi> 2012-03-28 00:41:47.984 : SendLogFileToLocalFileSystem_Process/STARTED (Tx
nId:Truck_100_1332888107984_1)
2012-03-28 00:41:48.031 : SendLogFileToLocalFileSystem_Process/FINISHED (TxnId:T
ruck_100_1332888107984_1)
-
```

Figure 4-10 Lotus Expeditor integrator OSGi console output when SendLogToLocalFileSystem.flow is executed

4.5.3 Example 2: Messaging-based back-end integration of devices and protocols

This example explains how messages from micro broker clients (MQTT clients) can be forwarded by the edge server to the back-end messaging system. Data preprocessing, filtering, auditing, and so on, take place during the forwarding process.

Use case

This use case uses the following flow:

1. The MQTT client application (Publisher) sends an MQTT message containing example text data (such as truck status log information) to the edge server at company headquarters. This data is of interest to the company and must be further forwarded to the back-end messaging system. Truck usage data is collected and further analyzed for possible usage optimization, for truck maintenance and for purchase and investment planning.
2. Lotus Expeditor integrator monitors for published messages from trucks at a defined MQTT topic. If a truck status message is received, the message content is forwarded to the corresponding WebSphere MQ queue in the back end XPDintegResInQ100 (using the `SendStateInfoMessageForward.flow`). See Figure 4-11 for a successful example of this process.
3. When the received truck position log data is detached as a local file, this file can be easily used for data input into established applications.

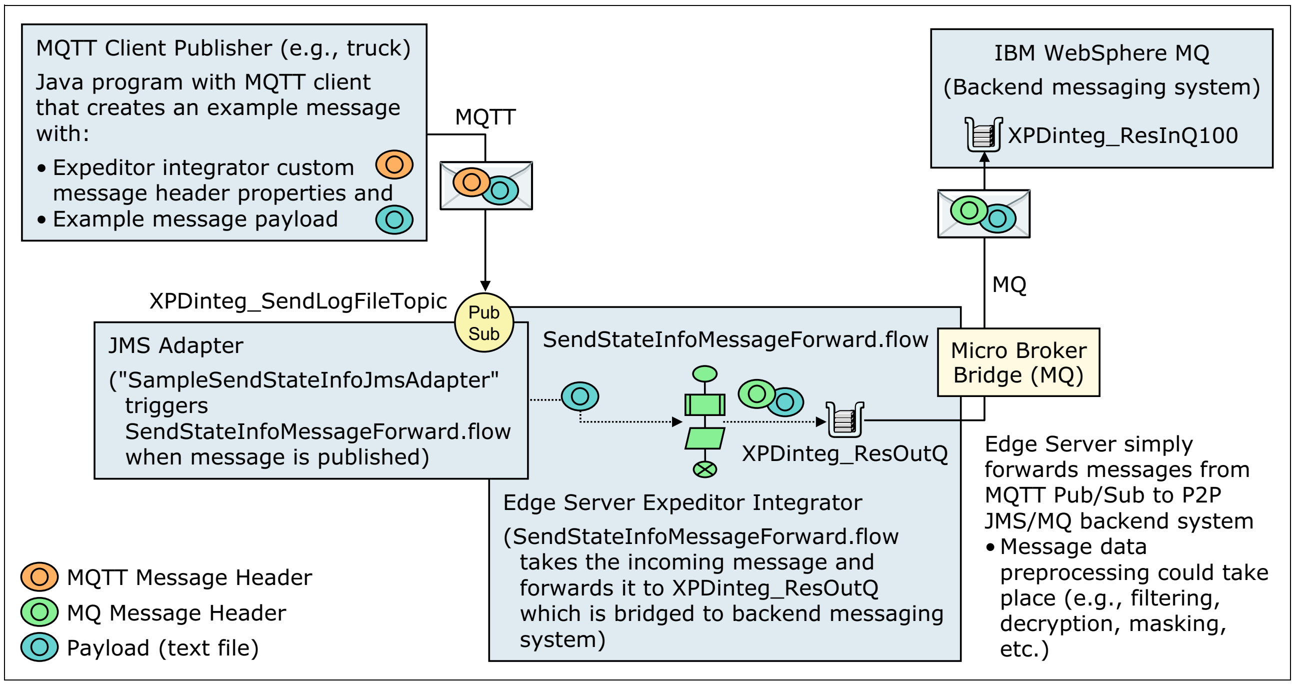

MQTT Client Publisher (e.g., truck)
Java program with MQTT client that creates an example message with:
• Expeditor integrator custom message header properties and
• Example message payload
MQTT
IBM WebSphere MQ
(Backend messaging system)
XPDinteg_ResInQ100
MQ
XPDinteg_SendLogFileTopic
Pub Sub
JMS Adapter
("SampleSendStateInfoJmsAdapter" triggers SendStateInfoMessageForward.flow when message is published)
SendStateInfoMessageForward.flow
Micro Broker Bridge (MQ)
XPDinteg_ResOutQ
Edge Server Expeditor Integrator
(SendStateInfoMessageForward.flow takes the incoming message and forwards it to XPDinteg_ResOutQ which is bridged to backend messaging system)
Edge Server simply forwards messages from MQTT Pub/Sub to P2P JMS/MQ backend system
• Message data preprocessing could take place (e.g., filtering, decryption, masking, etc.)
MQTT Message Header
MQ Message Header
Payload (text file)

Figure 4-11 Example for sending MQTT messages containing truck status information to the MQ back end

Prerequisites

This example requires the following prerequisites:

- Installed Lotus Expeditor integrator run time

  Make note of the installation directory (=`<XPDINTEGHOME>`)

- The Lotus Expeditor integrator `XPDinteg.xml` configuration file, which is available in the `<XPDINTEGHOME>/config` directory
- A simple text file, `myFile2.txt`, that contains sample truck status information and that can be copied into the text message body of the MQTT client application
- A messaging back-end system (for example, WebSphere MQ server) with the following settings:
  - IP Address/Hostname: `myBroker`
  - Queue manager: `TestQM`
  - Port: `14100`
  - Channel: `TestChannel`
- The following back-end queues for the Lotus Expeditor integrator run time
  - XPDintegResInQ100
  - XPDintegDeadletterQ100
  - XPDintegSyncQ100
  - Optionally: XPDintegReqOutQ100, XPDintegCtrlOutQ100, XPDintegLogQ100, XPDintegEventQ100

Configuration steps

Complete the following to configure this example.

1. Configure the SampleSendStateInfoJmsAdapter JMS adapter in the `XPDinteg.xml` configuration file as shown in Example 4-13.

Example 4-13 Step 1 for receiving log messages

```
<adapter type="XPDINTEGJMSDESTINATIONADAPTER"
name="SampleSendStateInfoJmsAdapter">
<listener>
<topic>com/ibm/integrator/flowtriggerevent/MessageForward/MESSAGE/SampleSendSta
teInfoJmsAdapter</topic>
</listener>
<configuration>
<param name="JndiConnectionFactoryKey" value="jms/XPDintegConnectionFactory"/>
<param name="JndiDestinationKey" value="jms/XPDintegSendStateInfoTopic"/>
<param name="DestinationName" value="jms/XPDintegResOutQ"/>
<param name="TruckNo" value="100"/>
<param name="JndiDeadLetterKey" value="jms/XPDintegServerDeadletterQ"/>
<param name="ValidateLocationId" value="OFF"/>
</configuration>
</adapter>
```

2. Configure the outgoing queue for delivering messages to the back-end system and provide the publish/subscribe topic under which the message carrying the log file will be published in the `XPDinteg.xml` file, as shown in Example 4-14.

Example 4-14 Step 2 for JMSDESTINATIONADAPTER configuration for receiving status messages

```
<queues> ...
<queue purpose="ResultOutQueue">
<queue-name>XPDintegResOutQ</queue-name>
<jndi-key>jms/XPDintegResOutQ</jndi-key>
</queue>
...
</queues>
```

```
<topics> ...
<topic purpose="SendStateInfoTopic">
<topic-name>XPDintegSendStateInfoTopic</topic-name>
<jndi-key>jms/XPDintegSendStateInfoTopic</jndi-key>
</topic>
```

3. Create the ACS flow for picking up the message and forwarding it to local `XPDintegResOutQ`. Locate examples of ACS flow definition files under `<XPDINTEGHOME>/samples/flowdefs`). Make sure a unique process name and the EventAdmin Service trigger topic of the configured SampleSendStateInfoJmsAdapter is used. Create a new flow file, `SendStateInfoMessageForward.flow`, in directory `<XPDINTEGHOME>/flowdefs` with the content shown in Example 4-15.

Example 4-15 ACS flow definition for forwarding payload

```
<?xml version="1.0" encoding="UTF-8"?>
<Process Name="SendStateInfoMessageForward\Process"
Trigger="com/ibm/integrator/flowtriggerevent/MessageForward/MESSAGE/SampleSendS
tateInfoJmsAdapter">
<XPDintegActivity
Name="SendStateInfoMessageForward\ReadResource"
ActivityName="XPDINTEGMESSAGEREAD"
PropertyKey="DATAREFERENCE"
/>
<XPDintegActivity
Name="SendStateInfoMessageForward\MessageWrite"
ActivityName="XPDINTEGMESSAGEWRITE"
JndiConnectionFactoryKey="jms/XPDintegConnectionFactory"
/>
</Process>
```

4. Configure the micro broker bridge connection to the back-end queues (find example configurations for back-end connectivity in the `<XPDINTEGHOME>/samples/config/SampleXPDinteg.xml` file). Make sure that all local and remote queues are defined and exist. The bridge flow entries must also point to existing queues.
5. Configure the WebSphere MQ server connectivity in 4.1: Objects for Back-end QUEUE CONFIG of the `XPDinteg.xml` configuration file.

   Example 4-16 shows the MQ server back-end connection configuration for the remote back-end `XPDintegResInQ100` in the `XPDinteg.xml` file.

Example 4-16 back-end connection configuration for the MQ server

```
<connection-factory name="ConnectionFactory1" type="MQ"
jndiKey="jms/XPDinteg-MQConnectionFactory">
<params>
<param name="hostname" value="myBroker"/>
<param name="port" value="14100"/>
<param name="queue-manager" value="TestQM"/>
<param name="clientid" value="XPDinteg100"/>
<param name="channel" value="TestChannel"/>
...
```

6. Configure the remote queue or queues of the WebSphere MQ server.

   Example 4-17 shows the Declaration of remote queues on the MQ server back end to which local queues can be bridged to in the `XPDinteg.xml` file.

   Example 4-17 Remote queues for bridging

```
..
<queue purpose="ServerSyncQ" type="MQ" jndiKey="jms/MQSyncQ">
<params>
<param name="queue-name" value="XPDintegSyncQ100"/>
</params>
</queue>
<queue purpose="ServerDeadletterQ" type="MQ" jndiKey="jms/MQServerDeadletterQ">
<params>
<param name="queue-name" value="XPDintegDeadletterQ100"/>
</params>
</queue>
<queue purpose="ServerResultInQueue" type="MQ" jndiKey="jms/MQServerResInQ">
<params>
<param name="queue-name" value="XPDintegResInQ100"/>
</params>
</queue>
...
```

7. Configure the queue bindings in the bridge (local queue to and from a remote queue) in 4.2: BRIDGE CONFIG of the `XPDinteg.xml` file. Make sure that the local and remote queues are declared in `XPDinteg.xml` and do exist.

   Example 4-18 shows the definition of micro broker bridge connections in `XPDinteg.xml` (for example, XPDintegResOutQ and XPDintegResInQ100).

   Example 4-18 Definition of micro broker bridge connections in XPDinteg.xml

```
<pipes>
<!-- Each pipe is exposed in the configuration. By using JNDI for the remote
endpoints we are able to achieve the requirement of different endpoints for
different queues all with configuration. -->
<pipe name="Back-end-pipe1">
<!-- As configured and referenced in the ConnectionFactory previously. -->
<jndi-connection connection-factory-key="jms/XPDinteg-MQConnectionFactory"
sync-queue-key="jms/MQSyncQ" dead-letter-queue-key="jms/MQServerDeadletterQ"/>
...
<flow name="resOutQFlow" direction="OUTBOUND">
<source name="ubQ" type="QUEUE">
<value>XPDintegResOutQ</value>
</source>
<target name="mqQ" type="JNDI" value="jms/MQServerResInQ"></target>
</flow>
<flow name="serverDeadetterQFlow" direction="OUTBOUND">
<source name="ubQ" type="QUEUE">
<value>XPDintegServerDeadletterQ</value>
</source>
<target name="mqQ" type="JNDI" value="jms/MQServerDeadletterQ"></target>
</flow>
...
```

Deploying and activating the configuration

To deploy and activate the configuration, follow these steps:

1. Stop the Lotus Expeditor integrator run time (click **Close** or **Exit** in the OSGi console).
2. Reset the Lotus Expeditor integrator run time (run `<XPDINTEGHOME>resetscript/XPDintegReset.bat` or `.sh`).
3. Start the Lotus Expeditor integrator run time (run `<XPDINTEGHOME>/XPDintegStart.bat` or `.sh`).
4. Wait until the Lotus Expeditor integrator run time is available (check this by pointing your browser to the admin GUI). You can use a local host (`http://localhost:8777/ui`). Under Linux, the run time starts in the background the first time it is started after a reset. Check for the expeditor process and end/kill it. Rerun `XPDintegStart.sh.`

MQTT client application and Publisher

Create the MQTT client application (Publisher) that sends the fictive truck status message to topic `XPDintegSendStateInfoTopic`. Example 4-19 shows the syntax for sending status messages to topic.

Example 4-19 Sending status messages to topic

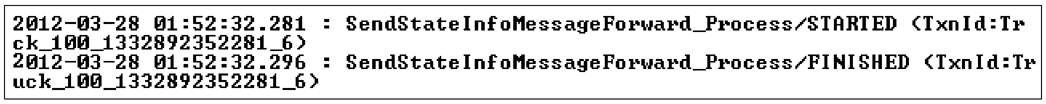
```
public static final String pubTopicString = System.getProperty("pubTopicString",
"XPDintegSendStateInfoTopic");
```

Run the example

To run the example, follow these steps:

1. Start Lotus Expeditor integrator (if not done yet).
2. Switch on ACS flow logging by entering `toggleFlowLog`.
3. Use your MQTT client application to publish a text message with some of the truck status sample text payload (for example, content from `myFile2.txt`).
4. The message is retrieved and its payload is forwarded to XPDintegResOutQ which is bridged to back-end queue XPDintegResInQ100 (check the content of the XPDintegResInQ100).
5. The execution of your ACS flow `SendStateInfoMessageForwardProcess` is displayed in the OSGi console.
6. You can repeat sending messages. The messages are received at the back end in the XPDintegResInQ.

Figure 4-12 shows the successful execution of this process.



