7.5

Mobile Messaging and M2M

Note

Before using this information and the product it supports, read the information in “Notices” on page 177.

This edition applies to version 7 release 5 of IBM® WebSphere® MQ and to all subsequent releases and modifications until otherwise indicated in new editions.

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

Contents

Mobile Messaging and M2M 5
Introduction to MQTT 7
Getting started with MQTT clients 9
Getting started with the MQTT client for Java 11
Getting started with the MQTT client for Java on Android 16
Getting started with the MQTT messaging client for JavaScript 21
Getting started with the MQTT client for C 23
Getting started with the MQTT client for C on iOS 43
MQTT command line sample programs 44
MQTT security 47
Building and running the secure MQTT client sample Java app 50
Connecting the MQTT client sample Java app on Android over SSL 58
Authenticating an MQTT client Java app with JAAS 67
Connecting the MQTT messaging client for JavaScript over SSL and WebSockets 72
Building and running the secure MQTT client sample C app 79
Generating keys and certificates 89
MQTT client identification, authorization, and authentication 95
Telemetry channel authentication using SSL 100
Publication privacy on telemetry channels 102
SSL configuration of MQTT clients and telemetry channels 103
Telemetry channel JAAS configuration 108
Programming concepts 109
The MQTT messaging client for JavaScript and web apps 110
How to program messaging apps in JavaScript 113
Callbacks and synchronization in MQTT client apps 117
Clean sessions 119
Client identifier 120
Delivery tokens 120
Last will and testament publication 121
Message persistence in MQTT clients 122
Publications 123
Qualities of service provided by an MQTT client 124
Retained publications and MQTT clients 126
Subscriptions 126
Topic strings and topic filters in MQTT clients 127
MQTT client programming reference 128
Getting started with MQTT servers 128
IBM WebSphere MQ as the MQTT server 130
IBM WebSphere MQ Telemetry daemon for devices concepts 140
Troubleshooting MQTT clients 151
Location of telemetry logs, error logs, and configuration files 152
MQTT v3 Java client reason codes 154
Tracing the telemetry (MQXR) service 155
Tracing the MQTT v3 Java client 156
Tracing the MQTT client for C 158
Tracing and debugging the MQTT (Paho) Java client 159
Tracing the MQTT JavaScript client 161
System requirements for using SHA-2 cipher suites with MQTT clients 162
Restrictions in browser support for mobile messaging web apps over SSL 163
Resolving problem: MQTT client does not connect 167
Resolving problem: MQTT client connection dropped 169
Resolving problem: Lost messages in an MQTT application 170

Resolving problem: Telemetry (MQXR) service does not start....171
Resolving problem: JAAS login module not called by the telemetry service....172
Resolving problem: Starting or running the daemon....175
Resolving problem: MQTT clients not connecting to the daemon....176

Notices....177
Programming interface information....178
Trademarks....178

Introduction to MQTT

Learn about sending messages between mobile apps using MQ telemetry transport (MQTT). The protocol is intended for use on wireless and low-bandwidth networks. A mobile application that uses MQTT sends and receives messages by calling an MQTT library. The messages are exchanged through an MQTT messaging server. The MQTT client and server handle the complexities of delivering messages reliably for the mobile app and keep the cost of network management small.

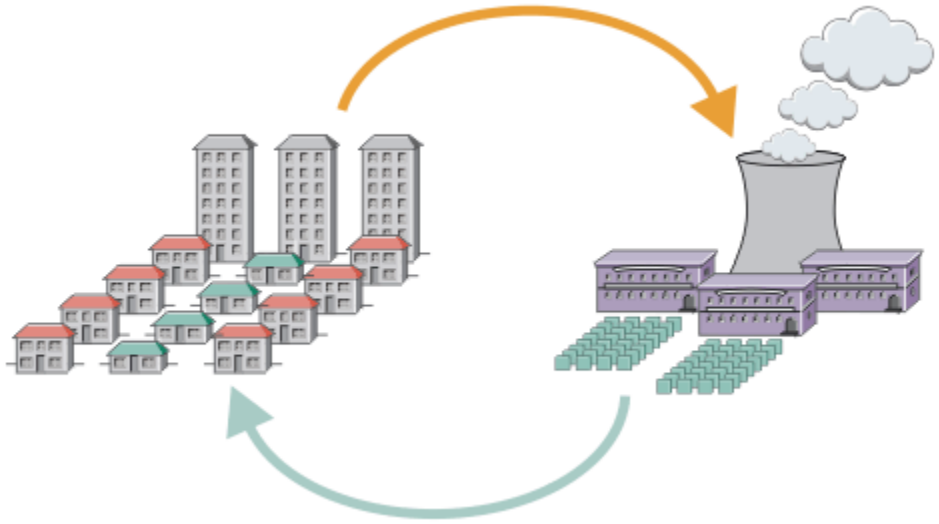
MQTT applications run on mobile devices, such as smartphones and tablets. MQTT is also used for telemetry to receive data from sensors, and to control them remotely. For mobile devices and sensors, MQTT offers a highly scalable publish/subscribe protocol with assured delivery. To send and receive MQTT messages, you add an MQTT client library to your application.

The MQTT client library is small. The library acts like a mail box, sending and receiving messages with other MQTT applications that are connected to an MQTT server. By sending messages, rather than staying connected to a server that is waiting for a response, MQTT applications conserve battery life. The library sends messages to other devices through an MQTT server that is running the MQTT version 3.1 protocol. You can send messages to a specific client, or use publish/subscribe messaging to connect many devices.

The MQTT client libraries connect applications for mobile devices and sensors to an MQTT server using the MQTT protocol.

IBM MessageSight and IBM WebSphere MQ are MQTT servers. They can connect large volumes of MQTT client applications, and they can connect MQTT and IBM WebSphere MQ networks together. See “Getting started with MQTT servers” on page 128. IBM WebSphere MQ and IBM MessageSight can both form a bridge between external web applications that are running on mobile devices and sensors, and other types of publish/subscribe and messaging applications that are running within the enterprise. The bridge makes it easier to build "smart solutions" that incorporate mobile devices and sensors.

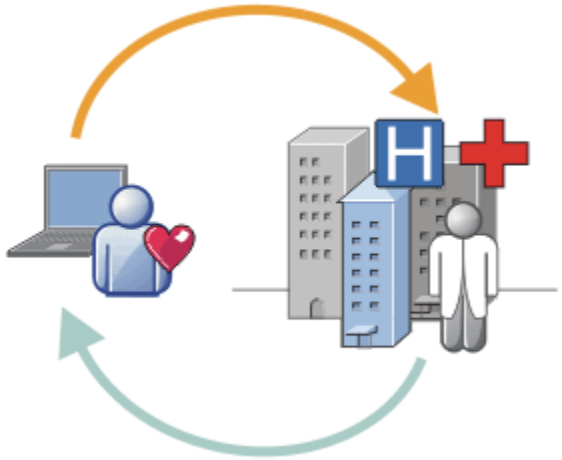
Smart solutions unlock the wealth of information available on the internet to applications running on mobile and sensor devices. Two examples of smart applications that are based on telemetry are smart electricity, and smart health services.



- An MQTT message that contains energy usage data sent to service provider.
- A telemetry application sends control commands that are based on analysis of energy usage data.
- For more information, see Telemetry scenario: Home energy monitoring and control.

Figure 1. Smart electricity metering

Figure 2. Smart health monitoring



- A telemetry application sends your health data to your hospital and doctor.
- MQTT message alerts or feedback can be sent based on analysis of your health data.
- For more information, see Telemetry scenario: Home patient monitoring.

You can build MQTT into small devices by writing your own app for the MQTT protocol. To help you do this, IBM provides client libraries that support apps that run over MQTT. See "Getting started with MQTT clients" on page 9. IBM provides client libraries for iOS apps, and for Android apps V 7.5.0.1, and a JavaScript browser client for platform-agnostic web apps. V 7.5.0.1 The JavaScript client pages connect to IBM MessageSight and IBM WebSphere MQ with the MQTT protocol over WebSockets. IBM also provides MQTT sample apps for C and Java on Linux® and Windows.

The C and Java libraries run on iOS, Android, Windows, and a number of UNIX and Linux platforms. You can port the C source code for the MQTT client library to other platforms. The MQTT client libraries for C and Java are available with an open source license from the Eclipse Paho project. See Eclipse Paho. The MQTT protocol specification is open, and available from MQTT.org.

MQTT protocol

The MQTT protocol is lightweight in the sense that clients are small, and it uses network bandwidth efficiently. The MQTT protocol supports assured delivery and fire-and-forget transfers. In the protocol, message delivery is decoupled from the application. The extent of decoupling in an application depends on the way an MQTT client and MQTT server are written. Decoupled delivery frees up an application from any server connection, and from waiting for messages. The interaction model is like email, but optimized for application programming.

The MQTT V3.1 protocol is published; see MQTT V3.1 Protocol Specification. The specification identifies a number of distinctive features about the protocol:

- It is a publish/subscribe protocol.

  In addition to providing one-to-many message distribution, publish/subscribe decouples applications. Both features are useful in applications that have many clients.
- It is not dependent in any way on the message content.
- It runs over TCP/IP, which provides basic network connectivity.
- It has three qualities of service for message delivery:

  "At most once"

  Messages are delivered according to the best efforts of the underlying Internet Protocol network. Message loss might occur.

  Use this quality of service with communicating ambient sensor data, for example. It does not matter if an individual reading is lost, if the next one is published soon after.

"At least once"
Messages are assured to arrive but duplicates might occur.

"Exactly once"
Messages are assured to arrive exactly once.

Use this quality of service with billing systems, for example. Duplicate or lost messages might lead to inconvenience or imposing incorrect charges.

- It is economical in the way it manages the flow of messages on the network. For example, the fixed-length header is only 2 bytes long, and protocol exchanges are minimized to reduce network traffic.
- It has a "Last Will and Testament" feature that notifies subscribers of the abnormal disconnection of a client from the MQTT server. See “Last will and testament publication” on page 121.

MQTT version 3.1 is supported by IBM IBM WebSphere MQ and IBM MessageSight. MQTT is implemented over TCP/IP. Another version of the protocol, MQTT-S, is available for non-TCP/IP networks. See MQTT-S version 1.2 specification.

MQTT communities

IBM is running IBM Developer Messaging community for MQTT developers that are writing applications for IBM MessageSight and IBM WebSphere MQ.

MQTT.org is a good place to go to learn about and discuss implementations and extensions to the MQTT protocol.

MQTT is an open source Eclipse project, under the Eclipse Technology Project. The Paho community is developing open source clients and servers. See Eclipse Paho.

Introduction to MQTT

Learn about sending messages between mobile apps using MQ telemetry transport (MQTT). The protocol is intended for use on wireless and low-bandwidth networks. A mobile application that uses MQTT sends and receives messages by calling an MQTT library. The messages are exchanged through an MQTT messaging server. The MQTT client and server handle the complexities of delivering messages reliably for the mobile app and keep the cost of network management small.

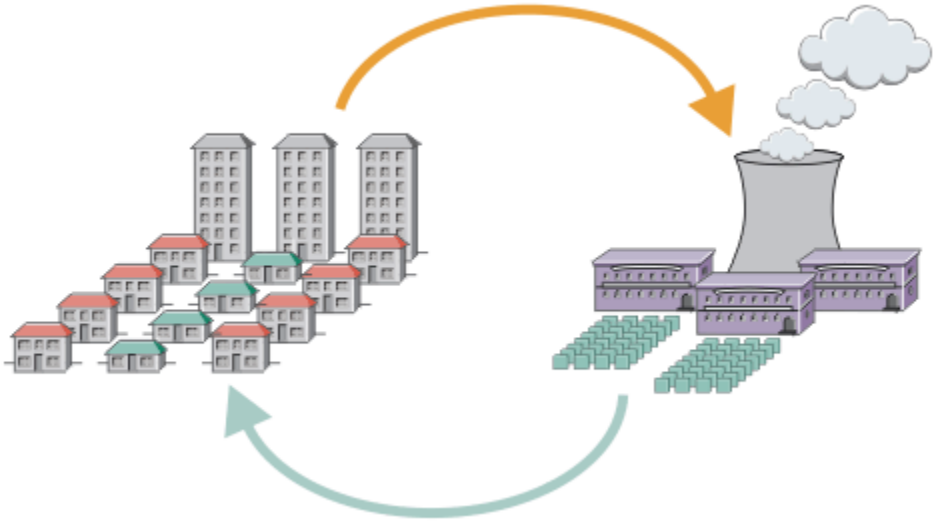
MQTT applications run on mobile devices, such as smartphones and tablets. MQTT is also used for telemetry to receive data from sensors, and to control them remotely. For mobile devices and sensors, MQTT offers a highly scalable publish/subscribe protocol with assured delivery. To send and receive MQTT messages, you add an MQTT client library to your application.

The MQTT client library is small. The library acts like a mail box, sending and receiving messages with other MQTT applications that are connected to an MQTT server. By sending messages, rather than staying connected to a server that is waiting for a response, MQTT applications conserve battery life. The library sends messages to other devices through an MQTT server that is running the MQTT version 3.1 protocol. You can send messages to a specific client, or use publish/subscribe messaging to connect many devices.

The MQTT client libraries connect applications for mobile devices and sensors to an MQTT server using the MQTT protocol.

IBM MessageSight and IBM WebSphere MQ are MQTT servers. They can connect large volumes of MQTT client applications, and they can connect MQTT and IBM WebSphere MQ networks together. See “Getting started with MQTT servers” on page 128. IBM WebSphere MQ and IBM MessageSight can both form a bridge between external web applications that are running on mobile devices and sensors, and other types of publish/subscribe and messaging applications that are running within the enterprise. The bridge makes it easier to build "smart solutions" that incorporate mobile devices and sensors.

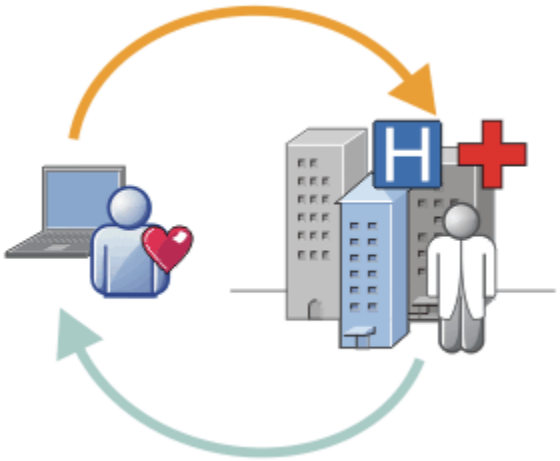
Smart solutions unlock the wealth of information available on the internet to applications running on mobile and sensor devices. Two examples of smart applications that are based on telemetry are smart electricity, and smart health services.



- An MQTT message that contains energy usage data sent to service provider.
- A telemetry application sends control commands that are based on analysis of energy usage data.
- For more information, see Telemetry scenario: Home energy monitoring and control.

Figure 3. Smart electricity metering

Figure 4. Smart health monitoring



- A telemetry application sends your health data to your hospital and doctor.
- MQTT message alerts or feedback can be sent based on analysis of your health data.
- For more information, see Telemetry scenario: Home patient monitoring.

You can build MQTT into small devices by writing your own app for the MQTT protocol. To help you do this, IBM provides client libraries that support apps that run over MQTT. See "Getting started with MQTT clients" on page 9. IBM provides client libraries for iOS apps, and for Android apps V 7.5.0.1, and a JavaScript browser client for platform-agnostic web apps. V 7.5.0.1 The JavaScript client pages connect to IBM MessageSight and IBM WebSphere MQ with the MQTT protocol over WebSockets. IBM also provides MQTT sample apps for C and Java on Linux and Windows.

The C and Java libraries run on iOS, Android, Windows, and a number of UNIX and Linux platforms. You can port the C source code for the MQTT client library to other platforms. The MQTT client libraries for C and Java are available with an open source license from the Eclipse Paho project. See Eclipse Paho. The MQTT protocol specification is open, and available from MQTT.org.

MQTT protocol

The MQTT protocol is lightweight in the sense that clients are small, and it uses network bandwidth efficiently. The MQTT protocol supports assured delivery and fire-and-forget transfers. In the protocol, message delivery is decoupled from the application. The extent of decoupling in an application depends on the way an MQTT client and MQTT server are written. Decoupled delivery frees up an application from any server connection, and from waiting for messages. The interaction model is like email, but optimized for application programming.

The MQTT V3.1 protocol is published; see MQTT V3.1 Protocol Specification. The specification identifies a number of distinctive features about the protocol:

- It is a publish/subscribe protocol.

  In addition to providing one-to-many message distribution, publish/subscribe decouples applications. Both features are useful in applications that have many clients.
- It is not dependent in any way on the message content.
- It runs over TCP/IP, which provides basic network connectivity.
- It has three qualities of service for message delivery:

  "At most once"
  Messages are delivered according to the best efforts of the underlying Internet Protocol network. Message loss might occur.

  Use this quality of service with communicating ambient sensor data, for example. It does not matter if an individual reading is lost, if the next one is published soon after.

  "At least once"
  Messages are assured to arrive but duplicates might occur.

  "Exactly once"
  Messages are assured to arrive exactly once.

  Use this quality of service with billing systems, for example. Duplicate or lost messages might lead to inconvenience or imposing incorrect charges.
- It is economical in the way it manages the flow of messages on the network. For example, the fixed-length header is only 2 bytes long, and protocol exchanges are minimized to reduce network traffic.
- It has a "Last Will and Testament" feature that notifies subscribers of the abnormal disconnection of a client from the MQTT server. See “Last will and testament publication” on page 121.

MQTT version 3.1 is supported by IBM IBM WebSphere MQ and IBM MessageSight. MQTT is implemented over TCP/IP. Another version of the protocol, MQTT-S, is available for non-TCP/IP networks. See MQTT-S version 1.2 specification.

MQTT communities

IBM is running IBM Developer Messaging community for MQTT developers that are writing applications for IBM MessageSight and IBM WebSphere MQ.

MQTT.org is a good place to go to learn about and discuss implementations and extensions to the MQTT protocol.

MQTT is an open source Eclipse project, under the Eclipse Technology Project. The Paho community is developing open source clients and servers. See Eclipse Paho.

Getting started with MQTT clients

You can get started developing a mobile or machine-to-machine (M2M) app by building and running a sample MQTT client app that uses an MQTT client library. The sample apps, and associated client libraries, are available in the Mobile Messaging and M2M Client Pack from IBM. There are versions of the apps and client libraries written in Java, in JavaScript, and in C. You can run these apps on most platforms and devices, including Android devices and products from Apple.

To build and run your app, you need some experience of building apps for the target device or platform and the programming language being used. A little experience is usually enough to get a sample app up and running on your chosen device or platform.

If you use an enterprise-strength MQTT server such as IBM WebSphere MQ or IBM MessageSight, you can exchange information from your sample app with your existing enterprise apps.

Your objectives are as follows:

1. Choose an MQTT server to which you can connect the client app.
2. Download the Mobile Messaging and M2M Client Pack.
3. Build, for your target device or platform, the sample apps from the client pack.
4. Verify that the samples behave as expected by connecting them to the MQTT server.

As a result of building and testing the sample apps for your device or platform, you create a working development environment that you can then use to build your own client apps.

The Mobile Messaging and M2M Client Pack contains the MQTT SDK. This SDK provides you with the following resources:

- Sample MQTT client apps written in Java, in JavaScript, and in C.
- MQTT client libraries that support these client apps, and enable them to run on most platforms and devices.

The SDK also includes the source code for the MQTT client for C. You can adapt this source code to build MQTT client libraries for C for other platforms. For help to do this, see "Building the MQTT client for C libraries" on page 27. The source code for the MQTT client for C is also available with an open source license from Eclipse Paho.

The following articles guide you through the platform-specific steps for building and running a sample MQTT app on a desktop computer, or on a mobile device for Android or from Apple:

- "Getting started with the MQTT client for Java" on page 11
- "Getting started with the MQTT client for Java on Android" on page 16
- V 7.5.0.1 "Getting started with the MQTT messaging client for JavaScript" on page 21
- "Getting started with the MQTT client for C" on page 23
- "Getting started with the MQTT client for C on iOS" on page 43

To develop a new MQTT application, you must have or acquire the following skills:

- Programming in the language that is required for the device or platform.
- Programming for the target device or platform.
- Designing publish/subscribe applications.
- Designing programs for the MQTT programming model.
- Designing programs to run on your chosen mobile device.
- Using SSL and JAAS to secure programs.

You do not need any network programming skills to connect an MQTT client with another device or application, because MQTT is a messaging and queuing system. The MQTT client libraries manage the network connections for your application.

To integrate your MQTT client with existing enterprise applications, you have two choices. You can share the MQTT publish/subscribe topics with (for example) an IBM WebSphere MQ or JMS application, or you can write your own integration adapter as another MQTT client.

Sources of information to consult today are:

- Developing applications for WebSphere MQ Telemetry
- MQTT.org

- Eclipse Paho

Related concepts

"Getting started with MQTT servers" on page 128

Getting started with the MQTT client for Java

Get up and running with the MQTT client for Java sample applications, using either IBM MessageSight or IBM WebSphere MQ as the MQTT server. The sample applications use a client library from the MQTT software development toolkit (SDK) from IBM. The `SampleAsyncCallBack` sample application is a model for writing MQTT applications for Android and other event-driven operating systems.

- You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible". See System requirements for IBM Mobile Messaging and M2M Client Pack.
- If there is a firewall between your client and the server, check that it does not block MQTT traffic.

The purpose of the task is to check that you can build and run an MQTT client for Java sample application, connect it to IBM WebSphere MQ or IBM MessageSight as the MQTT version 3 server, and exchange messages.

Follow this task to run the sample application from the Eclipse workbench, or from a command line. The steps in the example are for Windows. With small modifications, you can run the sample application on any platform that supports JSE 1.5 or above.

You can run applications on the same server as IBM WebSphere MQ, where the environment for running applications that connect to IBM WebSphere MQ is configured for you. Follow the task "Configuring the MQTT service from the command line" on page 132 to install and configure IBM WebSphere MQ with the IBM WebSphere MQ Telemetry option on Windows or Linux. When the environment is installed and configured, run the sample application, `MQTTV3Sample`, to verify the installation.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See "Getting started with MQTT servers" on page 128.
2. Optional: Configure the MQTT server.
   - On IBM WebSphere MQ, you must complete one or other the following tasks to set up a queue manager and configure its telemetry (MQXR) service:
     - "Configuring the MQTT service from the command line" on page 132
     - "Configuring the MQTT service with IBM WebSphere MQ Explorer" on page 134
   - On other servers, consult the server documentation. No configuration steps are required for the Really Small Message Broker. See Really Small Message Broker.
3. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

   There is no installation program, you just expand the downloaded file.

   a. Download the Mobile Messaging and M2M Client Pack.

   b. Create a folder where you are going to install the SDK.

      You might want to name the folder `MQTT`. The path to this folder is referred to here as *`sdkroot`*.

   c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *`sdkroot`*. The expansion creates a directory tree that starts at *`sdkroot`*\SDK.
4. Install a Java development kit (JDK) Version 6 or later.

   Because you are developing a Java app for Android, the JDK must come from Oracle. You can get the JDK from Java SE Downloads.
5. Compile and run one or more of the MQTT client for Java sample applications:
   - "Compile and run the Paho sample programs from the command line" on page 12
   - "Compile and run all the MQTT client sample Java apps from Eclipse" on page 14

- "Getting started with the MQTT client for Java on Android" on page 16

The following MQTT client sample Java apps are included in the SDK:

MQTTV3Sample
The sample is also included with IBM WebSphere MQ and links to the `com.ibm.micro.client.mqttv3.jar` package.

Sample
`Sample` is in the Paho package and links to the `org.eclipse.paho.client.mqttv3` package. It is similar to `MQTTV3Sample`; it waits until each MQTT action is completed.

SampleAsyncWait
`SampleAsyncWait` is in the `org.eclipse.paho.client.mqttv3` package. It uses the asynchronous MQTT API; it waits on a different thread until an action completes. The main thread can do other work until it synchronizes on the thread that is waiting for the MQTT action to complete.

SampleAsyncCallBack
`SampleAsyncCallBack` is in the `org.eclipse.paho.client.mqttv3` package. It calls the asynchronous MQTT API. The asynchronous API does not wait for MQTT to complete processing a call; it returns to the application. The application carries on with other tasks, then waits for the next event to arrive for it to process. MQTT posts an event notification back to the application when it completes processing. The event driven MQTT interface is suited to the service and activity programming model of Android and other event driven operating systems.

As an example, look at how the `mqttExerciser` sample integrates MQTT into Android using the service and activity programming model.

mqttExerciser
`mqttExerciser` is a sample program for Android. Because it is built and run differently, it is described separately. See "Getting started with the MQTT client for Java on Android" on page 16.

You compiled and ran the MQTT Java sample applications that are connected to IBM WebSphere MQ or IBM MessageSight as the MQTT server.

Study the Javadoc reference information; see step "3" on page 15 of "Compile and run all the MQTT client sample Java apps from Eclipse" on page 14. Alternatively open the Javadoc html files that are in the SDK`\clients\java\doc\javadoc` directory in the Mobile Messaging and M2M Client Pack.

Compile and run the Paho sample programs from the command line

Compile and run the Paho sample application `Sample.java` from the command line. The sample is in the MQTT SDK. The sample is built with the MQTT Paho client libraries in the `org.eclipse.paho.client.mqttv3` package. It demonstrates an MQTT publisher and subscriber. Two other Paho samples in the same directory can be built and run the same way. They differ by calling the MQTT library asynchronously.

Do steps "1" on page 11 to "4" on page 11 in the main task to configure an MQTT server and download the Mobile Messaging and M2M Client Pack.

Compile and run `Sample.java` from the SDK client samples subdirectory `SDK\clients\java\samples`. The Java code is in the directory, `SDK\clients\java\samples\org\eclipse\paho\sample\mqttv3app`.

1. Create a script in the client samples directory to compile and run `Sample` on your chosen platform.

   The following script compiles and runs the sample on Windows.

```
@echo on
setlocal
set classpath=
set JAVADIR=C:\Program Files\IBM\Java70\bin
"%JAVADIR%\javac"
-cp ..\org.eclipse.paho.client.mqttv3.jar .\org\eclipse\paho\sample\mqttv3app\Sample.java
start "Sample Subscriber" "%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -a subscribe -b localhost -p 1883
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
"%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -b localhost -p 1883
pause
endlocal
```

Figure 5. Compile and run `Sample.java`

2. Run the script.

   Results:

```
Connected to tcp://localhost:1883 with client ID SampleJavaV3_subscribe
Subscribing to topic "Sample/#" qos 2
Press <Enter> to exit
Time:   2012-11-09 17:23:22.718  Topic: Sample/Java/v3  Message:        Message
from blocking MQTTv3 Java client sample  QoS:   2
```

Figure 6. `MQTTV3Sample` Subscriber

```
Connecting to tcp://localhost:1883 with client ID SampleJavaV3_publish
Connected
Publishing at: 2012-11-09 17:22:07.734 to topic "Sample/Java/v3" qos 2
Disconnected
```

Figure 7. `MQTTV3Sample` Publisher

   If you leave the subscriber application running, you cannot run the same subscriber again. The client identifier of the new subscriber is the same as the old subscriber. You cannot run two MQTT clients with the same client identifier at the same time. You can set the `-i` option to set the client identifier so you can run different subscribers at the same time.

   If you run the same client again, you can take advantage of the `-c` option to start and start the client with `cleansession` set to false. With that option you can explore the behavior of interrupted client sessions.

3. End the subscriber by pressing enter or closing the window.

Create scripts to compile and run the other samples in the `SDK\clients\java\samples\org\eclipse\paho\sample\mqttv3app` subdirectory. Copy the script in Figure 5 on page 13 and replace `Sample` with `SampleAsyncWait` or `SampleAsyncCallBack`. The other samples are asynchronous versions of the synchronous `Sample` program.

`SampleAsyncWait`
: `SampleAsyncWait` is in the `org.eclipse.paho.client.mqttv3` package. It uses the asynchronous MQTT API; it waits on a different thread until an action completes. The main thread can do other work until it synchronizes on the thread that is waiting for the MQTT action to complete.

`SampleAsyncCallBack`
: `SampleAsyncCallBack` is in the `org.eclipse.paho.client.mqttv3` package. It calls the asynchronous MQTT API. The asynchronous API does not wait for MQTT to complete processing a call; it returns to the application. The application carries on with other tasks, then waits for the next event to arrive for it to process. MQTT posts an event notification back to the application when it completes processing. The event driven MQTT interface is suited to the service and activity programming model of Android and other event driven operating systems.

  As an example, look at how the `mqttExerciser` sample integrates MQTT into Android using the service and activity programming model.

The asynchronous samples demonstrate how to reduce the amount of time that an MQTT application blocks while it is waiting for the MQTT client. It is important to eliminate blocking calls in the main thread to increase responsivity and battery life in a mobile environment.

The samples demonstrate two patterns for calling asynchronous interfaces in the MQTT client.

1. `SampleAsyncWait` does not block while the MQTT is waiting for network interactions.
2. `SampleAsyncCallBack` does not block waiting for the MQTT client to complete any actions. The latter is necessary when a JavaScript page is calling the MQTT client from a browser. JavaScript pages must not block. Responses to actions must be posted back to the main browser thread, which calls the MQTT event handler you write to process the notification.

Compile and run all the MQTT client sample Java apps from Eclipse

Compile and run the MQTT client sample Java apps that are in the Mobile Messaging and M2M Client Pack. They demonstrate a MQTT publisher and subscriber.

Compile and run the MQTT Java samples, `Sample`, and `MQTTV3Sample` in Eclipse. `Sample` is in the *sdkroot*`\SDK\clients\java\samples\org\eclipse\paho\sample\mqttv3app` SDK clients subdirectory and `MQTTV3Sample.java` is in *sdkroot*`\SDK\clients\java\samples`.

1. Download Eclipse IDE for Java Developers.
2. Create a Java project that is called `MQTT Samples` in Eclipse.
   a) **File** > **New** > **Java project** and type `MQTT Samples`. Click **Next**.

   Check the JRE is at the correct or later version.JSE must be at version 1.5 or later.

   b) In the **Java Settings** window, click **Link additional source folders**.

   c) Browse to the directory where you installed the MQTT Java SDK folder to. Select the *sdkroot*`\SDK\clients\java\samples` folder and click **OK** > **Next** > **Finish**.

   d) In the **Java Settings** window, click **Libraries** > **Add External Jars**

   e) Browse to the directory where you installed the MQTT Java SDK folder to. Locate the *sdkroot*`\SDK\clients\java` folder and select the `org.eclipse.paho.client.mqttv3.jar` and `com.ibm.micro.client.mqttv3.jar` files; click **Open** > **Finish**.

   The `MQTTV3Sample.java` sample links to `com.ibm.micro.client.mqttv3.jar`, and the samples in the paho directory tree link to `org.eclipse.paho.client.mqttv3.jar`. The `com.ibm.micro.client.mqttv3.jar` is retained so that existing MQTT applications continue to build and run without change.

The `MQTT Samples` project builds with some warnings, but no errors.

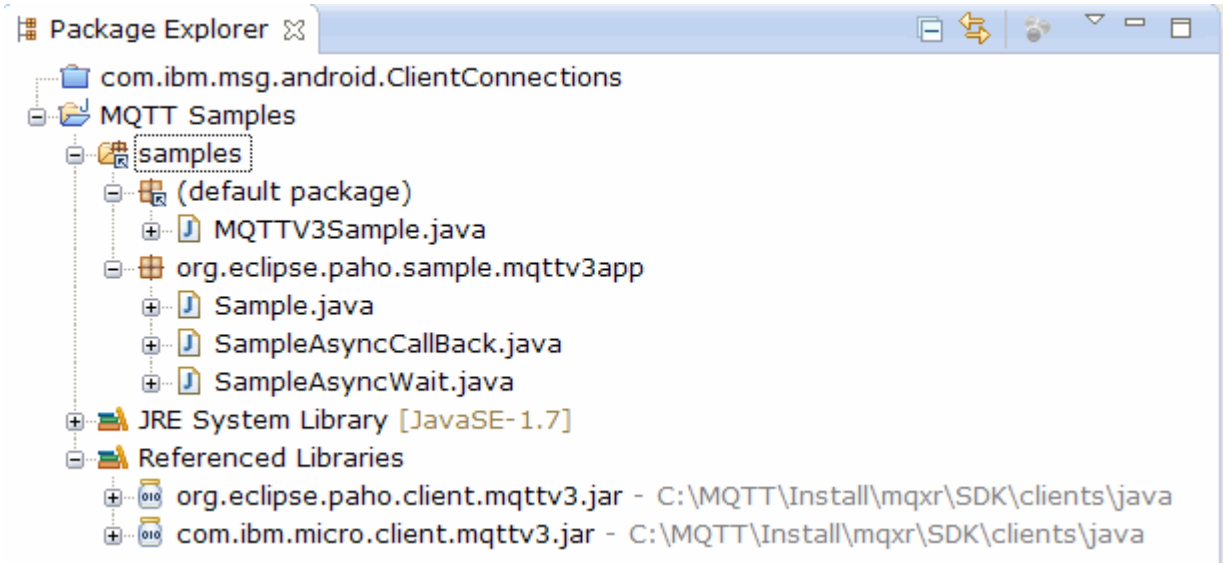


Figure 8. MQTT client for Java project

3. Optional: Install the MQTT client Javadoc.

   With the MQTT client Javadoc installed, the Java editor describes the MQTT classes in hover help.

   a) Open **Package Explorer** > **Referenced Libraries** in your Java project. Right click `org.eclipse.paho.client.mqttv3.jar` **> Properties**.

   b) In the Properties navigator click **Javadoc Location**.

   c) Click **Javadoc URL** > **Browse** in the **Javadoc Location** page, and find the `SDK\clients\java\doc\javadoc` folder **> OK**.

   d) Click **Validate** > **OK**

      You are prompted to open a browser to view the documentation.

   e) Repeat this procedure for the `com.ibm.micro.client.mqttv3.jar` file.

4. Create a publisher and subscriber runtime configuration to run the `mqttv3app.Sample` application.

   a) Right-click the **Sample** class, click **Run as** > **Run configurations**.

   b) Right-click **Java Application** > **New** and type the name `SampleSubscriber`.

   c) Click the arguments tab, and type the program arguments followed by **Apply**.

   ```
   -a subscribe -b localhost -p 1883
   ```

   d) Repeat the last step to create a `SamplePublisher` configuration by omitting the `-a subscribe` parameter.

5. Run the `mqttv3app.Sample` subscriber followed by the publisher.

   a) Click **Run** > **Run configurations**

   b) Click **SampleSubscriber** > **Run**.

      Open the **Console** view. The subscriber is waiting for a publication.

   ```
   Connected to tcp://localhost:1883 with client ID SampleJavaV3_subscribe
   Subscribing to topic "Sample/#" qos 2
   Press <Enter> to exit
   ```

   c) Click **SamplePublisher** > **Run**.

      Open the **Console** view. You see the publication that is created by the publisher:

   ```
   Connecting to tcp://localhost:1883 with client ID SampleJavaV3_publish
   Connected
   Publishing at: 2012-11-09 14:09:29.859 to topic "Sample/Java/v3" qos 2
   Disconnected
   ```

   d) Switch console views to the subscriber console.

      The switch consoles icon is  .

      The subscriber received the publication:

   ```
   ...
   Time: 2012-11-09 14:09:30.593  Topic: Sample/Java/v3  Message: Message from blocking
   MQTTv3 Java client sample  QoS: 2
   ```

6. Optional: Create a publisher and subscriber runtime configuration for `MQTTV3Sample`.

   a) Right-click the **MQTTV3Sample** class, click **Run as** > **Run configurations**.

   b) Right-click **Java Application** > **New** and type the name `MQTTV3SampleSubscriber`.

   c) Click the arguments tab, and type the program arguments followed by **Apply**.

   ```
   -a subscribe -b localhost -p 1883
   ```

   d) Repeat the last step to create an `MQTTV3SamplePublisher` configuration by omitting the `-a subscribe` parameter.

7. Optional: Run the `MQTTV3Sample` subscriber followed by the publisher.
   a) Click **Run** > **Run configurations**
   b) Click **MQTTV3SampleSubscriber** > **Run**.

      Open the **Console** view. The subscriber is waiting for a publication.

      ```
      Connected to tcp://localhost:1883
      Subscribing to topic "MQTTV3Sample/#" qos 2
      Press <Enter> to exit
      ```

   c) Click **MQTTV3SamplePublisher** > **Run**.

      Open the **Console** view. You can see the publication that is created by the publisher.

      ```
      Connected to tcp://localhost:1883
      Publishing to topic "MQTTV3Sample/Java/v3" qos 2
      Disconnected
      ```

   d) Switch console views to the subscriber console.

      The switch consoles icon is  .

      The subscriber received the publication:

      ```
      MQTTV3Sample/Java/v3
      Message: Message from MQTTv3 Java client
      QoS:  2
      ```

Getting started with the MQTT client for Java on Android

You can install an MQTT client sample Java app for Android that exchanges messages with an MQTT server. The app uses a client library from the MQTT SDK from IBM. You can either build the app yourself, or download a pre-built sample app.

- For supported and reference MQTT client platforms, see System requirements for IBM Mobile Messaging and M2M Client Pack.
- If there is a firewall between your client and the server, check that it does not block MQTT traffic.
- The MQTT client sample app works on Ice Cream Sandwich (Android 4.0) and up. This version of Android also gives a crisper display resolution on tablets.

The MQTT client sample Java app for Android is called "`mqttExerciser`". This app uses a client library from the MQTT SDK, and exchanges messages with an MQTT server.

You can either build the sample app yourself then export it from Eclipse as `mqttExerciser.apk`, or use the pre-built sample app that is available as file `mqttExerciser.apk` in the *sdkroot*`\SDK\clients\android\samples\apks` folder of the Mobile Messaging and M2M Client Pack. If you choose to build the app yourself, the development environment that you build is tailored to include mobile messaging into apps for Android. This should help you when you begin to include mobile messaging in your own apps.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See “Getting started with MQTT servers” on page 128.
2. Get the right tools.

   Install a Java development kit (JDK) Version 6 or later. Because you are developing a Java app for Android, the JDK must come from Oracle. You can get the JDK from Java SE Downloads.

   You also need an Eclipse development environment. This must be Eclipse 3.6.2 (Helios) or greater. Eclipse must have a Java compiler level of at least 6, to match your JDK. You can get all this from the Eclipse Foundation.

   Finally, you need the Android SDK. You can get this from Get the Android SDK.

3. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

   There is no installation program, you just expand the downloaded file.

   a. Download the Mobile Messaging and M2M Client Pack.

   b. Create a folder where you are going to install the SDK.

      You might want to name the folder `MQTT`. The path to this folder is referred to here as *`sdkroot`*.

   c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *`sdkroot`*. The expansion creates a directory tree that starts at *`sdkroot`*`\SDK`.

4. Optional: Build the mqttExerciser sample app for Android.

   Configure the Eclipse and Android tools, and import and build the `mqttExerciser` project from the MQTT SDK.

   Note: If you do not want to do this right now, you can use the pre-built sample app that is available as file `mqttExerciser.apk` in the *`sdkroot`*`\SDK\clients\android\samples\apks` folder of the MQTT SDK.

   a) Start the Eclipse development environment with the JRE from the JDK.

   ```
   eclipse -vm "JRE path"
   ```

   b) Select and install a set of packages and platforms from the Android SDK.

      See Adding Platforms and Packages for the list of platforms and packages that Google recommends.

      Note: The SDK platform must be Android API level 16 or later. With earlier API levels, the project cannot be compiled successfully.

   c) Add the Android Development Tools (ADT) plug-in to Eclipse.

   d) Import the sample `mqttExerciser` app project into Eclipse, and fix errors.

      i) Import the sample app project from the MQTT SDK, in the path *`sdkroot`*`\SDK\clients\android\samples\mqttExerciser`.

      The **Problems** view lists many build errors. You resolve the build errors in the next few steps.

      ii) Copy the `org.eclipse.paho.client.mqttv3.jar` library into the **libs** folder in the Android project. 

For example, on Windows, this is under the *`sdkroot`*`\SDK\clients\java` folder. A **File Operation** window is displayed. Accept the **Copy files** selection then click **OK**.

      iii) Right-click the project folder, `com.ibm.msg.android`; click **Android tools...** > **Add Support Library...**. Read and accept the license terms, then click **Install**.

      iv) Right-click the project folder, `com.ibm.msg.android`; click **Android tools...** > **Fix Project Properties**.

      v) If the workspace still has about 84 errors, referring to overriding a super class method, the compiler compliance level is probably set to 1.5 or lower. Android SDK version 16 expects the compiler compliance level to be no greater than 1.5. To fix the remaining errors, complete the following steps:

         a) Check and (if necessary) update your Android SDK and the corresponding Eclipse plugins to Android SDK version 17.

         b) Right-click the **com.ibm.msg.android** project folder, then select **Properties** > **Java Compiler**. Check the Compiler compliance level, set it to at least 1.6, then rebuild the workspace.

   The project builds, with some warnings, and no errors.

5. Install and start the MQTT client sample Java app on an Android device.

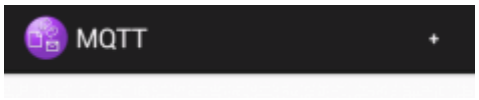
   See the `developer.android.com` page Running your app.

If you built the app yourself as an Eclipse project, you can start the app from Eclipse.

If you have the application package (APK) file `mqttExerciser.apk`, you can install it outside of Eclipse by using the Android Debug Bridge (ADB) install command. This command takes the location of the APK file as an argument. If you are using the pre-built sample app, the location is *sdkroot*`\SDK\clients\android\samples\apks\mqttExerciser.apk`.

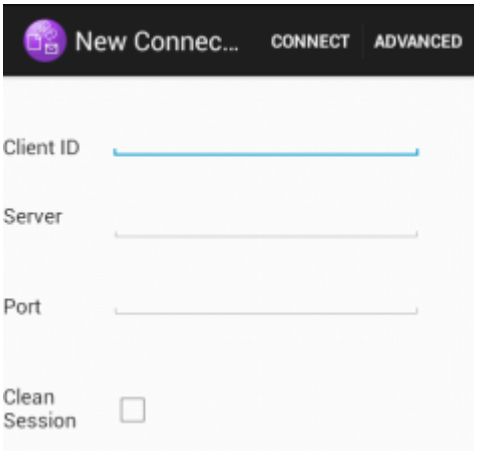
6. Use the mqttExerciser sample app for Android to connect, subscribe, and publish to a topic.
   a) Open the MQTT client sample Java app for Android.

      This window is open in your Android device:



   b) Connect to an MQTT server.

      i) Click the **+** sign to open a new MQTT connection.



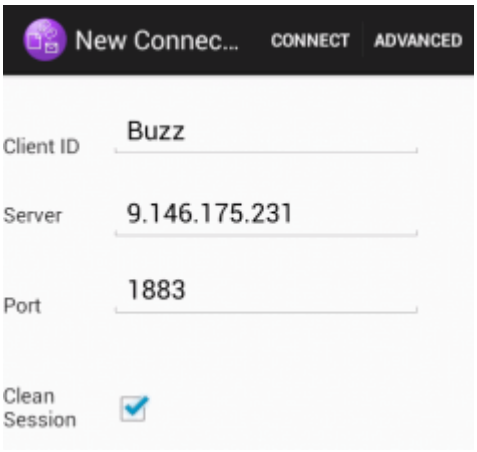
      ii) Enter any unique identifier into the **client ID** field. Be patient, the keystrokes can be slow.

      iii) Enter into the **Server** field the IP address of your MQTT server.

      This is the server that you chose in the first main step. The IP address must not be `127.0.0.1`

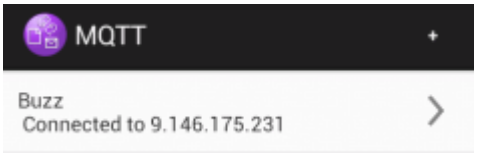
      iv) Enter the port number of the MQTT connection.

      The default port number for a normal MQTT connection is 1883.



      v) Click **Connect**.

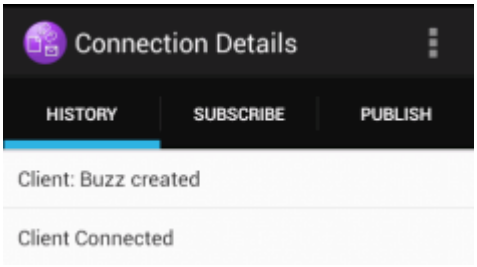
      If the connection is successful, you see a "Connecting" message followed by this window:



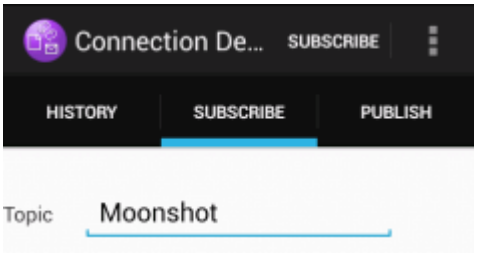
c) Subscribe to a topic.

i) Click the **Connected** message.

The **Connection Details** window opens with the history listed:



ii) Click the **Subscribe** tab, and enter a topic string.

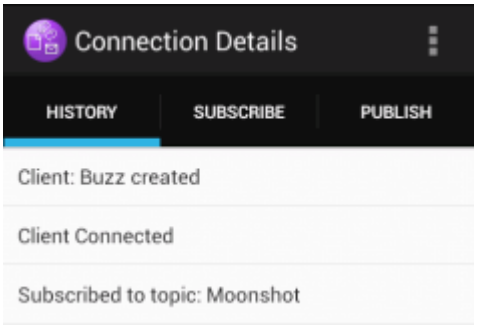


iii) Click the **Subscribe** action.

A "Subscribed" message appears for a short time.

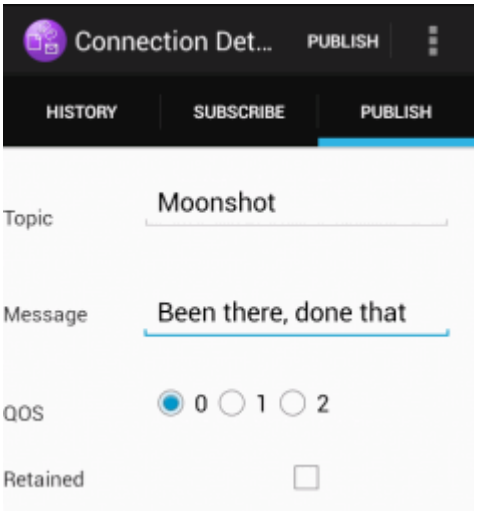
iv) Click the **History** tab.

The history now includes the subscription:



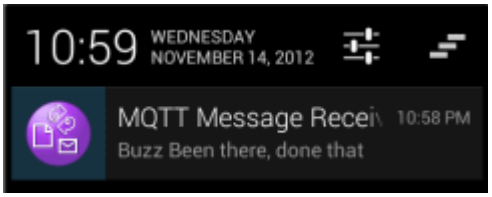
d) Now publish to the same topic.

i) Click the **Publish** tab, and enter the same topic string as you did for subscribing. Enter a message.

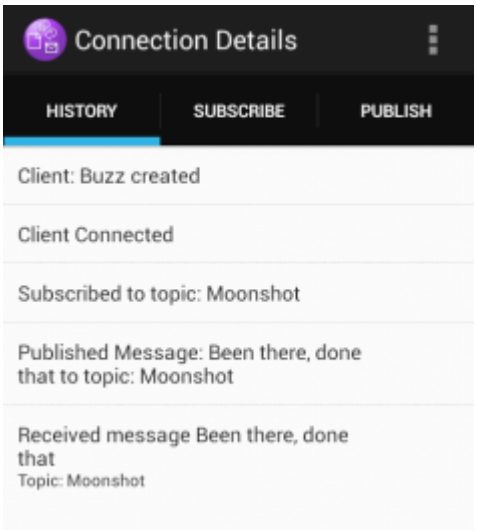


ii) Click the **Publish** action.

Two messages are displayed for a short time, "Published" followed by "Subscribed". The publication is displayed in the status area (pull the separator bar down to open the status window).



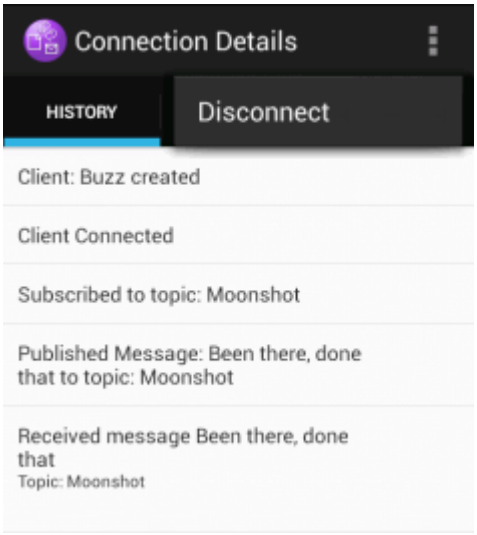
iii) Click the **History** tab to view the full history.



e) Disconnect the client instance.

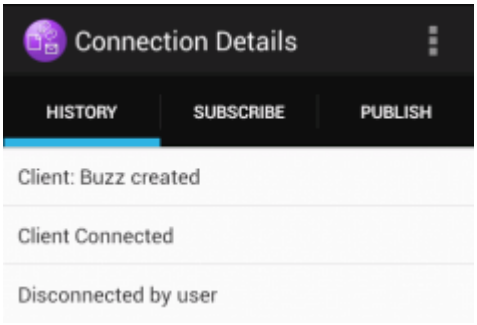
i) Click the menu icon in the action bar. 

The MQTT client sample Java app for Android adds a **Disconnect** button to the MQTT **Connection Details** window.

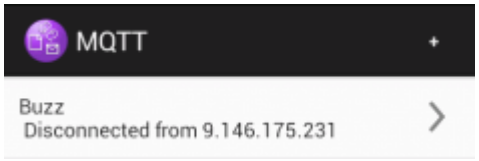


ii) Click **Disconnect**.

The connected status changes to disconnected:



f) Click **Back** to return to the list of MQTT client sample Java app sessions.



- Click the plus sign to start a new MQTT client sample Java app session.
- Click the disconnected client to reconnect it.
- Click **Back** to return to the launchpad.

g) Click the task button to list running apps. Locate the MQTT client sample Java app. Swipe the icon off the screen to close it.

If you built the sample app yourself, you are ready to start developing your own Android apps that call MQTT libraries to exchange messages. You can model your Android apps on the classes in `mqttExerciser`. To study the sample, generate the Javadoc for the classes in `com.ibm.msg.android` and `com.ibm.msg.android.service` in the `mqttExerciser` project.

Related information

Managing Projects from Eclipse with ADT

Getting started with the MQTT messaging client for JavaScript

You can get started with the MQTT messaging client for JavaScript by displaying the messaging client sample home page, and browsing the resources to which it links. To display this home page, you configure an MQTT server to accept connections from the MQTT messaging client sample JavaScript pages, then you type the URL that you have configured on the server into a web browser. The MQTT messaging

client for JavaScript automatically starts on your device, and the messaging client sample home page is displayed. This page contains links to utilities, programming interface documentation, a tutorial, and other useful information.

For advanced use, or use in production, you will want to reshape or remove the messaging client sample home page. Please note that user interfaces resulting from the sample code are not warranted to be compliant to any accessibility standards or accessibility requirements.

You need an MQTT server to support the MQTT messaging client for JavaScript. This server must support the MQTT V3.1 protocol over WebSockets. IBM MessageSight, and IBM WebSphere MQ Version 7.5.0, Fix Pack 1 and later versions, support the MQTT protocol over WebSockets. See "Getting started with MQTT servers" on page 128. To install IBM WebSphere MQ for a free 90-day evaluation, see "Installing IBM WebSphere MQ" on page 130.

The WebSocket protocol is recently established. If there is a firewall between your client and the server, check that it does not block WebSockets traffic. Similarly, if your browser does not yet support the WebSocket protocol[1] you won't be able to use the client utility or the tutorials that are available from the messaging client sample home page. The Table 1 on page 22 table lists the browsers whose most recent versions have been tested and shown to work with the messaging client.

Table 1. Supported browsers for use with the MQTT messaging client for JavaScript

Android	**iOS**	**Linux**	**Windows**
Firefox for Android 19.0 and later Chrome for Android 25.0 and later	Safari 6.0 and later Chrome 14.0 and later	Firefox 6.0 and later Chrome 14.0 and later	Firefox 6.0 and later Chrome 14.0 and later

Most of the steps in this task are to configure the MQTT server. All that is required to access the messaging client for JavaScript is to run a browser that supports the WebSocket protocol.

On IBM WebSphere MQ, follow the steps to enable IBM WebSphere MQ Telemetry by creating the sample channels. Connect to the sample default MQTT WebSockets channel on port 1883. The messaging client sample home page URL is `http://`*`hostname`*`:1883` on IBM WebSphere MQ.

On IBM MessageSight, install and set up the appliance, configure the messaging hub to accept connections, and create an MQTT WebSockets endpoint.

1. Download the Mobile Messaging and M2M Client Pack, and choose an MQTT server to which you can connect the client app.

   See "Getting started with MQTT clients" on page 9.
2. Configure your MQTT server to accept connections from the MQTT messaging client for JavaScript sample HTML pages.
   - On IBM WebSphere MQ:
     - If you already have a IBM WebSphere MQ queue manager that is configured for MQTT, alter the protocol in the channel definition to support both MQTT and HTTP. See **ALTER CHANNEL**.
     - To create an IBM WebSphere MQ queue manager and configure the sample MQTT WebSockets endpoint, complete either of the following tasks:
       - "Configuring the MQTT service from the command line" on page 132
       - "Configuring the MQTT service with IBM WebSphere MQ Explorer" on page 134
3. Open a web browser on your device.
4. Type the URL of the messaging client sample home page.
   - On IBM WebSphere MQ, this is `http://`*`hostname`*`:1883`

[1] Specifically, if it does not support the RFC 6455 (WebSocket) standard.

- On IBM MessageSight, this is `http://hostname:port`

  where *hostname* is the DNS name or IP address of the Ethernet socket that you configured on your IBM MessageSight appliance as the endpoint to which the client is to connect, and *port* is the TCP/IP port number you assigned to the endpoint for the client.