```
2012-03-28 01:52:32.281 : SendStateInfoMessageForward_Process/STARTED (TxnId:Tr
ck_100_1332892352281_6)
2012-03-28 01:52:32.296 : SendStateInfoMessageForward_Process/FINISHED (TxnId:Tr
uck_100_1332892352281_6)
```

Figure 4-12 Lotus Expeditor integrator OSGi Console output during successful execution of SendStateInfoMessageForward.flow

Note: These scenarios can be secured by using end-to-end messaging security based on TLS/SSL.

5

Typical topologies and patterns of use

The network topology is one part of the overall physical architecture of a system. It describes the placement of the various components in different physical domains and the interaction between those.

In this chapter we provide an overview of the typical network topologies for MQTT-based messaging. We discuss the common layers and define the components within each layer, and describe two common topology patterns that developers can follow based on the type of server device to be used.

This chapter contains the following sections:

- Typical topology
- Client to enterprise server
- Client to edge server to enterprise server

5.1 Typical topology

The network topology of a machine-to-machine (M2M) system varies depending on the type and number of devices being connected.The following lists important factors that influence this decision:

- Some remote devices are not allowed to embed the MQTT client, even if delivered by means of a client library.
- Server performance will drop if too many remote devices are connected to it directly.
- Certain devices can require private or specific protocols, necessitating the use of an edge gateway to act as a proxy.
- Network outage might influence the SLAs that need to be fulfilled.

To handle these challenges, a variety of different topologies can be used. However, to speed design and implementation, the selected topology should build upon standard connectivity patterns and components, to the extent possible.

The basic topology patterns that are most relevant to this discussion of WebSphere MQ and the MQTT protocol are:

- The client connects to the server directly.
- The client connects to the server through an edge gateway.

The IBM MQTT connectivity solution works well in these topologies. It provides a lightweight messaging option that can be critical if the network between the devices and the central system components is constrained due to latency, limited bandwidth or high incremental costs. And because the MQTT client libraries are small and require limited processing capacity on the devices which host them, the solution can work even with constrained devices.

5.1.1 Pattern 1: Client to enterprise server

For some systems, it is best to connect the remote devices to the Enterprise Integration Layer directly, as shown in Figure 5-1, also called the basic connectivity pattern. This approach works when third-party software (such as the MQTT client) can be embedded in the devices, and the number of devices is relatively small, with only hundreds or perhaps a few thousand devices connected.

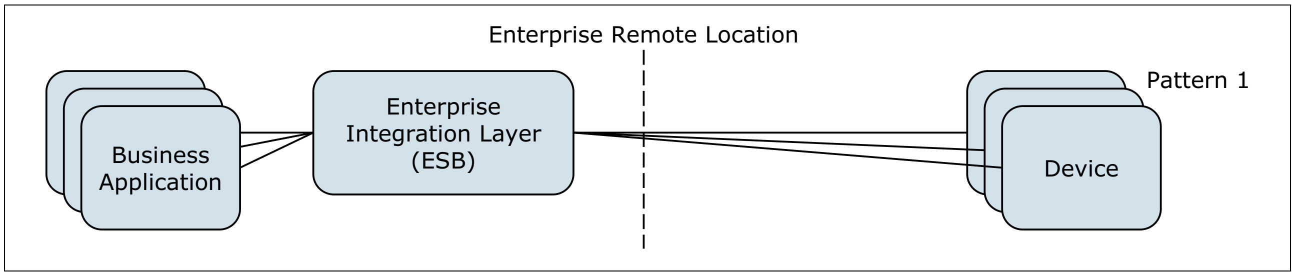


Figure 5-1 Client to enterprise server

Table 5-1 shows the components of the basic connectivity pattern.

Table 5-1 Components of a basic connectivity pattern

Component	Purpose
Business application(s)	An application that wishes to send and receive messages to or from a device. It connects to the server in the Enterprise Integration Layer using a supported transport protocol, such as JMS, MQ, or MQTT.
Enterprise Integration Layer	The server here acts as a central concentrator that remote devices connect to. The following list reflects other activities that the server supports: ► Acts as an enterprise service bus for business applications message exchange. ► Capable of storing events and notifications for later delivery, if connectivity was not available when first attempted. ► Capable of enforcing connection security by means of TSL/SSL to authenticate remote devices. ► Capable of mediation for filtering and aggregating messages, such as to reduce the volume of messages traveling over the network. ► Capable of event correlation, such as merging or mapping an event or alarm between the devices and business applications
Devices	Remote sensors or applications that collect information or carry out tasks and which must communicate to and from the central system.

5.1.2 Pattern 2: Client to edge server to enterprise server

Pattern 2 connects the devices to the Enterprise Integration Layer through an Edge Gateway. The Edge Gateway acts as a hub or concentrator for devices to connect to, and transports messages between the devices and the Enterprise Integration Layer, illustrated in Figure 5-2. This makes Pattern 2 ideal for systems with an unreliable or insecure network, or where the number of devices can cause a performance slowdown if they were connected directly to the server in the Enterprise Integration Layer.

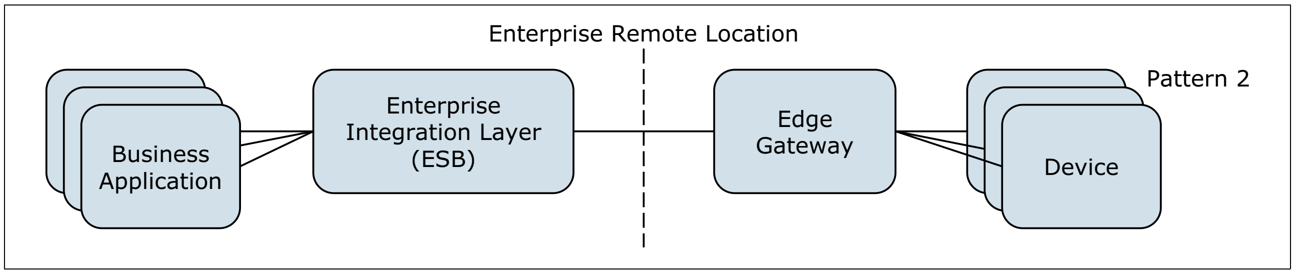

Enterprise Remote Location
Pattern 2
Business Application
Enterprise Integration Layer (ESB)
Edge Gateway
Device

Figure 5-2 Client to edge server to enterprise server

Table 5-2 lists the components of a client-to-edge-server-to-enterprise server topology.

Table 5-2 Components of a client-to-edge-server-to-enterprise server topology

Component	Purpose
Business applications	See Table 5-1 on page 161 for Pattern 1.
Enterprise Integration Layer	See Table 5-1 on page 161 for Pattern 1.
Edge gateway	Serves as a hub or concentrator for devices to connect; supports the following functions: ► Acts as a gateway to and from the Enterprise Integration Layer using a single connection. ► Allows devices to communicate with each other without going through the Enterprise Integration Layer. ► Capable of storing events and notifications for later delivery, if connectivity was not available when first attempted. ► Capable of applying application logic, such as filtering or aggregation of messages. ► Capable of applying application logic, such as event correlation, which is used to merge or map the event or alarm between device and enterprise integration layer.
Devices	See Table 5-1 on page 161 for Pattern 1.

5.1.3 Pattern 2A: Client to edge server (proxy) to enterprise server

Some organizations prohibit embedding third-party software, such as a messaging client, in remote devices. Or devices might use a private protocol or unusual communications format that the server in the Enterprise Integration Layer cannot understand. And the devices are distributed one by one. In such cases, the Edge Gateway can be used as a 1-1 proxy (Pattern 2A) shown in Figure 5-3.

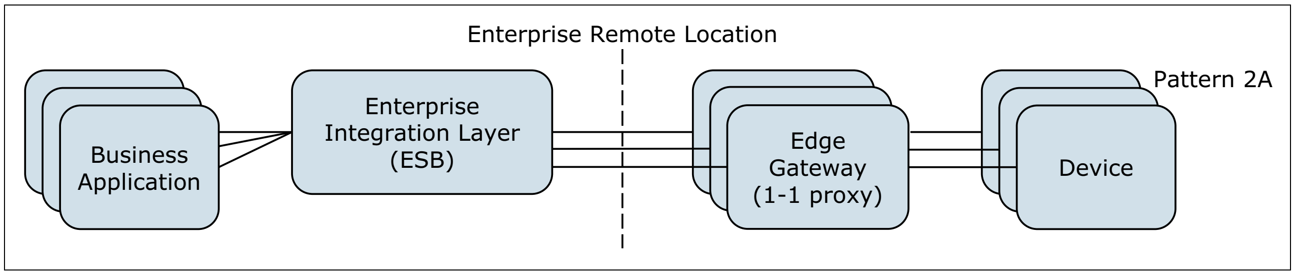


Figure 5-3 Client to edge server (proxy) to enterprise server

By acting as a proxy to and from the Enterprise Integration Layer, the Edge Gateway in Pattern 2A allows devices to communicate using their preferred private or specific protocol, such as ANO.

5.1.4 Pattern 2B: Client to edge server (with adapter) to enterprise server

Pattern 2B is similar to Pattern 2, except that it includes an adapter application in the Edge Gateway to translate messages that arrive from the remote device or devices using a protocol that cannot be understood. This pattern with adapter application is shown in Figure 5-4.

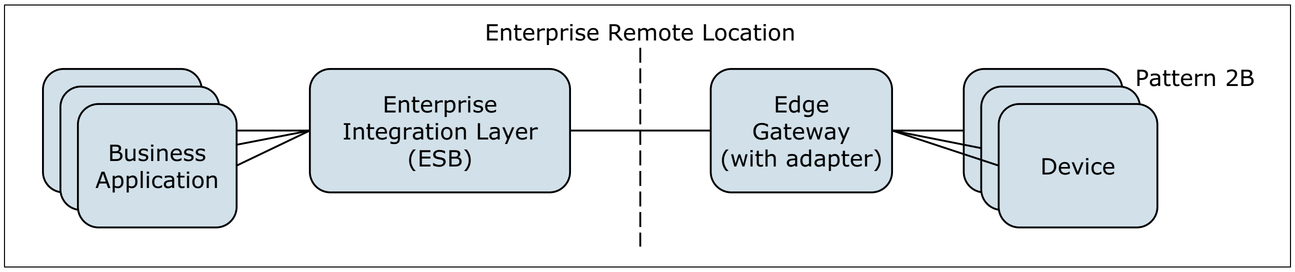

Enterprise Remote Location
Pattern 2B
Business Application
Enterprise Integration Layer (ESB)
Edge Gateway (with adapter)
Device

Figure 5-4 Client to edge server (with adapter) to enterprise server

5.1.5 Pattern 2 extended: Multiple edge servers connected to each other

This extended pattern is quite similar to the basic Pattern 2 with the option that some Edge Gateways are directly connected with each other; see Figure 5-5. This might be appropriate when remote locations need to communicate but without any data transmission to enterprise systems and vice versa. It will reduce the data traffic to a necessary amount but requires additional monitoring of the Edge Gateway instances.

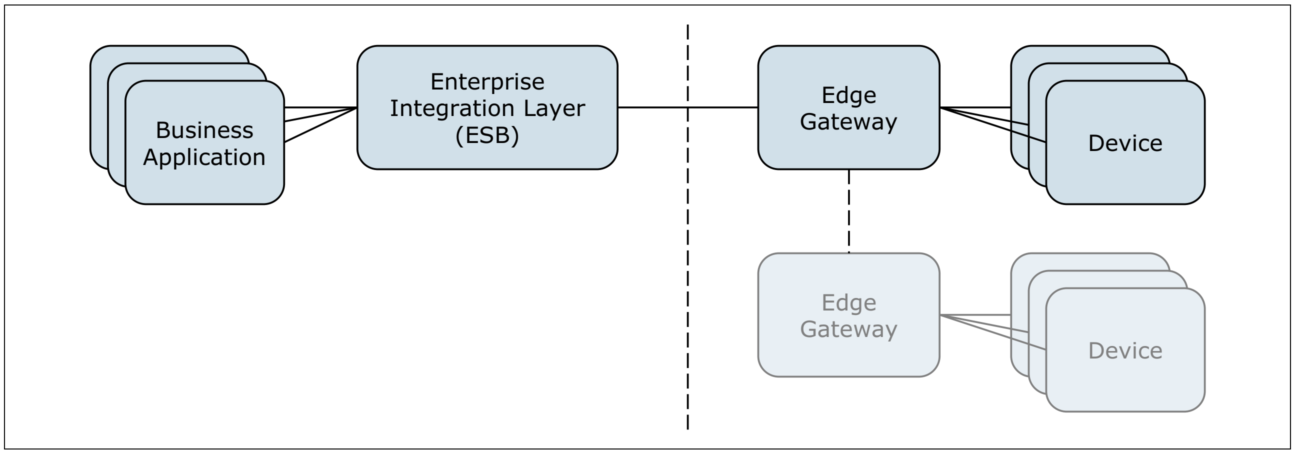


Figure 5-5 Multiple edge servers connected to each other

5.1.6 Selection criteria for the main pattern

Table 5-3 gives an overview about the typical requirements and the selection criteria for Pattern 1 or pattern 2.

Table 5-3 Criteria for selecting a messaging topology pattern

Decision criteria	**Pattern 1: Client to enterprise server**	**Pattern 2: Client to edge server to enterprise server**
Use case and topology related		
Data processing at remote location	None to less; only data transmission between remote locations and enterprise	Moderate to extended (for example filtering, data exchange between devices at one physical location)
Number of devices per remote location	One to less	Multiple and different devices
Device related		
Available resources (for example, processor or memory)	Moderate resources on the devices; able to handle some additional workload	None to less resources available on the devices for extensions
Extensibility (for example, API or connectivity options)	API or connectivity options available on devices	Almost closed device implementations with nearly zero access to functions and data; proprietary device interfaces
Network related		
Reliability of connection	Almost reliable online connectivity or none to less need of data buffering at remote locations	Unreliable connectivity with a moderate to huge amount of data that needs to be transferred
Security of lower protocols	Lower protocols secured with no need of additional encryption on-top	Unsecured connectivity with the need of additional security layer on-top

5.2 Client to enterprise server

In this section, we discuss the basic pattern of connecting devices to the enterprise server directly.

5.2.1 The topology

The client to enterprise server relationship is an example of basic connectivity topology, as illustrated in Figure 5-6.

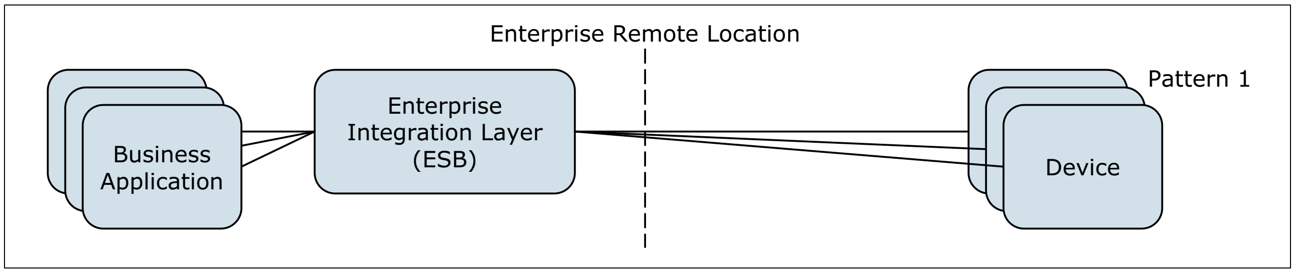

Enterprise Remote Location
Pattern 1
Business Application
Enterprise Integration Layer (ESB)
Device

Figure 5-6 Client to enterprise server (basic connectivity topology)

The main characteristics for selecting the basic pattern are described based on the decision table in Table 5-3 on page 164. Table 5-4 includes several areas of criteria for selecting the basic connectivity pattern. This table divides the information based on use case, device, and network relation.

Table 5-4 Criteria for selecting the basic connectivity pattern

Decision criteria	**Pattern 1: Client to enterprise server**
Use case and topology related	
Data processing at remote location	None to less; only data transmission between remote locations and enterprise. Remote locations with 1:1 data transfer between the locations themselves and the enterprise can be connected directly.
Number of devices per remote location	One to less A small number of devices at a remote location allow direct connectivity to them.
Device related	
Available resources (for example processor, memory)	Moderate resources on the devices; able to handle some additional workload. If extended capabilities (data preprocessing or filtering, protocol extension) are required and the devices have enough resources, they can be extended directly. This makes sense when a lot of equal devices should be connected in the same manner.

Decision criteria	Pattern 1: Client to enterprise server
Extensibility (for example, API or connectivity options)	API or other connectivity options available on devices. If extended capabilities (data preprocessing or filtering, protocol extension) are required and the devices provide an API or extended connectivity options, they can be extended directly. This makes even sense as well when a lot of equal devices should be connected in the same manner.
Network related	
Reliability of connection	Almost reliable online connectivity or none to less need of data buffering at remote locations An almost online connectivity requires none to less buffering on the devices; this reduces the overhead of data processing dramatically and allows direct connectivity.
Security of lower protocols	Lower protocols secured with no need of additional encryption on-top If the network transport between the remote locations and the enterprise is secured, there is no need for additional security layer on top. Uses almost online connectivity, which simplifies the connectivity and allows direct coupling.

5.2.2 The architecture

The main architecture looks quite similar to the topology pattern. The solution approach is based on the extension of the simple ESB Pattern. Figure 5-7 shows an enterprise service bus that consists of a Backbone Service Bus and the directly connected devices.

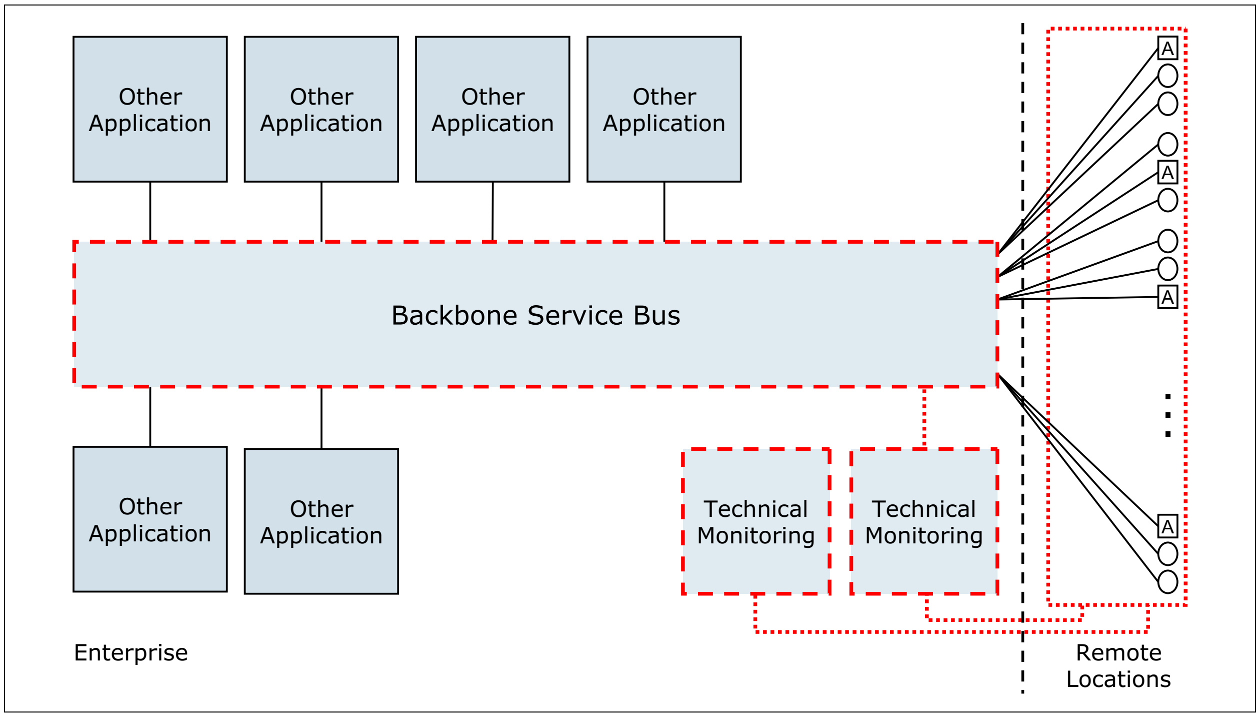


Figure 5-7 Client to enterprise server architecture

When designing a system architecture using Pattern 1, consideration must be given to the selection of the messaging application and the communication protocol. The server software will act as broker for messages to and from the devices, and the ESB provides the flexible connectivity infrastructure for integrating applications and services. The communication protocol should be as lightweight as possible due to a large number of connections. The MQTT protocol was specifically designed for communicating with remote devices across constrained networks, so it is an ideal choice for messages between the devices and the server.

IBM WebSphere MQ Telemetry can act as an MQTT server with high performance and flexible scaling capability which can handle current connections to multiple devices. Furthermore, the MQTT client libraries provided by IBM WebSphere MQ Telemetry can be used in constrained devices that have little storage and computing capability. In addition, with MQTT publish/subscribe client implementations in Eclipse Project Paho donated by IBM, users can save effort when designing and implementing systems. As a full-blown powerful ESB product, IBM WebSphere Message Broker provides the flexible connectivity infrastructure for integrating business applications and services, using MQTT natively.

Figure 5-8 on page 168 shows one possible implementation of the architecture previously described. The example shows IBM WebSphere MQ Telemetry as a central messaging layer between the devices and the enterprise application layer.

MQTT protocol: IBM WebSphere Message Broker can also implement the MQTT protocol natively. You need to decide whether the devices are connected to IBM WebSphere MQ or IBM WebSphere Message Broker, depending on whether the data that is coming from the devices can be processed directly by other applications or must be modified (transformed or routed).

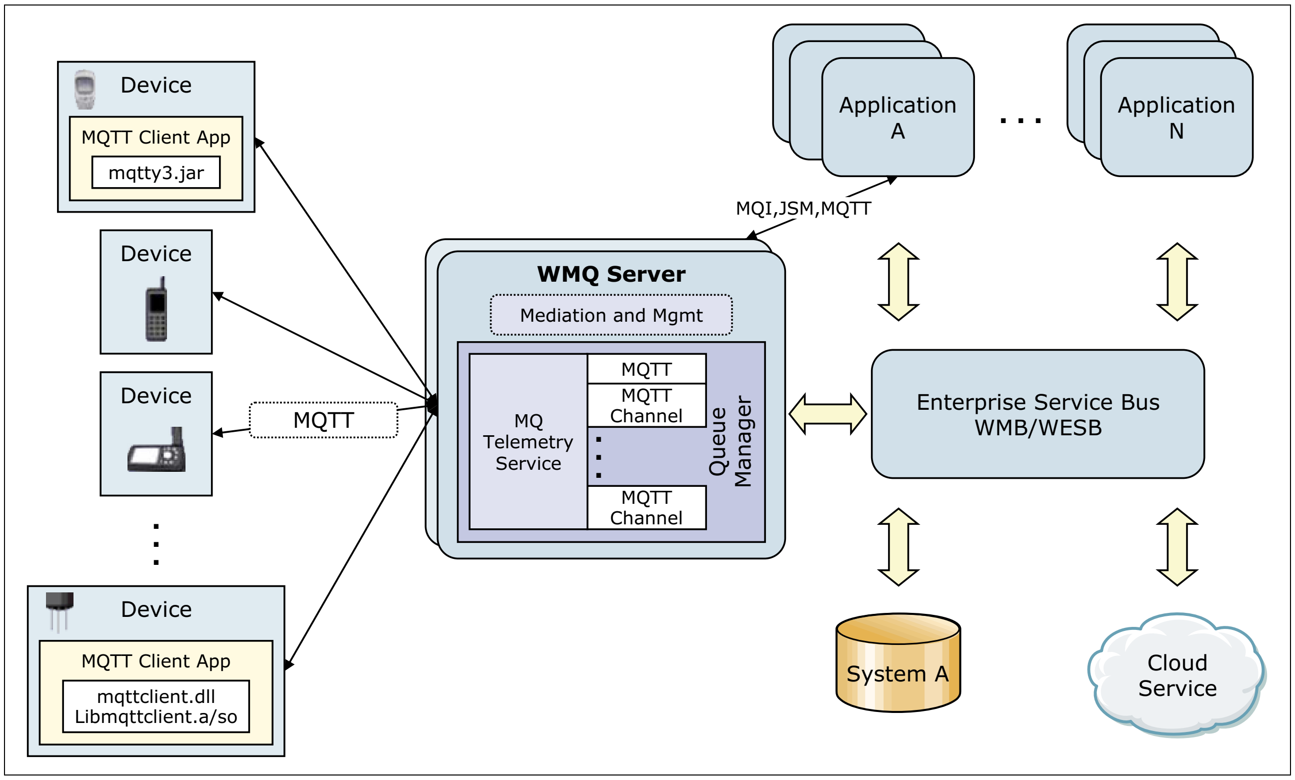


Figure 5-8 Architecture of the client to enterprise server topology

5.2.3 Sample using the client to enterprise server connectivity pattern

Figure 5-9 on page 169 provides an example of client to enterprise server architecture in a telematics system that is used for vehicle monitoring. The company needs to monitor information such as running speed, tire pressure, interior temperature, and other information. Thus, multiple sensors are attached to each vehicle. The sensor readings are to be transmitted to company headquarters on a predetermined schedule, but if an updated report is not received on time, the central system needs to send a message instructing the devices to submit the data that has been recorded.

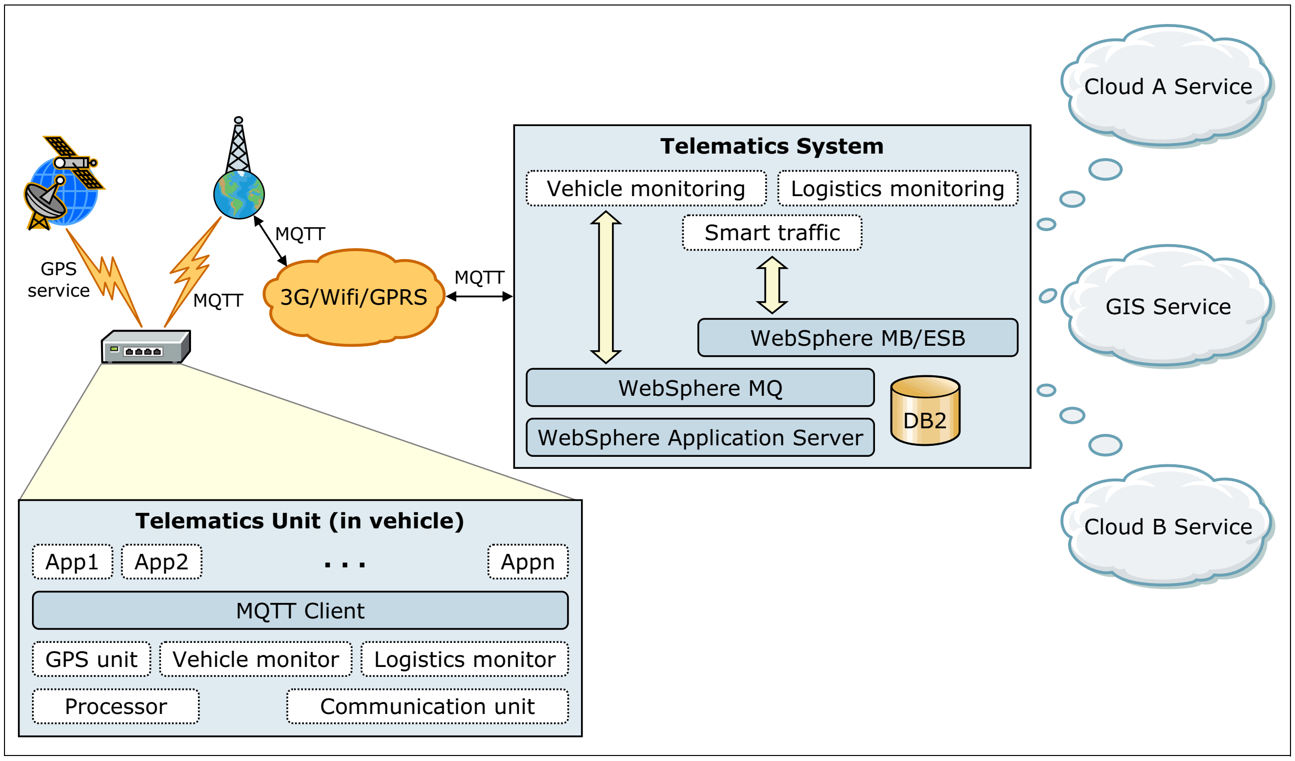


Figure 5-9 Architecture of a sample telematics messaging system in logistics

The client to enterprise server pattern works well for telematics, particularly in the field of logistics. The following are factors that influence this decision:

- There is direct data transfer between trucks and the enterprise without additional data preprocessing or filtering.
- The number of vehicles and sensors is unlikely to go higher than 100,000 (generally considered the limit for this pattern).
- The sensors and other remote devices are generally newer and, therefore, have the ability to embed and use third-party software, such as a messaging application.
- If no network is available, the devices are able to buffer the data until the next data transfer.
- The data that is transferred is not security critical for a single truck because the data is mostly truck healthy information or position data.

The following examples are specific to this case and its implementation:

- Trucks: MQTT client library: IBM WebSphere MQ Telemetry Software Development Toolkit, which contains IBM WebSphere MQ Client C and Java libraries
  - MQTT client library must be compatible with the device platform.
  - The persistence capability avoids loss of data when MQTT client crashes or powers off.
  - The connection between the device and enterprise server can be configured securely, such as SSL (Java only).
  - The devices that want to connect to the enterprise server can be authorized, such as JAAS (Java only).

- Enterprise: MQTT server/broker: IBM WebSphere MQ Telemetry

  Consider the following important factors:

  – WebSphere MQ Telemetry has good performance during the high number of concurrent connections from devices.

  – The applications can be developed and connect to the MQTT server and broker easily by using different transports, such as MQTT, MQI, or JMS.

  – Strong security management allows authorizing device registration and securing the connection.

- Enterprise: ESB: IBM WebSphere Message Broker and IBM WebSphere ESB

  – Strong capability of integration with other systems
  – Loose coupling model for updating, refactoring, and integrating

5.3 Client to edge server to enterprise server

Here we discuss the second topology pattern, client to edge server to enterprise server. Two examples of this pattern also will be discussed, one from retail and another from the utility industry.

5.3.1 The topology

The topology for this pattern is shown in Figure 5-10.

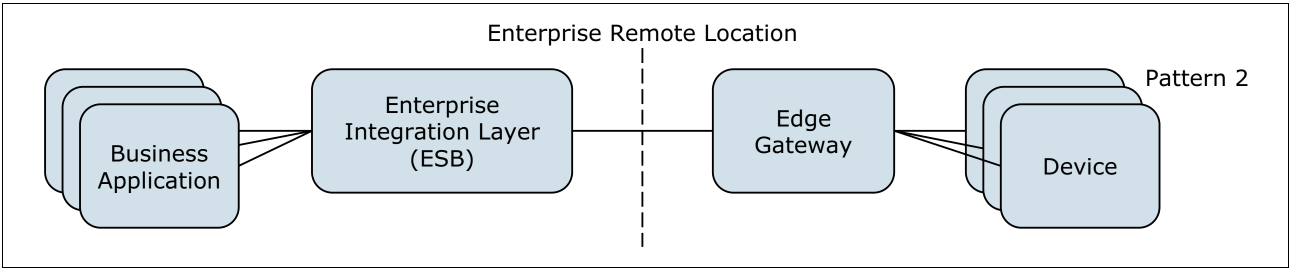

Enterprise Remote Location
Pattern 2
Business Application
Enterprise Integration Layer (ESB)
Edge Gateway
Device

Figure 5-10 Pattern 2: Client to edge server to enterprise server topology

Numerous industry requirements or system constraints might lead a developer to use the client to edge server to enterprise server topology, but the main characteristics of systems that can benefit from this pattern are listed in Table 5-5.

Table 5-5 Characteristics of systems benefited by client to edge server to enterprise server topology

Decision criteria	**Pattern 2: Client to edge server to enterprise server**
Use case and topology related	
Data processing at remote location	Moderate to extended (for example, filtering or data exchange between devices at one physical location Remote locations with data exchange, preprocessing and filtering require additional mediation capabilities that lead to a dedicated integration endpoint: Edge Gateway. Examples are Retail Stores or Buildings with Smart Meter Devices.

Decision criteria	Pattern 2: Client to edge server to enterprise server
Number of devices per remote location	Multiple and different devices Use an Edge Gateway to reduce the number of connections and make the data preprocessing more transparent at a single integration endpoint.
Device related	
Available resources (for example, processor or memory)	None to less resources available on the devices for extensions At remote locations a lot of devices do have limited resources and there is no room for extended integration functionality on these devices. An Edge Gateway can implement these extensions and reduce the impact on the devices.
Extensibility (for example, API or connectivity options)	Almost closed device implementations with nearly zero access to functions and data; proprietary device interfaces Often the number of device types is limited to proprietary protocols and connectivity options; there is a mediation required in between the devices and the enterprise layer. These mediations can be implemented in a configurable and flexible manner on an Edge Gateway with extended capabilities for remote monitoring and remote configurability and extensibility.
Network related	
Reliability of connection	Unreliable connectivity with a moderate to large amount of data that needs to be transferred Unsecured and unreliable networks require a data buffering at remote locations for further delivery. This can be done in an effective manner by an Edge Gateway. All data buffering and preprocessing can be provided in a transactional context that no data will be lost or compromised.
Security of lower protocols	Unsecured connectivity with the need of additional security layer on-top If the device connectivity options are limited to unsecured protocols (for example, FTP or HTTP) and the connectivity between remote locations and the enterprise layer is not secured, and there is a need for additional security layer on top. This should be implemented a single time, at the Edge Gateway.

5.3.2 The architecture

The solution approach is based on the extension of the Federated ESB Pattern. Figure 5-11 on page 172 shows a Federated ESB that consists of a Backbone Service Bus and managed integration endpoints, called *Edge Gateways*. These endpoints help to extend the reach of an ESB to the edge of an enterprise's IT infrastructure in a reliable and controlled manner.

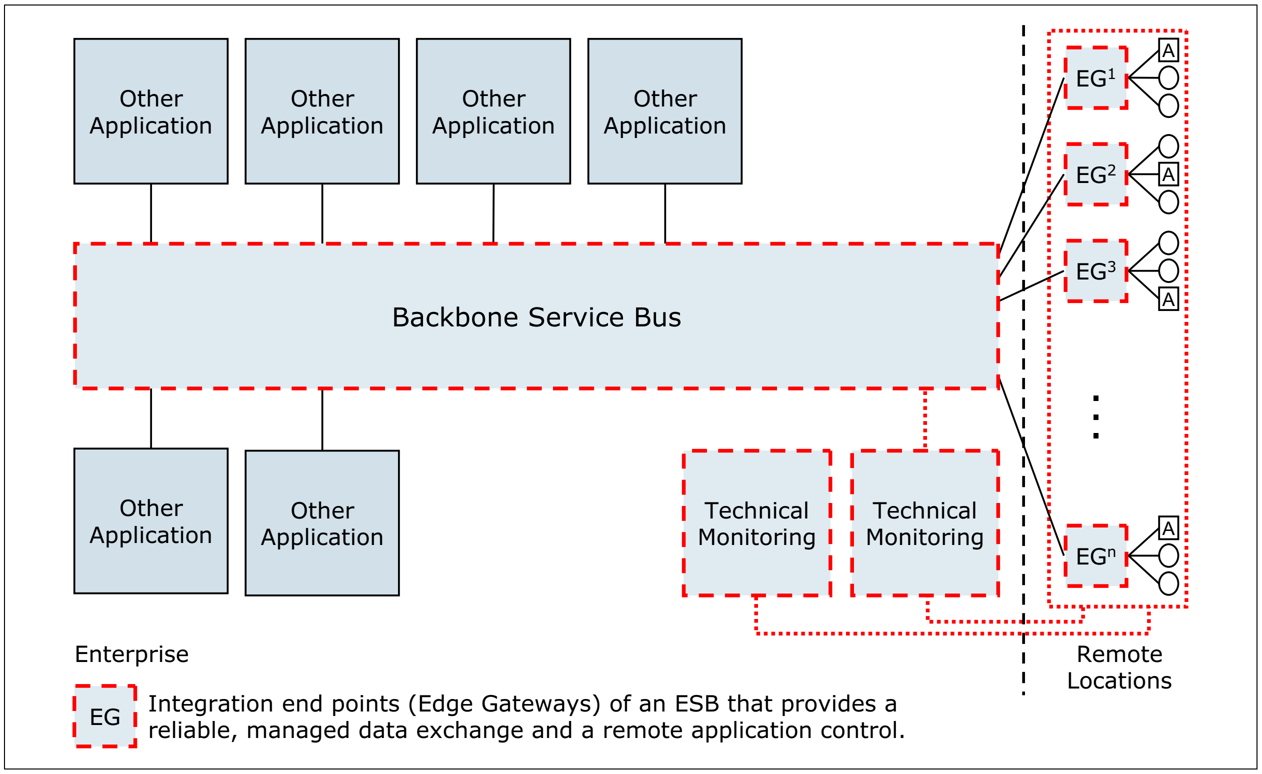


Figure 5-11 Client to edge server to enterprise server architecture

Today, many enterprises have already established a central enterprise integration facility and monitoring for technical reasons. Nevertheless, data exchange with branch offices is still handled batch-oriented with file transfer over unsecured and non-reliable connections in many cases. Changing these business processes from batch to reliable and near real-time processing requires a paradigm shift: The Federated ESB Pattern helps to extend the control of centrally managed integration processes to the edge.

One implementation of such an Edge Gateway that has been developed based on the requirements noted previously and especially for the retail industry is IBM Lotus Expeditor integrator, which provides platform and application management as well as access services such as the messaging and transaction services. These services allow for integration and control of the managed integration endpoint of an Enterprise Service Bus.

5.4 Using the client to edge server to enterprise server connectivity pattern

In this section we include samples from different industries of using the client to edge server to enterprise server connectivity pattern.

5.4.1 Retail industry

Retail companies face hard competition. The only way to further reduce costs is seen in streamlining business processes. Business process flexibility and automation requires state-of-the-art IT systems. Furthermore, expanding business to new locations and most of all moving into remote markets challenges existing IT systems. Reliable and secure data transmission between remote stores and back-end infrastructure is key to assure continuous business and availability.

Business analytics and optimization supports ad-hoc decisions and reactions to market and environment changes. Analytical investigations need most current data (for example, store revenue and transaction data). Subsequent actions must be processed instantly (for example, distribution of price updates to stores).

Existing classical time-triggered batch processing and data transport technologies, such as FTP or even file system shares, are not able to fulfill these requirements reliably, securely, and economically if network connections are slow and unreliable. Messaging technology addresses these drawbacks and adds possibilities of data-triggered transmission and transactional processing. Messaging is already recognized as the backbone data transport mechanism in service-oriented IT infrastructures.

Now, data transmission based on messaging technology is extended not only to the store but into the store. In order to achieve a seamless integration, it is recommended that all connected devices and appliances in the store also use messaging technology.

Figure 5-12 shows typical retail store integration scenarios. In terms of a centralized IT strategy of a retail enterprise the IBM Lotus Expeditor integrator can be implemented as a lightweight Edge Gateway to collect, distribute and, in some cases, preprocess the data. As the devices in retail stores do provide a large variety of protocols, it is important that an Edge Gateway supports these. IBM Lotus Expeditor is part of the IBM Retail Industry Framework as one option for retail store integration scenarios.

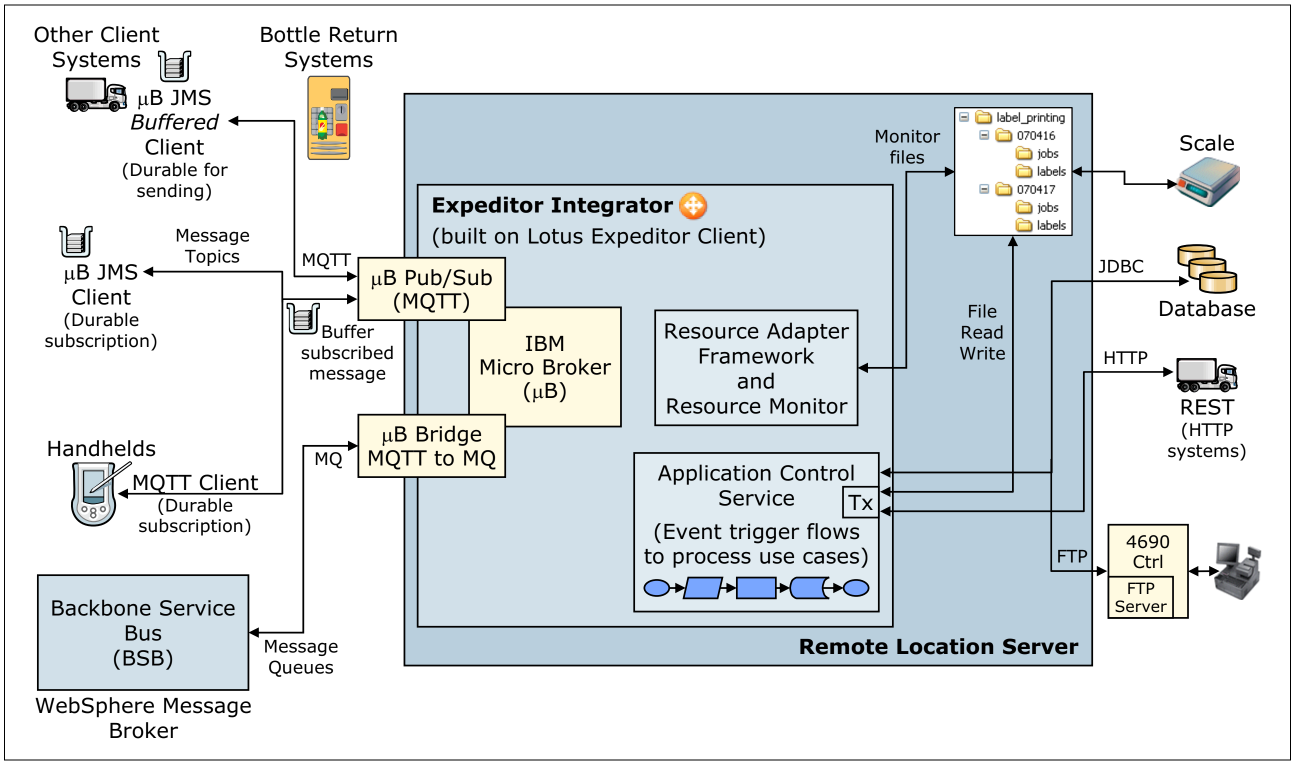


Figure 5-12 Retail integration example

A typical retail store integration scenario includes the following key components:

- Backbone Service Bus

  This component is the central messaging bus in the enterprise infrastructure. All retail stores are connected to this ESB. A typical implementation is IBM WebSphere Message Broker.

- Remote Location Server

  This server hosts all applications for back office processes within a retail store. Depending on store size this server might exist multiple times. It is the physical server on which the Edge Gateway runs.

- IBM Lotus Expeditor integrator (micro broker)

  Lotus Expeditor integrator as an implementation of the Edge Gateway provides a lightweight integration hub for remote locations, such as retail stores. It enables straightforward business and application integration of the enterprise central office with its retail stores in a reliable, flexible, and managed way. It can even be deployed at remote locations with limited hardware and software resources and with no skilled IT management available.

  Lotus Expeditor integrator interacts with the POS systems, label printers, and scales for forwarding price update data, label data, and scale data to the retail stores and to pick up transaction log and other data from the retail store. It supports the following protocols to talk with devices:

  – File system
  – (S)FTP
  – JDBC
  – HTTP
  – Other protocols through custom implementation based on an adapter API

One of the most important interfaces of IBM Lotus Expeditor integrator is the MQTT protocol-based connectivity to different types of messaging clients that can be implemented on devices:

- Micro broker JMS Buffered Client for buffered message-based connectivity (intended for durable sending, supports network outages)
- Micro broker JMS Client for simple message-based connectivity (intended for durable subscription)
- MQTT Client on non-Java platforms for simple message-based connectivity (intended for durable subscription)

The challenge today is to extend the reliable and secure data connectivity to the last mile between Edge Gateway and devices. Typically, retailers have started to roll out secure and reliable messaging for the following reasons:

- Provide data, such as price updates, scale and label printer data to the stores
- Collect data, such as transaction data, invoice data from the stores in a transactional context

In 4.5, “Integrate MQTT with IBM Lotus Expeditor integrator” on page 147, we describe in detail the message-based connectivity between the IBM Lotus Expeditor integrator and devices running the micro broker JMS Client.

5.4.2 Energy and utilities industry

The energy and utility industry is undergoing a big change due to the constantly increasing demand for energy at the same time as traditional energy generation resources are exhausting. The energy production and distribution must be more efficient and sustainable. The trend to *green* energy generation is noticeable.

One way of achieving the higher efficiency is better balancing of energy generation, storage and consumption. This can be done most efficiently as near to the physical location of generation, storage, and consumption as possible. The need for centralized communications is reduced and autonomous operations of increasingly smaller sections of the distribution grid are enabled.

New energy management applications drive the establishment of smart grids. Their efficiency heavily depends on the ability to capture fine-grained, real-time, energy-related sensor data, such as power consumption and power generation meter values, and control commands (for example, direct load-control commands for devices that support control functions).

The following basic architectural approaches exist for capturing data from sensors and actuators:

- Centralized data capturing and processing
- De-centralized data capturing with partly preprocessing de-centrally and centrally

In the following sections we explain the different methods to achieve data exchange with intelligent meters.

Centralized approach

The *centralized approach* is used for collection of (sensor/actuator) data that requires centralized processing. An example is the collection of energy consumption data for billing purposes. In addition, for better energy production planning, centralized captured data can be analyzed for consumption prediction.

Figure 5-13 on page 176 shows the centralized approach in which Integration tier 1 is centrally placed in the IT back-end system. High parallel processing power and high-bandwidth connections for further processing are easily made available and maintainable.

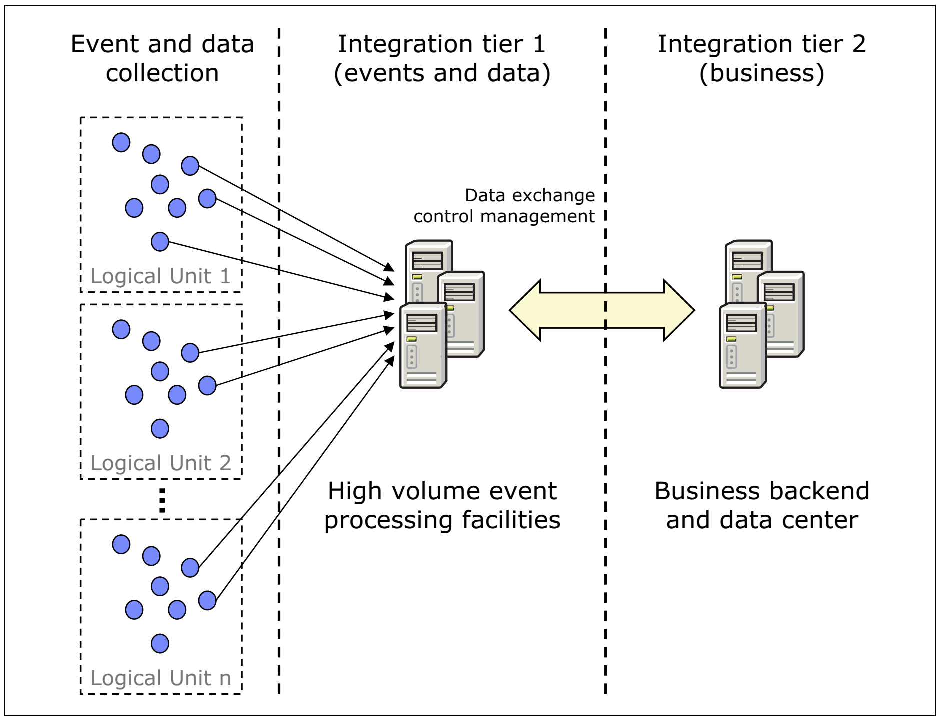

Event and data collection
Integration tier 1 (events and data)
Integration tier 2 (business)
Data exchange control management
Logical Unit 1
Logical Unit 2
Logical Unit n
High volume event processing facilities
Business backend and data center

Figure 5-13 Centralized architecture approach

Decentralized approach

The decentralized architectural approach is for use cases that rely on decentralized integration tiers (mini-server or integration hub), for example a smart grid server. If the captured data is required only for use cases that required further interaction with decentralized objects, instant processing close to these objects is more efficient. A typical example is a minicomputer or server that needs to control a device with sensors and actuators. The server collects the sensor data, processes it and interacts with the actuators to control the device based on the retrieved sensor data.

This approach includes the following typical requirements:

- Decentralized use cases are more efficiently executed near to the data source and close to the controlled objects (for example, leveling grid load by controlling photovoltaic plants as producers and hot water boilers as consumers).
- Constant back-end connectivity cannot be assured and is not desired or even not allowed.
- Data from home or premises needs to be transmitted anonymously, securely, and reliably (decentralized data transformation).
- Provide intermittently connected decentralized components that are intelligent enough to run decentralized use cases (decentralized transactional secure, reliable, remote manageable, and monitored execution unit).

Figure 5-14 on page 177 shows that Integration tier 1 is placed closer to the producers of the events. Only a subset of the collected data, maybe forwarded to the back-end system, depending on its relevance. The main objective for the data collection at this layer is preprocessing for further actions only relevant to the local environment (of tier 1).

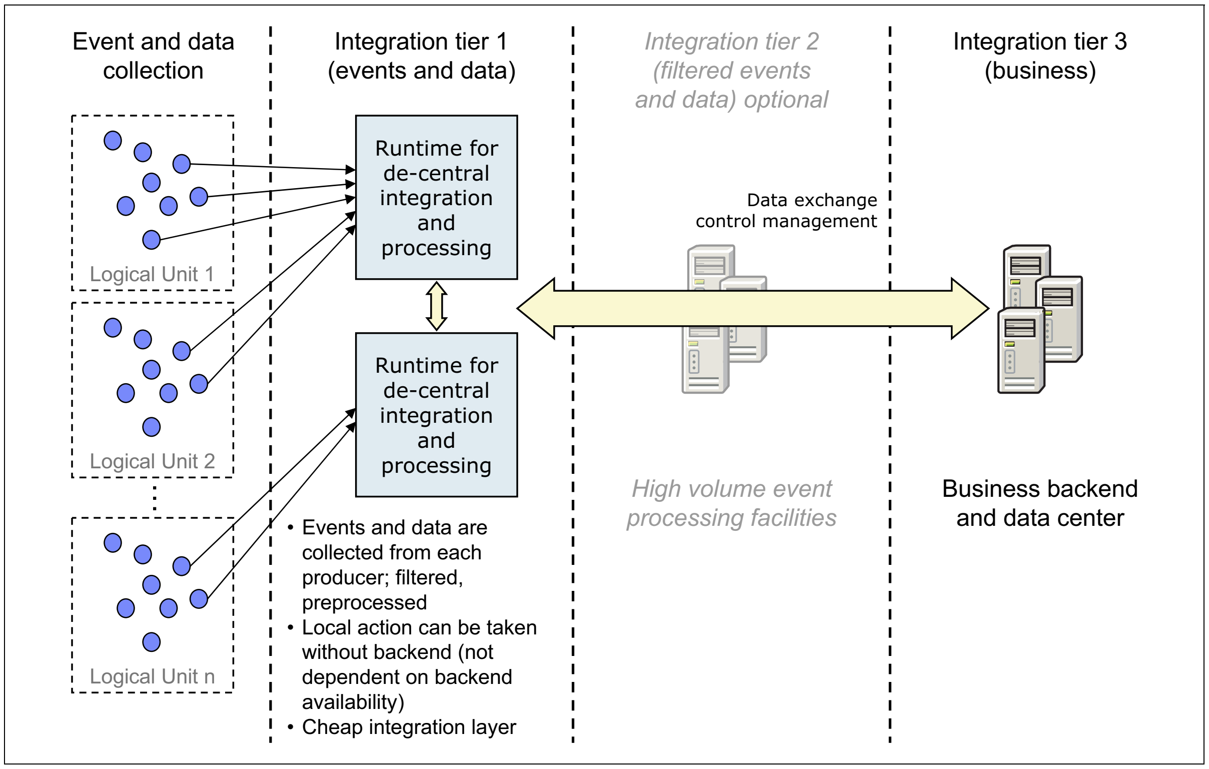
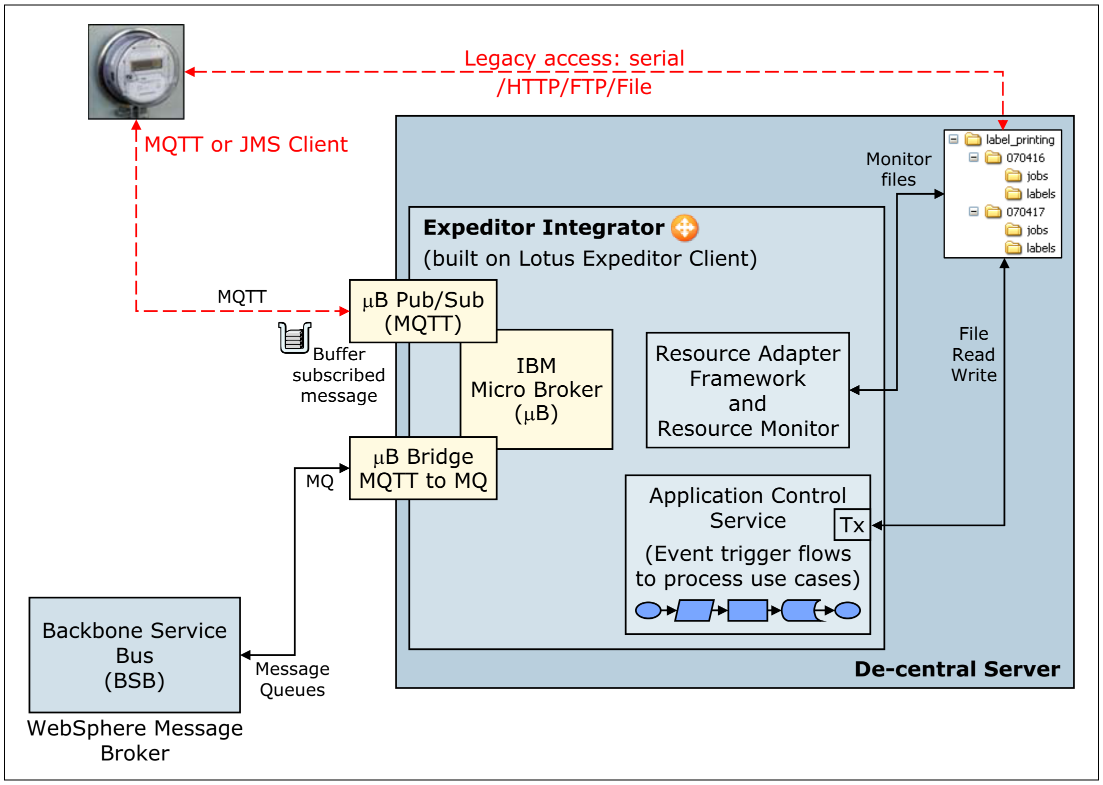

Event and data collection
Integration tier 1 (events and data)
Integration tier 2 (filtered events and data) optional
Integration tier 3 (business)
Logical Unit 1
Logical Unit 2
Logical Unit n
Runtime for de-central integration and processing
Runtime for de-central integration and processing
Data exchange control management
High volume event processing facilities
Business backend and data center
• Events and data are collected from each producer; filtered, preprocessed
• Local action can be taken without backend (not dependent on backend availability)
• Cheap integration layer

Figure 5-14 Decentralized architecture approach

5.4.3 Industry example with decentralized application logic

The decentralized approach can be used to collect data from sensor devices and to interact locally with connected actuators. The Edge Gateway acts as Integration tier 1 and MQTT is the messaging-based communication option for data exchange between the devices and appliances (logical units).

Figure 5-15 on page 178 contains the different integration options for smart meters, household and smart grid appliances, applications and services with IBM Lotus Expeditor integrator as the Edge Gateway. Currently many traditional meters are only accessible through serial communication interfaces. Meter Gateways bridge these meters into the TCP/IP world by using different protocols and standards (for example HTTP/SOAP). However, messaging becomes more popular as a communication technology because an integration into a messaging-based back-end services bus is more efficient. Protocol overhead is smaller and conversion between JMS messaging systems is simpler.



Legacy access: serial /HTTP/FTP/File
MQTT or JMS Client
Monitor files
label_printing
070416
jobs
labels
070417
jobs
labels
Expeditor Integrator
(built on Lotus Expeditor Client)
MQTT
µB Pub/Sub (MQTT)
Buffer subscribed message
IBM Micro Broker (µB)
Resource Adapter Framework and Resource Monitor
File Read Write
MQ
µB Bridge MQTT to MQ
Application Control Service
Tx
(Event trigger flows to process use cases)
Backbone Service Bus (BSB)
Message Queues
WebSphere Message Broker
De-central Server

Figure 5-15 Example for integration of meters with Edge Server

As already discussed for the retail industry, the micro broker clients can use Expeditor integrator (IBM micro broker as the MQTT provider) as the Edge Server (or Grid Server) for seamless messaging-based integration with the back-end system. This approach is reliable and can be end-to-end secured (for example, by using SSL or additional security features in IBM Lotus Expeditor integrator).

An Edge Server such as IBM Lotus Expeditor integrator provides a decentralized remotely manageable Java application run time based on open standards (OSGi) that can work without constant back-end connectivity. This allows not only for autonomous operation and execution of de-central use cases, for example control of accessible energy producers (private photovoltaic solar power plants) and consumers (hot water boilers, geo-thermal heating) for leveling power usage in low voltage grids, but also for preprocessing of data (for example, evaluate and filter meter readings, apply data privacy, and so forth).

The IBM Lotus Expeditor integrator Resource Adapters and the Resource Monitor Service are responsible for monitoring, reading and writing data to the different target resources. This also includes the publish/subscribe engine and queues in IBM micro broker to which the micro broker clients are connected. The Resource Monitor triggers configured ACS flows that perform the data exchange (and possible data processing). The Monitoring Service creates ACS flow status messages for transactional monitoring by the back-end service bus (Business Events). The Monitoring Service also creates log events that can be forwarded to operational monitoring facilities in the back-end system.

The selection for an optimal solution heavily depends on non-functional requirements. The following list notes most of those requirements:

- Number of connected meters; number of possible parallel connections
- Meter reading interval and absolute reading time; accepted delay between meter data read and meter data storage at the back end
- Meter data push (initiated by meter) versus meter data poll (back-end initiated reading)
- Security (encryption, authentication, authorization)
- Availability (central versus de-central approach)
- Assured delivery of meter data

If the collected data is required for local control interactions (for example, controlling energy consumers and producers) decentralized autonomous grid servers are preferred (independent of centralized back-end systems). Figure 5-16 shows the architecture for the utility scenario.

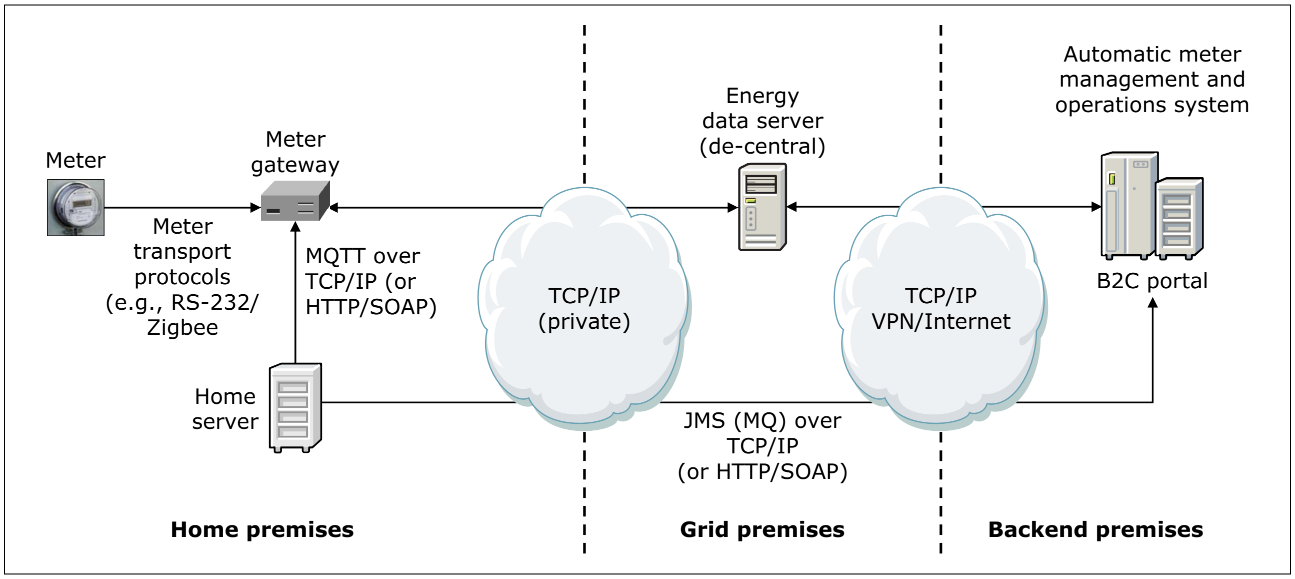

Meter
Meter gateway
Meter transport protocols (e.g., RS-232/ Zigbee
MQTT over TCP/IP (or HTTP/SOAP)
Home server
TCP/IP (private)
Energy data server (de-central)
JMS (MQ) over TCP/IP (or HTTP/SOAP)
TCP/IP VPN/Internet
Automatic meter management and operations system
B2C portal
Home premises
Grid premises
Backend premises

Figure 5-16 Energy and utility example architecture for integration of meters in SmartGrid infrastructures

A decentralized architecture provides the following benefits:

- Remote entities become more and more instrumented.
- Every device has computing power for different use cases but works in isolation.
- Remote entities are intermittently interconnected.
- Entities are not always online and depend often on a small and unreliable bandwidth.
- Intelligence is required immediately at a large number of remote entities.
- Processing of the data (preprocessing, filtering, preprocessing) where it occurs for security and performance reasons (forward only the needed data to the back-end system).
- Remote entities can work more intelligently when combining the isolated device capabilities. To achieve this there must exist basic and standardized services that can be leveraged by each device.
- The large number of independently developed devices will require an intelligent tool chain for test and maintenance.

It is recommended to push data from the meter to the back end, for example by using MQTT. rather than having back-end processes poll for data. This saves back-end resources and transmission channel overhead. The following list notes the important reasons to save resources:

- Currently, all networks with many hops and heavy load are based on a store-and-forward mechanism. The usage of the available transmission channel is more efficient for small arbitrary data packets when data transmission is packet-based rather than connection-oriented.
- Data packets are sent when available and travel over different nodes (routers and other nodes) to their destination. Intelligent routing algorithms from one node to the next optimize the transmission speed.
- Data transmission reliability is not always in focus in lower transport layers and must be applied by higher layers (for example TCP versus UDP).

There are different Edge Server functions in a Smart Metering environment. Smart Home Servers or Gateways and Meter Gateways also make use of meter data for local processing and execution of use cases. The following list shows the differences to the Grid Server:

- Smart home server or meter gateways
  - Support many protocols, many connectivity options, many sensors and actors, consumers and producers from different vendors.
  - Abstract end device APIs, provide data in standard format for further processing (for example, protocol conversion in MQTT and data in XML).
  - Provide a user friendly GUI for home customers to position new services in home premises.
  - Allow easy user-friendly wiring of home devices for home control functions.
- Decentralized server or grid server
  - Prepare data for seamless reliable back-end processing (masking, filtering, or securing).
  - Run advanced use cases (headless) for controlling more complex devices without requiring constant back-end connectivity, for example use cases that depend on optimization, prediction, modelling functions.
  - IBM can provide application development tooling, support, back-end integration, and remote management for de-central servers.

6

Scaling a system

In this chapter we provide information about the considerations you need to take into account when scaling or expanding an MQTT-based messaging system to include more and more interconnected devices.

This chapter contains the following sections:

- Scaling overview
- Provisioning first-time connections (PROV pattern)
- Pushing messages to a large number of devices (LSPN pattern)
- Pushing messages from a large number of devices (LSSP, LSSP2 patterns)

6.1 Scaling overview

As a system expands and the number of connected MQTT-enabled devices increases, developers must keep in mind several critical architectural considerations to ensure proper response time as the system grows or scales.

6.1.1 Types of scaling

Traditionally, scaling is achieved in two ways:

- Vertical scaling: Processing capacity is added to the processing nodes without adding any new nodes. This method is also called *scaling up*.
- Horizontal scaling: New nodes are added to balance the processing load between them. This method is also called *scaling out* or *clustering*.

When dealing with large sensor systems, horizontal scaling is the preferred approach. You can add multiple device handlers to manage the required number of devices, which helps maintain proper response times and adds architectural flexibility.

As the number of devices increases, you can also add new layers. For example, you can use Edge Device Hubs to group specific types of devices within certain areas of affinity.

6.1.2 Scalability patterns

Patterns are available to guide the solution architect when defining large scale systems. These patterns can be used alone or in combination with other patterns. The goal of these patterns is to add the required intermediary components in a transparent way. By doing this, the overall processing capacity of the system is increased without affecting other components.

Almost every component in a typical WebSphere MQ-MQTT topology can be scaled horizontally, provided certain considerations are taken into account. Refer to Chapter 5, "Typical topologies and patterns of use" on page 159.

For purposes of this analysis, scalability patterns are grouped into the following categories based on their primary purpose:

- PROV pattern: Configuring a device for its first-time use.
- LSPN pattern: Sending messages from the central system to devices.
- LSSP and LSSP2 patterns: Sending messages from devices to the central system.

In the remaining sections in this chapter we provide detailed information about these patterns.

6.2 Provisioning first-time connections (PROV pattern)

This pattern provisions first-time connections to new devices in a scalable way. Each device is configured to connect to a predefined provisioning server in its first use. The provisioning server sends connection information to the device, which is configured to always connect to a specific messaging service provider.

6.2.1 Applicability

Use the PROV pattern when you consider the following system characteristics to be most important:

- Auto-configuration

  Devices need to be configured on their first time use, without any user interaction. Include only provisioning server information in the devices before they are deployed. Depending on the capabilities of the device, the device can be configured such that if it loses its connection or configuration information or if it has just been started, the device can try using a previous connection or go back to the provisioning server to obtain the connection information again.

- Communication server independence

  Devices cannot be hard coded to use a specific queue manager. The only information needed at the device start is the provisioning server location, which can be changed later by sending information to reconfigure the device (assuming it is capable of receiving such information). The aim is to add flexibility to the model, allowing the devices to connect to different queue managers easily.

- Scalability

  The solution must be scalable without affecting the architecture. As the number of devices increases, there must be an ability to add more components to the system without having to change the model. There are usually far fewer provisioning messages than actual communication messages, but if provisioning needs are too high, consideration can be given to clustering the provisioning service.

6.2.2 Architecture

The architecture of the PROV pattern is shown in Figure 6-1.

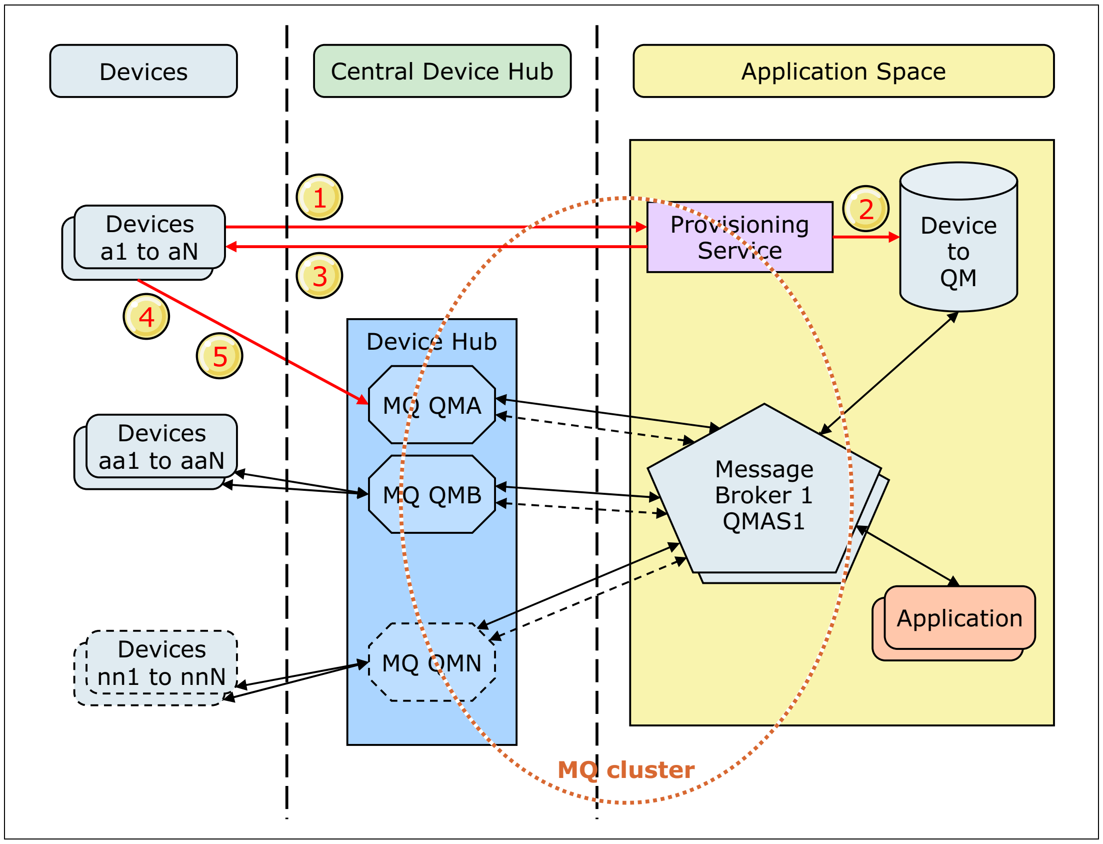


Figure 6-1 PROV pattern architecture

Table 6-1 lists the components of the PROV pattern.

Table 6-1 PROV pattern components

Component	Purpose
Devices *x*1 to *x*N	Individual devices connected to the same queue manager
Device Hub	A cluster of queue managers that handle messaging to and from devices
MQ QM*xx*	Individual queue managers, each handling a group of devices
Provisioning Service	Manages the communication parameters for each device
Device to QM	Database which records the device-manager association
Message Broker QMAS*x*	Routes messages to and from the appropriate queue manager
Application	Any application that exchanges messages with the system. These applications are not directly involved in the provisioning procedure.

6.2.3 Communication sequence

The following steps take place when provisioning a new connection under the PROV pattern:

1. The device that wants to communicate with the central system connects to the provisioning service and provides its unique device ID.
2. The provisioning service selects a queue manager for the device and records the association in the database.
3. The provisioning service returns queue manager connection information to the device, which stores it for future use.
4. The device uses the stored connection information to connect to the queue manager. The same connection information will be used each time until the device is reprovisioned or configured again.
5. The device subscribes to a unique topic specific to the device.

In an implementation of this pattern, the first communication between the device and the provisioning service can be executed using any protocol that is accessible to both parts, such as MQTT, HTTP, or REST.

The provisioning service selects the queue manager for each device. The rules for this selection can vary from one implementation to another, but typically this difference involves only the need to spread the devices equally to the available queue managers. Some scenario information can be taken into account, such as differences in network connectivity speed, the physical layout of the devices, or unbalanced use of the available queue managers.

Also, when provisioning a device that has been previously provisioned, the same queue manager should be assigned because the issue most likely relates to a device reset or dropped connection.

6.3 Pushing messages to a large number of devices (LSPN pattern)

This pattern manages communication with a large number of devices, particularly when the main communication direction is from the central system to the devices.

6.3.1 Applicability

Use the LSPN pattern when you consider the following system characteristics to be most important:

- Scalable to a large number of devices

  This pattern can be applied to any system, but its benefits are realized most fully when using a large number of remote devices or applications. It allows for multiple queue managers to be used to communicate with different groups of devices, balancing the workload between them. As the number of devices increases, more queue managers can be added to maintain required response times.

- Real-time messaging

  The number of queue managers is variable and can be adjusted to reduce communication latency, which is important because messages are supposed to be delivered in near real time. The queue managers also can temporarily store pending messages for later delivery in the event that communication to the receiving device or application is lost. And if the

LSPN pattern is used in conjunction with the PROV pattern, reconfiguration information can be sent to devices to change how they communicate with the central system.

- Architecture independent

  With the LSPN pattern, changes in architecture should be transparent to the rest of the system; when a new queue manager is added, it should not affect the operation of any existing ones. The device or application will always send messages to the same point, so it will not be affected by any changes in the queue manager structure.

- High availability

  To keep communication highly available, if a particular component fails, traffic can be redirected through other components. If the failure is in one of the queue managers, the affected devices can be reprovisioned to use another queue manager and continue working. If the failure is in the message broker, other components in the same MQ cluster can be assigned to cover the failure.

6.3.2 Architecture

The architecture of the LSPN pattern, shown in Figure 6-2, is more complex than that of the PROV pattern discussed in the previous section.

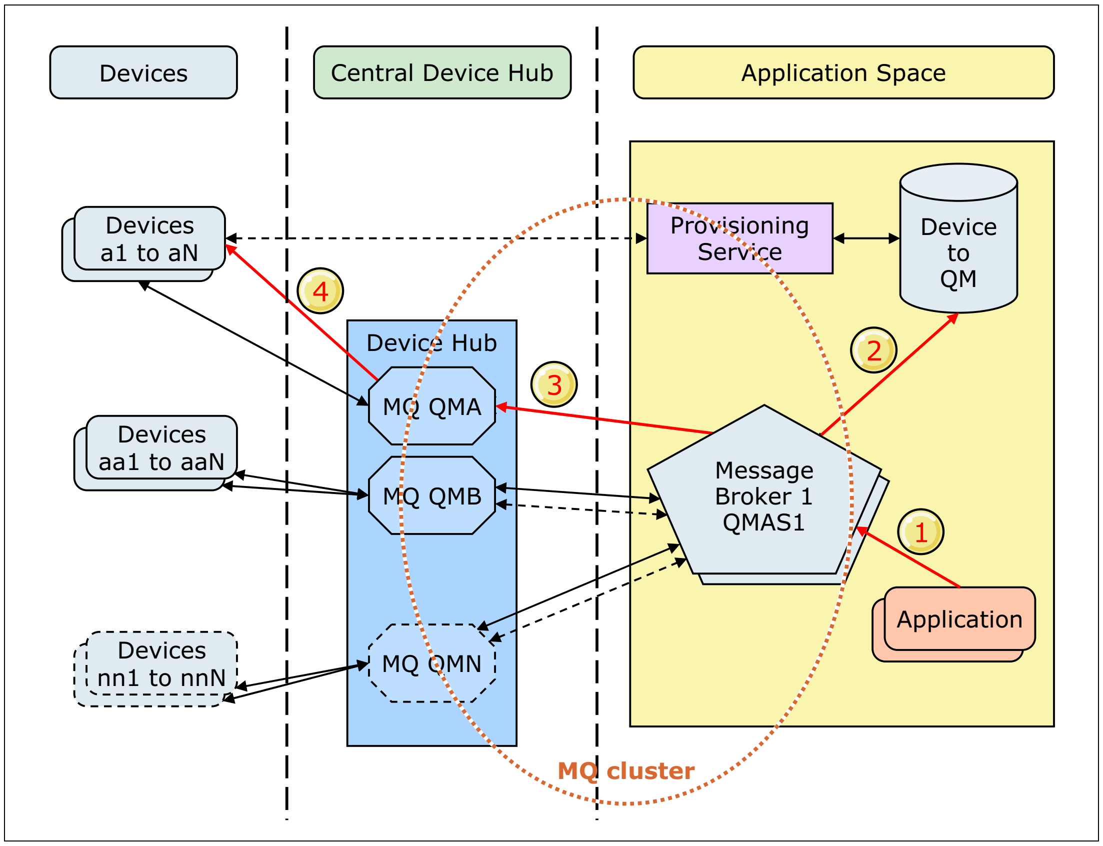


Figure 6-2 LSPN pattern architecture

Table 6-2 lists the components of the LSPN pattern.

Table 6-2 LSPN pattern components

Component	Purpose
Application	Any application that pushes messages to a device or group of devices
Message broker QMAS*x*	Routes messages to the appropriate queue manager; additional processing can be added to handle authorization, message transformation, and so on
Device to QM	Database that records the device-manager association
Device hub	A cluster of queue managers used to send messages to devices
MQ QM*xx*	Individual queue managers, each handling a group of connected devices
Devices *x*1 to *x*N	Individual devices connected to the same queue manager

6.3.3 Communication sequence

The following steps are executed when a message is sent from the application to a device under the LSPN pattern:

1. The application sends a request to the message broker using the appropriate communication protocol, which can be MQTT or another one (HTTP, JMS, REST, and so on).
2. The message broker determines which queue manager is responsible for the destination device.
3. The message broker publishes the message to the queue manager using the appropriate topic.
4. The queue manager sends the MQTT message to the destination device.

6.4 Pushing messages from a large number of devices (LSSP, LSSP2 patterns)

The LSSP pattern sends messages from a large number of devices to a central system. There are two versions of this pattern, one of which utilizes an edge device hub in an intermediate layer.

6.4.1 LSSP applicability

Use the LSSP pattern when you consider the following system characteristics to be most important:

- Scalable to a large number of devices

  This pattern can be applied to systems of any size. As long as the number of devices increases, additional intermediate components (such as queue managers) can be added. If the number of devices is too high or the devices are distributed too far geographically, then the LSSP2 pattern should be considered.

- Direct connection to central systems

  In this pattern, every device is directly connected to the central system when a connection is available. If the connection is lost, the device tries to reconnect to the same queue manager or, if that attempt fails, requests updated connection information from the provisioning manager.

- Fixed-interval messaging

  In most cases, the devices in this pattern are expected to send messages at predetermined time intervals or immediately if an exceptional event occurs.

- Parallel application processing

  The processing application can handle parallel message processing. Also, every message should be processed only once, so specific logic should be implemented by the application to prevent processing of duplicate messages, if they occur.

- Architecture independent

  With the LSSP pattern, changes in architecture are transparent to the rest of the system, so when a new queue manager is added, it will not affect the operation of any existing queue managers. And if a queue manager is taken out of service, only the devices that are directly connected to it will need to be reconfigured. To the rest of the system, this change remains transparent.

- High availability

  To keep communication highly available, if a particular component fails, traffic can be redirected through other components. If the failure is in one of the queue managers, the affected devices can be reprovisioned to use another queue manager and continue working.

6.4.2 LSSP architecture

The architecture of the LSSP pattern, illustrated in Figure 6-3, shows only a central device hub.

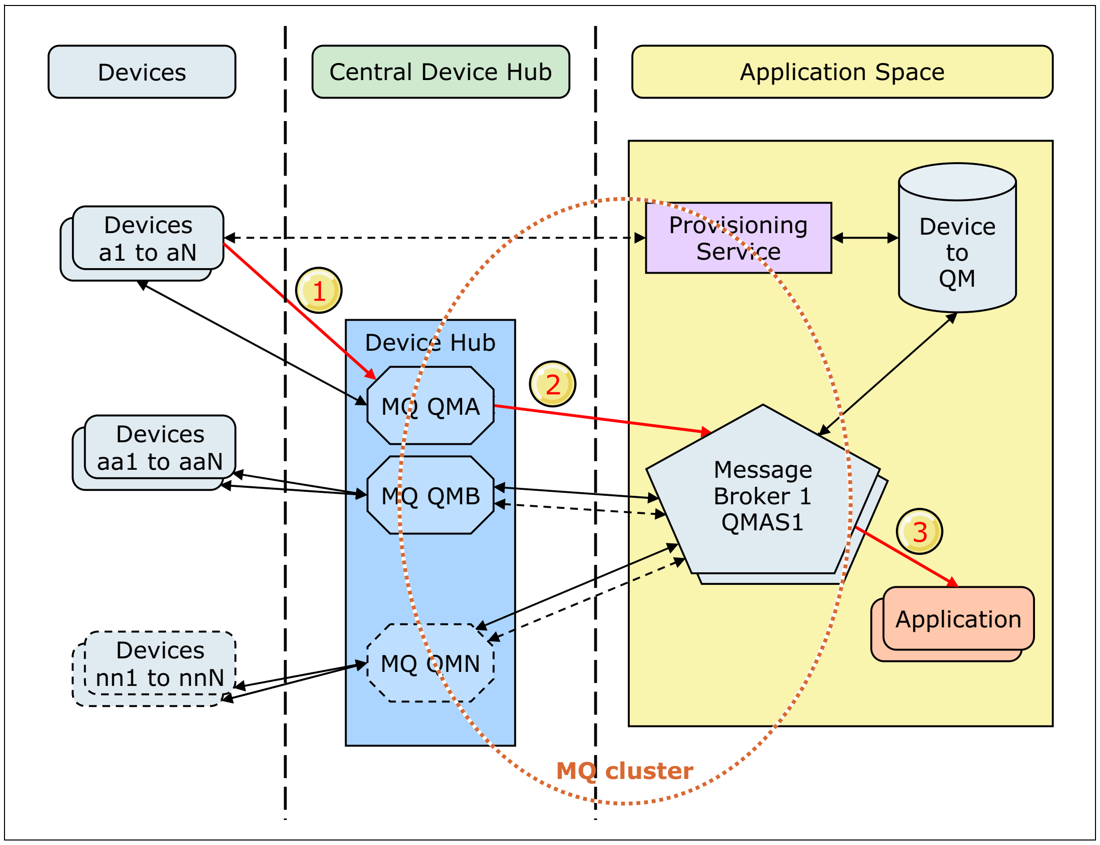


Figure 6-3 LSSP pattern architecture

Table 6-3 lists the components of the LSSP pattern.

Table 6-3 LSSP pattern components

Component	**Purpose**
Devices *x*1 to *x*N	Individual devices connected to the same queue manager
Device hub	Individual queue managers, each handling communication with a group of devices
MQ QM*xx*	Queue managers, defined as local to the application server, that receive messages from the workload balancing mechanism
Message Broker QM AS*x*	A cluster of queue managers used to send messages from devices to the broker
Device to QM	Database that records the device-manager association
Application	The final destination that picks the message from the queue manager for processing

6.4.3 LSSP communication sequence

The following sequence of steps is executed when a device sends a message to the application under the LSSP pattern:

1. The device publishes a message to its corresponding queue manager, using MQTT with a specified inbound topic.
2. The inbound topic is mapped to a clustered queue and published to the queue manager.
3. The queue manager defined as local in the application server receives the message sent by the workload balancing routine.
4. The application gets the message from the server queue manager and processes it.

6.4.4 LSSP2 applicability

The same system characteristics that support use of the LSSP pattern also support use of the LSSP2 pattern (as listed in 6.4.1, “LSSP applicability” on page 187). The main difference between the patterns is that the LSSP2 pattern inserts an intermediate layer to help manage message flow.

This layer, which contains an edge device hub, might be required for several reasons:

- Geographic proximity

  If the devices are widely distributed and the network to reach them is of poor quality or too expensive, it is better to have an edge hub located near the devices to optimize communication with the central system.

- Protocol incompatibility

  It the devices are unable to use the MQTT protocol, an edge hub can be used as a protocol converter.

- Device control

  Some devices, such as RFID sensors, require a specific controller to work properly. This controller can also act as an edge hub and handle the communication between the devices and the central system.

6.4.5 LSSP2 architecture

The architecture of the LSSP2 pattern, illustrated in Figure 6-4, adds an intermediate layer with an edge device hub.

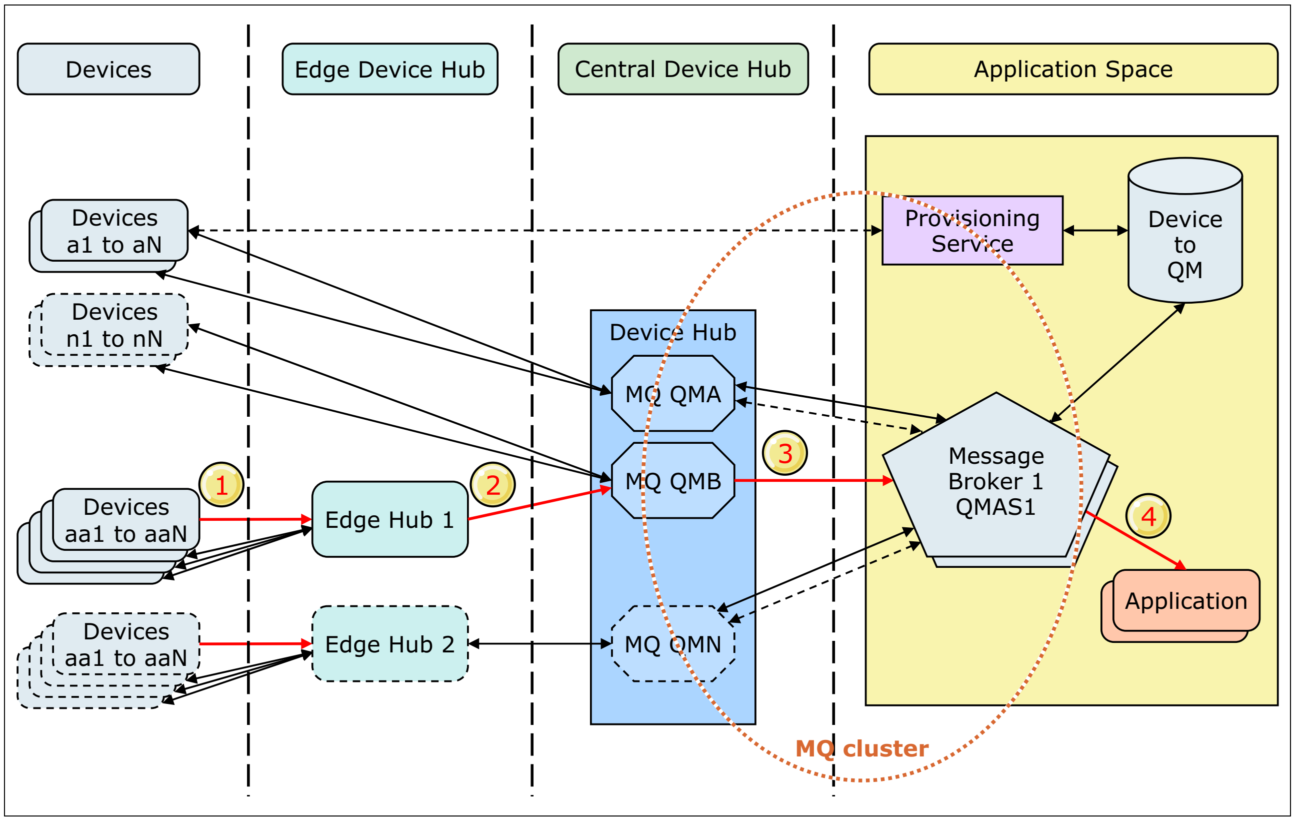


Figure 6-4 LSSP2 pattern architecture

Table 6-4 lists the components of the LSSP2 pattern.

Table 6-4 LSSP2 pattern components

Component	Purpose
Devices *x*1 to *x*N	Individual devices connected to the same queue manager
Edge Hub*x*	Individual hubs controlling a group of related devices and sending information to the corresponding queue manager
Device hub	Individual queue managers, each handling communication with a group of devices
MQ QM*xx*	Queue managers, defined as local to the application server, that will receive messages from the workload balancing mechanism
Message Broker QM AS*x*	A cluster of queue managers used to send messages from devices to the broker
Device to QM	Database that records the device-manager association
Application	The final destination that picks the message from the queue manager for processing

6.4.6 LSSP2 communication sequence

The following sequence of steps is executed when a device sends a message to the application under the LSSP2 pattern:

1. The device sends a message to its edge hub using an appropriate communication protocol.
2. The edge hub publishes the message to its corresponding queue manager, using MQTT with a specified inbound topic.
3. The inbound topic is mapped to a clustered queue and published to the queue manager.
4. The queue manager defined as local in the application server receives the message sent by the workload balancing routine.
5. The application receives the message from the application server queue manager and processes it.

A

MQTT and devices

The number of different types of devices that can benefit from MQTT implementations is growing every day. This appendix discusses some of the available MQTT implementations on specific devices to demonstrate their use and functionality. The first section shows how to use MQTT directly from a circuit board, in this case the Arduino platform. The second section covers the use of MQTT from mobile devices (including smartphones, PDAs, or tablets). Although there are a good number of different platforms, each one with its particularities, in this book we discuss devices based on the IBM Worklight and Android devices.

A.1 MQTT and Arduino devices

In this section we describe the use of the MQTT protocol for the devices that are built on top of the Arduino board.

A.1.1 Introducing Arduino

Arduino is an open source electronics platform that allows you to enhance the capabilities of sensors and actuators. Arduino is built on top of a micro-controller that has a number of digital and analogue pins. These pins allow the board to sense its environment and respond to it. After the wiring is done, the board can be programmed using the Arduino programming language.

The syntax of this programming language is similar to the C programming language. The similarity also extends to use of libraries in C programs. That is, a regularly used piece of code could be hived off as a library which other programmers could include in their subsequent programs.

Additional resource: You can learn more about the Arduino programming language at:

`http://arduino.cc/en/Reference/HomePage`

A.1.2 Simple Arduino circuit

This scenario shows a simple example of using Arduino and is taken from the Arduino development IDE. It provides an illustration of the circuit (Figure A-1) and includes the source code (Example A-1). The board is an Arduino Uno. In the circuit shown in Figure A-1, the LED has its cathode connected to GND and the anode to pin 13.

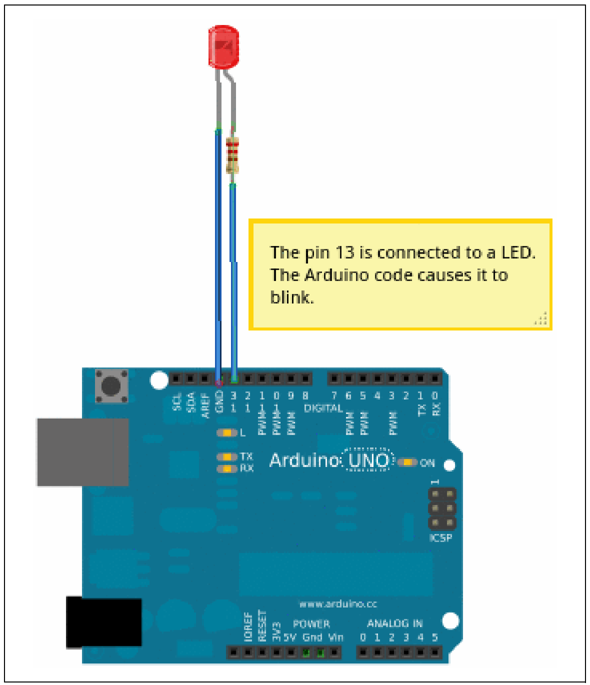

The pin 13 is connected to a LED. The Arduino code causes it to blink.
Arduino UNO
DIGITAL
POWER
ANALOG IN
ICSP
www.arduino.cc

Figure A-1 Illustration of the Arduino circuit

In general, the Arduino code runs such that the `setup()` is run once and only once. Whereas the `loop()` function runs all the time unless the reset button on the board is pressed. When reset is pressed, the code execution begins again with `setup()` followed by looping the `loop()` function. To run the code, it is first compiled in the IDE and then is uploaded to the board. The upload causes a reset automatically, and the execution begins as described before.

In Example A-1, the LED connected to pin 13 is first designated as an OUTPUT pin. Then, in the `loop()` function, the LED is turned on using the built-in `digitalWrite()` function, where HIGH turns the LED ON and LOW turns it OFF. A delay of 1000 milliseconds (or a second) is introduced between these functions to give an impression of a blinking LED.

Example A-1 Source code for the sample circuit

```
/
   Blink
   Turns on an LED on for one second, then off for one second, repeatedly.
   This example code is in the public domain.
/
void setup() {
   // initialize the digital pin as an output.
   // Pin 13 has an LED connected on most Arduino boards:
   pinMode(13, OUTPUT);
}
void loop() {
   digitalWrite(13, HIGH); // set the LED on
   delay(1000); // wait for a second
   digitalWrite(13, LOW); // set the LED off
   delay(1000); // wait for a second
}
```

A.1.3 MQTT support in Arduino

The MQTT support in Arduino is provided through a library that is an implementation of the MQTT protocol V3. You can download this library at:

http://knolleary.net/arduino-client-for-mqtt/

This library should be included in the Arduino code and then appropriate functions are called for CONNECT, PUBLISH, SUBSCRIBE, and so forth. In our simple example, the code for CONNECT is added in the `setup()` function because the connection is made only once. The SUBSCRIBE code is added in the `setup()` function, and the message that has arrived is handled separately in a `callback()` function. The PUBLISH code is added in the `loop()` function.

A.1.4 Arduino and MQTT with a publish scenario

The network support for the Arduino board is provided using a *shield*, such as Arduino Ethernet shield. This shield is connected to a modem or a network port using a standard network cable. This shield sits on top of the Arduino Uno board or other board.

For more information about the Arduino Ethernet shield, see:

http://arduino.cc/en/Main/ArduinoEthernetShield

Description of the scenario

This scenario illustrates an Arduino-based device that publishes to a topic using the MQTT protocol. The device is a simple temperature sensor that measures the ambient temperature every two minutes and publishes to a topic on an MQTT server.

Figure A-2 illustrates the circuit for this scenario.

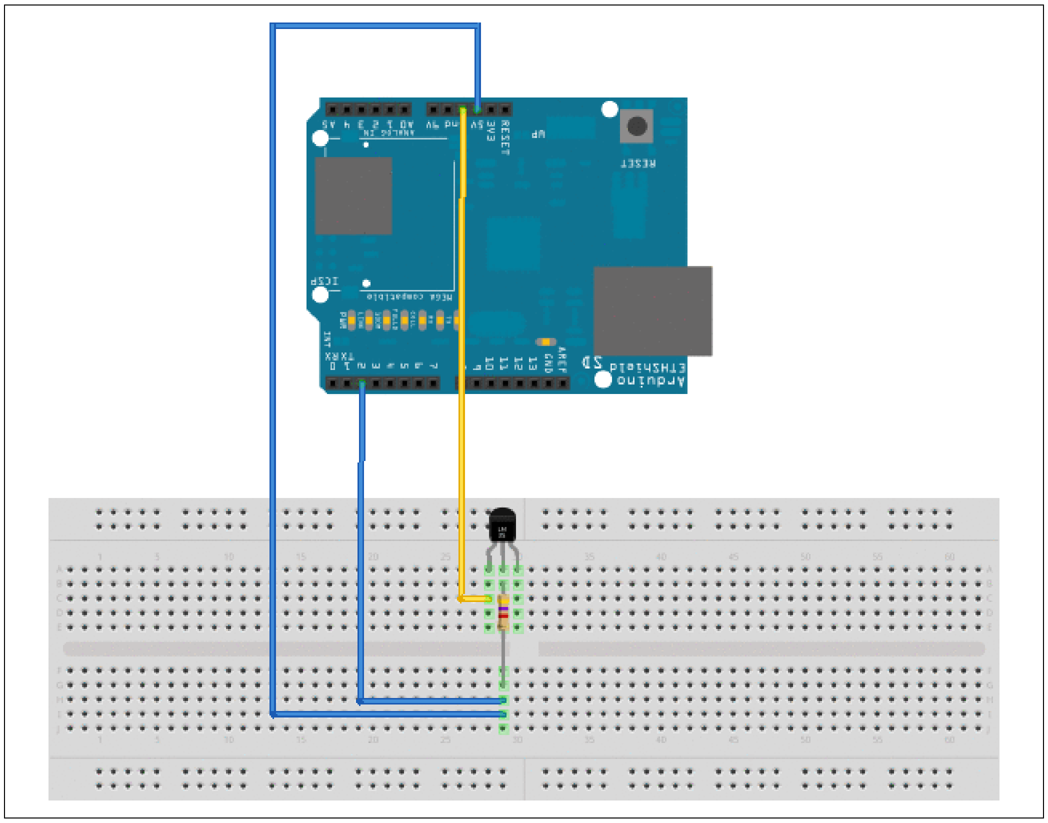


Figure A-2   Publish scenario circuit

Example A-2 lists the source code for this scenario.

Example A-2   Source code