The messaging client sample home page is displayed.

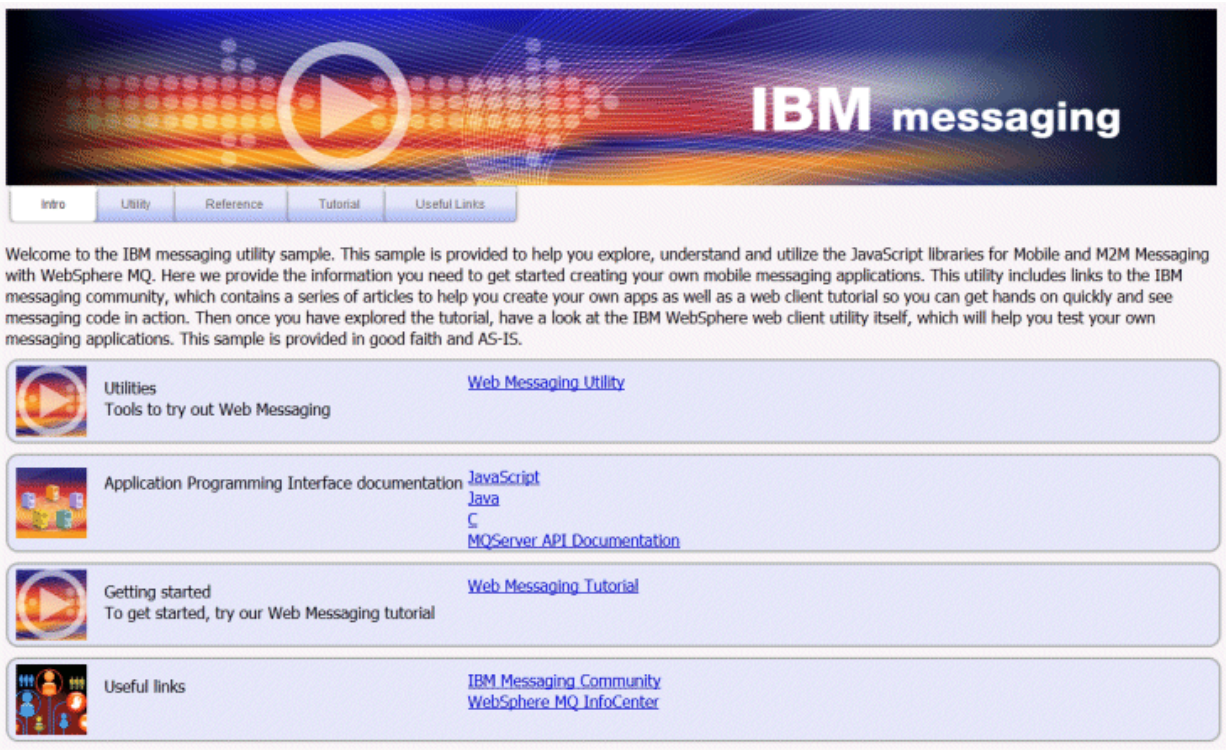


Figure 9. MQTT messaging client for JavaScript sample home page

You have configured an MQTT channel for WebSockets.

In the sample home page of the MQTT messaging client for JavaScript, click **Web Messaging Utility** to try out different functions in the messaging client API. For example, you can connect to the queue manager, subscribe for messages, then publish some messages. You can also click **Web Messaging Tutorial** to learn how to create a web page that calls the MQTT messaging client for JavaScript API.

Related concepts

"The MQTT messaging client for JavaScript and web apps" on page 110

"How to program messaging apps in JavaScript" on page 113

Related tasks

"Connecting the MQTT messaging client for JavaScript over SSL and WebSockets" on page 72

Connect your web app securely to IBM WebSphere MQ by using the MQTT messaging client for JavaScript sample HTML pages with SSL and the WebSocket protocol.

Getting started with the MQTT client for C

Get up and running with the sample MQTT client for C on any platform on which you can compile the C source. Verify that you can run the sample MQTT client for C with either IBM MessageSight or IBM WebSphere MQ as the MQTT server.

- If there is a firewall between your client and the server, check that it does not block MQTT traffic.

- For the supported and reference MQTT client for C platforms. See System requirements for IBM Mobile Messaging and M2M Client Pack.

Follow this task to compile and run the sample MQTT client for C on Windows from the command line or from Microsoft Visual Studio 2010. Microsoft Visual Studio 2010 is also used to compile the client in the command-line example. Modify the command-line scripts to compile and run the sample on other platforms.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See "Getting started with MQTT servers" on page 128.
2. Install a C development environment on the platform on which you are building.

   The makefiles in the examples in this topic target the following tools:

   - 

For iOS, on Apple Mac with OS X 10.8.2 with the iOS development tools from Xcode.
   - 

For Linux, gcc version 4.4.6 from Red Hat® Enterprise Linux version 6.2.

     The minimum supported level of the `glibc` C library is 2.12, and of the Linux kernel is 2.6.32.
   - 

For Microsoft Windows, Visual Studio version 10.0.
3. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

   There is no installation program, you just expand the downloaded file.

   a. Download the Mobile Messaging and M2M Client Pack.

   b. Create a folder where you are going to install the SDK.

      You might want to name the folder `MQTT`. The path to this folder is referred to here as *`sdkroot`*.

   c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *`sdkroot`*. The expansion creates a directory tree that starts at *`sdkroot`*`\SDK`.
4. Optional: Follow the steps in "Building the MQTT client for C libraries" on page 27.

   Do this step only if the Mobile Messaging and M2M Client Pack does not include the C client library for your target platform.
5. Compile and run the MQTT client sample C app, `MQTTV3Sample.c`.

   - From the command line, follow the steps in "Compile and run the MQTT client sample C app from the command line" on page 24.
   - From an IDE, follow the steps in "Compile and run the MQTT client sample C app from Microsoft Visual Studio" on page 25.

Compile and run the MQTT client sample C app from the command line

Compile and run the MQTT client sample C app from the command line. The sample is in the MQTT SDK. It demonstrates an MQTT publisher and subscriber.

Install a C development environment; for example the Microsoft Visual Studio 2010 as used in the example.

Compile and run the C sample, `MQTTV3Sample`, in the SDK clients subdirectory, *`sdkroot`*`\SDK\clients\c\samples`.

Create a script in the client samples directory to compile and run `Sample` on your chosen platform.

The following script compiles and runs the sample on a Windows 32-bit platform, built with Microsoft Visual Studio 2010. Run the script from the *`sdkroot`*`\SDK\clients\c\samples` subdirectory.

```
@echo off
setlocal
call "C:\Program Files\Microsoft Visual Studio 10.0\VC\vcvarsall.bat" x86
cl  /nologo /D "WIN32" /I "..\include" "MQTTV3Sample.c" /link /
nologo ..\windows_ia32\mqttv3c.lib
```

```
set path=%path%;..\windows_ia32;
start "MQTT Subscriber" MQTTV3Sample -a subscribe -b localhost -p 1883
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
MQTTV3Sample -b localhost -p 1883
pause
endlocal
```

The publisher and subscriber write output to their command windows:

```
Setting environment for using Microsoft Visual Studio 2008 x86 tools.
MQTTV3Sample.c
Connected to tcp://localhost:1883
Publishing to topic "MQTTV3Sample/C/v3" qos 2
Disconnected
Press any key to continue . . .
```

Figure 10. Output from the publisher

```
Connected to tcp://localhost:1883
Subscribing to topic "MQTTV3Sample/#" qos 2
Topic:          MQTTV3Sample/C/v3
Message:        Message from MQTTv3 C client
QoS:            2
```

Figure 11. Output from the subscriber

Compile and run the MQTT client sample C app from Microsoft Visual Studio

Compile and run the MQTT client sample C app from the Microsoft Visual Studio. The sample is in the Mobile Messaging and M2M Client Pack. It demonstrates an MQTT publisher and subscriber.

The example uses Microsoft Visual Studio 2010. You can use other C development environments on Windows and other platforms; for example Eclipse IDE for C/C++ Developers.

Compile and run the C sample, `MQTTV3Sample` with Microsoft Visual Studio 2010. `MQTTV3Sample.c` is in the SDK clients subdirectory, *sdkroot*`\SDK\clients\c\samples`.

1. Start Microsoft Visual Studio.
2. Create a new project from existing code.
   a) Click **File** > **New** > **Project from Existing Code**.
   b) Select **Visual C++** as the type of project to create.
   c) Click **Next**.
3. Specify the parameters in the **Project Location and Source Files** window.
   a) Click **Browse** and locate the *sdkroot*`\SDK\clients\c\samples` directory.
   b) Name the project `MQTTV3Sample`.
   c) Click **Next**.
4. Select **Console application project** in the **Project type** list. Click **Finish**
5. Configure only the Debug configuration.

   By default Microsoft Visual Studio creates both a release and debug configuration. In the tutorial, you configure the debug configuration. To suppress build errors, clear the **Build** option for the release configuration.

   a) Click **Project** > **MQTTV3Sample Properties** > **Configuration Manager**. Select **Release** as the **Active solution configuration** and clear **Build**.
   b) Select **Debug** as the **Active solution configuration** > **Close**.

   Check that you are modifying the Debug configuration in all the following steps.
6. Modify the **C/C++** settings in the **MQTTV3Sample Property Pages**.
   a) In the **MQTTV3Sample Property Pages** window, open **Configuration properties** > **C/C++** > **General**.

b) In the list of general properties, click **Additional Include Directories**, and add your directory path to *sdkroot*\SDK\clients\c\include and click **Apply**.

7. Modify the **Linker** settings
   a) Open **Configuration properties** > **Linker** > **General**.
   b) In the list of general properties, click **Additional Library Directories**, and add your directory path to *sdkroot*\SDK\clients\c\windows_ia32
   c) In the list of Linker properties, click **Command line**. Type `mqttv3c.lib` in the **Additional options** data entry area and click **Apply**.
8. Remove the `MQTTV3SSample.c` source file from the project.
   a) Open the **MQTTV3sample** > **Source Files** folder in the **Solution Explorer** window.
   b) Right-click **MQTTV3SSample.c** > **Exclude from Project**
9. Build the `MQTTV3Sample` project.
   a) Right-click the **MQTTV3sample** project in the Solution Explorer and click **Build**.

   The build completes without any errors.
10. Add two new projects to run **`MQTTV3Sample`** as both a Subscriber and Publisher runtime instance.

    The Publisher and Subscriber projects are to contain the commands to run the **`MQTTV3Sample`**. They are not built, and contain no code.
    a) In the **Solution Explorer**, right-click **Solution `MQTTV3Sample`** > **Add** > **New project**.
    b) Type `Subscriber` into the **Name** field. Leave **Win32 Console Application** selected. Click **OK**.

    The **Win32 Application Wizard** starts.
    c) In the **Win32 Application Wizard**, click **Next**. Check **Empty project** > **Finish**
    d) Repeat these steps to add a Publisher project.
    e) Right-click the Subscriber project and click **Set as StartUp project**

    The **Solution Explorer** window is shown in Figure 12 on page 26.

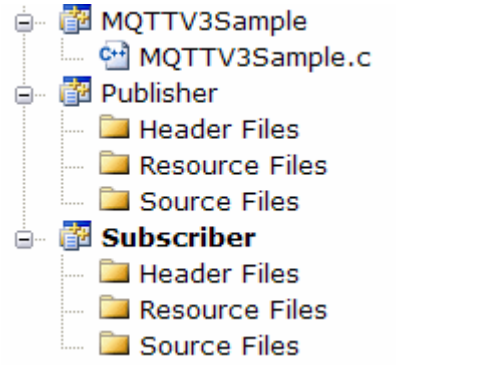


Figure 12. MQTTV3Sample solution

11. Configure the Subscriber property pages.
    a) Right-click **Subscriber** in the Solution Explorer **Properties** > **Configuration Properties** > **Properties** > **Debugging**

    Check the window title is **Subscriber Property Pages**.
    b) Click **Environment**. Type `path=%path%;`*sdkroot*`\SDK\clients\c\windows_ia32` and click **Apply**.

    Change *sdkroot* to suit your environment.
    c) Click **Command**, and replace `$(TargetPath)` by the path to the `MQTTV3Sample` module

    For example, *sdkroot*`\SDK\clients\c\samples\debug\MQTTV3Sample`

    Change *sdkroot* to suit your environment.

d) Click **Command Arguments** and type `-a subscribe -b localhost -p 1883` and click **Apply**.

12. Configure the Publisher property pages.

    Tip: You can switch the property pages project by clicking the projects in the **Solution Explorer** window.

    a) Repeat the Subscriber steps for the Publisher.

    The command argument is `-b localhost -p 1883`

13. Stop the build process that builds the Publisher and Subscriber projects.

    a) In the property pages of any of the projects, click **Configuration Manager** and clear **Build** for the Publisher and Subscriber in both the Release and Debug configurations. Click **Close**.

14. Run the sample.

    a) Click **F5** to start the Subscriber

    b) Right click **Publisher** in the Solution Explorer, **Debug** > **Start a new instance**

The publisher and subscriber output to command windows. Visual Studio closes the publisher window. Look at the subscriber window, which is shown in the following figure, then close the subscriber window.

```
Connected to tcp://localhost:1883
Subscribing to topic "MQTTV3Sample/#" qos 2
Topic:          MQTTV3Sample/C/v3
Message:        Message from MQTTv3 C client
QoS:            2
```

Figure 13. Output from the subscriber

Build and run the asynchronous publisher and subscriber. The examples are `MQTTV3ASample.c` and `MQTTV3ASSample.c` in *sdkroot*`\SDK\clients\c\samples`.

Building the MQTT client for C libraries

Follow these steps to build the MQTT client for C libraries. The topic includes the compile and link switches for a number of platforms, and examples of building the libraries on iOS and Windows.

1. Build the C client library only when necessary. Link the pre-built client libraries in the (Software Development Kit) SDK in the `SDK\clients\c` subdirectory if one matches your target platform.
2. Configure an MQTT server to test the library you build with the MQTT client sample C app. See "Getting started with MQTT servers" on page 128. Verify the server configuration by running one of the MQTT client sample apps.
3. If you are building a secure version of the C library, which supports (Secure Sockets Layer) SSL, you must also build the OpenSSL library. See "Building the OpenSSL package" on page 41.

   Important: Download and redistribution of the OpenSSL package is subject to stringent import and export regulation, and open source licensing conditions. Take careful note of the restrictions and warnings before you decide whether to download the package.

Follow the instructions in "Building the OpenSSL package" on page 41 to download and build the OpenSSL library. You must build the OpenSSL to build a secure version of the MQTT client for C library. You do not require OpenSSL to build an unsecured version of the MQTT library. The steps include examples of building the library for iOS and Windows.

Build the MQTT client for C library by downloading C development library tools, and the MQTT software development toolkit (SDK) onto your build platform. Write a makefile to build the library for your target platform, incorporating the options that are documented in "MQTT build options for different platforms" on page 28. Platform-specific steps to build and run a makefile are given here:

- iOS "Building the MQTT client libraries for C on an Apple Mac for use with iOS devices" on page 30
- Windows "Building the MQTT libraries on Windows" on page 35

1. Install a C development environment on the platform on which you are building.

   The makefiles in the examples in this topic target the following tools:

   - 

For iOS, on Apple Mac with OS X 10.8.2 with the iOS development tools from Xcode.
   - 

For Linux, gcc version 4.4.6 from Red Hat Enterprise Linux version 6.2.

     The minimum supported level of the `glibc` C library is 2.12, and of the Linux kernel is 2.6.32.
   - 

For Microsoft Windows, Visual Studio version 10.0.
2. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

   There is no installation program, you just expand the downloaded file.

   a. Download the Mobile Messaging and M2M Client Pack.

   b. Create a folder where you are going to install the SDK.

      You might want to name the folder `MQTT`. The path to this folder is referred to here as *`sdkroot`*.

   c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *`sdkroot`*. The expansion creates a directory tree that starts at *`sdkroot`*`\SDK`.
3. Expand the source code for the MQTT client for C libraries.

   The source code compressed file is *`sdkroot`*`\SDK\clients\c\source.zip`.
4. Optional: Build OpenSSL.

   See “Building the OpenSSL package” on page 41.
5. Build the MQTT client for C libraries.

   The commands and options to build the libraries are listed in “MQTT build options for different platforms” on page 28.

   Follow the steps in the following examples to write a makefile to build the MQTT client for C libraries for your target platform.

   - “Building the MQTT client libraries for C on an Apple Mac for use with iOS devices” on page 30
   - “Building the MQTT libraries on Windows” on page 35

MQTT build options for different platforms

The following table lists the compiler and build options to build the MQTT client for C libraries on various platforms.

Table 2. MQTT build options for different platforms

Platform	Compiler	Compiler Options	Linker Options	Extra Options
AIX®	gcc	-fPIC -Os -Wall -DREVERSED -I *MQTTCLIENT_DIR*	-Wl,-G	
Linux s390x			-shared -Wl,-soname, libmqttv3c.so	-m64
Linux x86-64		-fPIC -Os -Wall -I *MQTTCLIENT_DIR*		
Linux x86-32				-m32
Linux ARM (glibc)	arm-linux-gcc			
Linux ARM (uclibc)	arm-unknown-linux -uclibcgnueabi-gcc			
Windows 32-bit	cl	/D "WIN32" /D "_UNICODE" /D "UNICODE" /D "_CRT_SECURE_NO_WARNINGS" / nologo /c /O2 /W3 /Fd /MD /TC	/nologo /machine:x86 /manifest kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib ws2_32.lib / implib:mqttv3c.lib) /pdb:mqttv3c.pdb) / map:mqttv3c.map)	
iOS ARMv7	gcc -arch armv7	-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE -DOPENSSL -Os -Wall -fomit-frame-pointer -isysroot / Applications/Xcode.app/\ Contents/Developer/Platforms/\ iPhoneOS.platform/Developer/SDKs/\ iPhoneOS6.0.sdk	-L/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0.sdk/usr/lib/system	
iOS ARMv7s	gcc -arch armv7s			

Table 2. MQTT build options for different platforms (continued)				
Platform	**Compiler**	**Compiler Options**	**Linker Options**	**Extra Options**
iOS ARMi386	gcc -arch i386	-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE -DOPENSSL -Os -Wall -fomit-frame-pointer -isysroot / Applications/ Xcode.app/\ Contents/Developer/ Platforms/\ iPhoneSimulator.platform/ Developer/SDKs/ iPhoneSimulator6.0.sdk	-L/Applications/Xcode.app/ Contents/Developer/ Platforms/ iPhoneSimulator.platform/ Developer/SDKs/ iPhoneSimulator6.0.sdk/usr /lib/system	

iOS *Building the MQTT client libraries for C on an Apple Mac for use with iOS devices*

Follow these steps to write a makefile to build the MQTT client libraries for C on an Apple Mac, for subsequent use with iOS devices.

1. Install build tools, develop, and run the makefile on Apple Mac with OS X 10.8.2, or later.
2. Install the command-line tools for Xcode, which include the **make** program. Download the command-line tools from Xcode.

Create a makefile that builds MQTT client libraries for C for iPhone or iPad running an ARMv7 or ARMv7s processor, and the iPhone simulator that runs on an i386-64 bit processor. See List of iOS devices.

Tip: "MQTTios.mak makefile listing" on page 34 lists the complete makefile.

1. Copy and paste the listing into a file.
2. Convert the leading character of each line that follows a target to a tab; see step "8" on page 32.
3. Run it with the command listed in step "9" on page 34 of the procedure.

1. Download and install the iOS development tools.
   a. Log on with a user ID that has administrative privileges.
   b. Check your Apple Mac is at version 10.8.2 or later.
   c. Go to the website Xcode to download Xcode from the Mac app store.
   d. Install Xcode, the command-line environment, and the simulator.

   If the Mac app store offers multiple versions of the simulator, choose the version that is compatible with the level of iOS you are targeting for your app.
2. Create the makefile `MQTTios.mak`

   Add a prolog:

```
# Build output is produced in the current directory.
# MQTTCLIENT_DIR must point to the base directory containing the MQTT client source code.
```

```
# Default MQTTCLIENT_DIR is the current directory
# Default TOOL_DIR is /Applications/Xcode.app/Contents/Developer/Platforms
# Default OPENSSL_DIR is sdkroot/openssl, relative to sdkroot/sdk/clients/c/mqttv3c/src
# OPENSSL_DIR must point to the base directory containing the OpenSSL build.
# Example:  make -f MQTTios.mak MQTTCLIENT_DIR=sdkroot/sdk/clients/c/mqttv3c/src all
```

3. Set the location of the MQTT source code.

   Run the makefile in the same directory as the MQTT source files, or set the `MQTTCLIENT_DIR` command-line parameter:

   ```
   make -f makefile MQTTCLIENT_DIR=sdkroot/SDK/clients/c/mqttv3c/src
   ```

   Add the following lines to the makefile:

   ```
   ifndef MQTTCLIENT_DIR
     MQTTCLIENT_DIR = ${CURDIR}
   endif
   VPATH = ${MQTTCLIENT_DIR}
   ```

   The example sets VPATH to the directory where **make** searches for source files that are not explicitly identified; for example all the header files that are required in the build.

4. Optional: Set the location of the OpenSSL libraries.

   This step is required to build the SSL versions of the MQTT client for C libraries.

   Set the default path to the OpenSSL libraries to same directory as you expanded the MQTT SDK. Otherwise, set `OPENSSL_DIR` as a command-line parameter.

   ```
   ifndef OPENSSL_DIR
     OPENSSL_DIR = ${MQTTCLIENT_DIR}/../../../../../openssl-1.0.1c
   endif
   ```

   Tip: *OpenSSL* is the OpenSSL directory that contains all the OpenSSL subdirectories. You might have to move the directory tree from where you expanded it, because it contains unnecessary empty parent directories.

5. Set the development tools directories.

   If you installed Xcode in a different location, set `TOOL_DIR` in the command line.

   ```
   ifndef TOOL_DIR
      TOOL_DIR = /Applications/Xcode.app/Contents/Developer/Platforms endif
   IPHONE_SDK = iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0.sdk
   IPHONESIM_SDK = iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator6.0.sdk
   SDK_ARM = ${TOOL_DIR}/${IPHONE_SDK}
   SDK_i386 = ${TOOL_DIR}/${IPHONESIM_SDK}
   ```

6. Select all the source files that are required to build each MQTT library. Also, set the name and location of the MQTT library to build.

   Add the following line to the makefile to list all the MQTT source files:

   ```
   ALL_SOURCE_FILES = ${wildcard ${MQTTCLIENT_DIR}/*.c}
   ```

   The source files depend on whether you are building a synchronous or asynchronous library, and whether the library includes SSL or not.

   Add one or more of these lines, which depend on the targets to build. The shared libraries are created in the `darwin_x86_64` directory.

   - Synchronous, unsecured:

     ```
     MQTTLIB = mqttv3c
     SOURCE_FILES = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c ${MQTTCLIENT_DIR}/SSLSocket.c,
     ${ALL_SOURCE_FILES}}
     MQTTLIB_DARWIN = darwin_x86_64/lib${MQTTLIB}.a
     ```

   - Synchronous, secured:

```
MQTTLIB_S = mqttv3cs
SOURCE_FILES_S = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c, ${ALL_SOURCE_FILES}}
MQTTLIB_DARWIN_S = darwin_x86_64/lib${MQTTLIB_S}.a
```

- Asynchronous, unsecured:

```
MQTTLIB_A = mqttv3a
SOURCE_FILES_A = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c ${MQTTCLIENT_DIR}/
SSLSocket.c, ${ALL_SOURCE_FILES}}
MQTTLIB_DARWIN_A = darwin_x86_64/lib${MQTTLIB_A}.a
```

- Asynchronous secured:

```
MQTTLIB_AS = mqttv3as
SOURCE_FILES_AS = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c, ${ALL_SOURCE_FILES}}
MQTTLIB_DARWIN_AS = darwin_x86_64/lib${MQTTLIB_AS}.a
```

7. Define the compiler, and compiler options.

   See the options for different platforms that are shown in MQTT build options for different platforms.

   a) Set Gnu project C and C++ (**gcc**) as the compiler.

   Select three cross-compilers to build the library for different devices and the iPhone simulator:

```
CC = iPhoneOS.platform/Developer/usr/bin/gcc
CC_armv7 = ${TOOL_DIR}/${CC} -arch armv7
CC_armv7s = ${TOOL_DIR}/${CC} -arch armv7s
CC_i386 = ${TOOL_DIR}/${CC} -arch i386
```

   b) Add the compiler options.

```
CCFLAGS = -Os -Wall -fomit-frame-pointer
```

   c) Add the include paths.

```
CCFLAGS_SO_ARM =  ${CCFLAGS} -isysroot ${SDK_ARM} -I${OPENSSL_DIR}/include  -L$
{SDK_ARM}/usr/lib/system
CCFLAGS_SO_i386 = ${CCFLAGS} -isysroot ${SDK_i386} -I${OPENSSL_DIR}/include -L$
{SDK_i386}/usr/lib/system
```

8. Define the build targets.

   Tip: Each successive line that defines the implementation of a target must start with a tab character.

   a) Define the **all** target.

   The "all" target builds all the libraries.

```
all: ${MQTTLIB_DARWIN} ${MQTTLIB_DARWIN_A} ${MQTTLIB_DARWIN_AS} ${MQTTLIB_DARWIN_S}
```

   By listing it first, it is the default target.

   a) Build the synchronous, unsecured library, `libmqttv3c.a`.

```
${MQTTLIB_DARWIN}: ${SOURCE_FILES}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES} -DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES} -DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7s *.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES} -DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_i386} -o $@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@
```

   The statement `rm *.o` deletes all the object files that are created for each library. `lipo` concatenates all three libraries into one file.

   b) Build the asynchronous, unsecured library, `libmqttv3a.a`.

```
${MQTTLIB_DARWIN_A}: ${SOURCE_FILES_A}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_A} -DMQTT_ASYNC
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_A} -DMQTT_ASYNC
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7s *.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES_A} -DMQTT_ASYNC
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_i386} -o $@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@
```

The statement `rm *.o` deletes all the object files that are created for each library. `lipo` concatenates all three libraries into one file.

c) Build the synchronous, secured library, `libmqttv3cs.a`.

```
${MQTTLIB_DARWIN_S}: ${SOURCE_FILES_S}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_S} -DOPENSSL -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7 -lssl -lcrypto -o
$@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_S} -DOPENSSL -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7s -lssl -lcrypto -o
$@.armv7s *.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES_S} -DOPENSSL -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/i386 -lssl -lcrypto -o
$@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@
```

The statement `rm *.o` deletes all the object files that are created for each library. `lipo` concatenates all three libraries into one file.

d) Build the asynchronous, secured library, `libmqttv3as.a`.

```
${MQTTLIB_DARWIN_AS}: ${SOURCE_FILES_AS}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_AS} -DMQTT_ASYNC -DOPENSSL
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7 -lssl -lcrypto -o
$@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_AS} -DMQTT_ASYNC -DOPENSSL
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7s -lssl -lcrypto -o
$@.armv7s *.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES_AS} -DMQTT_ASYNC -DOPENSSL
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/i386 -lssl -lcrypto -o
$@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@
```

The statement `rm *.o` deletes all the object files that are created for each library. `lipo` concatenates all three libraries into one file.

e) Define the **clean** target.