```
#include <SPI.h>
#include <PubSubClient.h>
#include <DallasTemperature.h>
#include <OneWire.h>
#include <Ethernet.h>
#include <util.h>
#include <ctype.h>
/
OneWire - http://www.arduino.cc/playground/Learning/OneWire
Dallas Temperature Sensor -
https://code.google.com/p/dallas-temperature-control-library/source/checkout
MQTT - http://knolleary.net/arduino-client-for-mqtt/
/
#define CLIENTID "ArduinoSensor"
#define TOPICNAME "sensor/temperature"
#define POLLINTERVAL 120000
#define DS18S20Pin 2
```

```
//Some MAC - DEAD BEEF FEED !
byte mac [] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED} ;
//Connect to test.mosquitto.org server
byte server [] = { 85, 119, 83, 194 };
//Our MQTT client
PubSubClient arduinoClient(server, 1883, callback) ;
//Board on time
unsigned long boardTime = 0 ;
//Sensed temperature
float sensedTemperature = 0.0 ;
//Temperature in character format for publishing
char charTemperature [20]
;
//Temperature chip i/o
OneWire oneWire(DS18S20Pin); // on digital pin 2
DallasTemperature sensors(&oneWire);
//Handle message from mosquitto and set LEDs
void callback(char topic, byte payload, unsigned int length){
//Do nothing as we are publishing ONLY.
}
void setup(void) {
Serial.begin(9600);
// Start the sensor
sensors.begin();
//Connect to the MQTT server - test.mosquitto.org
beginConnection() ;
}
//Initialise MQTT connection
void beginConnection() {
Serial.begin(9600);
Ethernet.begin(mac) ;
int connRC = arduinoClient.connect(CLIENTID) ;
if (!connRC) {
Serial.println(connRC) ;
Serial.println("Could not connect to MQTT Server");
Serial.println("Please reset the arduino to try again");
delay(100);
exit(-1);
}
else {
Serial.println("Connected to MQTT Server...");
}
}
void loop(void) {
boardTime = millis();
if ((boardTime % POLLINTERVAL) == 0) {
getTemp() ;
dtostrf(sensedTemperature,5,2,charTemperature) ;
arduinoClient.publish(TOPICNAME, charTemperature) ;
}
}
void getTemp() {
// Send the command to get temperatures
sensors.requestTemperatures();
delay(100);
sensedTemperature = sensors.getTempCByIndex(0);
delay(150);
}
```

The description of the source code is as follows:

1. Add the `PubSubClient.h`, `OneWire.h`, and `DallasTemperature.h` header files and ensure that the associated `.cpp` files are available to the IDE.
2. The server to be connected to is defined as a byte array where each number represents the octets of an IP address.
3. Define a function `callback()` that receives control when a message for the subscribed topic arrives.
4. Define the client arduinoClient using the server IP address in the form of byte array, port number and the `callback()` function as the arguments.
5. Define the name of the client and the topic to be subscribed as constants.
6. As part of `setup()`, initialize the pins and initiate the connection. To initiate the connection, define (randomly) the MAC address for the device as `0xDEADBEEFFEED`. Change it to the MAC address as applicable to your installation.
7. As part of the `setup()` function, initialize the temperature sensor.
8. In the `loop()` function, obtain the temperature every 2.0 minutes, and publish it to an MQTT topic.

Running the scenario

To run the scenario, follow these steps:

1. Start an instance of the WMQTT Utility.
2. Enter the IP address and port for the MQTT broker.
3. Click **Connect**.
4. Enter a topic for the subscription as `sensor/temperature/`.
5. Turn on the board.
6. Click **Subscribe** on the WMQTT Utility.

Temperatures appear in the WMQTT Utility.

A.1.5 Arduino and MQTT with a subscribe scenario

The network support for the Arduino board is provided using a *shield*, such as Arduino Ethernet shield. This shield is connected to a modem or a network port using a standard network cable. This shield sits on top of the Arduino Uno board or other board.

For more information about the Arduino Ethernet shield, see:

http://arduino.cc/en/Main/ArduinoEthernetShield

Description of the scenario

This scenario shows a device that subscribes to a topic using the MQTT protocol. In this example, an application reads in the temperature that is published by the device described in A.1.4, “Arduino and MQTT with a publish scenario” on page 195. This application has logic in it to determine whether the temperature is within predetermined limits. If the temperature is within limits, a message as YES,YES is published. If the temperature is not within limits, a message as NO,YES or YES,NO is published, depending on whether the lower limit or higher limit, respectively, was breached.

Figure A-3 shows the circuit for this scenario.

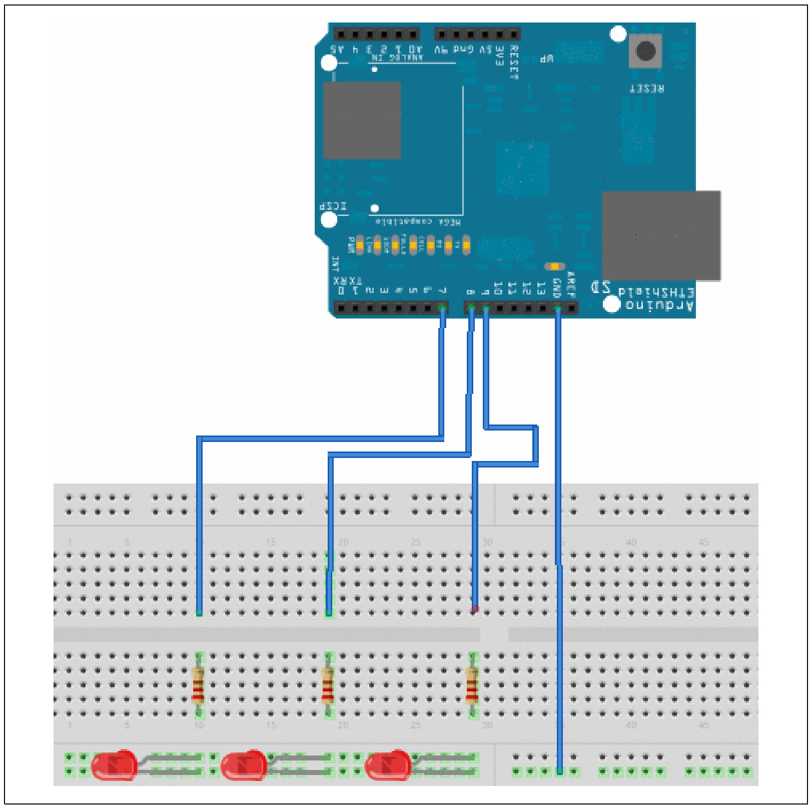


Figure A-3   Subscribe scenario circuit

In this circuit, three LEDs have anodes connected to pins 7, 8, and 9, respectively. Their cathodes are grounded. The resistors are 220 Ω to avoid damaging the LED. These three resistors approximately replicate an RGB LED behavior. That is, at any point, only one of the LEDs will be glowing to indicate NORMAL, HIGH, or LOW situations.

Example A-3 shows the source code for this scenario.

Example A-3   Source code

```
#include <Ethernet.h>
#include <EthernetUdp.h>
#include <EthernetServer.h>
#include <EthernetClient.h>
#include <util.h>
#include <Dns.h>
#include <Dhcp.h>
#include <SPI.h>
#include "PubSubClient.h"
//Pin definition
#define NORMALPIN 7
#define LOWPIN 8
#define HIGHPIN 9
//MQTT definition
#define CLIENTID "ArduinoActuator"
```

```
#define TOPICNAME "sensor/temperature/monitor"
/
By default Pin 7 will be ON to denote that the temperature is between HIGH and LOW
limits.
When the limits are breached, then Pin 7 goes OFF and either Pin 8 or Pin 9 goes
ON depending on which limit was breached.
Pin 7 Pin 8 Pin 9
NO,NO 0 1 1 Meaningless - this should not happen !
NO,YES 0 1 0 LOW limit breached
YES,NO 0 0 1 HIGH limit breached
YES,YES 1 0 0 NORMAL - No limits are breached.
/
//  CHANGE MAC ADDRESS TO THAT APPEARS ON THE ETHERNET SHIELD
byte mac [] = {0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED} ;
//
//Connect to test.mosquitto.org server
byte server [] = { 85, 119, 83, 194 };
//Handle message from mosquitto and set LEDs
void callback(char topic, byte payload, unsigned int length) {
int i=0; char buffer [length] ;
for(i=0;i<length;i++) {
buffer [i] = char((payload + i)) ;
}
String payLoadData = buffer ;
setPins(payLoadData) ;
}
//Our MQTT client
PubSubClient arduinoClient(server, 1883, callback) ;
void setup() {
//Set pins to indicate normal
beginPins() ;
//Connect to the MQTT server - test.mosquitto.org
beginConnection() ;
}
//Initialise LED pins
void beginPins() {
pinMode(NORMALPIN, OUTPUT) ;
pinMode(LOWPIN, OUTPUT) ;
pinMode(HIGHPIN, OUTPUT) ;
digitalWrite(NORMALPIN, HIGH) ;
digitalWrite(LOWPIN, LOW) ;
digitalWrite(HIGHPIN, LOW) ;
}
//Initialise MQTT connection
void beginConnection() {
//Serial.begin(9600);
Ethernet.begin(mac) ;
int connRC = arduinoClient.connect(CLIENTID) ;
if (connRC) {
arduinoClient.subscribe(TOPICNAME) ;
} else {
Serial.println(connRC) ;
}
}
void loop() {
```

```
arduinoClient.loop() ;
}
void setPins(String s) {
/
The code between BEGIN and END turns ON all the LED for a second before turning ON
or OFF the LED
to denote the temperature range. If the temperature source does not fluctuate too
much, then this
piece of code can give an impression that the Arduino is "working".
/
// BEGIN
digitalWrite(NORMALPIN, HIGH) ;
digitalWrite(LOWPIN, HIGH) ;
digitalWrite(HIGHPIN, HIGH) ;
delay(1000) ;
// END
s.trim();
int temperatureLimits[] = {1,1} ;
if (s.substring(0,s.indexOf(',')) == "YES") { temperatureLimits [0] = 1; }
if (s.substring(0,s.indexOf(',')) == "NO") { temperatureLimits [0] = 0 ; }
if (s.substring(s.indexOf(',')+1) == "YES") { temperatureLimits [1] = 1 ;}
if (s.substring(s.indexOf(',')+1) == "NO") { temperatureLimits [1] = 0 ;}
int ledPins [] = {NORMALPIN, LOWPIN, HIGHPIN} ;
int ledStatus [] = {1, 0, 0} ; //Default is NORMAL, so NORMAL pin is ON
for (int i=0; i<2; i++) {
if (temperatureLimits [i] == 0) { ledStatus [0] = 0 ; ledStatus [i+1] = 1 ; }
}
for (int j=0; j<3; j++) {
digitalWrite(ledPins[j],ledStatus[j]
);
}
}
```

The description of the source code is as follows:

1. Add the `PubSubClient.h` header file, and ensure that this header file and the associated `.cpp` files are available to the IDE.
2. Define the server to be connected to as a byte array, where each number represents the octets of an IP address.
3. Define a function `callback()` that receives control when a message for the subscribed topic arrives.
4. Define the `arduinoClient` client using the server IP address in the form of byte array, port number, and the `callback()` function as the arguments.
5. Define the name of the client and the topic to be subscribed as constants.
6. As part of the `setup()` function, initialize the pins and initiate the connection. To initiate the connection, define (randomly) the MAC address for the device as `0xDEADBEEFFEED`. Change it to the MAC address that is burnt onto the device.
7. After initiating the connection, subscribe to the topic name in which you are interested.

The `callback()` function calls the `setPins()` function to turn ON or OFF the LEDs as described previously.

Running the scenario

To run the scenario, follow these steps:

1. Start an instance of the WMQTT Utility.
2. Enter the IP address and port for MQTT broker.
3. Click **Connect**.
4. Enter a topic for publication as `sensor/temperature/monitor`.
5. Type in a message as `YES,YES` and click **Publish**.
6. Watch the LEDs turn ON or OFF.

To run this scenario, set up an MQTT server and a client. Then, publish to the sensor/temperature/monitor topic a message as YES,YES, NO,YES, or other setting.

A.2 MQTT and mobile communication devices

Mobile communication devices are the fastest growing application segment. The traditional mobile phone has evolved into a powerful computing device, and new platforms and mobile operating systems are also now available.

Despite the wide variety of available technological capabilities, keep in mind the following factors when considering building applications for mobile devices:

- Preservation of battery life

  Mobile communication devices are designed, as the name implies, to be mobile. Thus, as a general rule, the time interval between charges needs to be as long as possible. Independent of the device's platform or battery technology, running applications consume resources (processor, memory, and network connectivity) and consume battery power.

  Even when application management is generally a platform's choice, applications should be designed to consume the least resources possible.

- Processing capacity

  Even when mobile devices have a relatively good processing capacity, make applications as simple as possible in terms of processing to speed up response time and to help preserve battery life.

- Network optimization

  Mobile devices should have at least one network connectivity option. In fact, most devices have a choice between several connection possibilities that vary in speed, connectivity range, battery consumption, and cost to the user.

  As a general rule, most types of connectivity are at some point limited in range. Mobile applications should be able to switch between available communication options seamlessly based on a set of user-defined preferences. By doing this, an application can use a preconfigured IEEE 802.11*x* (WiFi) connection when available and then switch to a 3G cellular connection when the previous connection is out of range. It is the application's responsibility to continue its functionality in a transparent way.

- Platform capabilities

  Base requirements to build applications can vary from one platform to another. Usually, the platform owner provides a set of APIs to develop applications for their platform. Sometimes, as with MQTT, free implementations of the protocol allow the developer to concentrate on the application functionality and use the MQTT protocol as a service.

For a list of available APIs, see Table 1-1 on page 7. For the benefits of using MQTT over other protocols, see 1.3, “Benefits of using MQTT” on page 12.

In this section we introduce two mobile technologies: Worklight and Android.

A.2.1 Worklight

Worklight, an IBM company, lets developers use JavaScript and HTML 5 to develop applications for different mobile devices. Worklight also provides a set of enterprise features that includes version management of client application, secure and scalable connectivity to back-end systems, and uniform push notification.

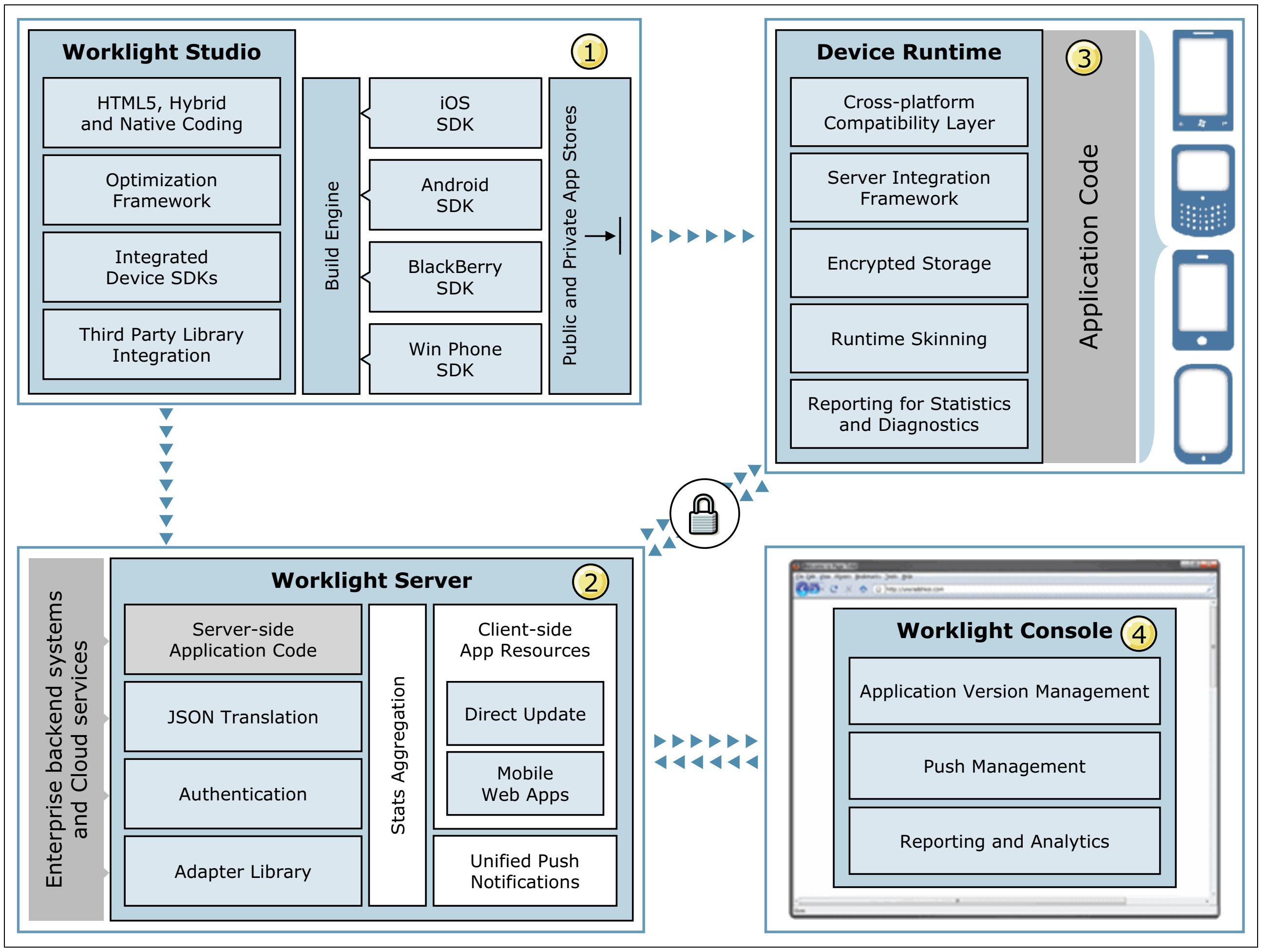
Worklight consists of the following components:

- *Worklight Studio* is an Eclipse-based integration development environment that allows developers to perform coding, generate client application packages for different platforms and deploy applications to Worklight Server.
- *Worklight Server* is a Java-based server application. It is designed to use enterprise systems and is accessed from mobile phones. It also provides administration functions such as authentication, direct update of web and hybrid applications, and operational functions.
- *Device Runtime* is a cross mobile platform run time. Worklight uses the PhoneGap framework with the Worklight plug-in to provide web applications, hybrid, or native applications for mobile devices.
- *Worklight Console* is a web-based console for administration of application version management, push notification management and reporting and analytics. It provides a user-friendly way to manage deployed applications, adapters, and other resources on the Worklight Server.

For more information about Worklight, see:

http://www.worklight.com/product/overview

Figure A-4 on page 204 shows an architectural diagram of the Worklight components.



Worklight Studio
HTML5, Hybrid and Native Coding
Optimization Framework
Integrated Device SDKs
Third Party Library Integration
Build Engine
iOS SDK
Android SDK
BlackBerry SDK
Win Phone SDK
Public and Private App Stores
1
Device Runtime
3
Cross-platform Compatibility Layer
Server Integration Framework
Encrypted Storage
Runtime Skinning
Reporting for Statistics and Diagnostics
Application Code
Enterprise backend systems and Cloud services
Worklight Server
2
Server-side Application Code
JSON Translation
Authentication
Adapter Library
Stats Aggregation
Client-side App Resources
Direct Update
Mobile Web Apps
Unified Push Notifications
Worklight Console
4
Application Version Management
Push Management
Reporting and Analytics

Figure A-4 Architecture diagram of Worklight

Integrating MQTT with Worklight

The integration between MQTT and Worklight is performed in the *Device Runtime* level. Because Worklight uses the PhoneGap framework to provide cross-platform run time for supported mobile devices, a PhoneGap plug-in needs to be built for supporting MQTT. A PhoneGap plug-in lets developers use a specific third-party native library or device function, which is not available in PhoneGap.

Because there is no official support of MQTT in Worklight, developers need to build a custom plug-in to develop MQTT applications.

Figure A-5 on page 205 illustrates the architecture of the Device Runtime plug-in. It consists of a JavaScript wrapper and native code. Native code is Objective-C that calls the MQTT C client library to communicate with the MQTT broker. A JavaScript wrapper provides a JavaScript API by invoking native code through the JavaScript/Native bridge of PhoneGap.

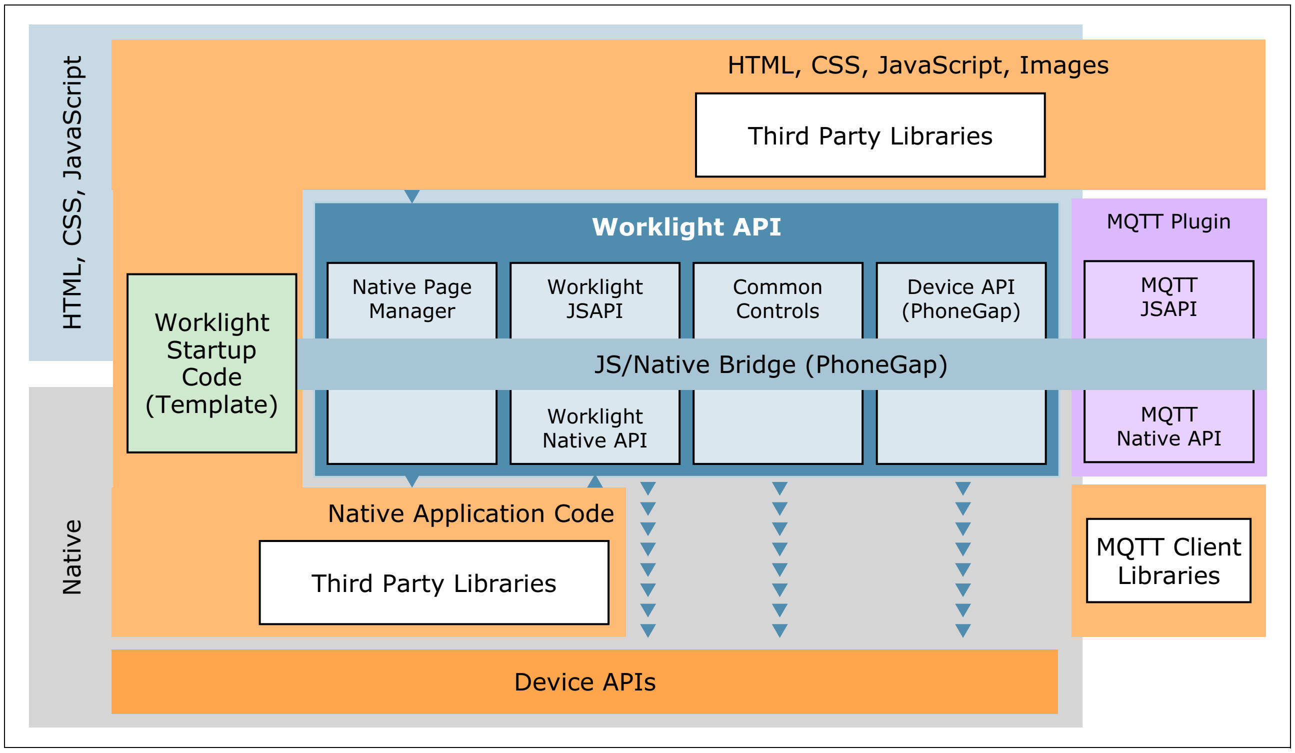


Figure A-5 Architecture diagram of Worklight

A.2.2 Android

Android is a mobile operating system. The Android software stack is shown in Figure A-6 on page 206. In this section we show how to develop a basic MQTT application for publishing location data on Android. With the powerful sensors support on Android, you can publish other data from sensors, such as gyroscopic sensor or digital compass to the MQTT broker.

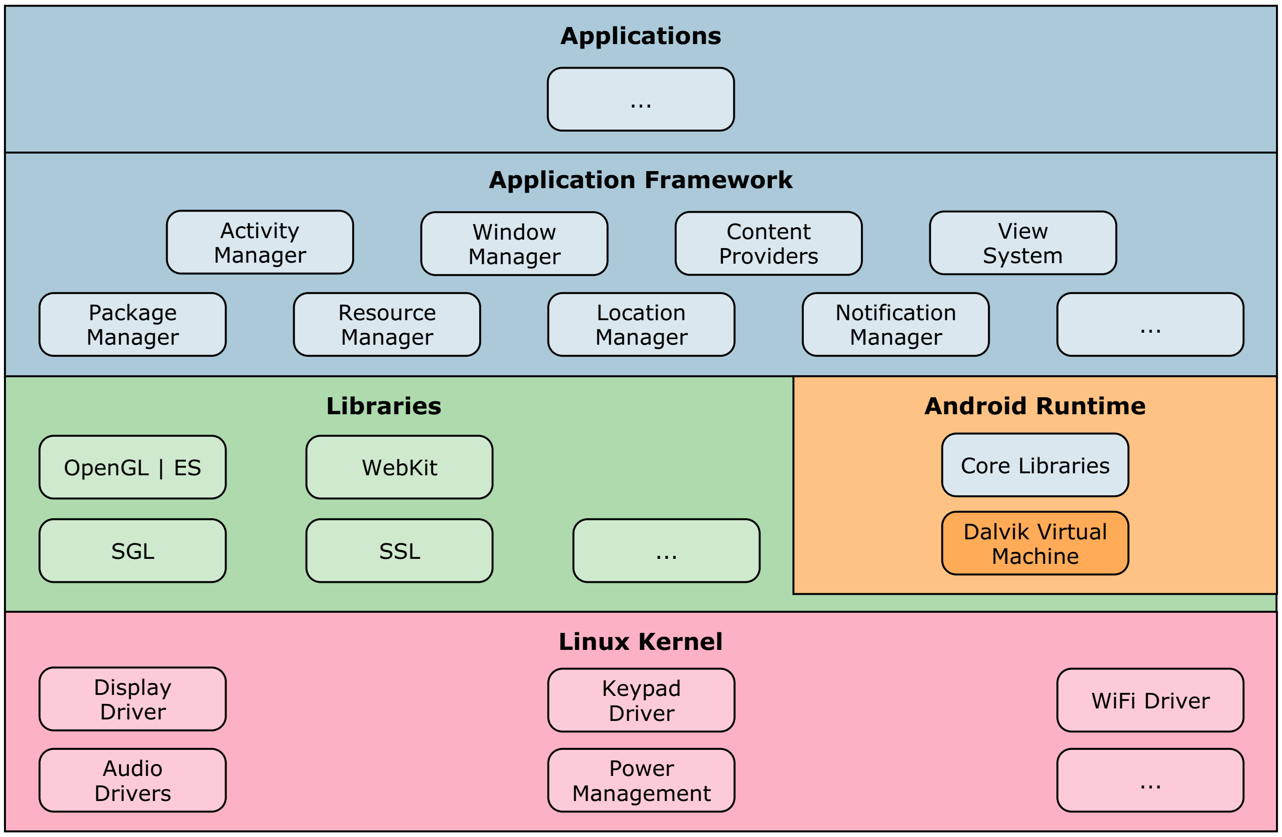


Figure A-6 Android software stack

Sample MQTT application

To develop Android applications, you need to set up a development environment, including the Android SDK, Eclipse, the ADT plug-in for Eclipse, and the AVD Manager. You can find more information about the Android development environment at:

`http://developer.android.com/`

Also to enable MQTT use in an Android application, include the corresponding Java libraries as part of your project. You can download the IA92 SupportPac client library from:

`http://www-01.ibm.com/support/docview.wss?uid=swg24006006`

Programming model

Even though it is Java based, there are some specific Android APIs that should be considered to ensure the correct functioning of our application.

The main difference between an Android application and a traditional JSE application is the need of Activities and Services. Each one plays a specific role inside the programming model, and ignoring some of their peculiarities can lead to unexpected application behavior.

Activities are user-centric tasks, probably involving a panel and some user interaction. They are short lived because only one activity is active at a specific moment in time. The operating system chooses to pause or destroy an activity based on its own needs. The application does not have much control about how activities are executed. So, activities let the user review application configuration information or type a message to be sent using MQTT. To execute more controllable processing, you need to use services.

Services are defined as possible long-time processing operations that are not directly related to the user's interaction. They are the place where you will manage the MQTT connection and messaging exchange because you want it to be an always-running background process.

There is another rule to take into consideration for our sample: Services can be stopped by the operating system at any time, if for example the device is in standby or if it is low on resources (low memory, low battery, and so forth). To partially prevent this from happening, this specific service should be marked as *sticky*, meaning that the operating system will try to restart the service as soon as possible if the service is killed.

For more information about this topic, see:

http://developer.android.com/

Sample code

In this section we show how to build a location publisher on the Android platform. An advantage to developing an application on a smartphone today is that it is easy to obtain data from sensors through the APIs of the mobile phone platform. Most smartphones on the market contain a branch of sensors, such as GPS, gyroscopic sensor, accelerometer, digital compass, and other types of sensors. Instead of publishing a fix string to the MQTT server, the sample program in this section publishes the location information from GPS sensors of the phone for a period of time.

To simplify the program, the MQTT application is defined inside an activity instead of a service. The difference between an activity and a service is the lifecycle. Although a service always runs in the background, an activity runs only when the application is in the foreground. The MQTT connection between client and server is interrupted if the activity is not in the foreground. You can modify the sample program to a long-running service after you learn how to develop an MQTT application on Android.

This sample is based on the Java sample that is shown in 2.5.3, "Publisher and subscriber in Java" on page 37. The Java sample is provided in the additional materials supplied with this book. The following paragraphs list and explain only the key parts related to MQTT client programming.

Requesting user permissions

In Android development, operations such as writing external disks, accessing the Internet, and obtaining location data require declaring permission in an Android manifest file. This sample needs to declare the following permissions (see Example A-4):

- `ACCESS`*`FINE`*`LOCATION`

  To receive location data updates, declare this permission.

- `ACCESS`*`MOCK`*`LOCATION`

  Allow this sample to create mock location providers for testing purposes.

- `INTERNET`

  MQTT client needs Internet access to connect with the MQTT broker.

Example A-4 Android manifest file sample

```
<manifest ...>
<uses-permission android:name="android.permission.ACCESSFINELOCATION" />
<uses-permission android:name="android.permission.ACCESSMOCKLOCATION" />
<uses-permission android:name="android.permission.INTERNET"/>
...
</manifest>
```

Initialize connection to the MQTT broker

First, define a MQTTActivity class that inherits from the *Activity* class. Then update onCreate with the text shown in Example A-5. onCreate is called during the construction of the *Activity* class. This sample needs to build up the connection to the MQTT broker at the start, so MQTTClient is initialized and the *connect* method is called.

For QoS=1 and QoS=2 message delivery, MQTTClient requires that temporary information is stored on a local file system. It saves files in the running directory of the sample by default, but this does not allow the sample to write in Android. This sample gets a temporary directory with the getCacheDir() method, and provides it to an MqttDefaultFilePersistence instance. Then MqttDefaultFilePersistence uses this directory to store temporary message delivery information.

Example A-5 Initialization code of the MQTTActivity class

```
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
try {
//a. Create an instance of MQTT client

String cacheDir = getCacheDir().getAbsolutePath();

client = new MqttClient(Example.TCPAddress, Example.clientId, new
MqttDefaultFilePersistence(cacheDir));

//b. Prepare connection options

//Use default connection options in this sample.

//c. Connect to broker with the connection options

client.connect();
} catch (MqttException e) {

throw new RuntimeException(e);
}
textView = (TextView) findViewById(R.id.text);
initLocationService();
}
```

The publishMessage method, shown in Example A-6, provides the caller with a way to publish a message easily. `Log.d` outputs messages to a logging system in Android. It is similar to standard output in Java Standard Edition. `textView` is a visible component in this sample. A message is appended to it if message delivery is successful.

Example A-6 Publish method code

```
public void publishMessage(String topic, String message) {
//d. Publish message to topics
MqttTopic mqttTopic = client.getTopic(topic);
MqttMessage mqttMessage = new MqttMessage(message.getBytes());
mqttMessage.setQos(Example.QoS);
Log.d(TAG, "Waiting for up to " + Example.sleepTimeout / 1000
+ " seconds for publication of \"" + mqttMessage.toString()


+ "\" with QoS = " + mqttMessage.getQos());
Log.d(TAG, "On topic \"" + mqttTopic.getName()


+ "\" for client instance: \"" + client.getClientId()


+ "\" on address " + client.getServerURI() + "\"");
MqttDeliveryToken token;
try {

token = mqttTopic.publish(mqttMessage);

token.waitForCompletion(Example.sleepTimeout);

textView.append("Delivery token \"" + token.hashCode()




+ "\" has been received: " + token.isComplete() + "\n");
} catch (MqttException e) {

Log.e(TAG, "Cannot publish message, " + message);
}
}
```

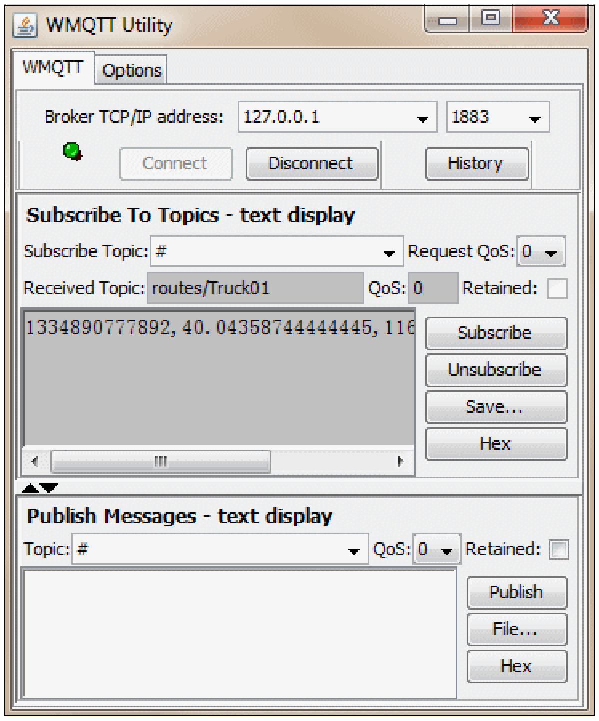
Example A-7 shows how a client application registers callback and defines behavior of callbacks for location data updates. This method uses the following flow:

1. Gets locationManager from the system service.
2. Defines a listener locationListener that responds to location updates. This sample implements only the onLocationChanged method. When location updates occur, this method is called, and this sample publishes a message to the routes/Truck01 topic with the format time, latitude, longitude, status to MQTT broker.
3. Registers the listener with the location manager.

Example A-7 Register callback of location data updates

```
private void initLocationService() {
LocationManager locationManager = (LocationManager)
this.getSystemService(Context.LOCATIONSERVICE);
//Define a listener which responds to location updates
LocationListener locationListener = new LocationListener() {
public void onLocationChanged(Location location) {

String message = System.currentTimeMillis() + "," + location.getLatitude() + ","

+ location.getLongitude() + ",B";

publishMessage("routes/Truck01", message);
}
public void onStatusChanged(String provider, int status, Bundle extras) {}
public void onProviderEnabled(String provider) {}
public void onProviderDisabled(String provider) {}
};
// Register the listener with the location manager.
locationManager.requestLocationUpdates(LocationManager.NETWORKPROVIDER, 0, 0,
locationListener);
}
```

For the remainder of this sample, download the sample project and refer to it. After importing the project, you can run this sample either on an emulator or your Android phone. Activate the GPS on your phone before starting the sample. Then, you can see update messages print to the window for a period of time. You can also verify the application with a IA-92 GUI client. You just need to start the GUI, connect to the server to which your application connects, subscribe to routes/Truck01 topic or all topic #, and then you will see that the IA-92 GUI client gets published messages from the sample Android application as shown in Figure A-7.



Broker TCP/IP address: 127.0.0.1 1883

Subscribe To Topics - text display

Subscribe Topic: # Request QoS: 0

Received Topic: routes/Truck01 QoS: 0 Retained:

```
1334890777892,40.04358744444445,116
```

Publish Messages - text display

Topic: # QoS: 0 Retained:

Figure A-7 Using the MQTT Utility

Using mock location data for testing

If you are using an emulator for testing or if you want to customize your current location, you can use mock location data for the testing. Follow the instructions in *"Providing mock location Data"* in the following document. It describes three methods (using Eclipse, DDMS, or the geo command in a console) to provide mock location data for your phone or emulator.

http://developer.android.com/guide/topics/location/obtaining-user-location.html

B

The MQTT protocol

The MQ Telemetry Transport (MQTT) protocol is a lightweight network protocol that is used for publish/subscribe messaging between devices. In this appendix we describe the various commands that flow back and forth between the publisher or subscriber and the server.

The traces shown in this appendix were captured using Wireshark, an open source tool to view packets traveling across a network. You can find details about Wireshark at:

http://www.wireshark.org/

The sections of this appendix discuss the following topics:

- MQTT concepts
- Set up for tracing
- Connect to the MQTT server
- Publishing with QoS 0
- Publishing with QoS 1
- Publishing with QoS 2
- Subscribing

Tip: The IBM developerWorks website has the complete MQTT protocol specification at:

http://www.ibm.com/developerworks/webservices/library/ws-mqtt/index.html

B.1 MQTT concepts

In this section we describe the following concepts of the MQ Telemetry Transport protocol:

- Quality of service levels and flows
- QoS determination
- QoS Impact on performance
- Client identifier
- Durable and non-durable subscribers with MQTT
- Understanding MQTT persistence
- Message overhead
- Keep alive handling
- Retry message delivery
- Last will and testament
- Retained Flag on Messages
- TCP/IP overhead
- Message format
- Command messages

B.1.1 Quality of service levels and flows

Quality of service (QoS) levels should be understood by every user of MQTT. A QoS value determines how each message will be delivered and must be specified for every message sent by means of MQTT.

A real-life comparison can be made with letters sent through a postal service. An informal letter to a friend can be dropped into a mailbox and the sender might never think about it again. The sender expects the letter will get to its destination, but there are no significant consequences if it does not arrive. In contrast, a letter to which the recipient must respond is more important, so the sender might choose a higher level of delivery service that provides proof that the letter was received at its destination. In each situation, the choices made by the sender of the letter are comparable to choosing QoS levels in MQTT.

It is important to choose the proper QoS value for every message, because this value determines how the client and the server will communicate to deliver the message.

QoS 0: At most once delivery

It is termed as “At most once message delivery” as messages are delivered according to the best effort of the underlying network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the broker either once or not at all. This is the least level of QoS. The MQTT client or the server would just attempt sending the message without waiting for any kind of receipt. There are no steps taken to ensure message delivery other than the features provided by the TCP/IP layer. Also, there is no retry attempted by the MQTT layer if there is a failure to deliver the message. Hence if the client is sending a message it can arrive at the broker once or never. A QoS 0 message can get lost if the client unexpectedly disconnects or if the server fails. From a performance perspective this adds value because it is the fastest way to send a message using MQTT.

The MQTT command message used is PUBLISH. No other command messages flow for the QoS 0 messages.

Table B-1 on page 213 shows the QoS level 0 protocol flow.

Table B-1 QoS level 0 protocol flow

Client	Message and direction	Broker
QoS = 0	PUBLISH ---------->	**Action**: Publish message to subscribers

QoS 1: At least Once Delivery

The message is delivered at least once. The MQTT client or the server would attempt to deliver the message at least once, but there can be a duplicate message. The receipt of a message by the broker is acknowledged by a PUBACK message. If there is an identified failure of either the communications link or the sending device, or the acknowledgment message is not received after a specified period of time, the sender resends the message with the DUP bit set in the message header. The message arrives at the broker at least once. Both SUBSCRIBE and UNSUBSCRIBE messages use QoS level 1.

If the client does not receive a PUBACK message (either within a time period defined in the application, or if a failure is detected and the communications session is restarted), the client resends the PUBLISH message with the DUP flag set. When it receives a duplicate message from the client, the broker republishes the message to the subscribers, and sends another PUBACK message. At the client side, the implementations of the MQTT protocol also provide an additional feature known as the MQTT persistence. The MQTT persistence layer is not described in the specification of MQTT but is normally available with MQTT client implementations. When a QoS 1 message is published to the server, the client would need to wait for the acknowledgement to arrive. There can be a program termination or a crash at the client device. When the client is started again it will need to resume from the point it left before the crash. Hence the message is stored in a persistence layer such as disk and retrieved soon after the reconnecting back to the broker.

The MQTT command messages used are PUBLISH and PUBACK. While the publish happens the message will be logged to the MQTT persistence and removed when PUBACK is received. A message with QoS level 1 has a Message ID in the message header.

Table B-2 shows the QoS level 1 protocol flow.

Table B-2 QoS level 1 protocol flow

Client	Message and direction	Broker

QoS = 1
DUP = 0
Message ID = x
	PUBLISH
	---------->
		Actions:
		-- Store message in database
		-- Publish message to subscribers

Action: Discard message	PUBACK <----------	

QoS 2: Exactly once delivery

This is the highest level of QoS. Additional protocol flows above QoS level 1 ensure that duplicate messages are not delivered to the receiving application. The message is delivered once and only once when QoS 2 is used. The MQTT client or the server will ensure that the message is sent only once. This QoS must be used only when duplicate messages are not desired. From a performance perspective, there is a price to be paid in terms of network traffic and processing power.

The MQTT command messages used are PUBLISH, PUBREC, PUBREL and PUBCOMP. The message is sent in the PUBLISH flow and the client will store that message in the MQTT persistence layer if used. The message will remain locked on the server. PUBREC is sent by the server in response to PUBLISH. PUBREL will be dispatched to the server from the client in response to PUBREC. Once PUBREL is received by the broker it can dispatch the messages to the subscribers and send back PUBCOMP to the PUBREL. A message with QoS level 2 has a Message ID in the message header.

Table B-3 shows the QoS level 2 protocol flow.

Table B-3 QoS level 2 protocol flow

Client	Message and direction	Broker

QoS = 2
 	
DUP = 0
 	
Message ID = x
	PUBLISH
	
	---------->
		Action: Store message in database

	PUBREC <----------	Message ID = x
Message ID = x	PUBREL ---------->	**Actions:** -- Update database -- Publish message to subscribers
Action: Discard message	PUBCOMP <----------	Message ID = xb

If a failure is detected, or after a defined time period, each part of the protocol flow is retried with the DUP bit set. The additional protocol flows ensure that the message is delivered to subscribers once only.

Because QoS1 and QoS2 indicate that messages must be delivered, the broker stores messages in a database. If the broker has problems accessing this data, messages might be lost.

Assumptions for QoS levels 1 and 2

In any network, it is possible for devices or communication links to fail. If this happens, one end of the link might not know what is happening at the other end; these are known as in doubt windows. In these scenarios assumptions have to be made about the reliability of the devices and networks involved in message delivery.

WebSphere MQ Telemetry Transport assumes that the client and broker are generally reliable, and that the communications channel is more likely to be unreliable. If the client device fails, it is typically a catastrophic failure, rather than a transient one. The possibility of recovering data from the device is low. Some devices have non-volatile storage, for example flash ROM. The provision of more persistent storage on the client device protects the most critical data from some modes of failure.

Beyond the basic failure of the communications link, the failure mode matrix becomes complex, resulting in more scenarios than the specification for WebSphere MQ Telemetry Transport can handle.

The time delay (retry interval) before resending a message that has not been acknowledged is specific to the application, and is not defined by the protocol specification.

B.1.2 QoS determination

When the client subscribes to the topic it will specify a QoS at which it would like to receive messages from the server. Let us consider a scenario where the publisher A is sending the messages to the topic at QoS 2. A subscriber can subscribe at QoS 0 and then the messages to client will be delivered with QoS 0. The QoS value is further used in SUBSCRIBE and UNSUBSCRIBE requests. A subscribe request from the client is a QoS 1 request, the server will respond with SUBACK, which will ensure that the subscription has happened.

B.1.3 QoS impact on performance

There is a simple rule when considering performance impact of QoS. It is: “The higher the QoS, the lower the performance.” Let us evaluate performance corresponding with higher QoS. Suppose the time taken for sending a PUBLISH message is pt. If QoS is used, the total time taken to transfer n number of messages will be npt. Now in case of QoS 1, the PUBACK message (that is reply for the PUBLISH message) will flow from server to client. This is a 2-byte message and might take a lot less time than pt, hence call it mt. So the time taken for transferring n messages will be n(pt + mt). And for QoS 2, the PUBREC, PUBREL and PUBCOMP messages would be flowing. Hence the n number of messages would take approximately n(pt + 3mt). So if 10 messages need to be transferred from client to server and pt is 1 second and mt is 0.4 seconds, a QoS 0 message would take 101 = 10 seconds, QoS 1 message would take 10(1 + 0.4) which is 14 seconds and QoS 2 message would take 22 seconds.

B.1.4 MQTT client identifier

MQTT protocol defines a “client identifier” (client ID) that uniquely identifies a client in a network. In simple terms, when connecting to a server a client needs to specify a unique string that is not used currently and will not be used by any other client that will connect to the MQTT server. There are several ways of choosing a client identifier. Here are some examples:

- A Sensor installed in a particular location can use location code as the client ID.
- A mobile device having network ability can choose the MAC address or some unique device ID as the client ID

But MQTT restricts the client ID length to 23 characters. Hence there will be a situation when the client ID needs to be shortened. While shortening the client identifier, it needs to be ensured that the client ID is not the same as any other client ID used in the network. In order to keep the identifier short and unique, you should introduce a reliable identifier generation mechanism. For instance, you might create a client identifier from the 48-bit device MAC address. If transmission size is not a critical issue, you might use the remaining 17 bytes to make the address easier to administer, like some human readable text in the identifier.

Now let us try to understand the implications of two clients getting the same client identifier. The MQTT server keeps track of the pending messages to be sent to a client based on the client identifier. Thus, if a client has been using QoS 1 or QoS 2 and subscribed to any topic and disconnected from the server, the server will save the messages that arrived for the client while it was disconnected. Once the client reconnects, the server will send those messages to the client. If some other MQTT device uses the same client ID and connects to the server, the server will send the messages it had saved to it.

Another scenario related to the client identifiers is the duplicate connections. Let us say a particular device using client ID DeviceA is connected to the broker. If another client comes with the same client ID DeviceA, the server can decide if it has to allow the new client to connect and disconnect the existing client or keep the old connection alive and disallow the new client. This is an optional feature of an MQTT server.

B.1.5 Durable and non-durable subscribers with MQTT

A durable subscriber in any client can receive all the messages published on a topic, including messages published while the client was inactive. Defining inactive is important at this point. A client is said to be inactive in the following cases:

- A client has connected and subscribed to a topic and later disconnected with the server without unsubscribing to that topic.
- A client has connected and subscribed to a topic and later disappeared from the network due to some network issue or lost connection to the server.

A non-durable subscriber is any client that does not intend to receive messages published while it is inactive. A client is said to be non-durable in the following cases:

- A client always uses the clean session flag set to true when it connects to the server.
- A client always unsubscribes for all the topics it subscribed for, before disconnecting.

In MQTT, QoS 0 messages are never persisted while a durable subscriber is inactive. Only QoS 1 and QoS 2 messages are persisted by the server and sent when the client becomes active. MQTT further provides a facility that allows MQTT clients to refrain from receiving messages while they were disconnected by setting the clean start flag to true.

Here is an example flow for a typical durable MQTT subscriber:

1. MQTT ClientA connects to the server by specifying the clean session flag as false.
2. ClientA subscribes to the topic topic/test.
3. MQTT ClientB connects to the server and publishes messages of QoS 1 and QoS 2 to topic topic/test.
4. ClientA receives them.
5. ClientA disconnects from the server (note ClientA did not unsubscribe).
6. ClientB publishes more messages of QoS1 and QoS 2.
7. ClientA connects to the server by specifying the clean session flag as false.
8. ClientA receives all the messages that were published while it was inactive and all the future messages that will be published to the topic topic/test.

Durable subscribers are useful when the subscribing client needs all the messages that are published to the topic it subscribes to. So a client needs not really worry about being on the network because it is assured to receive the messages while it was disconnected or even lost connection.

While durable subscriptions is a nice feature to have, it adds additional responsibility to the server to hold the messages until the client connects back to the server.

B.1.6 Understanding MQTT persistence

MQTT protocol is designed with the assumption that the client and server are generally reliable. What this means is that the client machine and the server machine will not crash or hang or get into power failure issues. The assumption may prove to be costlier in some situations for the clients and hence a feature known as MQTT client persistence is provided by the implementations. It is used with QoS 1 and QoS 2 message flows. Here is how it works:

- Before the client actually sends the PUBLISH command over the wire, it will store it on disk or any other volatile storage.
- The client, when it receives an acknowledgement, deletes the message from the disk. Thus, in case of a power failure and restart of the client application, the first action soon after the reconnect is to check the pending messages in the client persistence and send them. On the server side, this feature is generally managed by the messaging engine itself.

B.1.7 MQTT message overhead

MQTT has a very low message overhead. Let us try to understand how many additional bytes of data will flow from client to server for one of the scenarios. Consider a scenario where a client needs to send a message, say "Hello" 10 times. Additionally, the client needs to tell the server that the destination is "a/b" for every message.

For sending it using MQTT here is the computation of total number of bytes:

1. CONNECT -> Fixed (2 bytes) + Variable (12 bytes) = 14 bytes
2. CONNACK -> Fixed (2 bytes) + Variable (2 bytes) = 4 bytes
3. PUBLISH -> Fixed (2 bytes) + Variable:

   Length(a/b) = 3.

   So 2 bytes for representing the topic name, followed by topic name, followed by msg ID, followed by payload, which is 2 + 3 + 2 + 5 = 12 bytes.
4. There are 10 publications happening... hence (12) 10 = 120 bytes
5. DISCONNECT - Fixed (2 bytes) = 2 bytes

   Hence this scenario needs 14 + 4 + 120 + 2 = 140 bytes.

As you can see from the above example, the overhead for every message is fixed at 2 bytes. And the total overhead is 140 - 80 = 60 bytes for this scenario.

B.1.8 MQTT keep alive handling

You might be wondering how an MQTT Server would know if the MQTT client is still on the network and vice-versa. That is the *keep alive* timer on the client and server. The TCP/IP timeout and error handling mechanisms are at the network layer, but the MQTT client and server need not depend on that. The client says hello to the server and the server responds back acknowledging it. That is how simple keep alive works.

But the interesting question is when should a client tell the server that it is alive. As per MQTT it is only when there has not been any interaction between them for some period of time. The time period is a configurable option. This option can the set while connecting to the server by the client. It is the client that will choose the keep alive time, the server just keeps a record of the value in the client information table on the server side.

There are two messages that constitute the keep alive interaction, PINGREQ and PINGRESP. The PINGREQ, the ping request, is sent by the client to the server when the keep alive timer expires. The PINGRESP, the ping response, is the reply sent by the server for a ping request from the client. Both these messages are short 2-byte messages. They are not associated with any QoS. The PINGREQ is sent only once and the client waits for the PINGRESP.

Internally there is a simple timer that triggers in the client It will check if there has been any other recent activity between the client and the server. If not, a PINGREQ is sent. The client will wait until a PINGRESP is received. On the server side also, if there is no message received from the client within the keep alive time period, it disconnects the client. But the server will offer a grace period to the client, that is an additional 50% of the keep alive time period.

When choosing a keep alive time for a client there are some considerations. If the MQTT client is less active and sends one or two messages in an hour's time and if the keep alive is set to a very low value, say 10 seconds, the network would be flooded with PINGREQ and PINGRES messages. On the other hand, if the keep alive timer is set to a high value, say 1 hour and the client goes out of the network, the server will not know that the client has gone away for a long time. This would affect administrators who monitor the connected clients and also the server would keep retrying PUBLISH messages. So based on the message flow and amount of time the client can be idle, the keep alive timer needs to be chosen.

B.1.9 Retry message delivery

Let us consider a scenario when a message of QoS 1 is published from a client to a server but the PUBACK is not received from the server, then the MQTT client will retry sending the message after a specified time period. This timeout is different from the keep alive. The retry timeout is the maximum amount of time the client will wait for the server to send a response or vice-versa. This retry will happen until there is an error from the TCP/IP layer. That is, if there is any type of socket exception, then retry processing ceases. In cases where QoS 2 message delivery is not used there could be duplicate messages seen due to retry by the MQTT client or server. Hence when a retry happens the message will be marked with a duplicate flag. The application receiving the message will know if the received message is a duplicate or not.

As a user of an MQTT client it would be possible that a duplicate message is received while using QoS 1 due to MQTT retry. On some slow and fragile networks, one can observe a large number of duplicate messages due to the retry processing. If the retry timeout is very low, the network can be clogged with a large number of duplicates.

B.1.10 Last will and testament

When a client connects to the server it can define a topic and a message that needs to be published automatically when it unexpectedly disconnects. Let us now try to understand this feature in detail and also how it can be used. Firstly, an unexpected disconnect will happen when the client does not send a DISCONNECT command to the server and goes out of the network. This can happen due to several reasons including loss in network connection or termination of the client program.

When a client connects to the server, the following parameters can be specified optionally:

1. The will topic at which the will message needs to be published.
2. The will message that needs to be published.
3. The will QoS of the message that will get published on the topic.
4. The will retain flag that signifies if the message should be retained.

We have already learned about the keep alive feature of MQTT. When the client unexpectedly disconnects, the keep alive timer at the server side will detect that the client has not sent any message or the keep alive PINGREQ. Hence it will immediately publish the Will message on the Will topic specified by the client.

The LWT feature can be very useful in some scenarios as a basis for the design of the solution. If the fire alarms or the sensors in the field are equipped with an MQTT client, this feature can be used to detect when the device goes out of the network. Normally in a solution there would be health monitors that give a complete overview of the health of the running devices. The LWT feature can be used to create notifications on the health monitor so that the user can take action.

There could be a case when the health monitor that needs to know about the unexpected disconnection of the client is itself not connected to the server. That is when retained messages can be used for LWT. The broken Will retains the last published messages and immediately sends the Will message when the subscription happens. There are cases when the health monitor itself unexpectedly disconnects and reconnects back. If QoS 1 or 2 messages are used for LWT those messages can be received.

B.1.11 Retained flag on messages

MQTT provides a feature of holding a message for a topic even after it is delivered to the connected subscribers. This is achieved through a “retained” flag on the publish message. The publisher of the message sets this flag on the message while publishing. Here is an example flow to understand retained messages:

1. Client A connects to the server and subscribes to topic “a/b”.
2. Client B connects to the server and publishes the message “Hello” with Retain flag to “a/b”.
3. Client A receives the message without Retain flag set.
4. Client C connects to the server and subscribes to “a/b”.
5. Client C receives the message with Retain flag set.

Even if the server is restarted, the retained message will not be lost. It is also important to note that there is only one retained message that is held per topic. The retained publications are primarily used to maintain state information. If a particular topic is used to publish a state message from a device, the messages can be retained messages. The advantage is that the new monitoring program that connects can subscribe to this topic and gets to know the last published state messages from the device.

B.1.12 TCP/IP overhead

When designing applications for networks such as cellular networks the TCP/IP overhead should be considered. MQTT is just like any other protocol such as HTTP or FTP for the TCP/IP stack. TCP/IP is the communication protocol of the Internet. It defines how the client and server need to communicate with each other. All the major operating systems on the server support TCP/IP. The client devices also come with the operating system that has built-in capability of TCP/IP. Although MQTT adds only 2 bytes of additional header to a message, there is more data that flows on the network layers. The additional header data cannot be exactly computed on a per MQTT message basis. IP header, frame header and other keep alive flows at the network layer add to additional data flows while using MQTT clients.

B.1.13 WebSphere MQ Telemetry Transport message format

The message header for each WebSphere MQ Telemetry Transport command message contains a fixed header. Some messages also require a variable header and a payload.

WebSphere MQ Telemetry Transport fixed header

The message header for each WebSphere MQ Telemetry Transport command message contains a fixed header. Table B-4 shows the fixed header format.

Table B-4 Fixed header format

Bit	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
byte 1	Message type				DUP flag	Qos level		RETAIN
byte 2	Remaining length							

WebSphere MQ Telemetry Transport payload

The following types of WebSphere MQ Telemetry Transport command message have a payload in the message header:

- CONNECT

  The payload contains one or three UTF-8 encoded strings. The first string uniquely identifies the client to the broker. The second string is the Will topic, and the third string is the Will message. The second and third strings are present only if the Will flag is set in the CONNECT Flags byte.
- SUBSCRIBE

  The payload contains a list of topic names to which the client can subscribe, and the QoS level. These strings are UTF-encoded.
- SUBACK

  The payload contains a list of granted QoS levels. These are the QoS levels at which the administrators for the broker have permitted the client to subscribe to a particular topic name. Granted QoS levels are listed in the same order as the topic names in the corresponding SUBSCRIBE message.

The payload part of a PUBLISH message contains application-specific data only. No assumptions are made about the nature or content of the data, and this part of the message is treated as a BLOB.

If you want an application to apply compression to the payload data, you need to define in the application the appropriate payload flag fields to handle the compression details. You cannot define application-specific flags in the fixed or variable headers.

WebSphere MQ Telemetry Transport variable header

The message header for some types of WebSphere MQ Telemetry Transport command message contains a variable header. It resides between the fixed header and the payload.

The format of the variable header fields are described in the following topics, in the order in which they must appear in the header:

- Protocol name
- Protocol version
- Connect flags
- Keep alive timer
- Connect return code

- Topic name
- Message identifier

The variable length Remaining Length field is not part of the variable header. The bytes of the Remaining Length field do not contribute to the byte count of the Remaining Length value. This value only takes account of the variable header and the payload.

WebSphere MQ Telemetry Transport command messages

Here is the list of supported WebSphere MQ Telemetry Transport command messages:

- CONNACK

  Acknowledges connection request. The CONNACK message is the message sent by the broker in response to a CONNECT request from a client.

- CONNECT

  A client requests a connection to a broker. When a TCP/IP socket connection is established between the client and the broker, a protocol level session is required. It is assumed that the direction of connection is client to broker, and that the client supports broker listener functionality.

- DISCONNECT

  This is the disconnection notification. The DISCONNECT message is sent from the client to the broker to indicate that it is about to close its TCP/IP connection. This allows for a clean disconnection, rather than just dropping the line. Sending the DISCONNECT message does not affect existing subscriptions. They are persistent until they are either explicitly unsubscribed, or if there is a clean start. The broker retains QoS 1 and QoS 1 messages for topics to which the client is unsubscribed until the client reconnects. QoS 0 messages are not retained, since they are delivered on a best efforts basis.

- PINGREQ

  PING request. The PINGREQ message is an “are you alive” message that is sent from, or received by, a connected client.

- PINGRESP

  PING response. A PINGRESP message is the response to a PINGREQ message and means “yes I am alive”. Keep Alive messages flow in either direction, sent either by a connected client or the broker.

- PUBACK

  Publish acknowledgement. A PUBACK message is the response to a PUBLISH message with QoS level 1. A PUBACK message is sent by a broker in response to a PUBLISH message from a publishing client, and by a subscriber in response to a PUBLISH message from the broker.

- PUBCOMP

  Assured publish complete. This message is either the response from the broker to a PUBREL message from a publisher, or the response from a subscriber to a PUBREL message from the broker. It is the fourth and last message in the QoS 2 protocol flow.

- PUBLISH

  Publish message. A PUBLISH message is sent by a client to a broker for distribution to interested subscribers. Each PUBLISH message is associated with a topic name (also known as the Subject or Channel). This is a hierarchical name space that defines a taxonomy of information sources for which subscribers can register an interest. A message that is published to a specific topic name is delivered to connected subscribers for that topic. To maintain symmetry, if a client subscribes to one or more topics, any message published to those topics is sent by the broker to the client as a PUBLISH message.

- PUBREC

  Assured publish received. A PUBREC message is the response to a PUBLISH message with QoS level 2. It is the second message of the QoS level 2 protocol flow. A PUBREC message is sent by the broker in response to a PUBLISH message from a publishing client, or by a subscriber in response to a PUBLISH message from the broker.

- PUBREL

  Assured publish release. A PUBREL message is the response either from a publisher to a PUBREC message from the broker, or from the broker to a PUBREC message from a subscriber. It is the third message in the QoS 2 protocol flow.

- SUBACK

  Subscription acknowledgment. A SUBACK message is sent by the broker to the client to confirm receipt of a SUBSCRIBE message. A SUBACK message contains a list of granted QoS levels. These are the levels at which the administrators for the broker permit the client to subscribe to a specific topic name. In the current version of the protocol, the broker always grants the QoS level requested by the subscriber. The order of granted QoS levels in the SUBACK message matches the order of the topic Names in the corresponding SUBSCRIBE message.

- SUBSCRIBE

  Subscribe to named topics. The SUBSCRIBE message allows a client to register an interest in one or more topic names with the broker. Messages published to these topics are delivered from the broker to the client as PUBLISH messages. The SUBSCRIBE message also specifies the QoS level at which the subscriber wants to receive published messages.

- UNSUBACK

  Unsubscribe acknowledgment. The UNSUBACK message is sent by the broker to the client to confirm receipt of an UNSUBSCRIBE message.

- UNSUBSCRIBE

  Unsubscribe from named topics. An UNSUBSCRIBE message is sent by the client to the broker to unsubscribe from named topics.

For more details about message format and command messages, see:

`http://publib.boulder.ibm.com/infocenter/wmbhelp/v6r0m0/index.jsp?topic=%2Fcom.ibm.etools.mft.eb.doc%2Fac10840_.htm`

Set up for tracing

In this section we describe how to configure tracing with Wireshark and the WMQTT Utility.

B.1.14 Wireshark

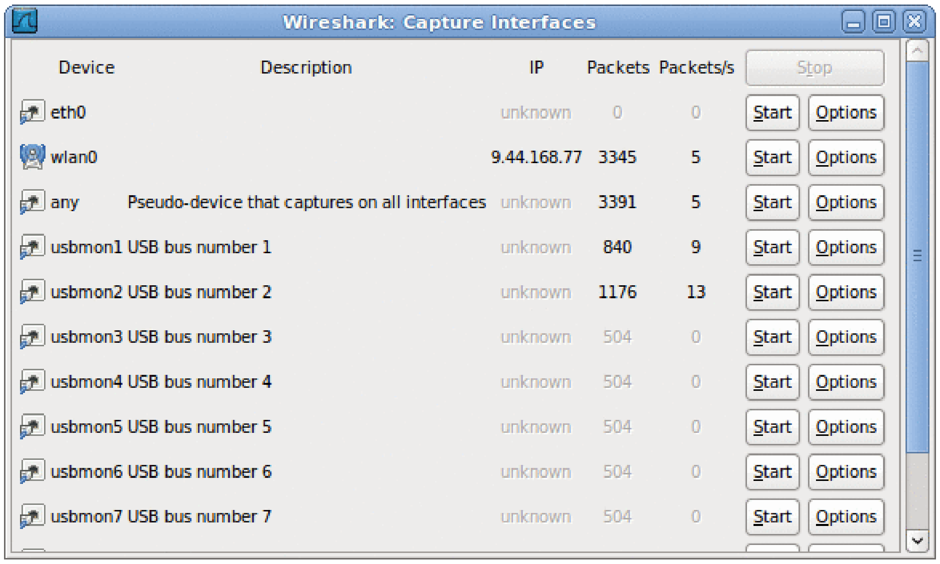
To initiate the trace, download and install the Wireshark product. The remainder of this section is written with Wireshark installed on a Linux system (Ubuntu 10.04) that communicates with an MQTT server using a wireless network. The network interface that is traced is *wlan0*, which in this book is the name of the interface.

To download and install Wireshark:

1. Set up an ID that has access to run the trace.
2. The tracing is done using the Wireshark GUI. If you are using the root user ID, start the GUI with the **`gksudo`** command, which is documented at:

   `https://help.ubuntu.com/community/RootSudo#Graphical_sudo|gksudo`

   Type `gksudo wireshark` on the command line, and press Enter.
3. To see the list of interfaces that are available in the Wireshark GUI, go to **Capture** → **Interfaces**. See Figure B-1. Do *not* click **Start** yet.

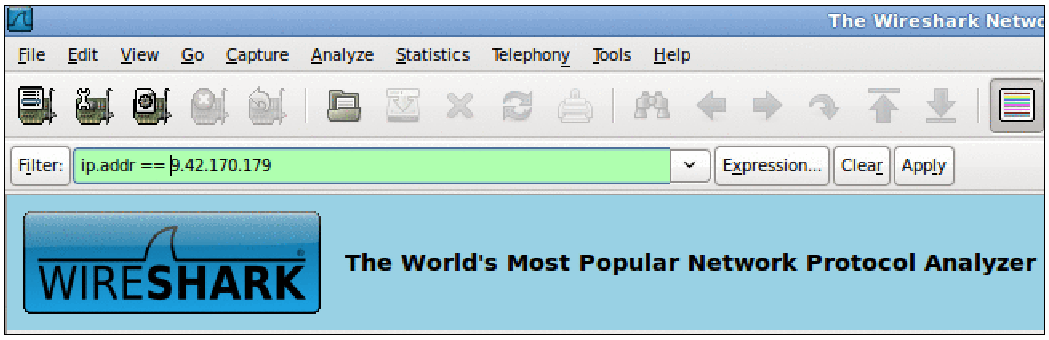


Device	Description	IP	Packets	Packets/s	Stop
eth0		unknown	0	0	Start Options
wlan0		9.44.168.77	3345	5	Start Options
any	Pseudo-device that captures on all interfaces	unknown	3391	5	Start Options
usbmon1	USB bus number 1	unknown	840	9	Start Options
usbmon2	USB bus number 2	unknown	1176	13	Start Options
usbmon3	USB bus number 3	unknown	504	0	Start Options
usbmon4	USB bus number 4	unknown	504	0	Start Options
usbmon5	USB bus number 5	unknown	504	0	Start Options
usbmon6	USB bus number 6	unknown	504	0	Start Options
usbmon7	USB bus number 7	unknown	504	0	Start Options

Figure B-1 Available interfaces

For the sake of brevity, we also add a filter in this example so that only those packets that are destined to the server are traced (Figure B-2):

1. In the main Wireshark window, click **Filter**.
2. Enter the IP address of the MQTT server.
3. Click **Apply**.


The Wireshark Netwo
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Expression... Clear Apply
WIRESHARK
The World's Most Popular Network Protocol Analyzer

Filter: ip.addr == 9.42.170.179

Figure B-2 Add a filter

B.1.15 WMQTT Utility: IA92 SupportPac

We set up the IA92 client for the trace. Be sure that the server IP address matches the server IP address that you enter for the filter. In the Options tab of the WMQTT Utility, shown in

Figure B-3, enter a name for the Client ID. After you enter the server IP address, port number, and the client ID, the setup is completed. Do *not* click **Connect** yet.

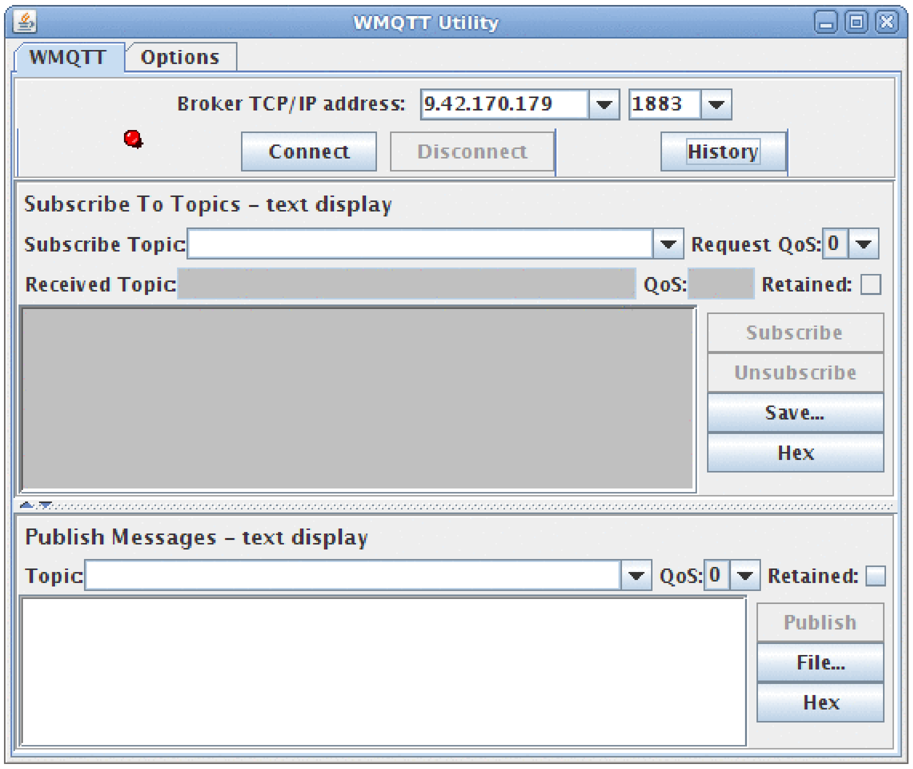


Figure B-3 Set up the IA92 client for the trace

B.2 Connect to the MQTT server

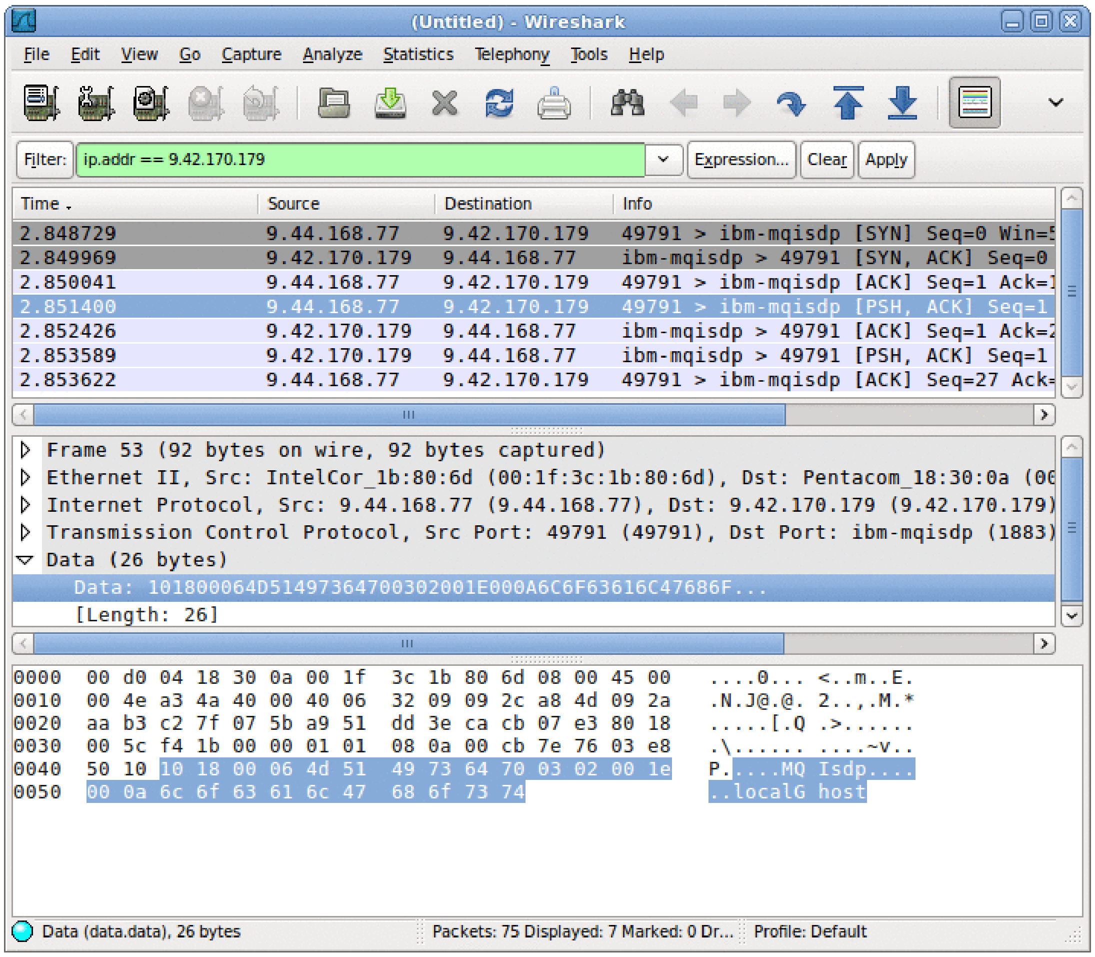
With the setup complete, you can begin the trace when the client connects to the server:

1. In the Options tab of the WMQTT Utility, ensure that the name of the client identifier is set to localGhost.
2. In the Options tab of the WMQTT Utility, ensure that the **Clean session** option is selected.
3. In the Options tab of the WMQTT Utility, set the number of seconds for the Keep Alive option to 30.
4. Click **Connect** in the WMQTT Utility.
5. To start the Wireshark trace, go to **Capture** → **Interfaces**. Be sure to apply the filter. Then, click **Start** to initiate the trace.

The remainder of this appendix uses Wireshark and the WMQTT Utility to show messages sent on the MQTT protocol. We start by looking at the CONNECT and CONNACK messages.

B.2.1 The CONNECT message

The client sends a CONNECT message to the server to initiate the connection as shown in the Wireshark trace in Figure B-4. The data flowing from the client to the server displays in the bottom pane of the window in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply

```
Filter: ip.addr == 9.42.170.179

Time           Source          Destination     Info
2.848729       9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [SYN] Seq=0 Win=5
2.849969       9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [SYN, ACK] Seq=0
2.850041       9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [ACK] Seq=1 Ack=1
2.851400       9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [PSH, ACK] Seq=1
2.852426       9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [ACK] Seq=1 Ack=2
2.853589       9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [PSH, ACK] Seq=1
2.853622       9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [ACK] Seq=27 Ack=

▷ Frame 53 (92 bytes on wire, 92 bytes captured)
▷ Ethernet II, Src: IntelCor_1b:80:6d (00:1f:3c:1b:80:6d), Dst: Pentacom_18:30:0a (00
▷ Internet Protocol, Src: 9.44.168.77 (9.44.168.77), Dst: 9.42.170.179 (9.42.170.179)
▷ Transmission Control Protocol, Src Port: 49791 (49791), Dst Port: ibm-mqisdp (1883)
▽ Data (26 bytes)
    Data: 101800064D514973647003022001E000A6C6F63616C47686F...
    [Length: 26]

0000  00 d0 04 18 30 0a 00 1f  3c 1b 80 6d 08 00 45 00   ....0... <..m..E.
0010  00 4e a3 4a 40 00 40 06  32 09 09 2c a8 4d 09 2a   .N.J@.@. 2..,.M.*
0020  aa b3 c2 7f 07 5b a9 51  dd 3e ca cb 07 e3 80 18   .....[.Q .>......
0030  00 5c f4 1b 00 00 01 01  08 0a 00 cb 7e 76 03 e8   .\...... ....~v..
0040  50 10 10 18 00 06 4d 51  49 73 64 70 03 02 00 1e   P.....MQ Isdp....
0050  00 0a 6c 6f 63 61 6c 47  68 6f 73 74               ..localG host

Data (data.data), 26 bytes    Packets: 75 Displayed: 7 Marked: 0 Dr...    Profile: Default
```

Figure B-4 Wireshark trace

Table B-5 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html#connect|CONNECT

Table B-5 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	0	0	1	0	0	0	0	Message Type 1 (upper nibble) denoting a connection request. Lower nibble is not used.
2	0	0	0	1	1	0	0	0	Remaining length 0x18 or 24 bytes
3	0	0	0	0	0	0	0	0	Length Most Significant Byte - 0
4	0	0	0	0	0	1	1	0	Length Least Significant Byte - 6
5	0	1	0	0	1	1	0	1	M

Byte	7	6	5	4	3	2	1	0	Remarks
6	0	1	0	1	0	0	0	1	Q
7	0	1	0	0	1	0	0	1	I
8	0	1	1	1	0	0	1	1	s
9	0	1	1	0	0	1	0	0	d
10	0	1	1	1	0	0	0	0	p
11	0	0	0	0	0	0	1	1	Protocol version - 3
12	0	0	0	0	0	0	1	0	Connect flags - 0x02 User name - 0 (Not set) Password - 0 (Not set) Will retain - 0 (Not set) Will QoS - 00 (Not set) Will flag - 0 (Not set) Clean session - 1 (yes) Reserved - 0
13	0	0	0	0	0	0	0	0	Keep Alive time - Most Significant Byte - 0
14	0	0	0	1	1	1	1	0	Keep Alive time - Least Significant Byte - 0x1e - 30 seconds
15	0	0	0	0	0	0	0	0	Client Identifier length - Most Significant Byte - 0
16	0	0	0	0	1	0	1	0	Client Identifier length - Least Significant Byte - 0x0a - 10
17	0	1	1	0	1	1	0	0	l
18	0	1	1	0	1	1	1	1	o
19	0	1	1	0	0	0	1	1	c
20	0	1	1	0	0	0	0	1	a
21	0	1	1	0	1	1	0	0	l
22	0	1	0	0	0	1	1	1	G
23	0	1	1	0	1	0	0	0	h
24	0	1	1	0	1	1	1	1	o
25	0	1	1	1	0	0	1	1	s
26	0	1	1	1	0	1	0	0	t

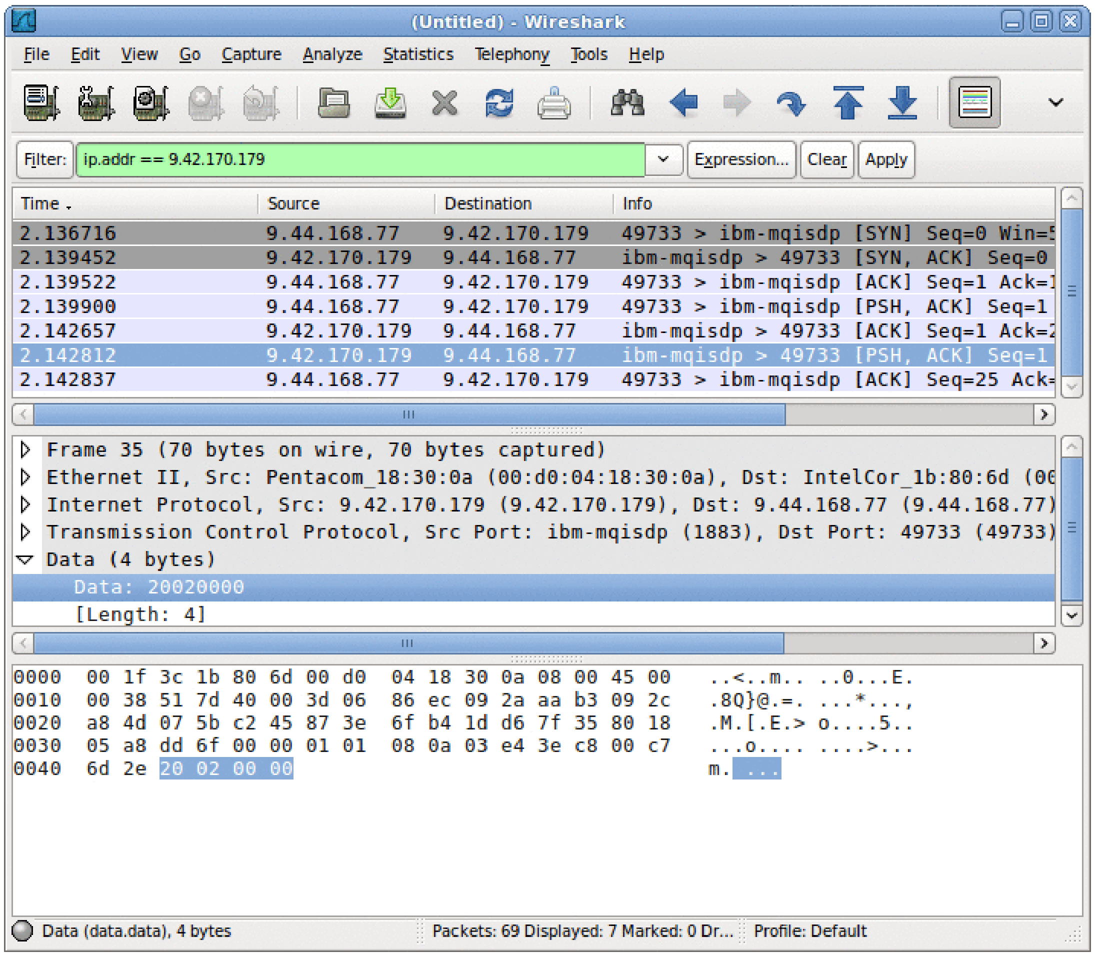
The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload. These categories are:

1. The first two bytes make up the fixed header of the message.
   a. The first byte has the message type as 1 denoting a CONNECT message.
   b. The second byte has the remaining length of the message. The trace shows a total of 26 bytes of which the first two are for the fixed header. Therefore, the balance is 24 bytes or 0x18 in byte 2.

2. Then, we have the variable header that is broken up as follows:
   a. The first two bytes (3 and 4 in the table) contain the length of the protocol name as 6 bytes.
   b. The next six bytes (5 thru 10) are the actual name of the protocol - MQIsdp.
   c. The 11th byte stores the protocol version - 3.
   d. The 12th byte holds the connection flags with each bit of this byte denoting if a connection flag is turned ON or OFF. Since in our example we have set the Clean session flag as ON and others are not set, the value of this byte is 0x02.
   e. The next two bytes comprise the number of seconds for Keep Alive of the client. Since this was set as 30 seconds in the WMQTT Utility, it is seen here as 0x1e.
3. The rest of the bytes describe the length and the actual value of the client identifier.
   a. The first two bytes (15 and 16 in the table) contain the length of the client identifier as 10 bytes.
   b. The rest of the bytes (17 thru 26) are the actual client identifier - localGhost as set in the GUI.
4. The payload does not exist for a CONNECT message.

B.2.2 The CONNACK message

The server responds to the connection request from the client with a CONNACK message as shown in the Wireshark trace in Figure B-5. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time      Source          Destination     Info
2.136716  9.44.168.77     9.42.170.179    49733 > ibm-mqisdp [SYN] Seq=0 Win=5
2.139452  9.42.170.179    9.44.168.77     ibm-mqisdp > 49733 [SYN, ACK] Seq=0
2.139522  9.44.168.77     9.42.170.179    49733 > ibm-mqisdp [ACK] Seq=1 Ack=1
2.139900  9.44.168.77     9.42.170.179    49733 > ibm-mqisdp [PSH, ACK] Seq=1
2.142657  9.42.170.179    9.44.168.77     ibm-mqisdp > 49733 [ACK] Seq=1 Ack=2
2.142812  9.42.170.179    9.44.168.77     ibm-mqisdp > 49733 [PSH, ACK] Seq=1
2.142837  9.44.168.77     9.42.170.179    49733 > ibm-mqisdp [ACK] Seq=25 Ack=
```

```
▷ Frame 35 (70 bytes on wire, 70 bytes captured)
▷ Ethernet II, Src: Pentacom_18:30:0a (00:d0:04:18:30:0a), Dst: IntelCor_1b:80:6d (00
▷ Internet Protocol, Src: 9.42.170.179 (9.42.170.179), Dst: 9.44.168.77 (9.44.168.77)
▷ Transmission Control Protocol, Src Port: ibm-mqisdp (1883), Dst Port: 49733 (49733)
▽ Data (4 bytes)
    Data: 20020000
    [Length: 4]
```

```
0000  00 1f 3c 1b 80 6d 00 d0  04 18 30 0a 08 00 45 00   ..<..m.. ..0...E.
0010  00 38 51 7d 40 00 3d 06  86 ec 09 2a aa b3 09 2c   .8Q}@.=. ...*...,
0020  a8 4d 07 5b c2 45 87 3e  6f b4 1d d6 7f 35 80 18   .M.[.E.> o....5..
0030  05 a8 dd 6f 00 00 01 01  08 0a 03 e4 3e c8 00 c7   ...o.... ....>...
0040  6d 2e 20 02 00 00                                  m. ...
```

Data (data.data), 4 bytes | Packets: 69 Displayed: 7 Marked: 0 Dr... | Profile: Default

Figure B-5 Wireshark trace

Table B-6 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-6 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	0	1	0	0	0	0	0	Message type 2 (upper nibble) denoting a connection acknowledgment. The lower nibble is not used.
2	0	0	0	0	0	0	1	0	Remaining length 0x02 or 2 bytes
3	0	0	0	0	0	0	0	0	Reserved byte - set to 0
4	0	0	0	0	0	0	0	0	Actual return code of the connection - 0

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

1. The first two bytes make up the fixed header of the message.

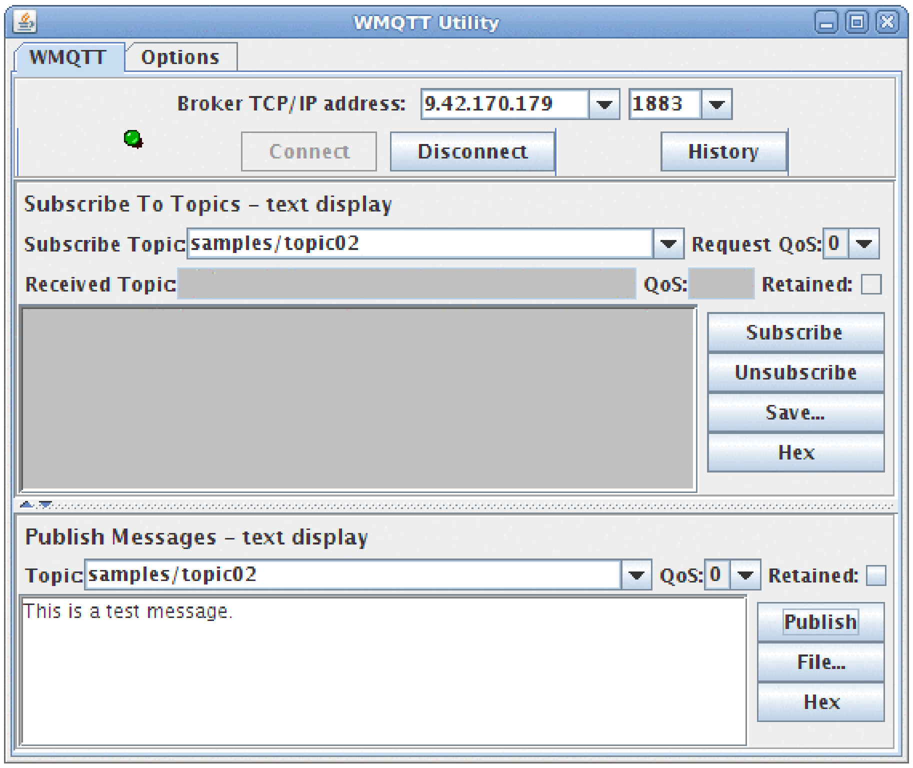
   a. The first byte has the message type as 2 denoting a CONNACK message.
   b. The second byte has the remaining length of the message as 2 bytes.
2. Then, we have the variable header that is broken up as follows.
   a. The first byte is Reserved and is therefore set to 0.
   b. The second byte has the return code. It is 0 denoting a successful connection.
3. The payload does not exist for a CONNACK message.

B.3 Publishing with QoS 0

After the client is connected to the server, you can publish as described here:

1. Set up Wireshark as described in "Set up for tracing" on page 222.
2. In the WMQTT Utility, enter a topic name, such as `samples/topic02`.
3. Enter a test message to be published, such as `This is a test message`.
4. Start the trace as described in B.2, "Connect to the MQTT server" on page 224.
5. Click **Publish** to publish this message.

Figure B-6 shows the client setup to publish this message.



Broker TCP/IP address: 9.42.170.179 1883

Subscribe Topic: samples/topic02 Request QoS: 0

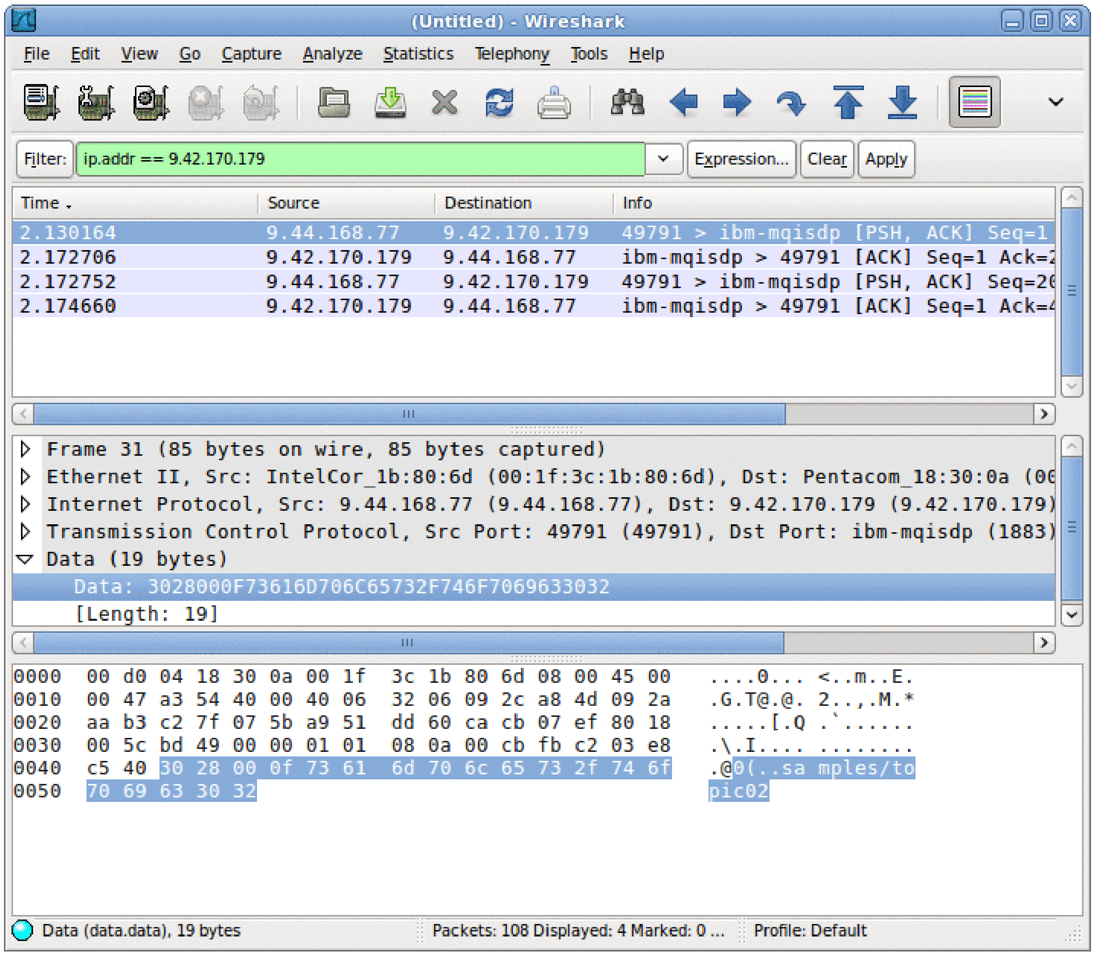
Received Topic: QoS: Retained:

Topic: samples/topic02 QoS: 0 Retained:

This is a test message.