The "clean" target removes all the files and directories that are generated by the makefile

```
.PHONY : clean
clean:
    -rm -f *.obj
    -rm -f -r darwin_x86_64
```

9. Run the makefile.

```
make -f  MQTTCLIENT_DIR=sdkroot/sdk/clients/c/mqttv3c/src
```

The following files are created in the *sdkroot/sdk/clients/c/mqttv3c/src*/darwin_x86_64 directory.

```
libmqttv3c.a.armv7
libmqttv3c.a.armv7s
libmqttv3c.a.i386
libmqttv3c.a
libmqttv3a.a.armv7
libmqttv3a.a.armv7s
libmqttv3a.a.i386
libmqttv3a.a
libmqttv3cs.a.armv7
libmqttv3cs.a.armv7s
libmqttv3cs.a.i386
libmqttv3cs.a
libmqttv3as.a.armv7
libmqttv3as.a.armv7s
libmqttv3as.a.i386
libmqttv3as.a
```

MQTTios.mak makefile listing

```
# Build output is produced in the current directory.
# MQTTCLIENT_DIR must point to the base directory containing the MQTT client source code.
# Default MQTTCLIENT_DIR is the current directory
# Default TOOL_DIR is /Applications/Xcode.app/Contents/Developer/Platforms
# Default OPENSSL_DIR is sdkroot/openssl, relative to sdkroot/sdk/clients/c/mqttv3c/src
# OPENSSL_DIR must point to the base directory containing the OpenSSL build.
# Example:  make -f MQTTios.mak MQTTCLIENT_DIR=sdkroot/sdk/clients/c/mqttv3c/src all

ifndef MQTTCLIENT_DIR
  MQTTCLIENT_DIR = ${CURDIR}
endif
VPATH = ${MQTTCLIENT_DIR}
ifndef OPENSSL_DIR
  OPENSSL_DIR = ${MQTTCLIENT_DIR}/../../../../../openssl-1.0.1c
endif
ALL_SOURCE_FILES = ${wildcard ${MQTTCLIENT_DIR}/*.c}
ifndef TOOL_DIR
   TOOL_DIR = /Applications/Xcode.app/Contents/Developer/Platforms endif
IPHONE_SDK = iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0.sdk
IPHONESIM_SDK = iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator6.0.sdk
SDK_ARM = ${TOOL_DIR}/${IPHONE_SDK}
SDK_i386 = ${TOOL_DIR}/${IPHONESIM_SDK}

MQTTLIB = mqttv3c
SOURCE_FILES = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c ${MQTTCLIENT_DIR}/SSLSocket.c, $
{ALL_SOURCE_FILES}}
MQTTLIB_DARWIN = darwin_x86_64/lib${MQTTLIB}.a
MQTTLIB_S = mqttv3cs
SOURCE_FILES_S = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c, ${ALL_SOURCE_FILES}}
MQTTLIB_DARWIN_S = darwin_x86_64/lib${MQTTLIB_S}.a
MQTTLIB_A = mqttv3a
SOURCE_FILES_A = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c ${MQTTCLIENT_DIR}/SSLSocket.c, $
{ALL_SOURCE_FILES}}
MQTTLIB_DARWIN_A = darwin_x86_64/lib${MQTTLIB_A}.a
MQTTLIB_AS = mqttv3as
SOURCE_FILES_AS = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c, ${ALL_SOURCE_FILES}}
MQTTLIB_DARWIN_AS = darwin_x86_64/lib${MQTTLIB_AS}.a

# compiler
CC = iPhoneOS.platform/Developer/usr/bin/gcc
CC_armv7 = ${TOOL_DIR}/${CC} -arch armv7
CC_armv7s = ${TOOL_DIR}/${CC} -arch armv7s
CC_i386 = ${TOOL_DIR}/${CC} -arch i386
CCFLAGS = -Os -Wall -fomit-frame-pointer
CCFLAGS_SO_ARM =  ${CCFLAGS} -isysroot ${SDK_ARM} -I${OPENSSL_DIR}/include  -L$
{SDK_ARM}/usr/lib/system
CCFLAGS_SO_i386 = ${CCFLAGS} -isysroot ${SDK_i386} -I${OPENSSL_DIR}/include -L$
{SDK_i386}/usr/lib/system

# targets
all: ${MQTTLIB_DARWIN} ${MQTTLIB_DARWIN_A} ${MQTTLIB_DARWIN_AS} ${MQTTLIB_DARWIN_S}
```

```
${MQTTLIB_DARWIN}: ${SOURCE_FILES}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES} -DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES} -DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7s *.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES} -DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_i386} -o $@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@

${MQTTLIB_DARWIN_A}: ${SOURCE_FILES_A}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_A} -DMQTT_ASYNC -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_A} -DMQTT_ASYNC -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -o $@.armv7s *.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES_A} -DMQTT_ASYNC -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_i386} -o $@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@

${MQTTLIB_DARWIN_S}: ${SOURCE_FILES_S}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_S} -DOPENSSL -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7 -lssl -lcrypto -o $@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_S} -DOPENSSL -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7s -lssl -lcrypto -o $@.armv7s
*.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES_S} -DOPENSSL -DUSE_NAMED_SEMAPHORES
-DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/i386 -lssl -lcrypto -o $@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@

${MQTTLIB_DARWIN_AS}: ${SOURCE_FILES_AS}
    -mkdir darwin_x86_64
    ${CC_armv7} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_AS} -DMQTT_ASYNC -DOPENSSL
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7 -lssl -lcrypto -o $@.armv7 *.o
    rm *.o
    ${CC_armv7s} ${CCFLAGS_SO_ARM} -c ${SOURCE_FILES_AS} -DMQTT_ASYNC -DOPENSSL
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/armv7s -lssl -lcrypto -o $@.armv7s
*.o
    rm *.o
    ${CC_i386} ${CCFLAGS_SO_i386} -c ${SOURCE_FILES_AS} -DMQTT_ASYNC -DOPENSSL
-DUSE_NAMED_SEMAPHORES -DNOSIGPIPE
    libtool -static -syslibroot ${SDK_ARM} -L${OPENSSL_DIR}/i386 -lssl -lcrypto -o $@.i386 *.o
    rm *.o
    lipo -create $@.armv7 $@.armv7s $@.i386 -output $@

.PHONY : clean
clean:
    -rm -f *.obj
    -rm -f -r darwin_x86_64
```

Windows *Building the MQTT libraries on Windows*

Follow these steps to write a makefile to build the MQTT client libraries for C on Windows.

1. If necessary, install a version of **Make** on your build workstation that is compatible with makefiles written for Gnu make; otherwise download Gnu make and build it. See Gnu Make. The website, Make for Windows, provides an installable version of **Make** for Windows.
2. You also require Linux commands for Windows to use the clean target in the makefile example. You can obtain Linux commands for Windows from websites such as Cygwin.

Create a makefile that builds MQTT client libraries for C for Windows 32 bit.

Tip: "MQTTwin.mak makefile listing" on page 39 lists the complete makefile.

1. Copy and paste the listing into a file.
2. Convert the leading character of each line that follows a target to a tab; see step "7" on page 38.
3. Run it with the command listed in step "9" on page 39 of the procedure.

1. Create the makefile `MQTTwin.mak`

   Add a prolog:

   ```
   # Build output is produced in the current directory.
   # MQTTCLIENT_DIR must point to the base directory containing the MQTT client source code.
   # Default MQTTCLIENT_DIR is the current directory
   # Default OPENSSL_DIR is sdkroot\openSSL, relative to sdkroot\sdk\clients\c\mqttv3c\src
   # OPENSSL_DIR must point to the base directory containing the OpenSSL build.
   # Example:  make -f MQTTwin.mak MQTTCLIENT_DIR=sdkroot/sdk/clients/c/mqttv3c/src
   # Set the build environment, for example:
   #    %comspec% /k ""C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"" x86
   #    set path=%path%;C:\Program Files\GnuWin32\bin;C:\cygwin\bin
   ```

2. Set the location of the MQTT source code.

   Run the makefile in the same directory as the MQTT source files, or set the `MQTTCLIENT_DIR` command-line parameter:

   ```
   make -f makefile MQTTCLIENT_DIR=sdkroot/SDK/clients/c/mqttv3c/src
   ```

   Add the following lines to the makefile:

   ```
   ifndef MQTTCLIENT_DIR
     MQTTCLIENT_DIR = ${CURDIR}
   endif
   VPATH = ${MQTTCLIENT_DIR}
   ```

   The example sets VPATH to the directory where **make** searches for source files that are not explicitly identified; for example all the header files that are required in the build.

3. Optional: Set the location of the OpenSSL libraries.

   This step is required to build the SSL versions of the MQTT client for C libraries.

   Set the default path to the OpenSSL libraries to same directory as you expanded the MQTT SDK. Otherwise, set `OPENSSL_DIR` as a command-line parameter.

   ```
   ifndef OPENSSL_DIR
     OPENSSL_DIR = ${MQTTCLIENT_DIR}/../../../../../openssl-1.0.1c
   endif
   ```

   Tip: *OpenSSL* is the OpenSSL directory that contains all the OpenSSL subdirectories. You might have to move the directory tree from where you expanded it, because it contains unnecessary empty parent directories.

4. Select all the source files that are required to build each MQTT library. Also, set the name and location of the MQTT library to build.

   Add the following line to the makefile to list all the MQTT source files:

   ```
   ALL_SOURCE_FILES = ${wildcard ${MQTTCLIENT_DIR}/*.c}
   ```

   The source files depend on whether you are building a synchronous or asynchronous library, and whether the library includes SSL or not.

   Add one or more of these lines, which depend on the targets to build. The shared libraries and manifests are created in the `windows_ia32` directory.

   - Synchronous, unsecured:

```
MQTTLIB = mqttv3c
MQTTDLL = windows_ia32/${MQTTLIB}.dll
SOURCE_FILES = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c ${MQTTCLIENT_DIR}/SSLSocket.c,
${ALL_SOURCE_FILES}}
MANIFEST = mt -manifest ${MQTTDLL}.manifest -outputresource:${MQTTDLL}\;2
```

- Synchronous, secured:

```
MQTTLIB_S = mqttv3cs
MQTTDLL_S  = windows_ia32/${MQTTLIB_S}.dll
SOURCE_FILES_S = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c, ${ALL_SOURCE_FILES}}
MANIFEST_S = mt -manifest ${MQTTDLL_S}.manifest -outputresource:${MQTTDLL_S}\;2
```

- Asynchronous, unsecured:

```
MQTTLIB_A = mqttv3a
MQTTDLL_A  = windows_ia32/${MQTTLIB_A}.dll
SOURCE_FILES_A = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c ${MQTTCLIENT_DIR}/
SSLSocket.c, ${ALL_SOURCE_FILES}}
MANIFEST_S = mt -manifest ${MQTTDLL_S}.manifest -outputresource:${MQTTDLL_S}\;2
```

- Asynchronous secured:

```
MQTTLIB_AS = mqttv3as
MQTTDLL_AS  = windows_ia32/${MQTTLIB_AS}.dll
SOURCE_FILES_AS = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c, ${ALL_SOURCE_FILES}}
MANIFEST_S = mt -manifest ${MQTTDLL_S}.manifest -outputresource:${MQTTDLL_S}\;2
```

5. Define the compiler, and compiler options.

   See the options for different platforms that are shown in MQTT build options for different platforms.

   a) Set Microsoft Visual C++ as the compiler.

   ```
   CC = cl
   ```

   b) Add the pre-processor options.

   ```
   CPPFLAGS = /D "WIN32" /D "_UNICODE" /D "UNICODE" /D "_CRT_SECURE_NO_WARNINGS"
   ```

   c) Add the compiler options.

   ```
   CFLAGS = /nologo /c /O2 /W3 /Fd /MD /TC
   ```

   d) Add the include paths.

   ```
   INC =  /I ${MQTTCLIENT_DIR} /I ${MQTTCLIENT_DIR}/..
   ```

   e) Optional: Add a pre-processor option, if you are building a secure library.

   ```
   CPPFLAGS_S = ${CPPFLAGS} /D "OPENSSL"
   ```

   f) Optional: Add the OpenSSL header files, if you are building a secure library.

   ```
   INC_S = ${INC} /I ${OPENSSL_DIR}/inc32/
   ```

   Tip: The header files are in `${OPENSSL_DIR}/inc32/openssl`, but the `ssl.h` file is included with "`openssl/ssl.h`".

6. Set the linker, and linker options.

   a) Set Microsoft Visual C++ as the linker.

   ```
   LD = link
   ```

   b) Add the linker options.

   ```
   LINKFLAGS = /nologo /machine:x86 /manifest /dll
   ```

   c) Add the library paths.

```
WINLIBS = kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib\
          advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib\
          odbc32.lib odbccp32.lib ws2_32.lib
```

d) Add the intermediate output files.

```
IMP = /implib:${@:.dll=.lib}
LIBPDB = /pdb:${@:.dll=.pdb}
LIBMAP = /map:${@:.dll=.map}
```

e) Optional: Add the OpenSSL libraries, if you are building a secure library.

```
WINLIBS_S = ${WINLIBS} crypt32.lib ssleay32.lib libeay32.lib
```

f) Optional: Add the OpenSSL library path, if you are building a secure library.

```
LIBPATH_S = /LIBPATH:${OPENSSL_DIR}/lib
```

7. Define the four build targets.

a) Define the **all** target.

Tip: Each successive line that defines the implementation of a target must start with a tab character.

The "all" target builds all the libraries.

```
all: ${MQTTDLL} ${MQTTDLL_A} ${MQTTDLL_AS} ${MQTTDLL_S}
```

b) Build the synchronous, unsecured library, `mqttv3c.dll`.

```
${MQTTDLL}: ${SOURCE_FILES}
    -mkdir windows_ia32
    -rm ${CURDIR}/MQTTAsync.obj
    ${CC} ${CPPFLAGS} ${CFLAGS} ${INC} /Fo ${SOURCE_FILES}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS} *.obj /out:${MQTTDLL}
    ${MANIFEST}
```

The statement `-rm ${CURDIR}/MQTTAsync.obj` deletes any `MQTTAsync.obj` created for an earlier target. `MQTTAsync.obj` and `MQTTClient.obj` are mutually exclusive.

c) Build the asynchronous, unsecured library, `mqttv3a.dll`.

```
${MQTTDLL_A}: ${SOURCE_FILES_A}
    -mkdir windows_ia32
    -rm ${CURDIR}/MQTTClient.obj
    ${CC} ${CPPFLAGS} ${CFLAGS} ${INC} /Fo ${SOURCE_FILES_A}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS} *.obj /out:${MQTTDLL_A}
    ${MANIFEST_A}
```

The statement `-rm ${CURDIR}/MQTTClient.obj` deletes any `MQTTClient.obj` created for an earlier target. `MQTTAsync.obj` and `MQTTClient.obj` are mutually exclusive.

d) Build the synchronous, secured library, `mqttv3cs.dll`.

```
${MQTTDLL_S}: ${SOURCE_FILES_S}
    -mkdir windows_ia32
    -rm ${CURDIR}/MQTTAsync.obj
    ${CC} ${CPPFLAGS_S} ${CFLAGS} ${INC_S} /Fo ${SOURCE_FILES}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS_S} ${LIBPATH_S} *.obj /out:$
{MQTTDLL_S}
    ${MANIFEST_S}
```

The statement `-rm ${CURDIR}/MQTTAsync.obj` deletes any `MQTTAsync.obj` created for an earlier target. `MQTTAsync.obj` and `MQTTClient.obj` are mutually exclusive.

e) Build the asynchronous, secured library, `mqttv3as.dll`.

```
${MQTTDLL_AS}: ${SOURCE_FILES_AS}
    -rm ${CURDIR}/MQTTClient.obj
    ${CC} ${CPPFLAGS_S} ${CFLAGS} ${INC_S} /Fo ${SOURCE_FILES_AS}
```

```
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS_S} ${LIBPATH_S} *.obj /out:$
(MQTTDLL_AS}
    $(MANIFEST_AS}
```

The statement `-rm $(CURDIR}/MQTTClient.obj` deletes any `MQTTClient.obj` created for an earlier target. `MQTTAsync.obj` and `MQTTClient.obj` are mutually exclusive.

f) Define the **clean** target.

The "clean" target removes all the files and directories that are generated by the makefile

```
.PHONY : clean
clean:
    -rm -f *.obj
    -rm -f -r windows_ia32
```

8. Set the Windows path to run the makefile.

Set the parts in italics to match your installation.

a) Set the Microsoft Visual Studio environment.

```
%comspec% /k ""C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"" x86
```

b) Set the `Path` variable to include the make program and the Linux command environment.

```
set path=%path%;C:\Program Files\GnuWin32\bin;C:\cygwin\bin
```

9. Run the makefile.

```
make -f MQTTwin.mak MQTTCLIENT_DIR=sdkroot/SDK/clients/c/mqttv3c/src
```

Tip: The file separator character must be a forward slash, not a backward slash.

The following files are created in the *sdkroot**SDK**clients**c**mqttv3c**src*\windows_ia32 directory.

```
mqttv3a.dll
mqttv3a.dll.manifest
mqttv3a.exp
mqttv3a.lib
mqttv3a.map
mqttv3as.dll
mqttv3as.dll.manifest
mqttv3as.exp
mqttv3as.lib
mqttv3as.map
mqttv3c.dll
mqttv3c.dll.manifest
mqttv3c.exp
mqttv3c.lib
mqttv3c.map
mqttv3cs.dll
mqttv3cs.dll.manifest
mqttv3cs.exp
mqttv3cs.lib
mqttv3cs.map
```

MQTTwin.mak makefile listing

```
# Build output is produced in the current directory.
# MQTTCLIENT_DIR must point to the base directory containing the MQTT client source code.
# Default MQTTCLIENT_DIR is the current directory
# Default OPENSSL_DIR is sdkroot\openSSL, relative to sdkroot\sdk\clients\c\mqttv3c\src
# OPENSSL_DIR must point to the base directory containing the OpenSSL build.
# Example:  make -f MQTTwin.mak MQTTCLIENT_DIR=sdkroot/sdk/clients/c/mqttv3c/src
# Set the build environment, for example:
#    %comspec% /k ""C:\Program Files\Microsoft Visual Studio 9.0\VC\vcvarsall.bat"" x86
#    set path=%path%;C:\Program Files\GnuWin32\bin;C:\cygwin\bin
ifndef MQTTCLIENT_DIR
  MQTTCLIENT_DIR = ${CURDIR}
endif
VPATH = ${MQTTCLIENT_DIR}
```

```
ifndef OPENSSL_DIR
  OPENSSL_DIR = ${MQTTCLIENT_DIR}/../../../../../openssl-1.0.1c
endif

ALL_SOURCE_FILES = ${wildcard ${MQTTCLIENT_DIR}/*.c}

MQTTLIB = mqttv3c
MQTTDLL = windows_ia32/${MQTTLIB}.dll
SOURCE_FILES = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c ${MQTTCLIENT_DIR}/SSLSocket.c, $
{ALL_SOURCE_FILES}}
MANIFEST = mt -manifest ${MQTTDLL}.manifest -outputresource:${MQTTDLL}\;2
MQTTLIB_S = mqttv3cs
MQTTDLL_S  = windows_ia32/${MQTTLIB_S}.dll
SOURCE_FILES_S = ${filter-out ${MQTTCLIENT_DIR}/MQTTAsync.c, ${ALL_SOURCE_FILES}}
MANIFEST_S = mt -manifest ${MQTTDLL_S}.manifest -outputresource:${MQTTDLL_S}\;2
MQTTLIB_A = mqttv3a
MQTTDLL_A  = windows_ia32/${MQTTLIB_A}.dll
SOURCE_FILES_A = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c ${MQTTCLIENT_DIR}/SSLSocket.c, $
{ALL_SOURCE_FILES}}
MANIFEST_S = mt -manifest ${MQTTDLL_S}.manifest -outputresource:${MQTTDLL_S}\;2
MQTTLIB_AS = mqttv3as
MQTTDLL_AS  = windows_ia32/${MQTTLIB_AS}.dll
SOURCE_FILES_AS = ${filter-out ${MQTTCLIENT_DIR}/MQTTClient.c, ${ALL_SOURCE_FILES}}
MANIFEST_S = mt -manifest ${MQTTDLL_S}.manifest -outputresource:${MQTTDLL_S}\;2

# compiler
CC = cl
CPPFLAGS = /D "WIN32" /D "_UNICODE" /D "UNICODE" /D "_CRT_SECURE_NO_WARNINGS"
CFLAGS = /nologo /c /O2 /W3 /Fd /MD /TC
INC =  /I ${MQTTCLIENT_DIR} /I ${MQTTCLIENT_DIR}/..
CPPFLAGS_S = ${CPPFLAGS} /D "OPENSSL"
INC_S = ${INC} /I ${OPENSSL_DIR}/inc32/

# linker
LD = link
LINKFLAGS = /nologo /machine:x86 /manifest /dll
WINLIBS = kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib\
          advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib\
          odbc32.lib odbccp32.lib ws2_32.lib
IMP = /implib:${@:.dll=.lib}
LIBPDB = /pdb:${@:.dll=.pdb}
LIBMAP = /map:${@:.dll=.map}
WINLIBS_S = ${WINLIBS} crypt32.lib ssleay32.lib libeay32.lib
LIBPATH_S = /LIBPATH:${OPENSSL_DIR}/lib

# targets
all: ${MQTTDLL} ${MQTTDLL_A} ${MQTTDLL_AS} ${MQTTDLL_S}

${MQTTDLL}: ${SOURCE_FILES}
    -mkdir windows_ia32
    -rm ${CURDIR}/MQTTAsync.obj
    ${CC} ${CPPFLAGS} ${CFLAGS} ${INC} /Fo ${SOURCE_FILES}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS} *.obj /out:${MQTTDLL}
    ${MANIFEST}

${MQTTDLL_A}: ${SOURCE_FILES_A}
    -mkdir windows_ia32
    -rm ${CURDIR}/MQTTClient.obj
    ${CC} ${CPPFLAGS} ${CFLAGS} ${INC} /Fo ${SOURCE_FILES_A}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS} *.obj /out:${MQTTDLL_A}
    ${MANIFEST_A}

${MQTTDLL_S}: ${SOURCE_FILES_S}
    -mkdir windows_ia32
    -rm ${CURDIR}/MQTTAsync.obj
    ${CC} ${CPPFLAGS_S} ${CFLAGS} ${INC_S} /Fo ${SOURCE_FILES}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS_S} ${LIBPATH_S} *.obj /out:$
{MQTTDLL_S}
    ${MANIFEST_S}

${MQTTDLL_AS}: ${SOURCE_FILES_AS}
    -rm ${CURDIR}/MQTTClient.obj
    ${CC} ${CPPFLAGS_S} ${CFLAGS} ${INC_S} /Fo ${SOURCE_FILES_AS}
    ${LD} ${LINKFLAGS} ${IMP} ${LIBPDB} ${LIBMAP} ${WINLIBS_S} ${LIBPATH_S} *.obj /out:$
(MQTTDLL_AS}
    $(MANIFEST_AS}

.PHONY : clean
clean:
    -rm -f *.obj
    -rm -f -r windows_ia32
```

Building the OpenSSL package

Build the OpenSSL package before you build the secure MQTT client libraries for C, `mqttv3cs` and `mqttv3as`. The build creates the libraries that are required to build a secure version of the MQTT client for C library, and the OpenSSL certificate management tool.

1. iOS The iOS makefile customization is for target devices that run iOS6. The customization might be different for earlier or later versions of iOS.
2. Windows The Windows makefile customization is for 32-bit windows.

Download and install the OpenSSL package and any prerequisite software. Customize the OpenSSL makefiles, and build OpenSSL libraries for your target platform. On Windows and Linux, make also builds the OpenSSL key creation and management tool.

1. Install the OpenSSL package.
   a) Download the OpenSSL package from OpenSSL

      Important: Download and redistribution of the OpenSSL package is subject to stringent import and export regulation, and open source licensing conditions. Take careful note of the restrictions and warnings before you decide whether to download the package.

   b) Expand the compressed file contents into *`sdkroot`*.

      Look under the **News** tab on the OpenSSL site to find the download location of the latest package. The package is compressed as a tar file with the extension `tar.gz`. When expanded, the package creates a top-level folder `openssl`*`version`*; for example *`openssl-1.0.1c`*. The examples refer to the path to the folder as %openssl% on Windows and $openssl on iOS; for example, on Windows, %openssl% is *`sdkroot`*`\openssl-1.0.1c`.

      Tip: Check the directory path that is created by extracting the OpenSSL package. Some packages have duplicate levels of the `openssl`*`version`* folder.
2. Windows
   Optional: On Windows, download and install perl. See perl.org.

   For the example, perl was downloaded from ActivePerl Downloads.
3. iOS
   Optional: On iOS, create three more directories.

   ```
   $ssarm7 ≡ $openssl/arm7
   $sslarm7s ≡ $openssl/arm7s
   $ssli386 ≡ $openssl/i386
   ```

   For iOS you must build the OpenSSL package for three different hardware platforms.
4. Generate the OpenSSL makefile to build the OpenSSL package for your hardware and operating system.
   a) Open a command window in the %openssl% or $openssl directory.
   b) Run the **`Configure`** perl command with the appropriate parameters.
   - Windows

     ```
     perl Configure VC-WIN32 enable-capieng no-asm no-idea no-mdc2 no-rc5 --prefix=%openssl%
     ms\do_ms.bat
     ```

   - iOS

     ```
     ./Configure BSD-generic32 no-idea no-mdc2 no-rc5 --prefix=$openssl
     ```
5. iOS
   On iOS, customize the generated OpenSSL makefile for different Apple devices.

a) Make three copies of the generated makefile, `$openssl/Makefile`

```
$openssl/Makefile_armv7
$openssl/Makefile_armv7s
$openssl/Makefile_i386
```

b) Change the "`CC=gcc`" statement in each makefile.

The `CC=gcc` statement is on line 62, or thereabouts. Change it to the following commands:

`$openssl/Makefile_armv7`

```
CC=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
Developer/usr/bin/gcc -arch armv7
```

`$openssl/Makefile_armv7s`

```
CC=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
Developer/usr/bin/gcc -arch armv7s
```

`$openssl/Makefile_i386`

```
CC=/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/
Developer/usr/bin/gcc -arch i386
```

c) Change the "`CFLAG=...`" statement in each makefile.