Figure B-6 Client setup

The client does the publication by sending a PUBLISH message, as shown in the Wireshark trace in Figure B-7. The data flowing from the client to server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time      Source         Destination    Info
2.130164  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=1
2.172706  9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=2
2.172752  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=26
2.174660  9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=4
```

```
▷ Frame 31 (85 bytes on wire, 85 bytes captured)
▷ Ethernet II, Src: IntelCor_1b:80:6d (00:1f:3c:1b:80:6d), Dst: Pentacom_18:30:0a (00
▷ Internet Protocol, Src: 9.44.168.77 (9.44.168.77), Dst: 9.42.170.179 (9.42.170.179)
▷ Transmission Control Protocol, Src Port: 49791 (49791), Dst Port: ibm-mqisdp (1883)
▽ Data (19 bytes)
    Data: 3028000F73616D706C65732F746F7069633032
    [Length: 19]
```

```
0000  00 d0 04 18 30 0a 00 1f  3c 1b 80 6d 08 00 45 00   ....0... <..m..E.
0010  00 47 a3 54 40 00 40 06  32 06 09 2c a8 4d 09 2a   .G.T@.@. 2..,.M.*
0020  aa b3 c2 7f 07 5b a9 51  dd 60 ca cb 07 ef 80 18   .....[.Q .`......
0030  00 5c bd 49 00 00 01 01  08 0a 00 cb fb c2 03 e8   .\.I.... ........
0040  c5 40 30 28 00 0f 73 61  6d 70 6c 65 73 2f 74 6f   .@0(..sa mples/to
0050  70 69 63 30 32                                     pic02
```

Data (data.data), 19 bytes | Packets: 108 Displayed: 4 Marked: 0 ... | Profile: Default

Figure B-7 Wireshark trace

The length of the topic name is 15 bytes and the length of the test message is 21 bytes. Adding 2 bytes for the length that is stored as a prefix to the topic name and another 2 bytes as the prefix to the actual message, the total comes to 40 or 0x28 bytes as seen in the bytes table above. The actual message that is published does not display in the trace that is shown in Figure B-7. It displays in another Wireshark window. We do not show this window for the sake of brevity.

Table B-7 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-7 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	0	1	1	0	0	0	0	Message Type 3 (upper nibble) denoting a publication. The lower nibble (bits 1 and 2) is zero to denote QoS 0.
2	0	0	1	0	1	0	0	0	Remaining length 0x28 or 40 bytes
3	0	0	0	0	0	0	0	0	Length of topic name - Most Significant Byte - 0
4	0	0	0	0	1	1	1	1	Length of topic name - Least Significant Byte - 15

Byte	7	6	5	4	3	2	1	0	Remarks
5	0	1	1	1	0	0	1	1	s
6	0	1	1	0	0	0	0	1	a
7	0	1	1	0	1	1	0	1	m
8	0	1	1	1	0	0	0	0	p
9	0	1	1	0	1	1	0	0	l
10	0	1	1	0	0	1	0	1	e
11	0	1	1	1	0	0	1	1	s
12	0	0	1	0	1	1	1	1	/
13	0	1	1	1	0	1	0	0	t
14	0	1	1	0	1	1	1	1	o
15	0	1	1	1	0	0	0	0	p
16	0	1	1	0	1	0	0	1	i
17	0	1	1	0	0	0	1	1	c
18	0	0	1	1	0	0	0	0	0
19	0	0	1	1	0	0	1	0	2

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

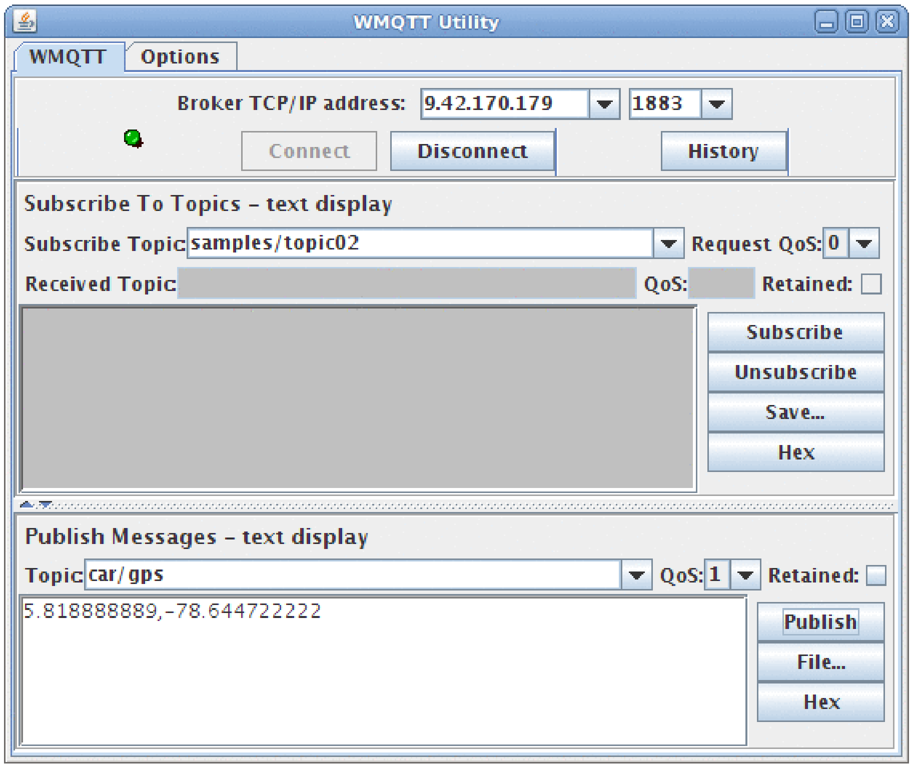
1. The first two bytes make up the fixed header of the message:
   a. The first byte has the message type as 3 denoting a PUBLISH message - bits 4 and 5 of upper nibble. It also denotes QoS as 0 in the bits 1 and 2 of the lower nibble.
   b. The second byte has the remaining length of the message as 0x28 or 40 bytes.
2. Then, we have the variable header that is broken up as follows.
   a. The first two bytes (3 and 4 in the table) contain the length of the topic name as 0x000f or 15 bytes.
   b. The next 11 bytes (5 thru 19) are the actual name of the topic, samples/topic02.
3. The rest of the message is the payload.
   a. The first two bytes (20-21) are the Message ID.
   b. The remaining bytes (22-42) is the actual message published.

B.4 Publishing with QoS 1

After the client is connected to the server, you can publish as described here:

1. Set up Wireshark as described in "Set up for tracing" on page 222.
2. In the WMQTT Utility, enter a topic name, such as `car/gps`.
3. Enter a message such as `35.818888889,-78.644722222` to be published.
4. Set the QoS value to `1`.
5. Start the trace as described in B.2, "Connect to the MQTT server" on page 224.
6. Click **Publish** to publish this message.

Figure B-8 shows the client setup to publish this message.



Broker TCP/IP address: 9.42.170.179 1883

Subscribe To Topics – text display

Subscribe Topic: samples/topic02 Request QoS: 0

Received Topic: QoS: Retained: ☐

Publish Messages – text display

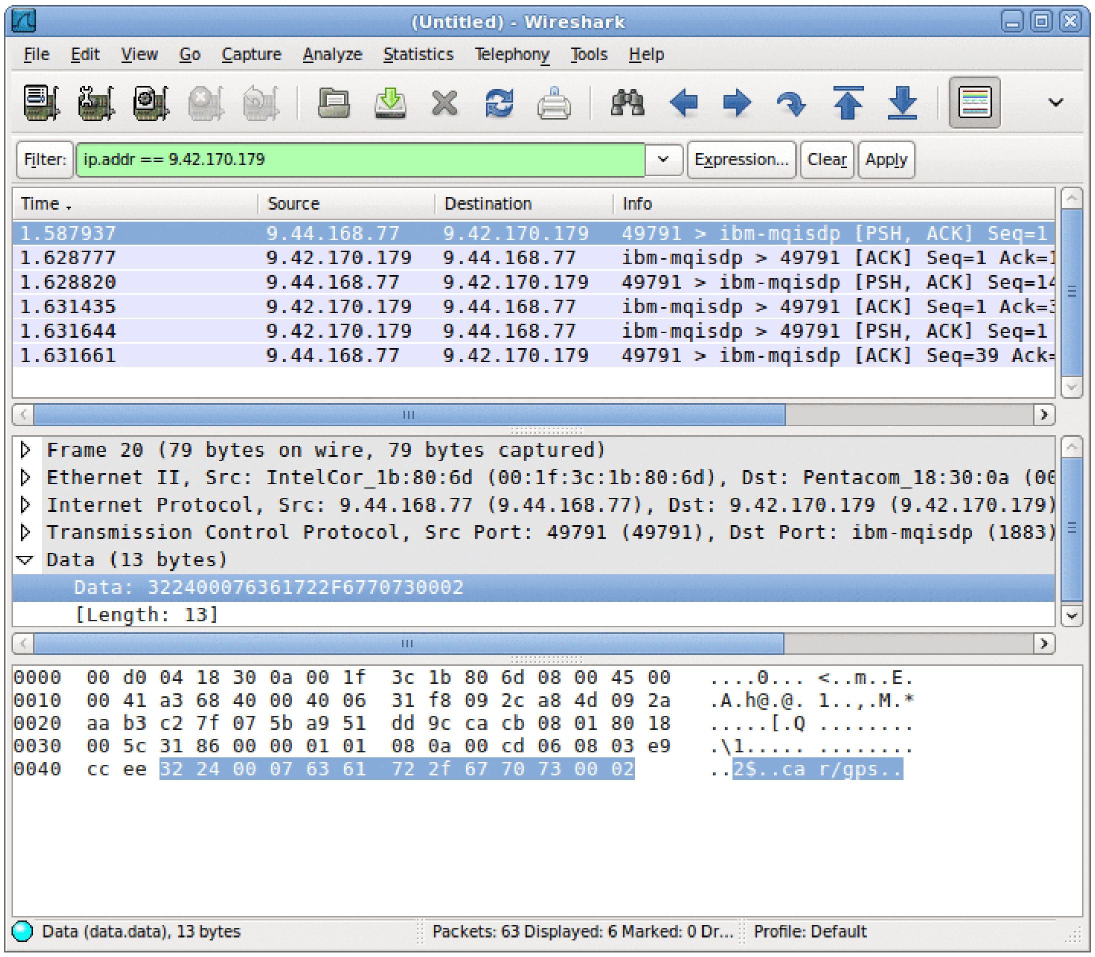
Topic: car/gps QoS: 1 Retained: ☐

```
5.818888889,-78.644722222
```

Figure B-8 Client setup

B.4.1 The PUBLISH message

The client performs the publication by sending a PUBLISH message, as shown in the Wireshark trace in Figure B-9. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time              Source          Destination     Info
1.587937          9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [PSH, ACK] Seq=1
1.628777          9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [ACK] Seq=1 Ack=1
1.628820          9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [PSH, ACK] Seq=14
1.631435          9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [ACK] Seq=1 Ack=3
1.631644          9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [PSH, ACK] Seq=1
1.631661          9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [ACK] Seq=39 Ack=
```

```
▷ Frame 20 (79 bytes on wire, 79 bytes captured)
▷ Ethernet II, Src: IntelCor_1b:80:6d (00:1f:3c:1b:80:6d), Dst: Pentacom_18:30:0a (00
▷ Internet Protocol, Src: 9.44.168.77 (9.44.168.77), Dst: 9.42.170.179 (9.42.170.179)
▷ Transmission Control Protocol, Src Port: 49791 (49791), Dst Port: ibm-mqisdp (1883)
▽ Data (13 bytes)
    Data: 322400076361722F6770730002
    [Length: 13]
```

```
0000  00 d0 04 18 30 0a 00 1f  3c 1b 80 6d 08 00 45 00   ....0... <..m..E.
0010  00 41 a3 68 40 00 40 06  31 f8 09 2c a8 4d 09 2a   .A.h@.@. 1..,.M.*
0020  aa b3 c2 7f 07 5b a9 51  dd 9c ca cb 08 01 80 18   .....[.Q ........
0030  00 5c 31 86 00 00 01 01  08 0a 00 cd 06 08 03 e9   .\1..... ........
0040  cc ee 32 24 00 07 63 61  72 2f 67 70 73 00 02      ..2$..ca r/gps..
```

Data (data.data), 13 bytes
Packets: 63 Displayed: 6 Marked: 0 Dr...
Profile: Default

Figure B-9 Wireshark trace

Table B-8 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-8 Bytes table

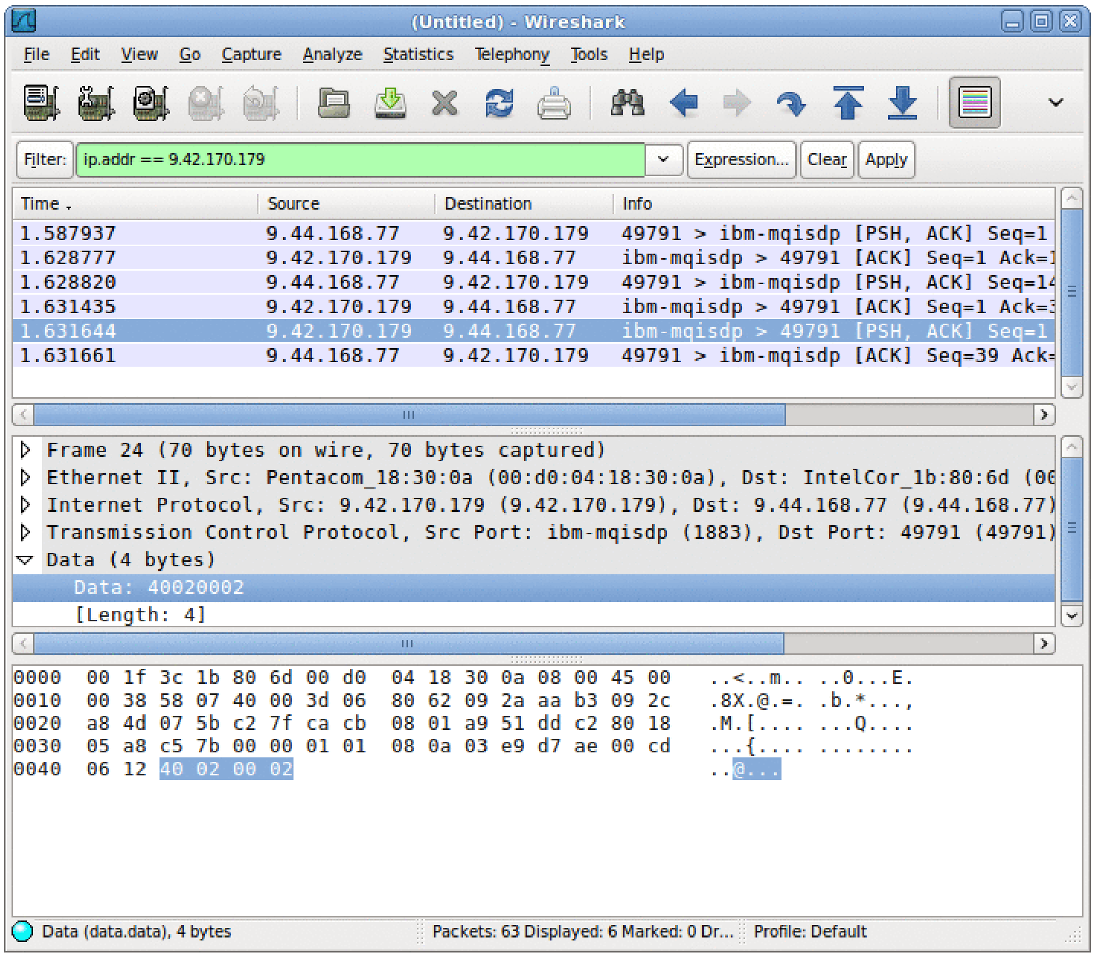
Byte	7	6	5	4	3	2	1	0	Remarks
1[a]	0	0	1	1	0	0	1	0	Message Type 3 (upper nibble) denoting a publication. The lower nibble (bits 1 and 2) shows QoS 1.
12[b]	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
13	0	0	0	0	0	0	1	0	Message ID - Least Significant Byte - 2

a. The first byte of the fixed header to show the QoS of 1.
b. The last byte in the variable header has the message ID as 2.

The bytes and description are similar to the trace seen for PUBLISH QoS 0 message except for the QoS value in the PUBLISH message sent.

B.4.2 The PUBACK message

The server responds to a QoS 1 PUBLISH message with a PUBACK message, as shown in the Wireshark trace in Figure B-10. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression...
Clear
Apply

```
Filter: ip.addr == 9.42.170.179

Time      Source         Destination    Info
1.587937  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=1
1.628777  9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=1
1.628820  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=14
1.631435  9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=3
1.631644  9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=1
1.631661  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=39 Ack=

Frame 24 (70 bytes on wire, 70 bytes captured)
Ethernet II, Src: Pentacom_18:30:0a (00:d0:04:18:30:0a), Dst: IntelCor_1b:80:6d (00
Internet Protocol, Src: 9.42.170.179 (9.42.170.179), Dst: 9.44.168.77 (9.44.168.77)
Transmission Control Protocol, Src Port: ibm-mqisdp (1883), Dst Port: 49791 (49791)
Data (4 bytes)
    Data: 40020002
    [Length: 4]

0000  00 1f 3c 1b 80 6d 00 d0  04 18 30 0a 08 00 45 00   ..<..m.. ..0...E.
0010  00 38 58 07 40 00 3d 06  80 62 09 2a aa b3 09 2c   .8X.@.=. .b.*...,
0020  a8 4d 07 5b c2 7f ca cb  08 01 a9 51 dd c2 80 18   .M.[.... ...Q....
0030  05 a8 c5 7b 00 00 01 01  08 0a 03 e9 d7 ae 00 cd   ...{.... ........
0040  06 12 40 02 00 02                                  ..@...

Data (data.data), 4 bytes    Packets: 63 Displayed: 6 Marked: 0 Dr...    Profile: Default
```

Figure B-10 Wireshark trace

Table B-9 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-9 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	1	0	0	0	0	0	0	Message Type 4 (upper nibble) denoting acknowledgment of publication. The lower nibble is not used.
2	0	0	0	0	0	0	1	0	The remaining length - 0x02 - 2 bytes.
3	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
4	0	0	0	0	0	0	1	0	Message ID - Least Significant Byte - 2

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

1. The first two bytes make up the fixed header of the message.

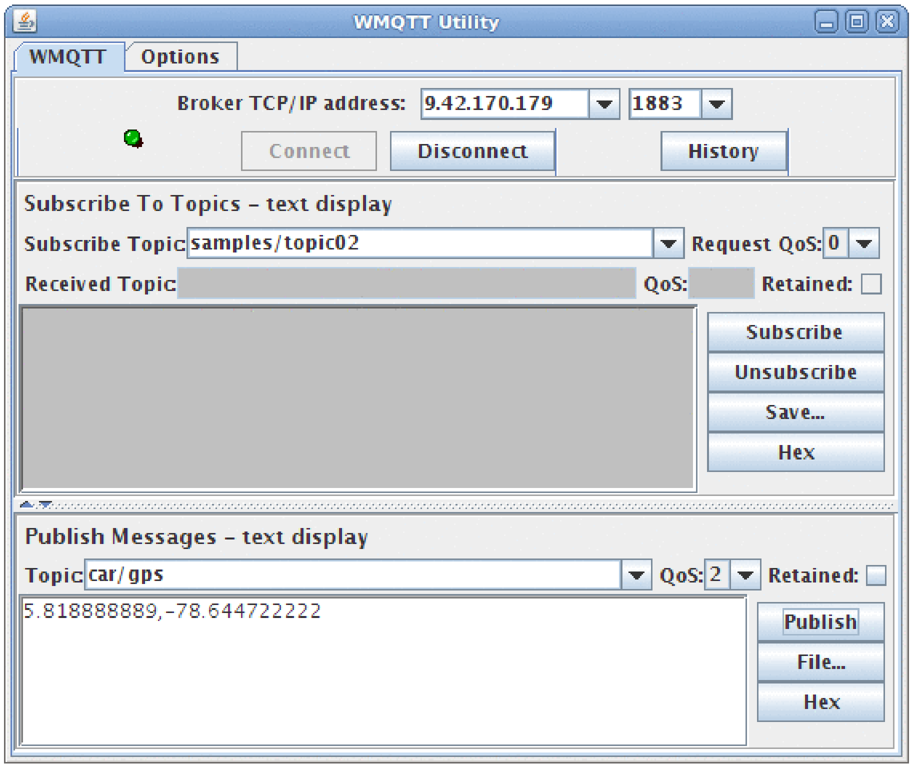
   a. The first byte has the message type as 4 denoting a PUBACK message - upper nibble. It also denotes QoS as 0 in the bits 1 and 2 of the lower nibble.
   b. The second byte has the remaining length of the message as 0x02 or 2 bytes.
2. Then, we have the variable header that is broken up as follows.
   a. The first two bytes (3 and 4 in the table) contain the message ID as 2.
3. There is no payload.

B.5 Publishing with QoS 2

After the client is connected to the server, you can publish as described here:

1. Set up Wireshark as described in "Set up for tracing" on page 222.
2. In the WMQTT Utility, enter a topic name, such as `car/gps`.
3. Enter a message such as `35.818888889,-78.644722222` to be published.
4. Set the QoS value to `2`.
5. Start the trace as described in B.2, "Connect to the MQTT server" on page 224.
6. Click **Publish** to publish this message.

Figure B-11 shows the client setup to publish this message.

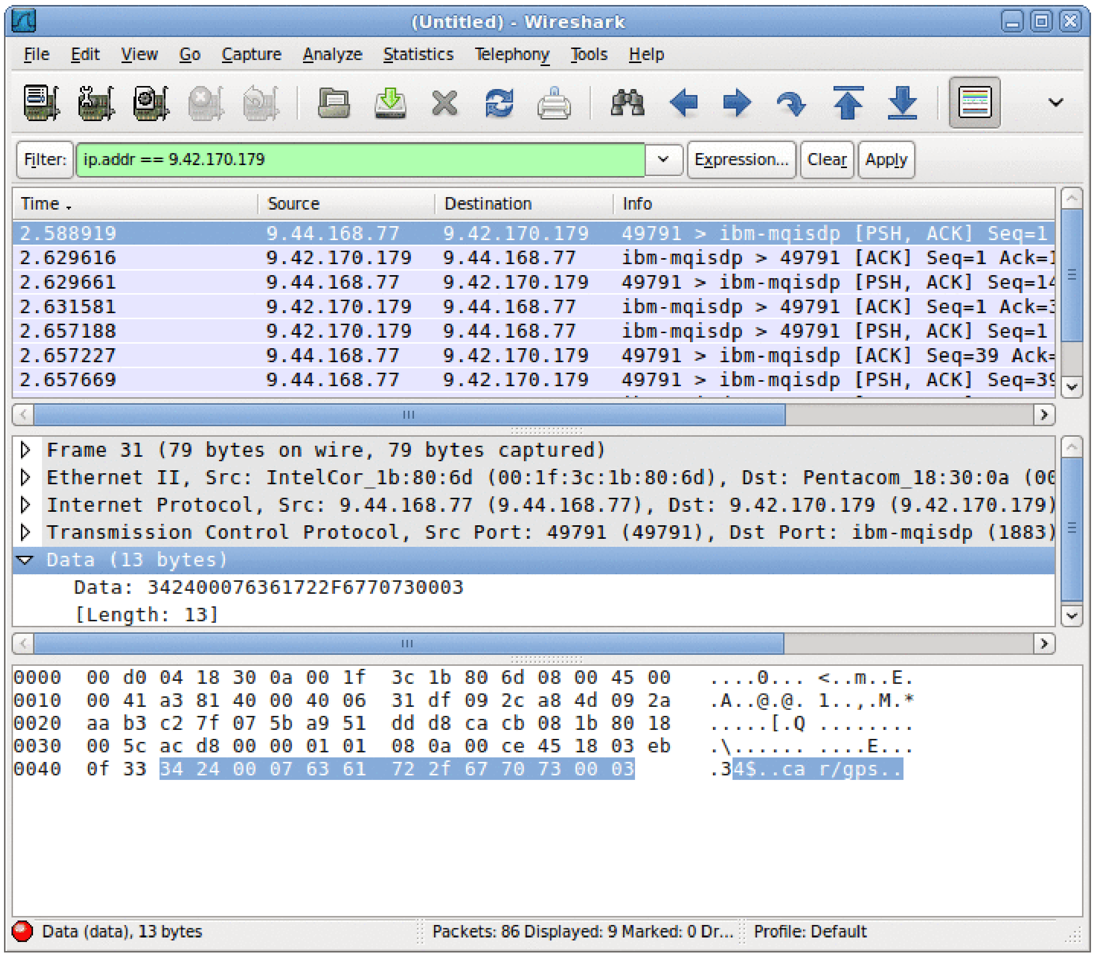

Broker TCP/IP address: 9.42.170.179 1883
Subscribe Topic: samples/topic02 Request QoS: 0
Received Topic: QoS: Retained: ☐
Topic: car/gps QoS: 2 Retained: ☐

```
5.818888889,-78.644722222
```

Figure B-11 Client setup

B.5.1 The PUBLISH message

The client does the publication by sending a PUBLISH message as shown in the Wireshark trace in Figure B-12 on page 236. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Expression... Clear Apply

```
Filter: ip.addr == 9.42.170.179

Time       Source         Destination    Info
2.588919   9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=1
2.629616   9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=1
2.629661   9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=14
2.631581   9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=3
2.657188   9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=1
2.657227   9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=39 Ack=
2.657669   9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=39
```

```
▷ Frame 31 (79 bytes on wire, 79 bytes captured)
▷ Ethernet II, Src: IntelCor_1b:80:6d (00:1f:3c:1b:80:6d), Dst: Pentacom_18:30:0a (00
▷ Internet Protocol, Src: 9.44.168.77 (9.44.168.77), Dst: 9.42.170.179 (9.42.170.179)
▷ Transmission Control Protocol, Src Port: 49791 (49791), Dst Port: ibm-mqisdp (1883)
▽ Data (13 bytes)
    Data: 342400076361722F6770730003
    [Length: 13]
```

```
0000  00 d0 04 18 30 0a 00 1f  3c 1b 80 6d 08 00 45 00   ....0... <..m..E.
0010  00 41 a3 81 40 00 40 06  31 df 09 2c a8 4d 09 2a   .A..@.@. 1..,.M.*
0020  aa b3 c2 7f 07 5b a9 51  dd d8 ca cb 08 1b 80 18   .....[.Q ........
0030  00 5c ac d8 00 00 01 01  08 0a 00 ce 45 18 03 eb   .\...... ....E...
0040  0f 33 34 24 00 07 63 61  72 2f 67 70 73 00 03      .34$..ca r/gps..
```

```
Data (data), 13 bytes    Packets: 86 Displayed: 9 Marked: 0 Dr...    Profile: Default
```

Figure B-12 Wireshark trace

Table B-10 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-10 Bytes table

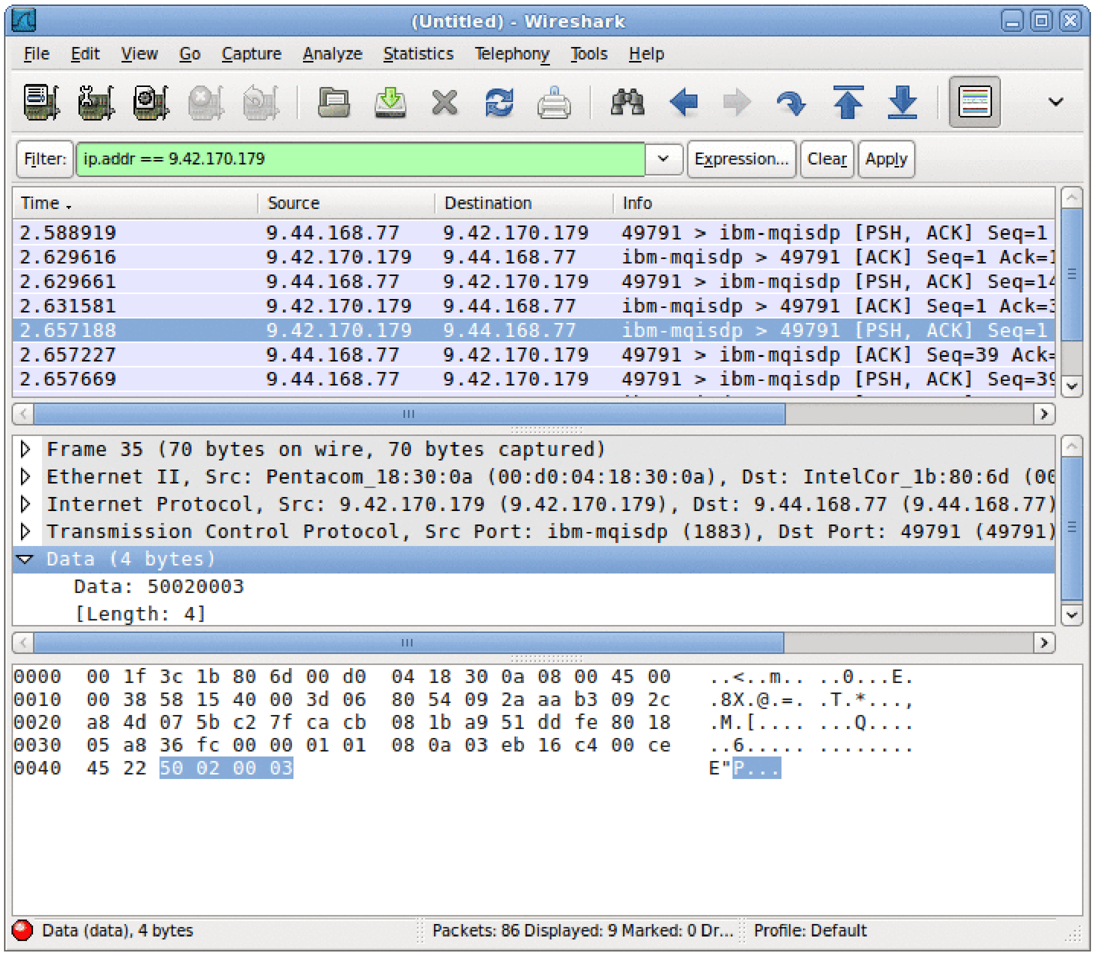
Byte	7	6	5	4	3	2	1	0	Remarks
1[a]	0	0	1	1	0	1	0	0	Message Type 3 (upper nibble) denoting a publication. The lower nibble (bits 1 and 2) shows QoS 2.
12[b]	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
13	0	0	0	0	0	0	1	1	Message ID - Least Significant Byte - 3

a. The first byte of the fixed header shows the QoS as 2.
b. The last byte in the variable header has the message ID as 3.

The bytes and description are similar to the trace seen for PUBLISH message QoS 0 except for the QoS value in the PUBLISH message.

B.5.2 The PUBREC message

The server responds back with a PUBREC message for the incoming QoS 2 message from the client, as shown in the Wireshark trace in Figure B-13 on page 237. This message is the second message in a QoS 2 publication. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time      Source        Destination   Info
2.588919  9.44.168.77   9.42.170.179  49791 > ibm-mqisdp [PSH, ACK] Seq=1
2.629616  9.42.170.179  9.44.168.77   ibm-mqisdp > 49791 [ACK] Seq=1 Ack=1
2.629661  9.44.168.77   9.42.170.179  49791 > ibm-mqisdp [PSH, ACK] Seq=14
2.631581  9.42.170.179  9.44.168.77   ibm-mqisdp > 49791 [ACK] Seq=1 Ack=3
2.657188  9.42.170.179  9.44.168.77   ibm-mqisdp > 49791 [PSH, ACK] Seq=1
2.657227  9.44.168.77   9.42.170.179  49791 > ibm-mqisdp [ACK] Seq=39 Ack=
2.657669  9.44.168.77   9.42.170.179  49791 > ibm-mqisdp [PSH, ACK] Seq=39
```

```
▷ Frame 35 (70 bytes on wire, 70 bytes captured)
▷ Ethernet II, Src: Pentacom_18:30:0a (00:d0:04:18:30:0a), Dst: IntelCor_1b:80:6d (00
▷ Internet Protocol, Src: 9.42.170.179 (9.42.170.179), Dst: 9.44.168.77 (9.44.168.77)
▷ Transmission Control Protocol, Src Port: ibm-mqisdp (1883), Dst Port: 49791 (49791)
▽ Data (4 bytes)
    Data: 50020003
    [Length: 4]
```

```
0000  00 1f 3c 1b 80 6d 00 d0  04 18 30 0a 08 00 45 00   ..<..m.. ..0...E.
0010  00 38 58 15 40 00 3d 06  80 54 09 2a aa b3 09 2c   .8X.@.=. .T.*...,
0020  a8 4d 07 5b c2 7f ca cb  08 1b a9 51 dd fe 80 18   .M.[.... ...Q....
0030  05 a8 36 fc 00 00 01 01  08 0a 03 eb 16 c4 00 ce   ..6..... ........
0040  45 22 50 02 00 03                                  E"P...
```

Figure B-13 Wireshark trace

Table B-11 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-11 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	1	0	1	0	0	0	0	Message Type 5 (upper nibble) denoting acknowledgment of publication. The lower nibble is not used.
2	0	0	0	0	0	0	1	0	The remaining length - 0x02 - 2 bytes.
3	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
4	0	0	0	0	0	0	1	1	Message ID - Least Significant Byte - 3

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

1. The first two bytes make up the fixed header of the message.
   a. The first byte has the message type as 5 denoting a PUBREC message - upper nibble. The lower nibble is not used.

b. The second byte has the remaining length of the message as 0x02 or 2.

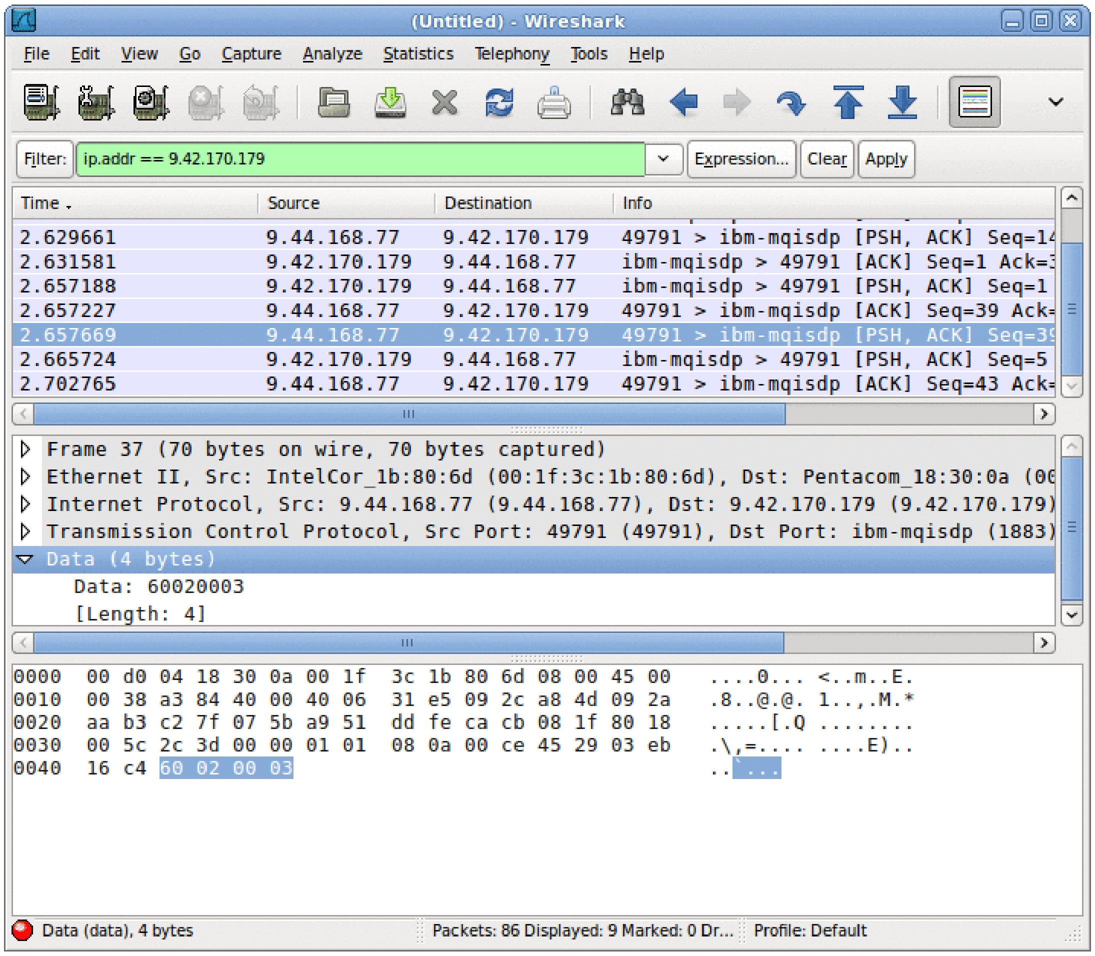
2. Then, we have the variable header that is broken up as follows.

   a. The first two bytes (3 and 4 in the table) contain the message ID of the message as 0x03 or 3.

3. There is no payload.

B.5.3 The PUBREL message

The client responds back with a PUBREL message for the incoming PUBREC message from the server, as shown in the Wireshark trace in Figure B-14. This message is the third message in a QoS 2 publication. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.



Filter: ip.addr == 9.42.170.179

```
Time        Source         Destination    Info
2.629661    9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=14
2.631581    9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [ACK] Seq=1 Ack=3
2.657188    9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=1
2.657227    9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=39 Ack=
2.657669    9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=39
2.665724    9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=5
2.702765    9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=43 Ack=
```

```
▷ Frame 37 (70 bytes on wire, 70 bytes captured)
▷ Ethernet II, Src: IntelCor_1b:80:6d (00:1f:3c:1b:80:6d), Dst: Pentacom_18:30:0a (00
▷ Internet Protocol, Src: 9.44.168.77 (9.44.168.77), Dst: 9.42.170.179 (9.42.170.179)
▷ Transmission Control Protocol, Src Port: 49791 (49791), Dst Port: ibm-mqisdp (1883)
▽ Data (4 bytes)
    Data: 60020003
    [Length: 4]
```

```
0000  00 d0 04 18 30 0a 00 1f  3c 1b 80 6d 08 00 45 00   ....0... <..m..E.
0010  00 38 a3 84 40 00 40 06  31 e5 09 2c a8 4d 09 2a   .8..@.@. 1..,.M.*
0020  aa b3 c2 7f 07 5b a9 51  dd fe ca cb 08 1f 80 18   .....[.Q ........
0030  00 5c 2c 3d 00 00 01 01  08 0a 00 ce 45 29 03 eb   .\,=.... ....E)..
0040  16 c4 60 02 00 03                                  ..`...
```

Data (data), 4 bytes   Packets: 86 Displayed: 9 Marked: 0 Dr...   Profile: Default

Figure B-14 Wireshark trace

Table B-12 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-12 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	1	1	0	0	0	0	0	Message Type 6 (upper nibble) denoting acknowledgment of publication.

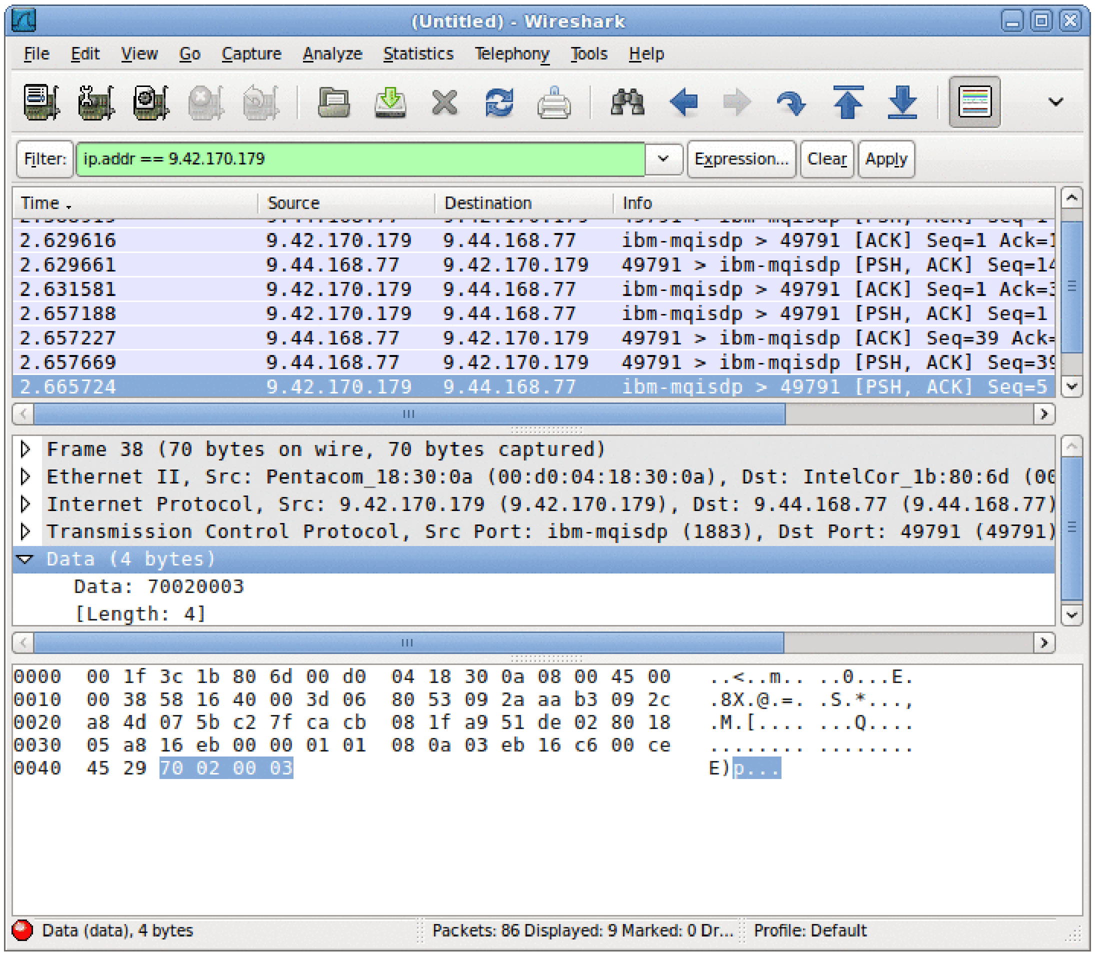
Byte	7	6	5	4	3	2	1	0	Remarks
2	0	0	0	0	0	0	1	0	The remaining length - 0x02 - 2 bytes.
3	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
4	0	0	0	0	0	0	1	1	Message ID - Least Significant Byte - 3

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

1. The first two bytes make up the fixed header of the message.
   a. The first byte has the message type as 6 denoting a PUBREL message - upper nibble. The lower nibble is not used.
   b. The second byte has the remaining length of the message as 0x02 or 2.
2. Then, we have the variable header that is broken up as follows.
   a. The first two bytes (3 and 4 in the table) contain the message ID of the message as 0x02 or 2.
3. There is no payload.

B.5.4 The PUBCOMP message

The server responds back with a PUBCOMP message for the incoming PUBREL message from the server, as shown in the Wireshark trace in Figure B-15 on page 240. This message is the fourth and final message in a QoS 2 publication. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time        Source          Destination     Info
2.629616    9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [ACK] Seq=1 Ack=1
2.629661    9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [PSH, ACK] Seq=14
2.631581    9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [ACK] Seq=1 Ack=3
2.657188    9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [PSH, ACK] Seq=1
2.657227    9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [ACK] Seq=39 Ack=
2.657669    9.44.168.77     9.42.170.179    49791 > ibm-mqisdp [PSH, ACK] Seq=39
2.665724    9.42.170.179    9.44.168.77     ibm-mqisdp > 49791 [PSH, ACK] Seq=5
```

```
▷ Frame 38 (70 bytes on wire, 70 bytes captured)
▷ Ethernet II, Src: Pentacom_18:30:0a (00:d0:04:18:30:0a), Dst: IntelCor_1b:80:6d (00
▷ Internet Protocol, Src: 9.42.170.179 (9.42.170.179), Dst: 9.44.168.77 (9.44.168.77)
▷ Transmission Control Protocol, Src Port: ibm-mqisdp (1883), Dst Port: 49791 (49791)
▽ Data (4 bytes)
    Data: 70020003
    [Length: 4]
```

```
0000  00 1f 3c 1b 80 6d 00 d0  04 18 30 0a 08 00 45 00   ..<..m.. ..0...E.
0010  00 38 58 16 40 00 3d 06  80 53 09 2a aa b3 09 2c   .8X.@.=. .S.*...,
0020  a8 4d 07 5b c2 7f ca cb  08 1f a9 51 de 02 80 18   .M.[.... ...Q....
0030  05 a8 16 eb 00 00 01 01  08 0a 03 eb 16 c6 00 ce   ........ ........
0040  45 29 70 02 00 03                                  E)p...
```

Figure B-15 Wireshark trace

Table B-13 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-13 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	0	1	1	1	0	0	0	0	Message Type 7 (upper nibble) denoting acknowledgment of publication. The lower nibble is not used.
2	0	0	0	0	0	0	1	0	The remaining length - 0x02 - 2 bytes.
3	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
4	0	0	0	0	0	0	1	1	Message ID - Least Significant Byte - 3

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

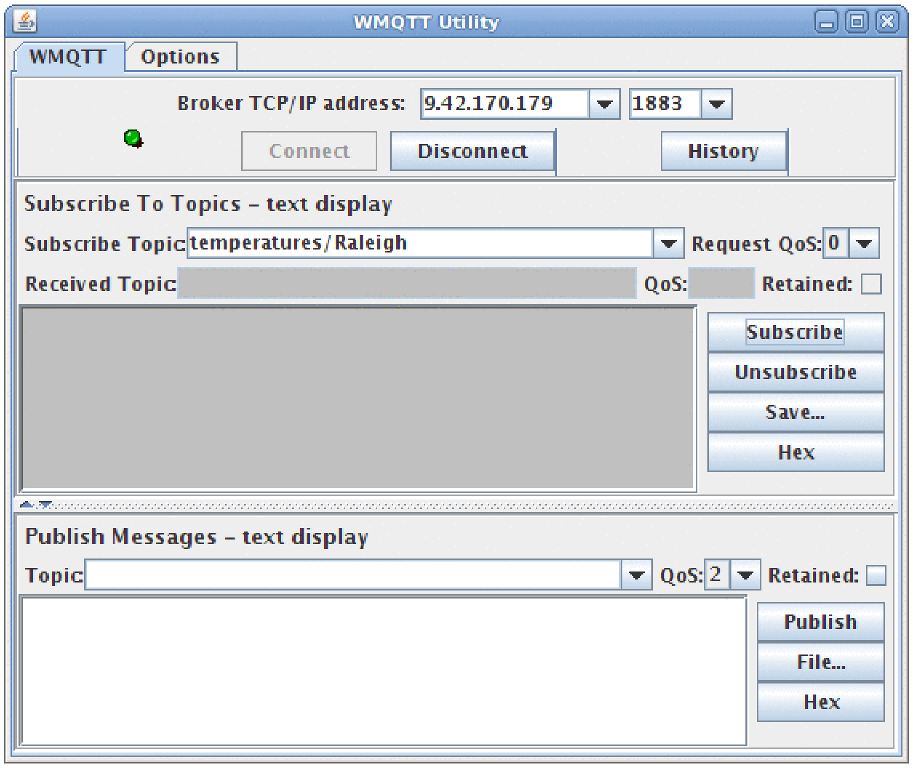
1. The first two bytes make up the fixed header of the message.
   a. The first byte has the message type as 7 denoting a PUBCOMP message - upper nibble. The lower nibble is not used.
   b. The second byte has the remaining length of the message as 0x02 or 2.

2. Then, we have the variable header that is broken up as follows.
   a. The first two bytes (3 and 4 in the table) contain the message ID of the message as 0x02 or 2.
3. There is no payload.

B.6 Subscribing

After the client is connected to the server, you can subscribe (see Figure B-16):

1. Set up Wireshark as described in “Set up for tracing” on page 222.
2. In the WMQTT Utility, enter a topic name, such as `temperatures/Raleigh`.
3. Start the trace as described in B.2, “Connect to the MQTT server” on page 224.
4. Click **Subscribe** to subscribe to a given topic name.



Broker TCP/IP address: 9.42.170.179 1883

Subscribe To Topics – text display

Subscribe Topic: temperatures/Raleigh Request QoS: 0

Received Topic: QoS: Retained:

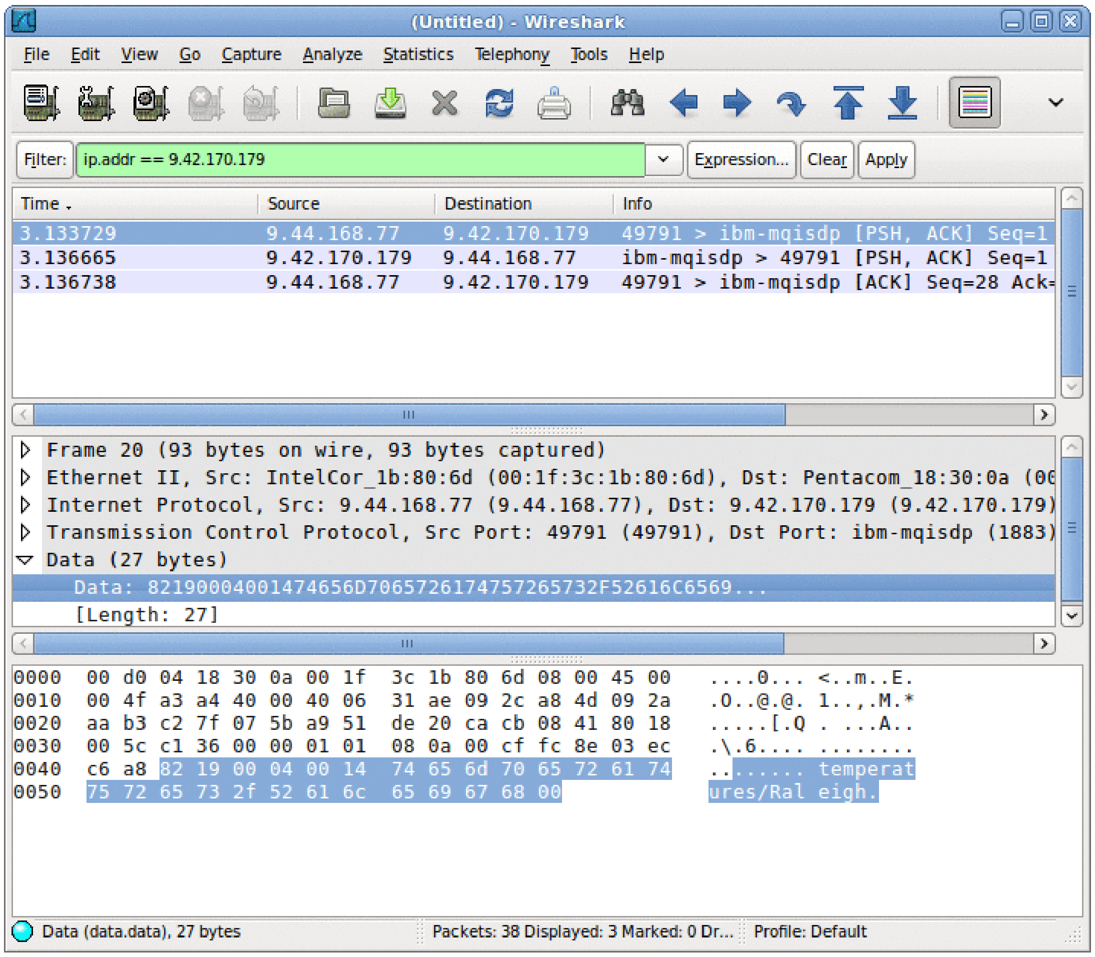
Publish Messages – text display

Topic: QoS: 2 Retained:

Figure B-16 Subscribe

B.6.1 The SUBSCRIBE message

The client sends out a SUBSCRIBE message to the server, as shown in the Wireshark trace in Figure B-17. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Expression... Clear Apply

Filter: ip.addr == 9.42.170.179

```
Time         Source         Destination    Info
3.133729     9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=1
3.136665     9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=1
3.136738     9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=28 Ack=
```

```
▷ Frame 20 (93 bytes on wire, 93 bytes captured)
▷ Ethernet II, Src: IntelCor_1b:80:6d (00:1f:3c:1b:80:6d), Dst: Pentacom_18:30:0a (00
▷ Internet Protocol, Src: 9.44.168.77 (9.44.168.77), Dst: 9.42.170.179 (9.42.170.179)
▷ Transmission Control Protocol, Src Port: 49791 (49791), Dst Port: ibm-mqisdp (1883)
▽ Data (27 bytes)
    Data: 821900040014746560706572617475726573 2F52616C6569...
    [Length: 27]
```

```
0000  00 d0 04 18 30 0a 00 1f  3c 1b 80 6d 08 00 45 00   ....0... <..m..E.
0010  00 4f a3 a4 40 00 40 06  31 ae 09 2c a8 4d 09 2a   .O..@.@. 1..,.M.*
0020  aa b3 c2 7f 07 5b a9 51  de 20 ca cb 08 41 80 18   .....[.Q . ...A..
0030  00 5c c1 36 00 00 01 01  08 0a 00 cf fc 8e 03 ec   .\.6.... ........
0040  c6 a8 82 19 00 04 00 14  74 65 6d 70 65 72 61 74   ........ temperat
0050  75 72 65 73 2f 52 61 6c  65 69 67 68 00            ures/Ral eigh.
```

Data (data.data), 27 bytes   Packets: 38 Displayed: 3 Marked: 0 Dr...   Profile: Default

Figure B-17 Wireshark trace

Table B-14 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-14 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	1	0	0	0	0	0	1	0	Message Type 8 (upper nibble) denoting a subscription. The lower nibble (bits 1 and 2) denotes QoS 1 of SUBSCRIBE messages. This is for acknowledging multiple subscription requests.
2	0	0	0	1	1	0	0	1	Remaining length 0x19 or 25 bytes
3	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
4	0	0	0	0	0	1	0	0	Message ID - Least Significant Byte - 4
5	0	0	0	0	0	0	0	0	Length of topic name - Most Significant Byte - 0
6	0	0	0	1	0	1	0	0	Length of topic name - Least Significant Byte - 0x14 - 20

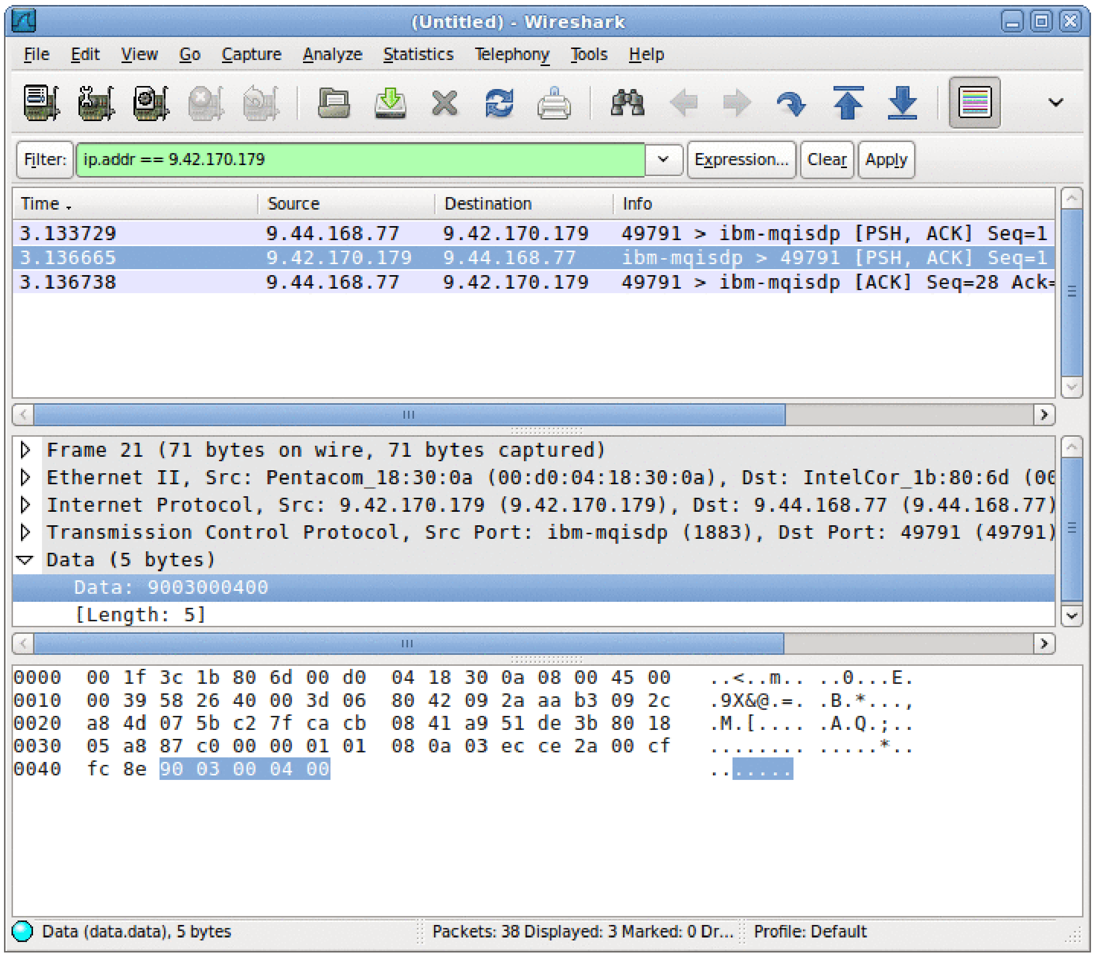
Byte	7	6	5	4	3	2	1	0	Remarks
7	0	1	1	1	0	1	0	0	t
8	0	1	1	0	0	1	0	1	e
9	0	1	1	0	1	1	0	1	m
10	0	1	1	1	0	0	0	0	p
11	0	1	1	0	0	1	0	1	e
12	0	1	1	1	0	0	1	0	r
13	0	1	1	0	0	0	0	1	a
14	0	1	1	1	0	1	0	0	t
15	0	1	1	1	0	1	0	1	u
16	0	1	1	1	0	0	1	0	r
17	0	1	1	0	0	1	0	1	e
18	0	1	1	1	0	0	1	1	s
19	0	0	1	0	1	1	1	1	/
20	0	1	0	1	0	0	1	0	R
21	0	1	1	0	0	0	0	1	a
22	0	1	1	0	1	1	0	0	l
23	0	1	1	0	0	1	0	1	e
24	0	1	1	0	1	0	0	1	i
25	0	1	1	0	0	1	1	1	g
26	0	1	1	0	1	0	0	0	h
27	0	0	0	0	0	0	0	0	Requested QoS is 0

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

1. The first two bytes make up the fixed header of the message.
   a. The first byte has the message type as 8 denoting a SUBSCRIBE message - bits 4 and 5 of the upper nibble. It also denotes QoS as 1 in the bits 1 and 2 of the lower nibble.
   b. The second byte has the remaining length of the message as 0x19 or 25 bytes.
2. Then, we have the variable header that is broken up as follows.
   a. The first two bytes (3 and 4 in the table) contain the message ID 0x02 or 2.
   b. The next two bytes (5 and 6 in the table) contain the length of the topic name as 0x0014 or 20 bytes.
3. The rest of the message is the payload. It is just the name of the topic being subscribed.
   a. The next twenty bytes (7 thru 26) are the actual name of the topic - temperatures/Raleigh.
   b. The last byte shows the requested QoS.

B.6.2 The SUBACK message

The server responds with a SUBACK message for the subscription request received by the client, as shown in the Wireshark trace in Figure B-18. The data flowing from the client to the server displays in the bottom pane in hexadecimal format.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time      Source         Destination    Info
3.133729  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [PSH, ACK] Seq=1
3.136665  9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=1
3.136738  9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=28 Ack=
```

```
▷ Frame 21 (71 bytes on wire, 71 bytes captured)
▷ Ethernet II, Src: Pentacom_18:30:0a (00:d0:04:18:30:0a), Dst: IntelCor_1b:80:6d (00
▷ Internet Protocol, Src: 9.42.170.179 (9.42.170.179), Dst: 9.44.168.77 (9.44.168.77)
▷ Transmission Control Protocol, Src Port: ibm-mqisdp (1883), Dst Port: 49791 (49791)
▽ Data (5 bytes)
    Data: 9003000400
    [Length: 5]
```

```
0000  00 1f 3c 1b 80 6d 00 d0  04 18 30 0a 08 00 45 00   ..<..m.. ..0...E.
0010  00 39 58 26 40 00 3d 06  80 42 09 2a aa b3 09 2c   .9X&@.=. .B.*...,
0020  a8 4d 07 5b c2 7f ca cb  08 41 a9 51 de 3b 80 18   .M.[.... .A.Q.;..
0030  05 a8 87 c0 00 00 01 01  08 0a 03 ec ce 2a 00 cf   ........ .....*..
0040  fc 8e 90 03 00 04 00                               .......
```

```
Data (data.data), 5 bytes    Packets: 38 Displayed: 3 Marked: 0 Dr...    Profile: Default
```

Figure B-18 Wireshark trace

Table B-15 deciphers each byte to show how this message correlates with the MQTT V3.1 Protocol Specification.

Table B-15 Bytes table

Byte	7	6	5	4	3	2	1	0	Remarks
1	1	0	0	1	0	0	0	0	Message Type 9 (upper nibble) denoting acknowledgement of subscription request. The lower nibble is not used.
2	0	0	0	0	0	0	1	1	Remaining length 0x03 or 3 bytes
3	0	0	0	0	0	0	0	0	Message ID - Most Significant Byte - 0
4	0	0	0	0	0	1	0	0	Message ID - Least Significant Byte - 4
5	0	0	0	0	0	0	0	0	Granted QoS 0 - upper 6 bits are not used.

The bytes transmitted can be broadly categorized into fixed header, variable header, and a payload as follows:

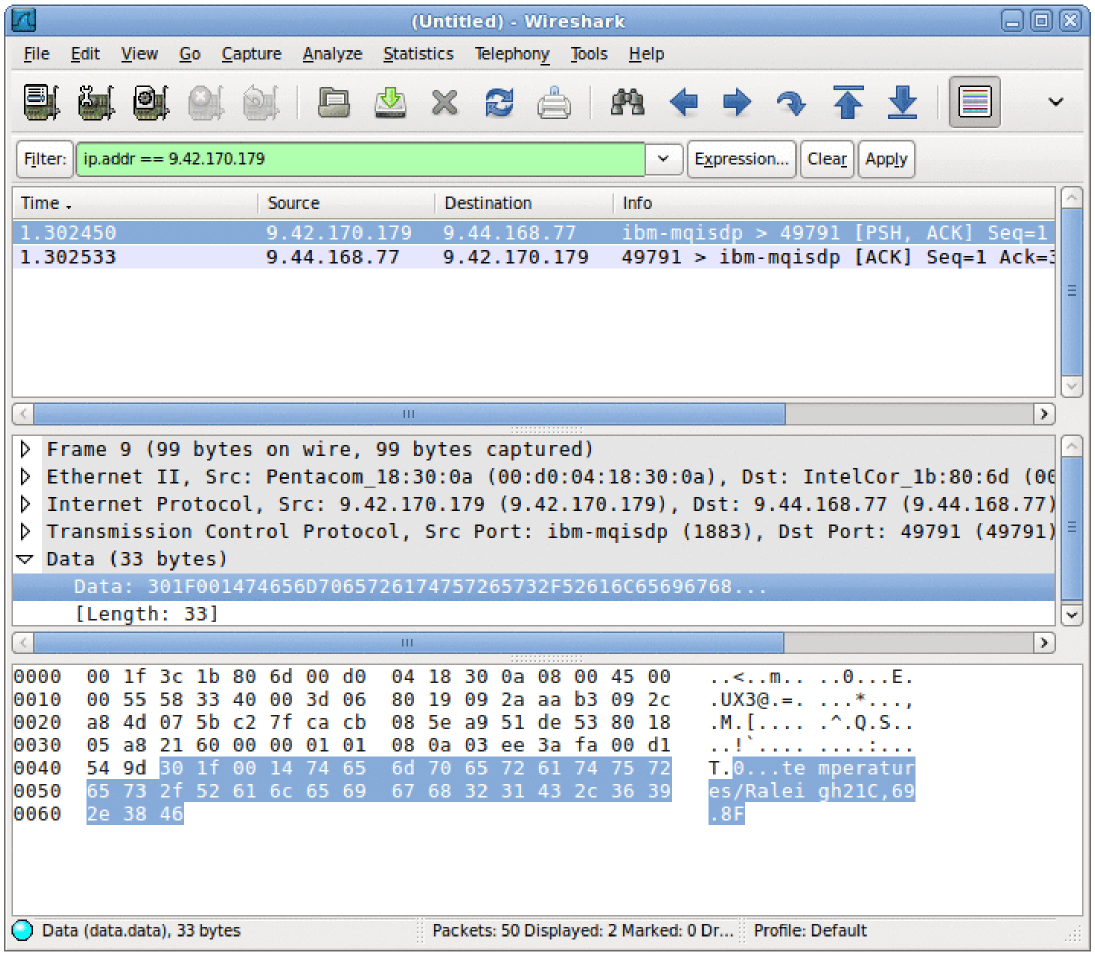
1. The first two bytes make up the fixed header of the message.
   a. The first byte has the message type as 9 denoting a SUBACK message - upper nibble. The lower nibble is not used.
   b. The second byte has the remaining length of the message as 0x03 or 3.
2. Then, we have the variable header that is broken up as follows.
   a. The first two bytes (3 and 4 in the table) contain the message ID of the message as 0x02 or 2.
3. The rest of the message is the payload which has the granted QoS as 0.

B.6.3 Subscribed message

We now trace the receipt of a message that was published by another client to the same server and topic (in this case, temperatures/Raleigh). To publish the message and initiate the trace, follow these steps:

1. In another instance of GUI, click **Connect** to initiate the connection from the client to a MQTT server.
2. Enter the name of a topic as `temperatures/Raleigh` in the bottom (Publish) pane.
3. Enter text as `21C,69.8F` to denote the temperature in Celsius and Fahrenheit.
4. In Wireshark, click **Start** in the list of live captures for the wlan0 interface to begin the trace.
5. Click **Publish** in the GUI to publish this message.
6. From the **Capture** menu option, select **Stop** to stop the capture.

The message received is actually a PUBLISH command from the server to client. Figure B-19 shows this message.


(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter: ip.addr == 9.42.170.179
Expression... Clear Apply

```
Time     Source         Destination    Info
1.302450 9.42.170.179   9.44.168.77    ibm-mqisdp > 49791 [PSH, ACK] Seq=1
1.302533 9.44.168.77    9.42.170.179   49791 > ibm-mqisdp [ACK] Seq=1 Ack=3

▷ Frame 9 (99 bytes on wire, 99 bytes captured)
▷ Ethernet II, Src: Pentacom_18:30:0a (00:d0:04:18:30:0a), Dst: IntelCor_1b:80:6d (06
▷ Internet Protocol, Src: 9.42.170.179 (9.42.170.179), Dst: 9.44.168.77 (9.44.168.77)
▷ Transmission Control Protocol, Src Port: ibm-mqisdp (1883), Dst Port: 49791 (49791)
▽ Data (33 bytes)
    Data: 301F001474656D70657261747572657332F52616C65696768...
    [Length: 33]

0000  00 1f 3c 1b 80 6d 00 d0  04 18 30 0a 08 00 45 00   ..<..m.. ..0...E.
0010  00 55 58 33 40 00 3d 06  80 19 09 2a aa b3 09 2c   .UX3@.=. ...*...,
0020  a8 4d 07 5b c2 7f ca cb  08 5e a9 51 de 53 80 18   .M.[.... .^.Q.S..
0030  05 a8 21 60 00 00 01 01  08 0a 03 ee 3a fa 00 d1   ..!`.... ....:...
0040  54 9d 30 1f 00 14 74 65  6d 70 65 72 61 74 75 72   T.0...te mperatur
0050  65 73 2f 52 61 6c 65 69  67 68 32 31 43 2c 36 39   es/Ralei gh21C,69
0060  2e 38 46                                           .8F

Data (data.data), 33 bytes   Packets: 50 Displayed: 2 Marked: 0 Dr...   Profile: Default
```

Figure B-19 Subscribed

The bytes table and description are the same as described for publications (B.3, "Publishing with QoS 0" on page 229).

C

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser at:

ftp://www.redbooks.ibm.com/redbooks/SG248054

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG248054.

Downloading and extracting the Web material

Create a subdirectory (folder) on your workstation, and extract the contents of the Web material .zip file into this folder.

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

References

- IBM Lotus Expeditor integrator product information

  `http://www.ibm.com/software/lotus/products/expeditor/integrator.html`
- IBM Lotus Expeditor integrator wiki

  `http://www.lotus.com/ldd/lewiki.nsf/dx/10092008022324PMJHEPKF.htm`
- IBM developerWorks, *Extending Java Message Service messaging to retail store devices*

  `http://www.ibm.com/developerworks/lotus/documentation/retailjms/`
- IBM developerWorks, *Messaging security guide for IBM Lotus Expeditor 6.2.3 integrator software*

  `http://www.ibm.com/developerworks/lotus/documentation/expeditorsecurity/`
- OSGi Alliance

  `http://www.osgi.org`
- Eclipse Consortium

  `http://www.eclipse.org`
- Equinox on-line information
- `http://www.eclipse.org/equinox/`
- Oracle Sun Developer Network, Java Messaging API documentation

  `http://java.sun.com/products/jms/docs.html`

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

IBM

Redbooks

Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ

(0.5” spine)
0.475”<->0.873”
250 <-> 459 pages

Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ

Introduces MQTT and includes scenarios that demonstrate its capabilities

Provides a quick guide to getting started with MQTT

Includes typical usage patterns and guidance on scaling a solution

MQ Telemetry Transport (MQTT) is a messaging protocol that is lightweight enough to be supported by the smallest devices, yet robust enough to ensure that important messages get to their destinations every time. With MQTT devices such as smart energy meters, cars, trains, satellite receivers, and personal health care devices can communicate with each other and with other systems or applications.

This IBM Redbooks publication introduces MQTT and takes a scenario-based approach to demonstrate its capabilities. It provides a quick guide to getting started and then shows how to grow to an enterprise scale MQTT server using IBM WebSphere® MQ Telemetry. Scenarios demonstrate how to integrate MQTT with other IBM products, including WebSphere Message Broker. This book also provides typical usage patterns and guidance on scaling a solution.

The intended audience for this book ranges from new users of MQTT and telemetry to those readers who are looking for in-depth knowledge and advanced topics.

SG24-8054-00 ISBN 0738437085