The statement is on line 63 or thereabouts (broken into three lines for readability):

```
CFLAG= -DOPENSSL_THREADS -pthread -D_THREAD_SAFE
-D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DTERMIOS
-O3 -fomit-frame-pointer -Wall
```

The location of the SDKs shown is dependent on your Xcode installation choices. The version of the SDKs is dependent on the operating system level you are building the makefile for.

iPhone simulator

The iPhone simulator makefile is `$openssl/Makefile_i386`.

```
CFLAG= -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator6.0.sdk
-DOPENSSL_THREADS -pthread -D_THREAD_SAFE
-D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DTERMIOS
-O3 -fomit-frame-pointer -Wall
```

iOS

The iOS makefiles are `$openssl/Makefile_arm7` and `$openssl/Makefile_arm7s`.

```
CFLAG= -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/
iPhoneOS.platform/Developer/SDKs/iPhoneOS6.0.sdk
-DOPENSSL_THREADS -pthread -D_THREAD_SAFE
-D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DTERMIOS
-O3 -fomit-frame-pointer -Wall
```

6. Run the generated makefile.

- Windows

```
nmake -clean
nmake -f ms\nt.mak
nmake -f ms\nt.mak install
```

- iOS On iOS:

```
make clean
make -f $openssl/Makefile_arm7
mv $openssl/libcrypto.a $ssarm7/libcrypto.a
mv $openssl/libssl.a $ssarm7/libssl.a
make clean
make -f $openssl/Makefile_arm7s
```

```
mv $openssl/libcrypto.a $sslarm7s/libcrypto.a
mv $openssl/libssl.a $sslarm7s/libssl.a
make clean
make -f $openssl/Makefile_i386
mv $openssl/libcrypto.a $ssli386/libcrypto.a
mv $openssl/libssl.a $ssli386/libssl.a
```

The build generates the shared libraries, lib, and header files that are required to build secure versions of the MQTT client library for C.

Getting started with the MQTT client for C on iOS

Learn how to get iOS applications to exchange messages with an MQTT server. For use on iOS devices (that is, iPhone and iPad), you must build the MQTT client library for C from the source code that is provided as part of the MQTT software development kit.

1. Link to the iOS Dev Center and know how to develop applications for iOS.
2. Obtain an Apple Mac with OS X 10.8.2, or later to run the Xcode integrated development environment (IDE).
3. (Optional) Configure a C development environment on Windows or Linux. It is useful to build and run the MQTT client sample C apps on Windows or Linux before you develop an MQTT iOS app. Alternatively, study the sample source code without building the samples.
4. For supported and reference MQTT client platforms for C, see System requirements for IBM Mobile Messaging and M2M Client Pack.
5. If there is a firewall between your client and the server, check that it does not block MQTT traffic.

The procedure guides you through the following steps:

1. Learn about programming for MQTT by studying, building, and running the MQTT client sample apps and MQTT client libraries for C.
2. Install the Xcode development environment for iOS on Apple Mac.
3. Do the task "Building the MQTT client for C libraries" on page 27 to build the MQTT client for C libraries for iOS devices.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See "Getting started with MQTT servers" on page 128.
2. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

   There is no installation program, you just expand the downloaded file.

   a. Download the Mobile Messaging and M2M Client Pack.

   b. Create a folder where you are going to install the SDK.

      You might want to name the folder `MQTT`. The path to this folder is referred to here as *`sdkroot`*.

   c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *`sdkroot`*. The expansion creates a directory tree that starts at *`sdkroot`*\SDK.
3. Optional: Familiarize yourself with the MQTT API by studying the MQTT client sample C app.

   a) Build the synchronous MQTT client sample C app `MQTTV3sample.c` for Windows or Linux. See "Getting started with the MQTT client for C" on page 23.

   b) Connect to an MQTT version 3 server, and publish and subscribe to topics on the server.

   c) Study the source code and MQTT API documentation. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

      Learn how to create and resume MQTT clients, and publish and subscribe to MQTT topics by studying the synchronous sample. The synchronous sample is simpler than the asynchronous sample. If you have not programmed MQTT before, write a synchronous MQTT program to become familiar with the MQTT programming model and API.

d) Build the asynchronous MQTT client library for C on Windows or Linux. See "Building the MQTT client for C libraries" on page 27.

e) Build and run the asynchronous MQTT client sample publish and subscribe C app.

f) Study the MQTT client sample asynchronous C app source code and MQTT reference documentation.

You must use the asynchronous interface to write an MQTT application for a mobile device. Well-written apps that call the asynchronous interface are more responsive and stretch battery life further than apps written to the synchronous interface.

The asynchronous interface has two degrees of asynchronicity:

i) The first degree is to unblock the application while the MQTT client library waits for publications from the server.

ii) The second degree is to unblock the application while the client library connects to the server, creates subscriptions, and posts publications.

4. Download and install the iOS development tools.

a. Log on with a user ID that has administrative privileges.

b. Check your Apple Mac is at version 10.8.2 or later.

c. Go to the website Xcode to download Xcode from the Mac app store.

d. Install Xcode, the command-line environment, and the simulator.

If the Mac app store offers multiple versions of the simulator, choose the version that is compatible with the level of iOS you are targeting for your app.

5. Build the MQTT client libraries for C on iOS. See "Building the MQTT client for C libraries" on page 27.

1. Verify the MQTT client libraries for C that you built:

a. Use the Xcode development environment to compile the asynchronous MQTT client sample C app, and link to the unsecured asynchronous MQTT client library for C.

b. Use the Xcode development environment to run the asynchronous MQTT client sample C app on an iOS device. Connect the sample to the MQTT version 3 server you configured; see "Configuring the MQTT service from the command line" on page 132.

2. Create an MQTT client C app for iOS. The coding examples for the asynchronous C application, might prove helpful. The examples are `MQTTV3ASample.c` and `MQTTV3ASSample.c` in *`sdkroot`*`\SDK\clients\c\samples`. As an exercise, start by implementing the MQTT publish/subscribe sample.

Tip: To get an idea of what the app might look like and do, look at screen captures of the MQTT client sample C app. See "Getting started with the MQTT client for Java on Android" on page 16.

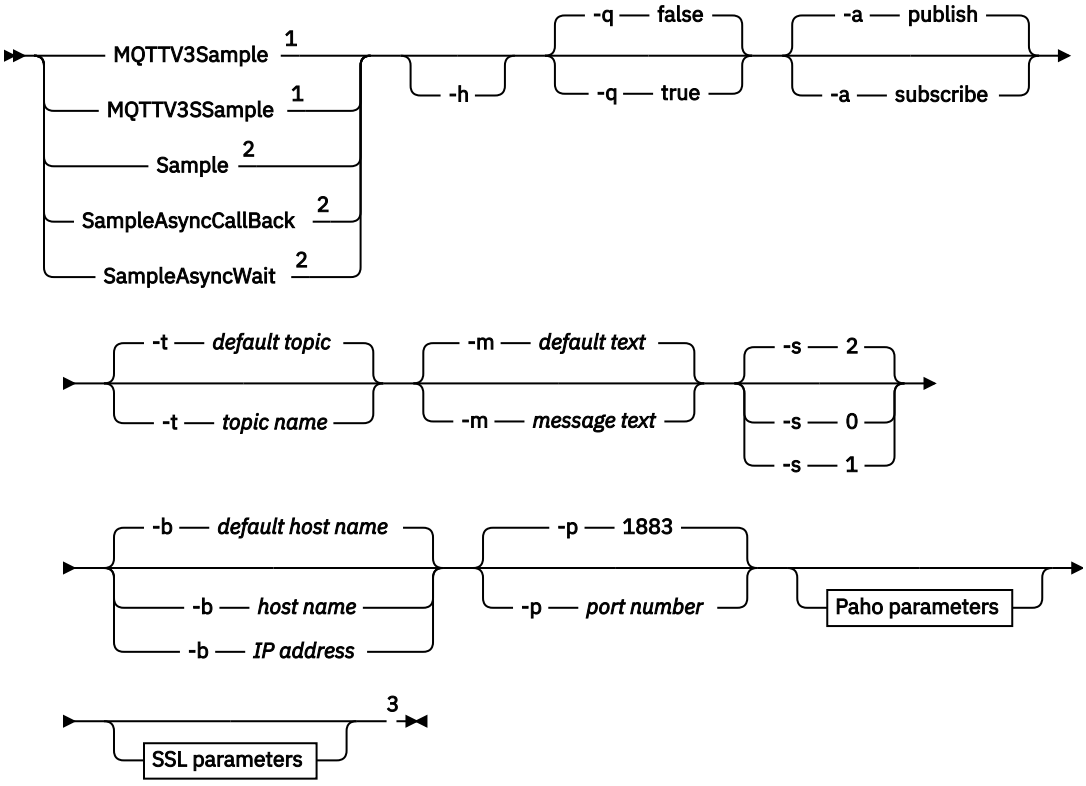
MQTT command line sample programs

The syntax and parameters of the MQTT command line sample programs .

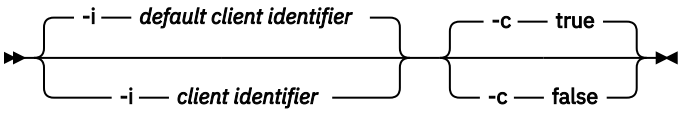
Purpose

Publish and subscribe to a topic.

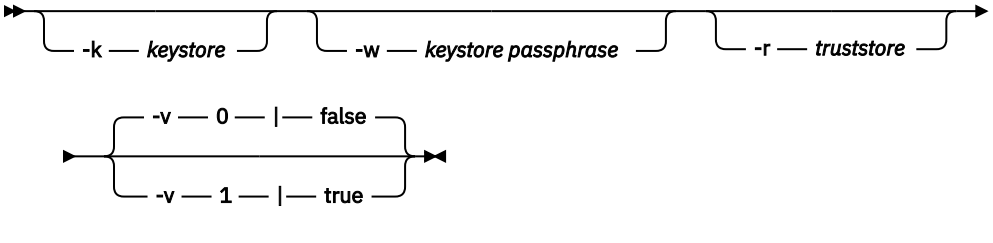
Syntax



Paho parameters



SSL parameters



Notes:

[1] IBM WebSphere MQ sample
[2] Paho sample
[3] *Not* `MQTTV3Sample`.

Parameters

`-h`
Print this help text and quit

`-q`
Set quiet mode, instead of using the default mode of false.

`-a publish|subscribe`
Set the action to `publish` or `subscribe`, instead of assuming the default action of publishing.

-t ***topic name***
Publish or subscribe to *topic name*, instead of publishing or subscribing to the default topic. The default topics are as follows:

Paho samples

Publish
`Sample/Java/v3`

Subscribe
`Sample/#`

IBM WebSphere MQ samples

Publish
`MQTTV3Sample/Java/v3` or `MQTTV3Sample/C/v3`

Subscribe
`MQTTV3Sample/#`

-m ***message text***
Publish *message text* instead of sending the default text. The default text is either "`Message from MQTTv3 C client`", or "`Message from MQTTv3 Java client`"

-s 0|1|2
Set the quality of service (QoS) instead of using the default QoS, 2.

-b ***host name***
Connect to *host name* or IP address instead of connecting to the default host name. The default host name for the Paho samples is `m2m.eclipse.org`. For the IBM WebSphere MQ samples it is `localhost`.

-p ***port number***
Use port *port number* instead of using the default port, 1883.

Paho parameters

-i ***client identifier***
Set the client identifier to *client identifier.* The default client identifier is `SampleJavaV3_"+action`, where `action` is `publish` or `subscribe`.

-c true|false
Set the clean session flag. The default is `true`: subscriptions are not durable.

SSL parameters

-k ***keystore***
Set the path to the keystore containing the private key that identifies the client to *keystore*. For the C samples, the store is a Privacy-Enhanced Mail (PEM) file. For the Java samples it is a Java keystore (JKS).

-w ***keystore passphrase***
Set the passphrase to authorize the client to access the keystore to *keystore passphrase*.

-r ***truststore***
Set the path to the keystore containing the public keys of MQTT servers the client trusts to *truststore*. The keystore is a Privacy-Enhanced Mail (PEM) file. For the C samples, the store is a Privacy-Enhanced Mail (PEM) file. For the Java samples it is a Java keystore (JKS).

-v 0|false|1|true
Set the verify option to `1|true` to require a server certificate. The default is `0|false`: the server certificate is not checked. The SSL channel is always encrypted.

Set the option to `0|1` for C programs and `true|false` for Java programs.

Related tasks
“Getting started with the MQTT client for Java” on page 11

Get up and running with the MQTT client for Java sample applications, using either IBM MessageSight or IBM WebSphere MQ as the MQTT server. The sample applications use a client library from the MQTT software development toolkit (SDK) from IBM. The `SampleAsyncCallBack` sample application is a model for writing MQTT applications for Android and other event-driven operating systems.

"Getting started with the MQTT client for C" on page 23
Get up and running with the sample MQTT client for C on any platform on which you can compile the C source. Verify that you can run the sample MQTT client for C with either IBM MessageSight or IBM WebSphere MQ as the MQTT server.

"Building the MQTT client for C libraries" on page 27
Follow these steps to build the MQTT client for C libraries. The topic includes the compile and link switches for a number of platforms, and examples of building the libraries on iOS and Windows.

MQTT security

Three concepts are fundamental to MQTT security: identity, authentication, and authorization. Identity is about naming the client that is being authorized and given authority. Authentication is about proving the identity of the client, and authorization is about managing the rights that are given the client.

Try the security samples

- "Building and running the secure MQTT client sample Java app" on page 50
- "Connecting the MQTT client sample Java app on Android over SSL" on page 58
- "Authenticating an MQTT client Java app with JAAS" on page 67
- V 7.5.0.1 "Connecting the MQTT messaging client for JavaScript over SSL and WebSockets" on page 72
- "Building and running the secure MQTT client sample C app" on page 79

Identity

Identify an MQTT client by its client identifier, user ID, or public digital certificate. One or other of these attributes defines the client identity. An MQTT server authenticates the certificate sent by the client with the SSL protocol, or the client identity with a password set by the client. The server controls which resources the client can access, based on the client identity.

The MQTT server identifies itself to the client with its IP address and its digital certificate. The MQTT client uses the SSL protocol to authenticate the certificate sent by the server. In some cases, it uses the DNS name of the server to verify the server that sent it the certificate is registered as the certificate holder.

Set the identity of the client in one of the following ways:

Client identifier
The `MqttClient` class (`MQTTClient_create` or `MQTTAsync_create` in C) sets the client identifier. Call the class constructor to set the client identifier as a parameter, or return a randomly generated client identifier. The client identifier must be unique across all clients that connect to the server, and must not be the same as the queue manager name on the server. All clients must have a client identifier, even if it is not used for identity checking. See "Client identifier" on page 120.

User ID
The `MqttClient` class (`MQTTClient_create` or `MQTTAsync_create` in C) sets the client user ID as an attribute of `MqttConnectOptions` (`MqttClient_ConnectOptions` in C). The user ID does not need to be unique to a client.

Client digital certificate
The client digital certificate is stored in the client keystore. The keystore location depends on the client:

- **Java**

Set the location and properties of the client keystore by calling the `setSSLProperties` method of `MqttConnectOptions` and passing the keystore properties. See SSL Modifications to Example.java. The **keytool** tool manages Java keys and keystores.

- **C**

  `MQTTClient_create` or `MQTTAsync_create` sets the keystore properties as attributes of `MQTTClient_SSLOptions ssl_opts`. The **openSSL** tool creates and manages the keys and keystores that are accessed by the MQTT client for C.

- **Android**

  Manage an Android device keystore from the **Settings** > **Security** menu. Load new certificates from the SD card.

Set the identity of the server by storing its private key in the server keystore:

IBM WebSphere MQ
The MQTT server keystore is an attribute of the telemetry channel the client is connected to.

Set the keystore location and attributes with IBM WebSphere MQ Explorer, or with the **DEFINE CHANNEL** command; see DEFINE CHANNEL (MQTT). Multiple channels can share a keystore.

Authentication

An MQTT client can authenticate the MQTT server that it connects to, and the server can authenticate the client that is connecting to it.

A client authenticates a server with the SSL protocol. An MQTT server authenticates a client with the SSL protocol, or with a password, or both.

If the client authenticates the server, but the server does not authenticate the client, the client is often known as an anonymous client. It is common to establish an anonymous client connection over SSL, and then authenticate the client with a password encrypted by the SSL session. It is much more common to authenticate a client with a password than with a client certificate, because of the certificate distribution and management problem. You are likely to find client certificates used in high value devices such as ATMs and chip-and-pin machines, and in custom devices, such as smart electricity meters.

Server authentication by a client
An MQTT client verifies that it is connected to the correct server by authenticating the server certificate with the SSL protocol. This form of verification is familiar to you, when you browse a website over the HTTPS protocol.

The server sends its public certificate, signed by a certificate authority, to the client. The client uses the public key of the certificate authority to verify the signature of the certificate authority on the server certificate. It also checks the certificate is current. Those checks establish that the certificate is valid.

Certificate authority certificates, often termed root certificates, are stored in the client truststore:

- **Java**

  Call the `setSSLProperties` method of `MqttConnectOptions` and pass the truststore properties to set the location and properties of the client truststore. See SSL Modifications to Example.java. Manage certificates and truststores with the **keytool** tool.

- **C**

  `MQTTClient_create` or `MQTTAsync_create` set the truststore properties as attributes of `MQTTClient_SSLOptions ssl_opts`. Manage certificates and truststores with the **openSSL** tool.

- **Android**

  Manage an Android device truststore from the **Settings** > **Security** menu. Load new root certificates from the SD card.

Client authentication by a server

An MQTT server verifies that it is connected to the correct client by authenticating the client certificate with the SSL protocol, or by authenticating the client identity with a password.

It authenticates the client with the password that is sent by the client to the server in an MQTT protocol header. The server might choose to authenticate the client identifier, user ID, or certificate with the password. It depends on the server. Usually, the server authenticates the user ID. Verify passwords over an SSL connection that has been secured by verifying the server, to avoid sending passwords in the clear.

- **IBM WebSphere MQ**

  IBM WebSphere MQ authenticates a client certificate with the SSL protocol. Store root certificates in the IBM WebSphere MQ Telemetry keystore. You can only authenticate a client certificate as part of mutual SSL authentication. That is, you must provide the client with the server public certificate as well as providing the server with the client public certificate.

  IBM WebSphere MQ Telemetry uses the same store for both its own private and public certificate, and other public certificates, such as the root certificates provided by certificate authorities.

  Set the keystore location and attributes with IBM WebSphere MQ Explorer, or with the **DEFINE CHANNEL** command; see DEFINE CHANNEL (MQTT). Multiple channels can share a keystore.

  IBM WebSphere MQ authenticates the client user ID, or the client identifier, by calling the Java authentication and authorization service (JAAS).

  Configure JAAS in an `MQXRConfig` configuration stanza that is stored in the `jaas.config` file. The file is stored in the `qmgrs\`*`QmgrName`*`\mqxr` directory in the IBM WebSphere MQ data path.

  Check the authenticity of the client by writing a `login` method for the `JAASLoginModule`. See “Telemetry channel JAAS configuration” on page 108.

  IBM WebSphere MQ Telemetry passes the `JAASLoginModule.login` method the following parameters:

  – User ID
  – Password
  – Client identifier
  – Network identifier
  – Channel name
  – ValidPrompts

Authorization

Authorization is not part of the MQTT protocol. It is provided by MQTT servers. What is authorized depends on what the server does. MQTT servers are publish/subscribe brokers, and useful MQTT authorization rules control which clients can connect to the server, and which topics a client can publish or subscribe to. If an MQTT client can administer the server, more authorization rules control which clients can administer different aspects of the server.

The number of possible clients is huge, so it is not feasible to authorize each client separately. An MQTT server will have a means to group clients by profiles, or groups.

The identity of a client, from the point of view of access and authorization, is not something that is unique to an MQTT client. Do not equate the identity of a client, with the client identifier. They might be the same, but are commonly different. For example, you probably have a user name that is common across a number of services, and some of these services co-operate in "single sign-on". An enterprise scale MQTT server is likely to call an authorization service that offers common identities and authorities for different applications.

IBM WebSphere MQ

IBM WebSphere MQ has a pluggable authorization service. The default authorization service that is provided on Windows and Linux is the object authority manager (OAM). See Controlling access to objects by using the OAM on UNIX, Linux and Windows systems. It associates operating system user IDs and groups with operations on IBM WebSphere MQ objects, such as topics and queues.

You can configure a telemetry channel to access IBM WebSphere MQ with a fixed user ID. This is how the sample channel is set up. Or you can access IBM WebSphere MQ with the user ID set by the MQTT client. Authorizing MQTT clients to access WebSphere MQ objects describes ways of setting up IBM WebSphere MQ Telemetry to achieve coarse, medium, and fined grained client access control.

Related tasks

"Building and running the secure MQTT client sample C app" on page 79
Based on a Windows example, you can get up and running with the secure sample C app on any operating system for which you can compile the C source. Verify that you can run the sample C app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server.

"Building and running the secure MQTT client sample Java app" on page 50
Based on a Windows example, you can get up and running with the secure sample Java app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible"

"Connecting the MQTT messaging client for JavaScript over SSL and WebSockets" on page 72
Connect your web app securely to IBM WebSphere MQ by using the MQTT messaging client for JavaScript sample HTML pages with SSL and the WebSocket protocol.

"Connecting the MQTT client sample Java app on Android over SSL" on page 58
Get up and running with the sample Android MQTT client connected to IBM WebSphere MQ over SSL.

"Authenticating an MQTT client Java app with JAAS" on page 67
Learn how to authenticate a client with JAAS. Complete the steps in this task to modify the sample program `JAASLoginModule.java` and configure IBM WebSphere MQ to authenticate an MQTT client Java app with JAAS.

Building and running the secure MQTT client sample Java app

Based on a Windows example, you can get up and running with the secure sample Java app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible"

1. You must have access to an MQTT version 3.1 server that supports the MQTT protocol over SSL.
2. If there is a firewall between your client and the server, check that it does not block MQTT traffic.
3. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible". See System requirements for IBM Mobile Messaging and M2M Client Pack.
4. The SSL channels must be started.

As an illustration, this article shows you how to compile and run the secure MQTT client sample Java app on Windows from the command line.

Secure the SSL channel with either certificate authority signed keys, or self-signed keys.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol over SSL. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See "Getting started with MQTT servers" on page 128.
2. Optional: Install a Java development kit (JDK) at Version 7 or later.

   Version 7 is required to run the **keytool** command to certify certificates. If you are not going to certify certificates, you do not require the Version 7 JDK.
3. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

There is no installation program, you just expand the downloaded file.

a. Download the Mobile Messaging and M2M Client Pack.

b. Create a folder where you are going to install the SDK.

You might want to name the folder MQTT. The path to this folder is referred to here as *sdkroot*.

c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *sdkroot*. The expansion creates a directory tree that starts at *sdkroot*\SDK.

4. Create and run the scripts to generate key-pairs and certificates, and configure IBM WebSphere MQ as the MQTT server.

Follow the steps in "Generating keys and certificates" on page 89 to create and run the scripts. The scripts are also listed in "Example scripts to configure SSL certificates for Windows" on page 53.

5. Check the SSL channels are running and are set up as you expect.

On IBM WebSphere MQ, type the following command into a command window:

- 


```
echo 'DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
echo 'DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
```

- 


```
echo DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
echo DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
```

6. Create the scripts to build and run the secure MQTT client sample Java app.

a) Create and run `ssjavaclient.bat` to test an SSL channel that is secured with self-signed certificates.

b) Create and run `cajavaclient.bat` to test an SSL channel that is secured with certificate authority signed certificates.

Scripts to run the MQTT secure Java client

Run the scripts in "Example scripts to configure SSL certificates for Windows" on page 53 before you run these scripts.

MQTT secure Java client with self-signed certificates.

Run this script with the self-signed certificates that you created by running the `sscerts.bat` script.

```
@echo off
setlocal
cd %jsamppath%
set classpath=
set JAVADIR=C:\Program Files\IBM\Java70\bin
cd %
"%JAVADIR%\javac"
-cp ..\org.eclipse.paho.client.mqttv3.jar .\org\eclipse\paho\sample\mqttv3app\Sample.java
ping -n 2 127.0.0.1 > NUL 2>&1
start "Sample Subscriber" "%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -a subscribe -b %host% -p %sslportopt% -k
%cltjkskeystore% -w %cltjkskeystorepass% -r %cltsrvjkstruststore% -v true
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
"%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -b %host% -p %sslportopt% -k %cltjkskeystore% -w
%cltjkskeystorepass% -r %cltsrvjkstruststore% -v true
pause
ping -n 2 127.0.0.1 > NUL 2>&1
start "Sample Subscriber" "%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -a subscribe -b %host% -p %sslportreq% -k
%cltjkskeystore% -w %cltjkskeystorepass% -r %cltsrvjkstruststore% -v true
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
"%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -b %host% -p %sslportreq% -k %cltjkskeystore% -w
%cltjkskeystorepass% -r %cltsrvjkstruststore% -v true
pause
endlocal
```

Figure 14. ssjavaclient.bat

Run the MQTT secure Java client with certificate authority signed certificates.

Run this script with the certificate authority signed certificates that you created by running the cacerts.bat script.

```
@echo off
setlocal
cd %jsamppath%
set classpath=
set JAVADIR=C:\Program Files\IBM\Java70\bin
cd %
"%JAVADIR%\javac"
-cp ..\org.eclipse.paho.client.mqttv3.jar .\org\eclipse\paho\sample\mqttv3app\Sample.java
ping -n 2 127.0.0.1 > NUL 2>&1
start "Sample Subscriber" "%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -a subscribe -b %host% -p %sslportopt% -k
%cltjkskeystore% -w %cltjkskeystorepass% -r %cltcajkstruststore% -v true
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
"%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -b %host% -p %sslportopt% -k %cltjkskeystore% -w
%cltjkskeystorepass% -r %cltcajkstruststore% -v true
pause
ping -n 2 127.0.0.1 > NUL 2>&1
start "Sample Subscriber" "%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -a subscribe -b %host% -p %sslportreq% -k
%cltjkskeystore% -w %cltjkskeystorepass% -r %cltcajkstruststore% -v true
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
"%JAVADIR%\java" -cp  .;..\org.eclipse.paho.client.mqttv3.jar
org.eclipse.paho.sample.mqttv3app.Sample -b %host% -p %sslportreq% -k %cltjkskeystore% -w
%cltjkskeystorepass% -r %cltcajkstruststore% -v true
pause
endlocal
```

Figure 15. cajavaclient.bat

Related concepts

“MQTT security” on page 47

Three concepts are fundamental to MQTT security: identity, authentication, and authorization. Identity is about naming the client that is being authorized and given authority. Authentication is about proving the identity of the client, and authorization is about managing the rights that are given the client.

Related tasks

"Generating keys and certificates" on page 89
Follow this procedure to generate keys and certificates for Java and C clients, including Android and iOS apps, and the IBM WebSphere MQ and IBM MessageSight servers.

"Connecting the MQTT client sample Java app on Android over SSL" on page 58
Get up and running with the sample Android MQTT client connected to IBM WebSphere MQ over SSL.

"Authenticating an MQTT client Java app with JAAS" on page 67
Learn how to authenticate a client with JAAS. Complete the steps in this task to modify the sample program `JAASLoginModule.java` and configure IBM WebSphere MQ to authenticate an MQTT client Java app with JAAS.

Example scripts to configure SSL certificates for Windows

** **

The example command files create the certificates and certificate stores as described in the steps in the task. In addition, the example sets up the MQTT client queue manager to use the server certificate store. The example deletes and re-creates the queue manager by calling the `SampleMQM.bat` script that is provided with IBM WebSphere MQ.

`initcert.bat`

`initcert.bat` sets the names and paths to certificates and other parameters that are required by the **keytool** and **openSSL** commands. The settings are described in comments in the script.

```
@echo off
@rem Set the path where you installed the MQTT SDK
@rem and short cuts to the samples directories.
set SDKRoot=C:\MQTT
set jsamppath=%SDKRoot%\sdk\clients\java\samples
set csamppath=%SDKRoot%\sdk\clients\c\samples
```

```
@rem Set the paths to Version 7 of the JDK
@rem and to the directory where you built the openSSL package.
@rem Set short cuts to the tools.
set javapath=C:\Program Files\IBM\Java70
set keytool="%javapath%\jre\bin\keytool.exe"
set ikeyman="%javapath%\jre\bin\ikeyman.exe"
set openssl=%SDKRoot%\openSSL
set runopenssl="%openssl%\bin\openssl"
```

```
@rem Set the path to where certificates are to be stored,
@rem and set global security parameters.
@rem Omit set password, and the security tools prompt you for passwords.
@rem Validity is the expiry time of the certificates in days.
set certpath=%SDKRoot%\Certificates
set password=password
set validity=5000
set algorithm=RSA
```

```
@rem Set the certificate authority (CA) jks keystore and certificate parameters.
@rem Omit this step, unless you are defining your own certificate authority.
@rem The CA keystore contains the key-pair for your own certificate authority.
@rem You must protect the CA keystore.
@rem The CA certificate is the self-signed certificate authority public certificate.
@rem It is commonly known as the CA root certificate.
set caalias=caalias
set cadname="CN=mqttca.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cakeypass=%password%
@rem ca key store
set cajkskeystore=%certpath%\cakeystore.jks
set cajkskeystorepass=%password%
```

```
@rem ca certificate (root certificate)
set cacert=%certpath%\cacert.cer
```

```
@rem Set the server jks keystore and certificate parameters.
@rem The server keystore contains the key-pair for the server.
@rem You must protect the server keystore.
@rem If you then export the server certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the server keystore for the server key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the server certificate to the CA.
@rem When you now export the server certificate,
@rem the exported certificate includes the certificate chain.
set srvalias=srvalias
set srvdname="CN=mqttserver.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set srvkeypass=%password%
@rem server key stores
set srvjkskeystore=%certpath%\srvkeystore.jks
set srvjkskeystorepass=%password%
@rem server certificates
set srvcertreq=%certpath%\srvcertreq.csr
set srvcertcasigned=%certpath%\srvcertcasigned.cer
set srvcertselfsigned=%certpath%\srvcertselfsigned.cer
```

```
@rem Set the client jks keystore and certificate parameters
@rem Omit this step, unless you are authenticating clients.
@rem The client keystore contains the key-pair for the client.
@rem You must protect the client keystore.
@rem If you then export the client certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the client keystore for the client key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the client certificate to the CA.
@rem When you now export the client certificate,
@rem the exported certificate includes the certificate chain.
set cltalias=cltalias
set cltdname="CN=mqttclient.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cltkeypass=%password%
@rem client key stores
set cltjkskeystore=%certpath%\cltkeystore.jks
set cltjkskeystorepass=%password%
set cltcertreq=%certpath%\cltcertreq.csr
set cltcertcasigned=%certpath%\cltcacertsigned.cer
set cltcertselfsigned=%certpath%\cltcertselfsigned.cer
```

```
@rem Set the paths to the client truststores signed by CA and signed by server key.
@rem You only need to define one of the trust stores.
@rem A trust store holds certificates that you trust,
@rem which are used to authenticate untrusted certificates.
@rem In this example, when the client authenticates the MQTT server it connects to,
@rem it authenticates the certificate it is sent by the server
@rem with the certificates in its trust store.
@rem For example, the MQTT server sends its server certificate,
@rem and the client authenticates it with either the same server certificate
@rem that you have stored in the cltsrvtruststore.jks trust store,
@rem or against the CA certificate, if the server certificate is signed by the CA.
set cltcajkstruststore=%certpath%\cltcatruststore.jks
set cltcajkstruststorepass=%password%
set cltsrvjkstruststore=%certpath%\cltsrvtruststore.jks
set cltsrvjkstruststorepass=%password%
```

```
@rem Set the paths to the client PKCS12 and PEM key and trust stores.
@rem Omit this step, unless you are configuring a C or iOS client.
@rem You only need to define either one of the trust stores for storing CA
@rem or server signed server certificates.
set cltp12keystore=%certpath%\cltkeystore.p12
set cltp12keystorepass=%password%
set cltpemkeystore=%certpath%\cltkeystore.pem
set cltpemkeystorepass=%password%
set cltcap12truststore=%certpath%\cltcatruststore.p12
set cltcap12truststorepass=%password%
set cltcapemtruststore=%certpath%\cltcatruststore.pem
set cltcapemtruststorepass=%password%
set cltsrvp12truststore=%certpath%\cltsrvtruststore.p12
set cltsrvp12truststorepass=%password%
```

```
set cltsrvpemtruststore=%certpath%\cltsrvtruststore.pem
set cltsrvpemtruststorepass=%password%
```

```
@rem set WMQ Variables
set authopt=NEVER
set authreq=REQUIRED
set qm=MQXR_SAMPLE_QM
set host=localhost
set mcauser='Guest'
set portsslopt=8884
set chlopt=SSLOPT
set portsslreq=8885
set chlreq=SSLREQ
V7.5.0.1 set portws=1886
set chlws=PLAINWS
set chlssloptws=SSLOPTWS
set portssloptws=8886
set chlsslreqws=SSLREQWS
set portsslreqws=8887
set mqlog=%certpath%\wmq.log
```

cleancert.bat

The commands in the `cleancert.bat` script delete the MQTT client queue manager to ensure that the server certificate store is not locked, and then delete all the keystores and certificates that are created by the sample security scripts.

```
@rem Delete the MQTT sample queue manager, MQXR_SAMPLE_QM
call "%MQ_FILE_PATH%\bin\setmqenv" -s
endmqm -i %qm%
dltmqm %qm%
```

```
@rem Erase all the certificates and key stores created by the sample scripts.
erase %cajkskeystore%
erase %cacert%
erase %srvjkskeystore%
erase %srvcertreq%
erase %srvcertcasigned%
erase %srvcertselfsigned%
erase %cltjkskeystore%
erase %cltp12keystore%
erase %cltpemkeystore%
erase %cltcertreq%
erase %cltcertcasigned%
erase %cltcertselfsigned%
erase %cltcajkstruststore%
erase %cltcap12truststore%
erase %cltcapemtruststore%
erase %cltsrvjkstruststore%
erase %cltsrvp12truststore%
erase %cltsrvpemtruststore%
erase %mqlog%
@echo Cleared all certificates
dir %certpath%\*.* /b
```

genkeys.bat

The commands in the `genkeys.bat` script create key-pairs for your private certificate authority, the server, and a client.

```
@rem
@echo ____________________________________________________________________________
@echo Generate %caalias%, %srvalias%, and %cltalias% key-pairs in %cajkskeystore%,
%srvjkskeystore%, and %cltjkskeystore%
@rem
@rem -- Generate a client certificate and a private key pair
@rem Omit this step, unless you are authenticating clients.
%keytool% -genkeypair -noprompt -alias %cltalias% -dname %cltdname% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass% -keypass %cltkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem -- Generate a server certificate and private key pair
%keytool% -genkeypair -noprompt -alias %srvalias% -dname %srvdname% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass% -keypass %srvkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem Create CA, client and server key-pairs
@rem  -- Generate a CA certificate and private key pair - The extension asserts this is a
certificate authority certificate, which is required to import into firefox
%keytool% -genkeypair -noprompt -ext bc=ca:true -alias %caalias%  -dname %cadname%
-keystore %cajkskeystore% -storepass %cajkskeystorepass%  -keypass %cakeypass% -keyalg
%algorithm% -validity %validity%
```

sscerts.bat

The commands in the `sscerts.bat` script export the client and server self-signed certificates from their keystores, and import the server certificate into the client truststore, and the client certificate into the server keystore. The server does not have a truststore. The commands create a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ____________________________________________________________________________
@echo Export self-signed certificates:  %srvcertselfsigned% and %cltcertselfsigned%
@rem Export Server public certificate
%keytool% -exportcert -noprompt -rfc -alias %srvalias% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass% -file %srvcertselfsigned%
@rem Export Client public certificate
@rem Omit this step, unless you are authenticating clients.
%keytool% -exportcert -noprompt -rfc -alias %cltalias% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass% -file %cltcertselfsigned%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned server certificate %srvcertselfsigned% to client trust store:
%cltsrvjkstruststore%
@rem Import the server certificate into the client-server trust store (for server self-
signed authentication)
%keytool% -import -noprompt -alias %srvalias% -file %srvcertselfsigned% -keystore
%cltsrvjkstruststore% -storepass %cltsrvjkstruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned client certificate %cltcertselfsigned% to server trust store:
%srvjkskeystore%
@rem Import the client certificate into the server trust store (for client self-signed
authentication)
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %cltalias% -file %cltcertselfsigned% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-server trust store from the jks client-server trust store:
%cltsrvpemtruststore%
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltsrvjkstruststore% -destkeystore
%cltsrvp12truststore% -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltsrvjkstruststorepass% -deststorepass %cltsrvp12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltsrvp12truststore% -out %cltsrvpemtruststore% -passin
pass:%cltsrvp12truststorepass% -passout pass:%cltsrvpemtruststorepass%@rem
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

cacerts.bat

The script imports the certificate authority root certificate into the private keystores. The CA root certificate is needed to create the keychain between the root certificate and the signed certificate. The `cacerts.bat` script exports the client and server certificate requests from their keystores. The script signs the certificate requests with the key of the private certificate authority in the `cajkskeystore.jks` keystore, then imports the signed certificates back into the same keystores

from which the requests came. The import creates the certificate chain with the CA root certificate. The script creates a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Export self-signed certificates: %cacert%
@rem
@rem Export CA public certificate
%keytool% -exportcert -noprompt -rfc -alias %caalias%  -keystore %cajkskeystore% -storepass
%cajkskeystorepass%  -file %cacert%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Add CA to server key and client key and trust stores: %srvjkskeystore%,
%cltjkskeystore%, %cltcajkstruststore%,
@rem The CA certificate is necessary to create key chains in the client and server key
stores,
@rem and to certify key chains in the server key store and the client trust store
@rem
@rem Import the CA root certificate into the server key store
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass%
@rem Import the CA root certificate into the client key store
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass%
@rem Import the CA root certificate into the client ca-trust store (for ca chained
authentication)
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltcajkstruststore%
-storepass %cltcajkstruststorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Create certificate signing requests: %srvcertreq% and %cltcertreq%
@rem
@rem Create a certificate signing request (CSR) for the server key
%keytool% -certreq -alias %srvalias% -file %srvcertreq% -keypass %srvkeypass% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Create a certificate signing request (CSR) for the client key
%keytool% -certreq -alias %cltalias% -file %cltcertreq% -keypass %cltkeypass% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Sign certificate requests: %srvcertcasigned% and %cltcertcasigned%
@rem The requests are signed with the ca key in the cajkskeystore.jks keystore
@rem
@rem Sign server certificate request
%keytool% -gencert -infile %srvcertreq% -outfile %srvcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
@rem Sign client certificate request
@rem Omit this step, unless you are authenticating clients.
%keytool% -gencert -infile %cltcertreq% -outfile %cltcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Import the signed certificates back into the key stores to create the key chain:
%srvjkskeystore% and %cltjkskeystore%
@rem
@rem Import the signed server certificate
%keytool% -import -noprompt -alias %srvalias% -file %srvcertcasigned% -keypass %srvkeypass%
-keystore %srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Import the signed client certificate and key chain back into the client keystore
%keytool% -import -noprompt -alias %cltalias% -file %cltcertcasigned% -keypass %cltkeypass%
-keystore %cltjkskeystore% -storepass %cltjkskeystorepass%
@rem
@rem The CA certificate is needed in the server key store, and the client trust store
@rem to verify the key chain sent from the client or server
@echo Delete the CA certificate from %cltjkskeystore%: it causes a problem in converting
keystore to pem
@rem Omit this step, unless you are authenticating clients.
%keytool% -delete -alias %caalias% -keystore %cltjkskeystore% -storepass
%cltjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-ca trust store from the jks client-ca trust store:
%cltcapemtruststore%
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltcajkstruststore%  -destkeystore
%cltcap12truststore%  -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltcajkstruststorepass%  -deststorepass %cltcap12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltcap12truststore%  -out %cltcapemtruststore%  -passin
pass:%cltcap12truststorepass%  -passout pass:%cltpemtruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

mqcerts.bat

The script lists the keystores and certificates in the certificate directory. It then creates the MQTT sample queue manager and configures the secure telemetry channels.

```
@echo ____________________________________________________________________________
@echo List keystores and certificates
dir %certpath%\*.* /b
```

```
@rem
@echo Create queue manager and define mqtt channels and certificate stores
call "%MQ_FILE_PATH%\mqxr\Samples\SampleMQM" >> %mqlog%
echo DEFINE CHANNEL(%chlreq%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreq%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlopt%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslopt%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
V 7.5.0.1
echo DEFINE CHANNEL(%chlsslreqws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreqws%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlssloptws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portssloptws%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlws%)       CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portws%)
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
@echo MQ logs saved in %mqlog%echo
```

Connecting the MQTT client sample Java app on Android over SSL

Get up and running with the sample Android MQTT client connected to IBM WebSphere MQ over SSL.

This article assumes that you are running at least Android API level 14 (ICS 4.0). Earlier levels had a key store, but only system apps could access it.

1. You must have access to an MQTT version 3.1 server that supports the MQTT protocol over SSL.
2. If there is a firewall between your client and the server, check that it does not block MQTT traffic.
3. If you are testing the connection on an early Android device, you might need an SD card to transfer the certificate to the device.
4. If you are testing the connection on a virtual Android device, configure an SD card for the virtual device.
5. The SSL channels must be started.

Complete this task to run the MQTT client sample Java app for Android over SSL. A successful SSL connection establishes a secure encrypted channel between your Android device and the MQTT server. The identity of the server is authenticated.

With Android, you can authenticate the server with SSL. You can also authenticate the device, although the sample app does not support this. To authenticate the device, either use the KeyChain API, or use JAAS to authenticate the client identifier, the client IP address, or the username and password provided by the MQTT Android app.

Any X.509 certificates you install into the Android trust store must be signed by a certificate authority. In the example, you create a certificate authority, which signs the certificate that you install in your Android device. Numerous root certificates are preinstalled into Android devices.

You must create a lock on your Android device before you install a trusted certificate. The lock prevents someone installing certificates on the device without your knowledge.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol over SSL. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See "Getting started with MQTT servers" on page 128.
2. Run the MQTT client sample app for Android "`MQTTExerciser`" on an unsecured MQTT channel. See "Getting started with the MQTT client for Java on Android" on page 16.

   You use the app again to test the secure channel.

   If you started an Android virtual device, leave it running.
3. Optional: Install a Java development kit (JDK) at Version 7 or later.

   Version 7 is required to run the **`keytool`** command to certify certificates. If you are not going to certify certificates, you do not require the Version 7 JDK.
4. Create and run the scripts to generate key-pairs and certificates, and configure IBM WebSphere MQ as the MQTT server.

   Follow the steps in "Generating keys and certificates" on page 89 to create and run the scripts. The scripts are also listed in "Example scripts to configure SSL certificates for Windows" on page 62.

   You need the certificate authority public certificate and the server key store. You do not need client certificates, or any certificates in `.pem` or `.p12` format.
5. Check the SSL channels are running and are set up as you expect.

   On IBM WebSphere MQ, type the following command into a command window:

   - Linux

     ```
     echo 'DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
     echo 'DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
     ```

   - Windows

     ```
     echo DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
     echo DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
     ```
6. Install the certificate authority certificate in the Android trust store.

   The certificate authority file in the example is `cacert.cer`.

   a) Rename the certificate to `cacert.crt`

   b) Copy the certificate to the root internal storage, or onto the SD card.

      For a running virtual device, open Eclipse or run the Android Debug Bridge (ADB) to copy the certificate to the virtual device:

      Eclipse

      i) Run Eclipse, and open the DDMS perspective.

ii) In the main view, open the **File Explorer** window.

iii) Open the `mnt/sdcard` directory.

iv) Drag the `cacert.crt` file to the `mnt/sdcard` directory.

ADB

i) Open a command window, and set its current directory to `android-sdk\platform-tools` in the android installation directory; for example, `C:\Program Files\Android\android-sdk\platform-tools`.

ii) Copy the certificate to the `mnt/sdcard` directory:

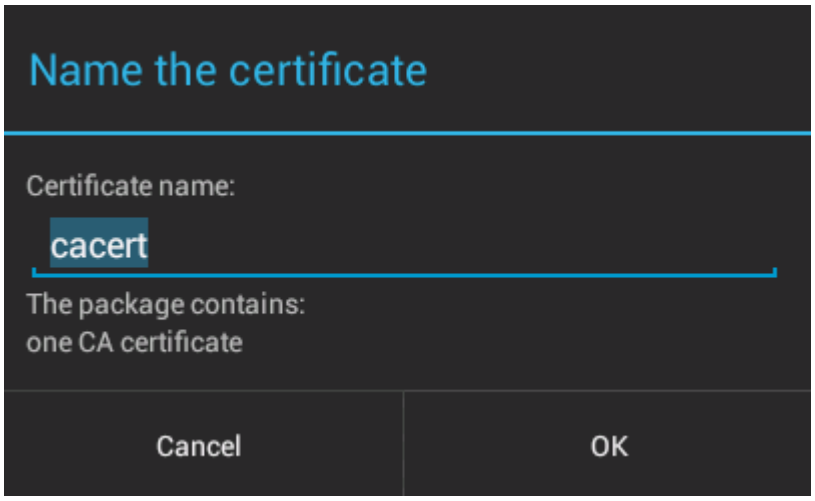
```
adb push %cacert% /mnt/sdcard/cacert.crt
```

7. Install the certificate in the certificate trust store on theAndroid device.

   The certificate must have a Basic Constraints clause with the value `Subject Type=CA`.

   a) Unlock the device and click the **widgets** button.

   b) Click **Settings** > **Security** > **Credential Storage** > **Install from SD card**.

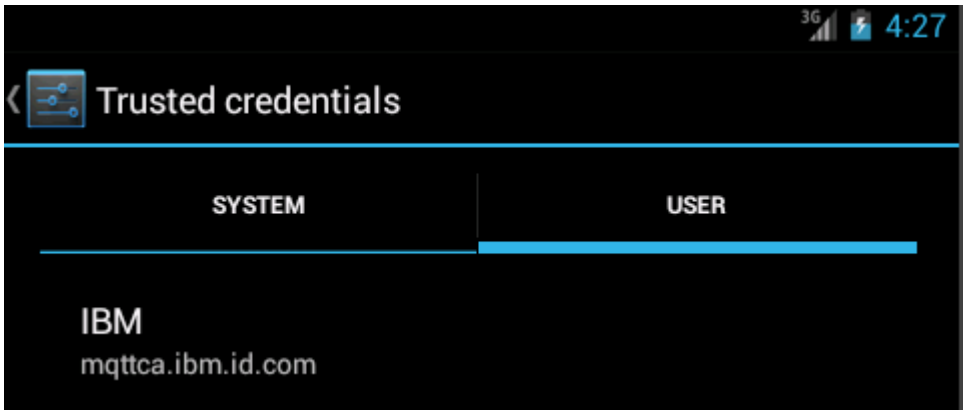


   c) Confirm the certificate file name is correct and click **OK**.

   Note: If you have not defined a lock for the device, you are now prompted by Android to set a lock up.

8. Confirm the certificate is installed on the device.

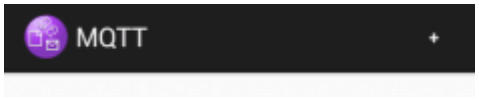
   a) Click **Trusted Credentials** > **User** and wait for several minutes or so for your certificate to be displayed in the list of user certificates.



9. Rerun the `MQTTExerciser` app, and connect to an MQTT channel that you have configured for anonymous SSL clients.

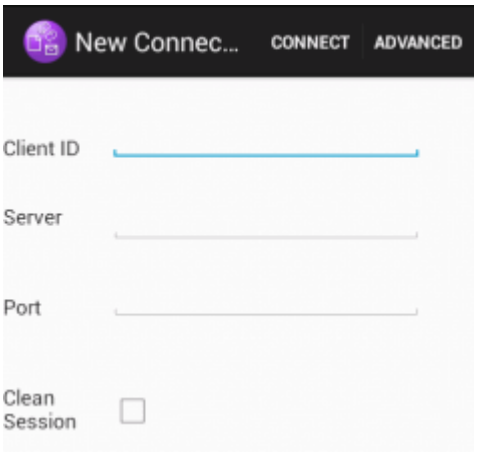
   a) Open the MQTT client sample Java app for Android.

This window is open in your Android device:



b) Connect to an MQTT server.

i) Click the **+** sign to open a new MQTT connection.



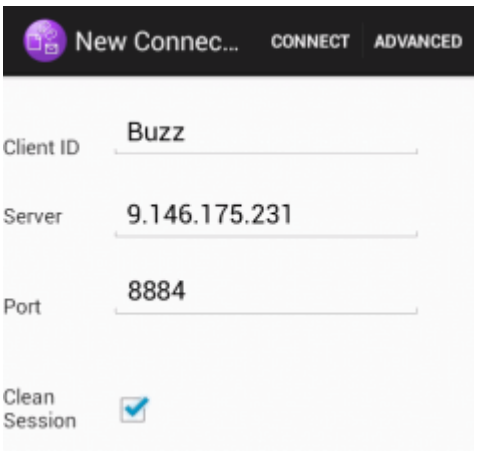
ii) Enter any unique identifier into the **client ID** field. Be patient, the keystrokes can be slow.

iii) Enter into the **Server** field the IP address of your MQTT server.

This is the server that you chose in the first main step. The IP address must not be `127.0.0.1`

iv) Enter the port number of the MQTT connection.

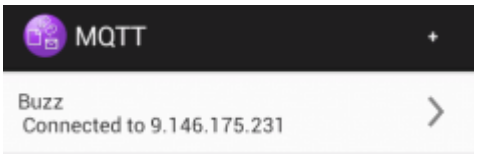
Set the port number to 8884, which is set by the variable `%sslportopt%` in the example scripts. This port number is the port number of the MQTT channel that you have configured for anonymous SSL clients by running the sample scripts in step "4" on page 59.



v) Click the **Advanced** tab, and select the **SSL** option. Click **Save**.

vi) Click **Connect**.

If the connection is successful, you see a "Connecting" message followed by this window:



The `MQTTExerciser` app takes a little longer to connect and exchange messages, but otherwise behaves no differently to being connected on an insecure connection.

Related concepts

“MQTT security” on page 47
Three concepts are fundamental to MQTT security: identity, authentication, and authorization. Identity is about naming the client that is being authorized and given authority. Authentication is about proving the identity of the client, and authorization is about managing the rights that are given the client.

Related tasks

“Generating keys and certificates” on page 89
Follow this procedure to generate keys and certificates for Java and C clients, including Android and iOS apps, and the IBM WebSphere MQ and IBM MessageSight servers.

“Building and running the secure MQTT client sample Java app” on page 50
Based on a Windows example, you can get up and running with the secure sample Java app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible"

“Authenticating an MQTT client Java app with JAAS” on page 67
Learn how to authenticate a client with JAAS. Complete the steps in this task to modify the sample program `JAASLoginModule.java` and configure IBM WebSphere MQ to authenticate an MQTT client Java app with JAAS.

Example scripts to configure SSL certificates for Windows

** **

The example command files create the certificates and certificate stores as described in the steps in the task. In addition, the example sets up the MQTT client queue manager to use the server certificate store. The example deletes and re-creates the queue manager by calling the `SampleMQM.bat` script that is provided with IBM WebSphere MQ.

`initcert.bat`

`initcert.bat` sets the names and paths to certificates and other parameters that are required by the **keytool** and **openSSL** commands. The settings are described in comments in the script.

```
@echo off
@rem Set the path where you installed the MQTT SDK
@rem and short cuts to the samples directories.
set SDKRoot=C:\MQTT
set jsamppath=%SDKRoot%\sdk\clients\java\samples
set csamppath=%SDKRoot%\sdk\clients\c\samples
```

```
@rem Set the paths to Version 7 of the JDK
@rem and to the directory where you built the openSSL package.
@rem Set short cuts to the tools.
set javapath=C:\Program Files\IBM\Java70
set keytool="%javapath%\jre\bin\keytool.exe"
set ikeyman="%javapath%\jre\bin\ikeyman.exe"
set openssl=%SDKRoot%\openSSL
set runopenssl="%openssl%\bin\openssl"
```

```
@rem Set the path to where certificates are to be stored,
@rem and set global security parameters.
@rem Omit set password, and the security tools prompt you for passwords.
@rem Validity is the expiry time of the certificates in days.
set certpath=%SDKRoot%\Certificates
set password=password
set validity=5000
set algorithm=RSA
```

```
@rem Set the certificate authority (CA) jks keystore and certificate parameters.
@rem Omit this step, unless you are defining your own certificate authority.
@rem The CA keystore contains the key-pair for your own certificate authority.
@rem You must protect the CA keystore.
@rem The CA certificate is the self-signed certificate authority public certificate.
@rem It is commonly known as the CA root certificate.
set caalias=caalias
set cadname="CN=mqttca.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
```

```
set cakeypass=%password%
@rem ca key store
set cajkskeystore=%certpath%\cakeystore.jks
set cajkskeystorepass=%password%
@rem ca certificate (root certificate)
set cacert=%certpath%\cacert.cer
```

```
@rem Set the server jks keystore and certificate parameters.
@rem The server keystore contains the key-pair for the server.
@rem You must protect the server keystore.
@rem If you then export the server certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the server keystore for the server key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the server certificate to the CA.
@rem When you now export the server certificate,
@rem the exported certificate includes the certificate chain.
set srvalias=srvalias
set srvdname="CN=mqttserver.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set srvkeypass=%password%
@rem server key stores
set srvjkskeystore=%certpath%\srvkeystore.jks
set srvjkskeystorepass=%password%
@rem server certificates
set srvcertreq=%certpath%\srvcertreq.csr
set srvcertcasigned=%certpath%\srvcertcasigned.cer
set srvcertselfsigned=%certpath%\srvcertselfsigned.cer
```

```
@rem Set the client jks keystore and certificate parameters
@rem Omit this step, unless you are authenticating clients.
@rem The client keystore contains the key-pair for the client.
@rem You must protect the client keystore.
@rem If you then export the client certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the client keystore for the client key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the client certificate to the CA.
@rem When you now export the client certificate,
@rem the exported certificate includes the certificate chain.
set cltalias=cltalias
set cltdname="CN=mqttclient.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cltkeypass=%password%
@rem client key stores
set cltjkskeystore=%certpath%\cltkeystore.jks
set cltjkskeystorepass=%password%
set cltcertreq=%certpath%\cltcertreq.csr
set cltcertcasigned=%certpath%\cltcacertsigned.cer
set cltcertselfsigned=%certpath%\cltcertselfsigned.cer
```

```
@rem Set the paths to the client truststores signed by CA and signed by server key.
@rem You only need to define one of the trust stores.
@rem A trust store holds certificates that you trust,
@rem which are used to authenticate untrusted certificates.
@rem In this example, when the client authenticates the MQTT server it connects to,
@rem it authenticates the certificate it is sent by the server
@rem with the certificates in its trust store.
@rem For example, the MQTT server sends its server certificate,
@rem and the client authenticates it with either the same server certificate
@rem that you have stored in the cltsrvtruststore.jks trust store,
@rem or against the CA certificate, if the server certificate is signed by the CA.
set cltcajkstruststore=%certpath%\cltcatruststore.jks
set cltcajkstruststorepass=%password%
set cltsrvjkstruststore=%certpath%\cltsrvtruststore.jks
set cltsrvjkstruststorepass=%password%
```

```
@rem Set the paths to the client PKCS12 and PEM key and trust stores.
@rem Omit this step, unless you are configuring a C or iOS client.
@rem You only need to define either one of the trust stores for storing CA
@rem or server signed server certificates.
set cltp12keystore=%certpath%\cltkeystore.p12
set cltp12keystorepass=%password%
set cltpemkeystore=%certpath%\cltkeystore.pem
set cltpemkeystorepass=%password%
set cltcap12truststore=%certpath%\cltcatruststore.p12
set cltcap12truststorepass=%password%
set cltcapemtruststore=%certpath%\cltcatruststore.pem
```

```
set cltcapemtruststorepass=%password%
set cltsrvp12truststore=%certpath%\cltsrvtruststore.p12
set cltsrvp12truststorepass=%password%
set cltsrvpemtruststore=%certpath%\cltsrvtruststore.pem
set cltsrvpemtruststorepass=%password%
```

```
@rem set WMQ Variables
set authopt=NEVER
set authreq=REQUIRED
set qm=MQXR_SAMPLE_QM
set host=localhost
set mcauser='Guest'
set portsslopt=8884
set chlopt=SSLOPT
set portsslreq=8885
set chlreq=SSLREQ
V7.5.0.1 set portws=1886
set chlws=PLAINWS
set chlssloptws=SSLOPTWS
set portssloptws=8886
set chlsslreqws=SSLREQWS
set portsslreqws=8887
set mqlog=%certpath%\wmq.log
```

cleancert.bat

The commands in the `cleancert.bat` script delete the MQTT client queue manager to ensure that the server certificate store is not locked, and then delete all the keystores and certificates that are created by the sample security scripts.

```
@rem Delete the MQTT sample queue manager, MQXR_SAMPLE_QM
call "%MQ_FILE_PATH%\bin\setmqenv" -s
endmqm -i %qm%
dltmqm %qm%
```

```
@rem Erase all the certificates and key stores created by the sample scripts.
erase %cajkskeystore%
erase %cacert%
erase %srvjkskeystore%
erase %srvcertreq%
erase %srvcertcasigned%
erase %srvcertselfsigned%
erase %cltjkskeystore%
erase %cltp12keystore%
erase %cltpemkeystore%
erase %cltcertreq%
erase %cltcertcasigned%
erase %cltcertselfsigned%
erase %cltcajkstruststore%
erase %cltcap12truststore%
erase %cltcapemtruststore%
erase %cltsrvjkstruststore%
erase %cltsrvp12truststore%
erase %cltsrvpemtruststore%
erase %mqlog%
@echo Cleared all certificates
dir %certpath%\*.* /b
```

genkeys.bat

The commands in the `genkeys.bat` script create key-pairs for your private certificate authority, the server, and a client.

```
@rem
@echo ____________________________________________________________________________
@echo Generate %caalias%, %srvalias%, and %cltalias% key-pairs in %cajkskeystore%,
%srvjkskeystore%, and %cltjkskeystore%
@rem
@rem -- Generate a client certificate and a private key pair
@rem Omit this step, unless you are authenticating clients.
%keytool% -genkeypair -noprompt -alias %cltalias% -dname %cltdname% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass% -keypass %cltkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem -- Generate a server certificate and private key pair
%keytool% -genkeypair -noprompt -alias %srvalias% -dname %srvdname% -keystore
```

```
%srvjkskeystore% -storepass %srvjkskeystorepass% -keypass %srvkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem Create CA, client and server key-pairs
@rem  -- Generate a CA certificate and private key pair - The extension asserts this is a
certificate authority certificate, which is required to import into firefox
%keytool% -genkeypair -noprompt -ext bc=ca:true -alias %caalias%  -dname %cadname%
-keystore %cajkskeystore% -storepass %cajkskeystorepass%  -keypass %cakeypass% -keyalg
%algorithm% -validity %validity%
```

sscerts.bat

The commands in the `sscerts.bat` script export the client and server self-signed certificates from their keystores, and import the server certificate into the client truststore, and the client certificate into the server keystore. The server does not have a truststore. The commands create a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ____________________________________________________________________________
@echo Export self-signed certificates:  %srvcertselfsigned% and %cltcertselfsigned%
@rem Export Server public certificate
%keytool% -exportcert -noprompt -rfc -alias %srvalias% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass% -file %srvcertselfsigned%
@rem Export Client public certificate
@rem Omit this step, unless you are authenticating clients.
%keytool% -exportcert -noprompt -rfc -alias %cltalias% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass% -file %cltcertselfsigned%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned server certificate %srvcertselfsigned% to client trust store:
%cltsrvjkstruststore%
@rem Import the server certificate into the client-server trust store (for server self-
signed authentication)
%keytool% -import -noprompt -alias %srvalias% -file %srvcertselfsigned% -keystore
%cltsrvjkstruststore% -storepass %cltsrvjkstruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned client certificate %cltcertselfsigned% to server trust store:
%srvjkskeystore%
@rem Import the client certificate into the server trust store (for client self-signed
authentication)
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %cltalias% -file %cltcertselfsigned% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-server trust store from the jks client-server trust store:
%cltsrvpemtruststore%
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltsrvjkstruststore% -destkeystore
%cltsrvp12truststore% -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltsrvjkstruststorepass% -deststorepass %cltsrvp12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltsrvp12truststore% -out %cltsrvpemtruststore% -passin
pass:%cltsrvp12truststorepass% -passout pass:%cltsrvpemtruststorepass%@rem
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

cacerts.bat

The script imports the certificate authority root certificate into the private keystores. The CA root certificate is needed to create the keychain between the root certificate and the signed certificate. The `cacerts.bat` script exports the client and server certificate requests from their keystores.

The script signs the certificate requests with the key of the private certificate authority in the `cajkskeystore.jks` keystore, then imports the signed certificates back into the same keystores from which the requests came. The import creates the certificate chain with the CA root certificate. The script creates a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Export self-signed certificates: %cacert%
@rem
@rem Export CA public certificate
%keytool% -exportcert -noprompt -rfc -alias %caalias%  -keystore %cajkskeystore% -storepass
%cajkskeystorepass%  -file %cacert%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Add CA to server key and client key and trust stores: %srvjkskeystore%,
%cltjkskeystore%, %cltcajkstruststore%,
@rem The CA certificate is necessary to create key chains in the client and server key
stores,
@rem and to certify key chains in the server key store and the client trust store
@rem
@rem Import the CA root certificate into the server key store
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass%
@rem Import the CA root certificate into the client key store
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass%
@rem Import the CA root certificate into the client ca-trust store (for ca chained
authentication)
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltcajkstruststore%
-storepass %cltcajkstruststorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Create certificate signing requests: %srvcertreq% and %cltcertreq%
@rem
@rem Create a certificate signing request (CSR) for the server key
%keytool% -certreq -alias %srvalias% -file %srvcertreq% -keypass %srvkeypass% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Create a certificate signing request (CSR) for the client key
%keytool% -certreq -alias %cltalias% -file %cltcertreq% -keypass %cltkeypass% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Sign certificate requests: %srvcertcasigned% and %cltcertcasigned%
@rem The requests are signed with the ca key in the cajkskeystore.jks keystore
@rem
@rem Sign server certificate request
%keytool% -gencert -infile %srvcertreq% -outfile %srvcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
@rem Sign client certificate request
@rem Omit this step, unless you are authenticating clients.
%keytool% -gencert -infile %cltcertreq% -outfile %cltcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Import the signed certificates back into the key stores to create the key chain:
%srvjkskeystore% and %cltjkskeystore%
@rem
@rem Import the signed server certificate
%keytool% -import -noprompt -alias %srvalias% -file %srvcertcasigned% -keypass %srvkeypass%
-keystore %srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Import the signed client certificate and key chain back into the client keystore
%keytool% -import -noprompt -alias %cltalias% -file %cltcertcasigned% -keypass %cltkeypass%
-keystore %cltjkskeystore% -storepass %cltjkskeystorepass%
@rem
@rem The CA certificate is needed in the server key store, and the client trust store
@rem to verify the key chain sent from the client or server
@echo Delete the CA certificate from %cltjkskeystore%: it causes a problem in converting
keystore to pem
@rem Omit this step, unless you are authenticating clients.
```

```
%keytool% -delete -alias %caalias% -keystore %cltjkskeystore% -storepass
%cltjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-ca trust store from the jks client-ca trust store:
%cltcapemtruststore%
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltcajkstruststore%  -destkeystore
%cltcap12truststore%  -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltcajkstruststorepass%  -deststorepass %cltcap12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltcap12truststore%  -out %cltcapemtruststore%  -passin
pass:%cltcap12truststorepass%  -passout pass:%cltpemtruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

mqcerts.bat

The script lists the keystores and certificates in the certificate directory. It then creates the MQTT sample queue manager and configures the secure telemetry channels.

```
@echo ____________________________________________________________________________
@echo List keystores and certificates
dir %certpath%\*.* /b
```

```
@rem
@echo Create queue manager and define mqtt channels and certificate stores
call "%MQ_FILE_PATH%\mqxr\Samples\SampleMQM" >> %mqlog%
echo DEFINE CHANNEL(%chlreq%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreq%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlopt%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslopt%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
```

V 7.5.0.1

```
echo DEFINE CHANNEL(%chlsslreqws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreqws%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlssloptws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portssloptws%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlws%)       CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portws%)
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
@echo MQ logs saved in %mqlog%echo
```

Authenticating an MQTT client Java app with JAAS

Learn how to authenticate a client with JAAS. Complete the steps in this task to modify the sample program `JAASLoginModule.java` and configure IBM WebSphere MQ to authenticate an MQTT client Java app with JAAS.

1. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible". See System requirements for IBM Mobile Messaging and M2M Client Pack.
2. If there is a firewall between your client and the server, check that it does not block MQTT traffic.
3. You must have access to the `MQXR JAASLoginModule` and `JAASPrincipal` Java samples in an IBM WebSphere MQ installation. The samples are in the path `%MQ_FILE_PATH%\mqxr\samples.`
4. Complete the steps on Windows or Linux; the examples are taken from Windows.
5. To complete step “1” on page 68, you must have authorization to create the `MQXR_SAMPLE_QM` queue manager on IBM WebSphere MQ.

In the task, you output the MQTT `Sample` client identification parameters from your version of `JAASLoginModule`. Writing the client parameters out entails modifying the sample `JAASLoginModule` program and configuring IBM WebSphere MQ to load your version of `JAASLoginModule`.

1. Complete the steps in “Compile and run all the MQTT client sample Java apps from Eclipse” on page 14 to run the Paho MQTT `Sample` client.

   Your objective is to prepare a development environment to develop and test JAAS authentication. You require a Java development environment to tailor the JAAS authentication module. In the example, you run the sample Paho client for Java to test your JAAS configuration. For simplicity, use the same development environment to modify both the sample client and the sample JAAS login module. Alternatively you test your JAAS login module with the MQTT client for C, or any other MQTT client.

2. Optional: Add a user name and password parameter to the MQTT Paho sample.

   Note: If your Paho client for Java includes the user name and password parameters, this step is unnecessary. Check the download site for an update. See IBM messaging community downloads, otherwise change your copy of `Sample.java`.

   a) Open the package explorer in the `org.eclipse.paho.sample.mqttv3app` package in the Paho samples project.

   b) Right click `Sample.java` **Copy** > **Paste**. In the **Name Conflict** window, type the name `SampleForJAAS`.

   c) Add the following lines of code to the `main` method.

      i) After the line "`boolean ssl = false;`", declare the `userName` and `password` variables:

      ```
      String password = null;
      String userName = null;
      ```

      For compatibility with older MQTT servers, by default do not set the password and user name parameters.

      ii) After the line, "`case 'v': ssl = Boolean.valueOf(args[++i]).booleanValue(); break;`", parse the two new input parameters:

      ```
      case 'u': userName = args[++i]; break;
      case 'z': password = args[++i]; break;
      ```

      iii) Before the line, "`if (action.equals("publish")) {`", add `userName` and `password` to the `Sample` constructor arguments:

      ```
      Sample sampleClient = new Sample(url, clientId, cleanSession, quietMode, userName,
      password);
      ```

   d) Add `userName` and `password` to the constructor of `Sample`.

      Change:

      ```
      public Sample(String brokerUrl, String clientId, boolean cleanSession,
             boolean quietMode) throws MqttException {
      ```

      To:

      ```
      public Sample(String brokerUrl, String clientId, boolean cleanSession,
             boolean quietMode, String userName, char[] password) throws MqttException {
      ```

   e) Add the following lines of code to the `Sample` constructor.

      After the line "`conOpt.setCleanSession(clean);`", set the `userName` and `password` variables in the `conOpt` object in the `Sample` method:

      ```
      if(password != null ) {
         conOpt.setPassword(this.password.toCharArray());
      }
      if(userName != null) {
      ```

```
    conOpt.setUserName(this.userName);
}
```

f) In the `publish` and `subscribe` methods, modify the following lines of code:

Change the line "`client.connect();`" to

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

3. Create a Java project, `JAASSample`, for your JAAS example.
   a) In your Eclipse workspace, open the **New Java Project** wizard: Click **File** > **New** > **Java project**.
   b) In the **Project name** field, type `JAASSample`.
   c) In the JRE options, select `J2SE-1.5` as the execution environment JRE, and click **Next**.

      The JRE must match the JRE that your IBM WebSphere MQ server runs. IBM WebSphere MQ Version 7.5 runs `J2SE-1.5`.
   d) In the **Java Settings** window, click the **Libraries** tab and click **Add External JARS...**. Browse to the `%MQ_FILE_PATH%\mqxr\lib` directory and select **MQXR.jar**; click **Finish**.
4. Import the JAAS samples `JAASLoginModule` and `JAASPrincipal` classes.
   a) Right-click the `JAASSample` project in the Package Explorer **Import...** > **General** > **File System** and click **Next**.
   b) Browse to `%MQ_FILE_PATH%\mqxr\samples` and check `JAASLoginModule.java` and `JAASPrincipal.java`; click **Finish**.
   c) Select and right click both Java files in the Package Explorer, **Refactor...** > **Move**.
   d) In the **Move** window, verify `JAASSample` is selected as the destination for both elements, and click **Create Package...**
   e) Type `samples` into the **Name** field in the **New Java Package** wizard; click **Finish** > **OK**

   Eclipse builds the imported Java classes with a number of warnings about unused values.
5. Rename the `JAASLoginModule` class

   Rename the class so it is easier to distinguish it from the sample `JAASLoginModule` class that is shipped with IBM WebSphere MQ.
   a) Right-click `JAASloginModule.java` in the package explorer **Refactor...** > **Rename**.
   b) In the **Rename Compilation Unit** window, change the **New name** field from `JAASloginModule` to `MyJAASloginModule`; click **Finish**.
6. Modify the `MyJAASloginModule` class to output the content of the callback fields.
   a) Add the following line of code to `MyJAASloginModule.java`.

```
System.out.println("Username=" + username
                 + "\nPassword=" + new String(password)
                 + "\nClientId=" + clientId
                 + "\nNetwork address=" + networkAddress);
```

      Place the lines just before the statement:"`if (true) loggedIn = true;`"
   b) Press `CTRL+Shift+O` to reorganize imports, and save the file.
7. Rename `JAASPrincipal` to `MyJAASPrincipal`.

   Rename the class to avoid confusion with the sample `JAASPrincipal` class. In the example, leave the content of the `MyJAASPrincipal` class unaltered.
8. Give the user ID that is running the queue manager processes `read` and `execute` permissions to your JAAS classes.
   a) In Windows Explorer, open your Eclipse workspace directory. In the example, the Eclipse workspace location is represented by the Eclipse variable, *`workspace_loc`*.
   b) Browse to the directory that contains your `MyJAASLoginModule` and `MyJAASPrincipal` classes.

The directory path is *workspace_loc*\JAASSample\bin\samples

c) Select and then right-click both classes, and click **Properties**; click the **Security** tab in the **Properties** window.

d) Click **Add...**, type the object name mqm, and click **Check names** to verify it; click **OK**.

e) Select **mqm** in the list of **Group or user names** and check **read and execute** and **read** in the list of permissions for mqm; click OK.

9. Configure IBM WebSphere MQ to run your MyJAASLoginModule class.

a) Add a service.env file to the IBM WebSphere MQ configuration to define the class paths to load your MyJAASLoginModule class.

Create a service.env file with the following class path statement in your *WMQ_DATA_PATH* directory:

```
CLASSPATH=user.dir\JAASSample\bin
```

Where *user.dir* is the directory root for the class files that are compiled in yourEclipse workspace. The *WMQ_DATA_PATH* directory contains the qmgrs directory. See Additional environment variables.

Tip: CLASSPATH=*user.dir*\JAASSample\bin might be the only statement in the service.env file

b) Add a stanza, MyJAASStanza, to the jaas.config file to identify your MyJAASLoginModule class relative to the class paths in the service.env file.

jaas.config is in the queue manager mqxr directory; *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM\mqxr.

The stanza is:

```
MyJAASStanza {
  samples.MyJAASLoginModule required debug=true;
};
```

c) Configure a IBM WebSphere MQ Telemetry channel with the name of your JAAS configuration stanza.

Run the following command from a command window:

```
echo DEFINE CHANNEL('MyJAAS') CHLTYPE(MQTT) TRPTYPE(TCP) PORT(1890)
JAASCFG('MyJAASStanza') | runmqsc MQXR_SAMPLE_QM
```

The command ties the MyJAAS channel to the MyJAASStanza in the jaas.config file. By not specifying an MCAUSER option, or specifying the USECLTID option on the channel definition, the channel authorizes access to queue manager resources with the user name supplied by the MQTT client program. In the example, the user name that is supplied by the client is set to "Guest". The existing authorizations for Guest set by the SampleMQM command file are used in this example too.

10. Restart the IBM WebSphere MQ Telemetry service to read the new configuration data.
To restart the IBM WebSphere MQ Telemetry service, start the queue manager, or the service from IBM WebSphere MQ Explorer, or run the following commands for the sample configuration:

```
echo stop  service(SYSTEM.MQXR.SERVICE) | runmqsc  MQXR_SAMPLE_QM
echo start service(SYSTEM.MQXR.SERVICE) | runmqsc  MQXR_SAMPLE_QM
```

11. Run the Sample program.

To configure a run configuration for SampleForJAAS follow the same procedure as in step “1” on page 68, with the following modifications:

a) Set the port number to 1890 to match the MQTT channel configuration.

b) Add the parameters -u Guest -z password to the passwords on the **(x)= Arguments** tab for both the Subscriber and Publisher configurations you created for the Sample program

The sample programs are run without any change in output, except the port number is now 1890 rather than 1883.

In the *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM directory, open the mqxr.stdout file. The output from MyJAASLoginModule is written to mqxr.stdout:

```
Username=Guest
Password=passord
ClientId=SampleJavaV3_subscribe
Network address=/127.0.0.1
```

If your example does not work read the troubleshooting topic for JAAS; "Resolving problem: JAAS login module not called by the telemetry service" on page 172, and try these debugging tips.

1. Add -verbose to the parameters in *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM\mqxr\java.properties. From this log, you can see whether your class loaded successfully.

   The output is written to *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM\mqxr.stderr.
2. Look in *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM\errors\mqxr.log for exceptions that are thrown in MyJAASLoginModule. For example, if you attempt to output a null password, which is a character array and not a string, an exception is thrown.
3. Look in *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM\errors\AMQERR01.log. If the username in userName is not authorized to access queue manager resources and the channel is configured with no MCAUSER or USECLTID option, any errors are reported here.
4. Check the name of the stanza in *WMQ_DATA_PATH*\Qmgrs\MQXR_SAMPLE_QM\mqxr\jaas.config is the same as the name in the MQTT channel that is configured for the port the Sample client is trying to connect to.
5. Check the path in the stanza matches the path to the MyJAASLoginModule class in Eclipse; for example:

   ```
   MyJAASStanza {
     samples.MyJAASLoginModule required debug=true;
   };
   ```
6. To eliminate whether the fault lies in the class path in the service.env file in *WMQ_DATA_PATH* is not being picked up correctly, change the line "set CLASSPATH=%MQXRCLASSPATH%;%CLASSPATH%" in %MQ_FILE_PATH%\mqxr\bin\controlMQXR.BAT to include your class path. You can also echo the class path. However, the class path does not contain the class path that is set in service.env, so this only works if you modify the controlMQXR.BAT file.

Related concepts

"MQTT security" on page 47
Three concepts are fundamental to MQTT security: identity, authentication, and authorization. Identity is about naming the client that is being authorized and given authority. Authentication is about proving the identity of the client, and authorization is about managing the rights that are given the client.

"Telemetry channel JAAS configuration" on page 108
Configure JAAS to authenticate the Username sent by the client.

Related tasks

"Resolving problem: JAAS login module not called by the telemetry service" on page 172
Find out if your JAAS login module is not being called by the telemetry (MQXR) service, and configure JAAS to correct the problem.

"Building and running the secure MQTT client sample Java app" on page 50
Based on a Windows example, you can get up and running with the secure sample Java app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible"

"Connecting the MQTT client sample Java app on Android over SSL" on page 58

Get up and running with the sample Android MQTT client connected to IBM WebSphere MQ over SSL.

Related information

Additional environment variables

Connecting the MQTT messaging client for JavaScript over SSL and WebSockets

Connect your web app securely to IBM WebSphere MQ by using the MQTT messaging client for JavaScript sample HTML pages with SSL and the WebSocket protocol.

1. You must have access to an MQTT version 3 server that supports the MQTT protocol over WebSockets.
2. The browser must support SSL and the WebSocket protocol. See "Restrictions in browser support for mobile messaging web apps over SSL" on page 163.
3. The SSL channels must be started.

Complete this task to run the MQTT messaging client for JavaScript sample pages over SSL. The task directs you to "Generating keys and certificates" on page 89 to create certificates and configure IBM WebSphere MQ.

Secure the SSL channel with either certificate authority signed keys, or self-signed keys.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT protocol over secure WebSockets.

   - IBM MessageSight, and releases of IBM WebSphere MQ Version 7.5.0.1 and later, do this.
2. Optional: Install a Java development kit (JDK) at Version 7 or later.

   If you are setting up a test system, and you want to use self-signed certificates, you need to use the JDK Version 7 **keytool** command to certify your certificates. If you are setting up a production system, and sending certificate signing requests (CSR) to an external certificate authority, you do not need the Version 7 JDK.
3. Create and run the scripts to generate key-pairs and certificates, and configure IBM WebSphere MQ as the MQTT server.

   Follow the steps in "Generating keys and certificates" on page 89 to create and run the scripts. The scripts are also listed in "Example scripts to configure SSL certificates for Windows" on page 74.

   The common name of the server certificate must match the DNS name of the server channel. Some browsers accept certificates that contain a list of common names; for example:

   ```
   "CN=localhost, CN=*.example.com"
   ```

   Other browsers accept only one common name. For example, Firefox, up to version 18, accepts only one common name. Later versions might be different.
4. Check the SSL channels are running and are set up as you expect.

   On IBM WebSphere MQ, type the following command into a command window:

   - Linux

     ```
     echo 'DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
     echo 'DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
     ```

   - Windows

     ```
     echo DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
     echo DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
     ```
5. Install the certificates in the browser certificate store.

   For the example, choose one of the following server certificates:

a. For a self-signed server certificate, the certificate is `srvcertselfsigned.cer`.

b. For a server certificate signed by your private certificate authority the certificate is `cacert.cer`.

c. For a server certificate signed by an external certificate authority, check the root certificate of the certificate authority is already installed in the certificate store.

Depending on the support available in your browser, install `cacert.cer` into the list of Trusted Root Certification Authorities. See "Restrictions in browser support for mobile messaging web apps over SSL" on page 163.

6. Optional: Authenticate the client.

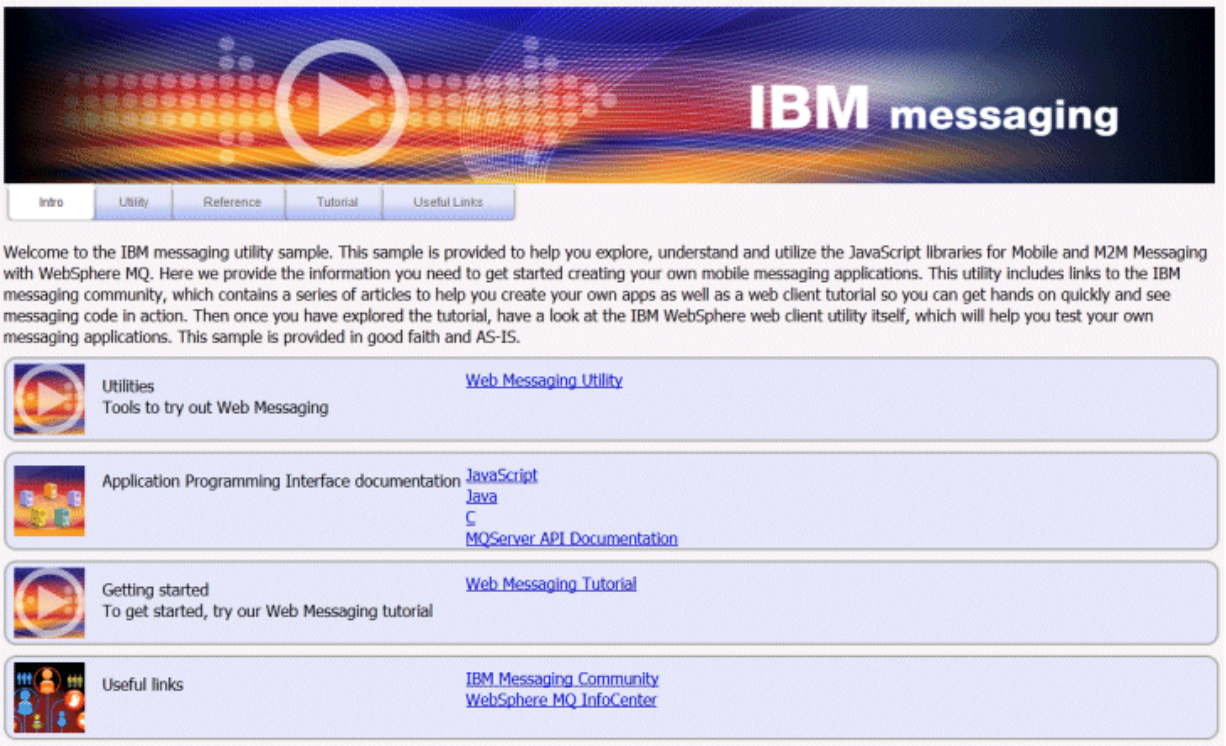
   In the example you install `cacert.cer` to authenticate the server and encrypt the channel, but not to authenticate the client. To authenticate the client you must install the client keystore, `cltkeystore.p12`, in the browser. Not all browsers support authenticating clients. See "Restrictions in browser support for mobile messaging web apps over SSL" on page 163.

7. Connect to the secure WebSockets channel.

   Open your browser and type the URL of the WebSockets channel in the address bar; in the example:

   ```
   https://localhost:8886
   ```

   IBM WebSphere MQ responds with the first page of the MQTT messaging client sample JavaScript pages.



   If the connection fails, and you ran the example scripts to set up the sample MQTT queue manager, try connecting to the normal WebSockets channel on port 1886. Success on port 1886 isolates the failure to the SSL connection.

   ```
   https://localhost:1886
   ```

Related concepts

"The MQTT messaging client for JavaScript and web apps" on page 110

"How to program messaging apps in JavaScript" on page 113

Related tasks

"Generating keys and certificates" on page 89

Follow this procedure to generate keys and certificates for Java and C clients, including Android and iOS apps, and the IBM WebSphere MQ and IBM MessageSight servers.

“Getting started with the MQTT messaging client for JavaScript” on page 21
You can get started with the MQTT messaging client for JavaScript by displaying the messaging client sample home page, and browsing the resources to which it links. To display this home page, you configure an MQTT server to accept connections from the MQTT messaging client sample JavaScript pages, then you type the URL that you have configured on the server into a web browser. The MQTT messaging client for JavaScript automatically starts on your device, and the messaging client sample home page is displayed. This page contains links to utilities, programming interface documentation, a tutorial, and other useful information.

Related reference

“Restrictions in browser support for mobile messaging web apps over SSL” on page 163
There are differences in capability between different browsers, on different platforms. Understanding these differences helps you configure your apps, certificate authorities (CAs) and client certificates to connect using the MQTT messaging client for JavaScript over SSL and WebSockets.

Example scripts to configure SSL certificates for Windows

** **

The example command files create the certificates and certificate stores as described in the steps in the task. In addition, the example sets up the MQTT client queue manager to use the server certificate store. The example deletes and re-creates the queue manager by calling the `SampleMQM.bat` script that is provided with IBM WebSphere MQ.

`initcert.bat`

`initcert.bat` sets the names and paths to certificates and other parameters that are required by the **`keytool`** and **openSSL** commands. The settings are described in comments in the script.

```
@echo off
@rem Set the path where you installed the MQTT SDK
@rem and short cuts to the samples directories.
set SDKRoot=C:\MQTT
set jsamppath=%SDKRoot%\sdk\clients\java\samples
set csamppath=%SDKRoot%\sdk\clients\c\samples
```

```
@rem Set the paths to Version 7 of the JDK
@rem and to the directory where you built the openSSL package.
@rem Set short cuts to the tools.
set javapath=C:\Program Files\IBM\Java70
set keytool="%javapath%\jre\bin\keytool.exe"
set ikeyman="%javapath%\jre\bin\ikeyman.exe"
set openssl=%SDKRoot%\openSSL
set runopenssl="%openssl%\bin\openssl"
```

```
@rem Set the path to where certificates are to be stored,
@rem and set global security parameters.
@rem Omit set password, and the security tools prompt you for passwords.
@rem Validity is the expiry time of the certificates in days.
set certpath=%SDKRoot%\Certificates
set password=password
set validity=5000
set algorithm=RSA
```

```
@rem Set the certificate authority (CA) jks keystore and certificate parameters.
@rem Omit this step, unless you are defining your own certificate authority.
@rem The CA keystore contains the key-pair for your own certificate authority.
@rem You must protect the CA keystore.
@rem The CA certificate is the self-signed certificate authority public certificate.
@rem It is commonly known as the CA root certificate.
set caalias=caalias
set cadname="CN=mqttca.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cakeypass=%password%
@rem ca key store
set cajkskeystore=%certpath%\cakeystore.jks
set cajkskeystorepass=%password%
```

```
@rem ca certificate (root certificate)
set cacert=%certpath%\cacert.cer
```

```
@rem Set the server jks keystore and certificate parameters.
@rem The server keystore contains the key-pair for the server.
@rem You must protect the server keystore.
@rem If you then export the server certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the server keystore for the server key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the server certificate to the CA.
@rem When you now export the server certificate,
@rem the exported certificate includes the certificate chain.
set srvalias=srvalias
set srvdname="CN=mqttserver.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set srvkeypass=%password%
@rem server key stores
set srvjkskeystore=%certpath%\srvkeystore.jks
set srvjkskeystorepass=%password%
@rem server certificates
set srvcertreq=%certpath%\srvcertreq.csr
set srvcertcasigned=%certpath%\srvcertcasigned.cer
set srvcertselfsigned=%certpath%\srvcertselfsigned.cer
```

```
@rem Set the client jks keystore and certificate parameters
@rem Omit this step, unless you are authenticating clients.
@rem The client keystore contains the key-pair for the client.
@rem You must protect the client keystore.
@rem If you then export the client certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the client keystore for the client key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the client certificate to the CA.
@rem When you now export the client certificate,
@rem the exported certificate includes the certificate chain.
set cltalias=cltalias
set cltdname="CN=mqttclient.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cltkeypass=%password%
@rem client key stores
set cltjkskeystore=%certpath%\cltkeystore.jks
set cltjkskeystorepass=%password%
set cltcertreq=%certpath%\cltcertreq.csr
set cltcertcasigned=%certpath%\cltcacertsigned.cer
set cltcertselfsigned=%certpath%\cltcertselfsigned.cer
```

```
@rem Set the paths to the client truststores signed by CA and signed by server key.
@rem You only need to define one of the trust stores.
@rem A trust store holds certificates that you trust,
@rem which are used to authenticate untrusted certificates.
@rem In this example, when the client authenticates the MQTT server it connects to,
@rem it authenticates the certificate it is sent by the server
@rem with the certificates in its trust store.
@rem For example, the MQTT server sends its server certificate,
@rem and the client authenticates it with either the same server certificate
@rem that you have stored in the cltsrvtruststore.jks trust store,
@rem or against the CA certificate, if the server certificate is signed by the CA.
set cltcajkstruststore=%certpath%\cltcatruststore.jks
set cltcajkstruststorepass=%password%
set cltsrvjkstruststore=%certpath%\cltsrvtruststore.jks
set cltsrvjkstruststorepass=%password%
```

```
@rem Set the paths to the client PKCS12 and PEM key and trust stores.
@rem Omit this step, unless you are configuring a C or iOS client.
@rem You only need to define either one of the trust stores for storing CA
@rem or server signed server certificates.
set cltp12keystore=%certpath%\cltkeystore.p12
set cltp12keystorepass=%password%
set cltpemkeystore=%certpath%\cltkeystore.pem
set cltpemkeystorepass=%password%
set cltcap12truststore=%certpath%\cltcatruststore.p12
set cltcap12truststorepass=%password%
set cltcapemtruststore=%certpath%\cltcatruststore.pem
set cltcapemtruststorepass=%password%
set cltsrvp12truststore=%certpath%\cltsrvtruststore.p12
set cltsrvp12truststorepass=%password%
```

```
set cltsrvpemtruststore=%certpath%\cltsrvtruststore.pem
set cltsrvpemtruststorepass=%password%
```

```
@rem set WMQ Variables
set authopt=NEVER
set authreq=REQUIRED
set qm=MQXR_SAMPLE_QM
set host=localhost
set mcauser='Guest'
set portsslopt=8884
set chlopt=SSLOPT
set portsslreq=8885
set chlreq=SSLREQ
V7.5.0.1 set portws=1886
set chlws=PLAINWS
set chlssloptws=SSLOPTWS
set portssloptws=8886
set chlsslreqws=SSLREQWS
set portsslreqws=8887
set mqlog=%certpath%\wmq.log
```

cleancert.bat

The commands in the `cleancert.bat` script delete the MQTT client queue manager to ensure that the server certificate store is not locked, and then delete all the keystores and certificates that are created by the sample security scripts.

```
@rem Delete the MQTT sample queue manager, MQXR_SAMPLE_QM
call "%MQ_FILE_PATH%\bin\setmqenv" -s
endmqm -i %qm%
dltmqm %qm%
```

```
@rem Erase all the certificates and key stores created by the sample scripts.
erase %cajkskeystore%
erase %cacert%
erase %srvjkskeystore%
erase %srvcertreq%
erase %srvcertcasigned%
erase %srvcertselfsigned%
erase %cltjkskeystore%
erase %cltp12keystore%
erase %cltpemkeystore%
erase %cltcertreq%
erase %cltcertcasigned%
erase %cltcertselfsigned%
erase %cltcajkstruststore%
erase %cltcap12truststore%
erase %cltcapemtruststore%
erase %cltsrvjkstruststore%
erase %cltsrvp12truststore%
erase %cltsrvpemtruststore%
erase %mqlog%
@echo Cleared all certificates
dir %certpath%\*.* /b
```

genkeys.bat

The commands in the `genkeys.bat` script create key-pairs for your private certificate authority, the server, and a client.

```
@rem
@echo ____________________________________________________________________________
@echo Generate %caalias%, %srvalias%, and %cltalias% key-pairs in %cajkskeystore%,
%srvjkskeystore%, and %cltjkskeystore%
@rem
@rem -- Generate a client certificate and a private key pair
@rem Omit this step, unless you are authenticating clients.
%keytool% -genkeypair -noprompt -alias %cltalias% -dname %cltdname% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass% -keypass %cltkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem -- Generate a server certificate and private key pair
%keytool% -genkeypair -noprompt -alias %srvalias% -dname %srvdname% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass% -keypass %srvkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem Create CA, client and server key-pairs
@rem  -- Generate a CA certificate and private key pair - The extension asserts this is a
certificate authority certificate, which is required to import into firefox
%keytool% -genkeypair -noprompt -ext bc=ca:true -alias %caalias%  -dname %cadname%
-keystore %cajkskeystore% -storepass %cajkskeystorepass%  -keypass %cakeypass% -keyalg
%algorithm% -validity %validity%
```

sscerts.bat

The commands in the `sscerts.bat` script export the client and server self-signed certificates from their keystores, and import the server certificate into the client truststore, and the client certificate into the server keystore. The server does not have a truststore. The commands create a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ____________________________________________________________________________
@echo Export self-signed certificates:  %srvcertselfsigned% and %cltcertselfsigned%
@rem Export Server public certificate
%keytool% -exportcert -noprompt -rfc -alias %srvalias% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass% -file %srvcertselfsigned%
@rem Export Client public certificate
@rem Omit this step, unless you are authenticating clients.
%keytool% -exportcert -noprompt -rfc -alias %cltalias% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass% -file %cltcertselfsigned%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned server certificate %srvcertselfsigned% to client trust store:
%cltsrvjkstruststore%
@rem Import the server certificate into the client-server trust store (for server self-
signed authentication)
%keytool% -import -noprompt -alias %srvalias% -file %srvcertselfsigned% -keystore
%cltsrvjkstruststore% -storepass %cltsrvjkstruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned client certificate %cltcertselfsigned% to server trust store:
%srvjkskeystore%
@rem Import the client certificate into the server trust store (for client self-signed
authentication)
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %cltalias% -file %cltcertselfsigned% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-server trust store from the jks client-server trust store:
%cltsrvpemtruststore%
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltsrvjkstruststore% -destkeystore
%cltsrvp12truststore% -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltsrvjkstruststorepass% -deststorepass %cltsrvp12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltsrvp12truststore% -out %cltsrvpemtruststore% -passin
pass:%cltsrvp12truststorepass% -passout pass:%cltsrvpemtruststorepass%@rem
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

cacerts.bat

The script imports the certificate authority root certificate into the private keystores. The CA root certificate is needed to create the keychain between the root certificate and the signed certificate. The `cacerts.bat` script exports the client and server certificate requests from their keystores. The script signs the certificate requests with the key of the private certificate authority in the `cajkskeystore.jks` keystore, then imports the signed certificates back into the same keystores

from which the requests came. The import creates the certificate chain with the CA root certificate. The script creates a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Export self-signed certificates: %cacert%
@rem
@rem Export CA public certificate
%keytool% -exportcert -noprompt -rfc -alias %caalias%  -keystore %cajkskeystore% -storepass
%cajkskeystorepass%  -file %cacert%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Add CA to server key and client key and trust stores: %srvjkskeystore%,
%cltjkskeystore%, %cltcajkstruststore%,
@rem The CA certificate is necessary to create key chains in the client and server key
stores,
@rem and to certify key chains in the server key store and the client trust store
@rem
@rem Import the CA root certificate into the server key store
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass%
@rem Import the CA root certificate into the client key store
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass%
@rem Import the CA root certificate into the client ca-trust store (for ca chained
authentication)
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltcajkstruststore%
-storepass %cltcajkstruststorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Create certificate signing requests: %srvcertreq% and %cltcertreq%
@rem
@rem Create a certificate signing request (CSR) for the server key
%keytool% -certreq -alias %srvalias% -file %srvcertreq% -keypass %srvkeypass% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Create a certificate signing request (CSR) for the client key
%keytool% -certreq -alias %cltalias% -file %cltcertreq% -keypass %cltkeypass% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Sign certificate requests: %srvcertcasigned% and %cltcertcasigned%
@rem The requests are signed with the ca key in the cajkskeystore.jks keystore
@rem
@rem Sign server certificate request
%keytool% -gencert -infile %srvcertreq% -outfile %srvcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
@rem Sign client certificate request
@rem Omit this step, unless you are authenticating clients.
%keytool% -gencert -infile %cltcertreq% -outfile %cltcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Import the signed certificates back into the key stores to create the key chain:
%srvjkskeystore% and %cltjkskeystore%
@rem
@rem Import the signed server certificate
%keytool% -import -noprompt -alias %srvalias% -file %srvcertcasigned% -keypass %srvkeypass%
-keystore %srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Import the signed client certificate and key chain back into the client keystore
%keytool% -import -noprompt -alias %cltalias% -file %cltcertcasigned% -keypass %cltkeypass%
-keystore %cltjkskeystore% -storepass %cltjkskeystorepass%
@rem
@rem The CA certificate is needed in the server key store, and the client trust store
@rem to verify the key chain sent from the client or server
@echo Delete the CA certificate from %cltjkskeystore%: it causes a problem in converting
keystore to pem
@rem Omit this step, unless you are authenticating clients.
%keytool% -delete -alias %caalias% -keystore %cltjkskeystore% -storepass
%cltjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-ca trust store from the jks client-ca trust store:
%cltcapemtruststore%
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltcajkstruststore%  -destkeystore
%cltcap12truststore%  -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltcajkstruststorepass%  -deststorepass %cltcap12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltcap12truststore%  -out %cltcapemtruststore%  -passin
pass:%cltcap12truststorepass%  -passout pass:%cltpemtruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

mqcerts.bat

The script lists the keystores and certificates in the certificate directory. It then creates the MQTT sample queue manager and configures the secure telemetry channels.

```
@echo ____________________________________________________________________________
@echo List keystores and certificates
dir %certpath%\*.* /b
```

```
@rem
@echo Create queue manager and define mqtt channels and certificate stores
call "%MQ_FILE_PATH%\mqxr\Samples\SampleMQM" >> %mqlog%
echo DEFINE CHANNEL(%chlreq%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreq%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlopt%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslopt%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
V 7.5.0.1
echo DEFINE CHANNEL(%chlsslreqws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreqws%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlssloptws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portssloptws%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlws%)       CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portws%)
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
@echo MQ logs saved in %mqlog%echo
```

Building and running the secure MQTT client sample C app

Based on a Windows example, you can get up and running with the secure sample C app on any operating system for which you can compile the C source. Verify that you can run the sample C app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server.

1. You must have access to an MQTT version 3.1 server that supports the MQTT protocol over SSL.
2. If there is a firewall between your client and the server, check that it does not block MQTT traffic.
3. Binary versions of the client for C libraries are provided for a number of operating systems. For some of these operating systems, the secure version of the client is not provided as a binary file. For those operating systems, you must follow the instructions in “Building the MQTT client for C libraries” on page 27.
4. For problem resolution, IBM support might require you to run the MQTT client for C on a reference platform.
5. The SSL channels must be started.

For an overview of supported and reference platforms, see System requirements for IBM Mobile Messaging and M2M Client Pack. For details of what's supported for the C client, see the relevant sections of System Requirements for WebSphere MQ V7.5 Telemetry.

As an illustration, this article shows you how to compile and run the secure MQTT client sample C app on Windows from the command line. In the illustration, Microsoft Visual Studio 2010 is used to compile the client. You can modify the command-line scripts to compile and run the sample app on other operating systems, such as Linux and iOS.

Note:

The Windows scripts provided in this article assume that you build the entire OpenSSL package from source. If you choose to use the precompiled libraries that IBM provides, you might also prefer to get a precompiled binary release of OpenSSL. Precompiled libraries are not available for use with iOS.

Secure the SSL channel with either certificate authority signed keys, or self-signed keys.

1. Choose an MQTT server to which you can connect the client app.

   The server must support the MQTT version 3.1 protocol over SSL. All MQTT servers from IBM do this, including IBM WebSphere MQ and IBM MessageSight. See "Getting started with MQTT servers" on page 128.
2. Optional: Install a Java development kit (JDK) at Version 7 or later.

   Version 7 is required to run the **keytool** command to certify certificates. If you are not going to certify certificates, you do not require the Version 7 JDK.
3. Install a C development environment on the platform on which you are building.

   The makefiles in the examples in this topic target the following tools:

   - iOS For iOS, on Apple Mac with OS X 10.8.2 with the iOS development tools from Xcode.
   - Linux For Linux, gcc version 4.4.6 from Red Hat Enterprise Linux version 6.2.

     The minimum supported level of the `glibc` C library is 2.12, and of the Linux kernel is 2.6.32.
   - Windows For Microsoft Windows, Visual Studio version 10.0.
4. Download the Mobile Messaging and M2M Client Pack and install the MQTT SDK.

   There is no installation program, you just expand the downloaded file.

   a. Download the Mobile Messaging and M2M Client Pack.

   b. Create a folder where you are going to install the SDK.

      You might want to name the folder `MQTT`. The path to this folder is referred to here as *sdkroot*.

   c. Expand the compressed Mobile Messaging and M2M Client Pack file contents into *sdkroot*. The expansion creates a directory tree that starts at *sdkroot*\SDK.
5. Optional: Follow the steps in "Building the MQTT client for C libraries" on page 27.

   Do this step only if the MQTT SDK does not include the secure C client library for your target operating system.

   - Windows The libraries are `mqttv3cs.lib` for compiling, and `mqttv3cs.dll` for running.
   - Linux The library is `libmqttv3cs.so`
   - iOS The library is `libmqttv3cs.a`
6. Create and run the scripts to generate key-pairs and certificates, and configure IBM WebSphere MQ as the MQTT server.

   Follow the steps in "Generating keys and certificates" on page 89 to create and run the scripts. The scripts are also listed in "Example scripts to configure SSL certificates for Windows" on page 83.
7. Check the SSL channels are running and are set up as you expect.

On IBM WebSphere MQ, type the following command into a command window:

- Linux

```
echo 'DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
echo 'DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL' | runmqsc MQXR_SAMPLE_QM
```

- Windows

```
echo DISPLAY CHSTATUS(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
echo DISPLAY CHANNEL(SSL*) CHLTYPE(MQTT) ALL | runmqsc MQXR_SAMPLE_QM
```

8. Create the scripts to build and run the secure MQTT client sample C app.
   a) Create and run `sscclient.bat` to test an SSL channel that is secured with self-signed certificates.
   b) Create and run `cacclient.bat` to test an SSL channel that is secured with certificate authority signed certificates.

The results are similar to running the unsecured client.

```
Connected to ssl://localhost:8884
Subscribing to topic "MQTTV3SSample/#" qos 2
Topic:          MQTTV3SSample/C/v3
Message:        Message from MQTTv3 SSL C client
QoS:            2
Connected to ssl://localhost:8885
Subscribing to topic "MQTTV3SSample/#" qos 2
Topic:          MQTTV3SSample/C/v3
Message:        Message from MQTTv3 SSL C client
QoS:            2
```

Figure 16. Secure subscriber

```
Setting environment for using Microsoft Visual Studio 2010 x86 tools.
MQTTV3SSample.c
Connected to ssl://localhost:8884
Publishing to topic "MQTTV3SSample/C/v3" qos 2
Disconnected
Press any key to continue . . .
Connected to ssl://localhost:8885
Publishing to topic "MQTTV3SSample/C/v3" qos 2
Disconnected
Press any key to continue . . .
```

Figure 17. Secure publisher

Scripts to run the secure MQTT client sample C app

Run the scripts in "Example scripts to configure SSL certificates for Windows" on page 83 before you run these scripts.

Secure MQTT client sample C app with self-signed certificates.
Run this script with the self-signed certificates that you created by running the `sscerts.bat` script.

```
@echo off
setlocal
cd %csamppath%
erase MQTTV3SSample.obj
erase MQTTV3SSample.exe
call "C:\Program Files\Microsoft Visual Studio 10.0\VC\vcvarsall.bat" x86
cl  /nologo /D "WIN32" /I "..\include" "MQTTV3SSample.c" /link /
nologo ..\windows_ia32\mqttv3cs.lib
set path=%path%;%csamppath%\..\windows_ia32
ping -n 2 127.0.0.1 > NUL 2>&1
@echo start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportopt% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportopt% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
@echo MQTTV3SSample -b %host% -p %sslportopt% -k %cltpemkeystore% -w %cltpemkeystorepass%
-r %cltsrvpemtruststore% -v 1
MQTTV3SSample -b %host% -p %sslportopt% -k %cltpemkeystore% -w %cltpemkeystorepass% -r
%cltsrvpemtruststore% -v 1
pause
ping -n 2 127.0.0.1 > NUL 2>&1
@echo start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportreq% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportreq% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
@echo MQTTV3SSample -b %host% -p %sslportreq% -k %cltpemkeystore% -w %cltpemkeystorepass%
-r %cltsrvpemtruststore% -v 1
MQTTV3SSample -b %host% -p %sslportreq% -k %cltpemkeystore% -w %cltpemkeystorepass% -r
%cltsrvpemtruststore% -v 1
pause
ping -n 2 127.0.0.1 > NUL 2>&1
endlocal
```

Figure 18. `sscclient.bat`

Run the MQTT secure client sample C app with certificate authority signed certificates.
Run this script with the certificate authority signed certificates that you created by running the `cacerts.bat` script.

```
@echo off
setlocal
cd %csamppath%
erase MQTTV3SSample.obj
erase MQTTV3SSample.exe
call "C:\Program Files\Microsoft Visual Studio 10.0\VC\vcvarsall.bat" x86
cl  /nologo /D "WIN32" /I "..\include" "MQTTV3SSample.c" /link /
nologo ..\windows_ia32\mqttv3cs.lib
set path=%path%;%csamppath%\..\windows_ia32
ping -n 2 127.0.0.1 > NUL 2>&1
@echo start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportopt% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportopt% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
@echo MQTTV3SSample -b %host% -p %sslportopt% -k %cltpemkeystore% -w %cltpemkeystorepass%
-r %cltsrvpemtruststore% -v 1
MQTTV3SSample -b %host% -p %sslportopt% -k %cltpemkeystore% -w %cltpemkeystorepass% -r
%cltsrvpemtruststore% -v 1
pause
ping -n 2 127.0.0.1 > NUL 2>&1
@echo start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportreq% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
start "MQTT Subscriber" MQTTV3SSample -a subscribe -b %host% -p %sslportreq% -k
%cltpemkeystore% -w %cltpemkeystorepass% -r %cltsrvpemtruststore% -v 1
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
@echo MQTTV3SSample -b %host% -p %sslportreq% -k %cltpemkeystore% -w %cltpemkeystorepass%
-r %cltsrvpemtruststore% -v 1
MQTTV3SSample -b %host% -p %sslportreq% -k %cltpemkeystore% -w %cltpemkeystorepass% -r
%cltsrvpemtruststore% -v 1
pause
ping -n 2 127.0.0.1 > NUL 2>&1
endlocal
```

Figure 19. `cacclient.bat`

Related concepts

“MQTT security” on page 47

Three concepts are fundamental to MQTT security: identity, authentication, and authorization. Identity is about naming the client that is being authorized and given authority. Authentication is about proving the identity of the client, and authorization is about managing the rights that are given the client.

Related tasks

“Generating keys and certificates” on page 89

Follow this procedure to generate keys and certificates for Java and C clients, including Android and iOS apps, and the IBM WebSphere MQ and IBM MessageSight servers.

Example scripts to configure SSL certificates for Windows

The example command files create the certificates and certificate stores as described in the steps in the task. In addition, the example sets up the MQTT client queue manager to use the server certificate store. The example deletes and re-creates the queue manager by calling the `SampleMQM.bat` script that is provided with IBM WebSphere MQ.

`initcert.bat`

`initcert.bat` sets the names and paths to certificates and other parameters that are required by the **keytool** and **openSSL** commands. The settings are described in comments in the script.

```
@echo off
@rem Set the path where you installed the MQTT SDK
@rem and short cuts to the samples directories.
set SDKRoot=C:\MQTT
set jsamppath=%SDKRoot%\sdk\clients\java\samples
set csamppath=%SDKRoot%\sdk\clients\c\samples
```

```
@rem Set the paths to Version 7 of the JDK
@rem and to the directory where you built the openSSL package.
@rem Set short cuts to the tools.
set javapath=C:\Program Files\IBM\Java70
```

```
set keytool="%javapath%\jre\bin\keytool.exe"
set ikeyman="%javapath%\jre\bin\ikeyman.exe"
set openssl=%SDKRoot%\openSSL
set runopenssl="%openssl%\bin\openssl"
```

```
@rem Set the path to where certificates are to be stored,
@rem and set global security parameters.
@rem Omit set password, and the security tools prompt you for passwords.
@rem Validity is the expiry time of the certificates in days.
set certpath=%SDKRoot%\Certificates
set password=password
set validity=5000
set algorithm=RSA
```

```
@rem Set the certificate authority (CA) jks keystore and certificate parameters.
@rem Omit this step, unless you are defining your own certificate authority.
@rem The CA keystore contains the key-pair for your own certificate authority.
@rem You must protect the CA keystore.
@rem The CA certificate is the self-signed certificate authority public certificate.
@rem It is commonly known as the CA root certificate.
set caalias=caalias
set cadname="CN=mqttca.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cakeypass=%password%
@rem ca key store
set cajkskeystore=%certpath%\cakeystore.jks
set cajkskeystorepass=%password%
@rem ca certificate (root certificate)
set cacert=%certpath%\cacert.cer
```

```
@rem Set the server jks keystore and certificate parameters.
@rem The server keystore contains the key-pair for the server.
@rem You must protect the server keystore.
@rem If you then export the server certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the server keystore for the server key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the server certificate to the CA.
@rem When you now export the server certificate,
@rem the exported certificate includes the certificate chain.
set srvalias=srvalias
set srvdname="CN=mqttserver.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set srvkeypass=%password%
@rem server key stores
set srvjkskeystore=%certpath%\srvkeystore.jks
set srvjkskeystorepass=%password%
@rem server certificates
set srvcertreq=%certpath%\srvcertreq.csr
set srvcertcasigned=%certpath%\srvcertcasigned.cer
set srvcertselfsigned=%certpath%\srvcertselfsigned.cer
```

```
@rem Set the client jks keystore and certificate parameters
@rem Omit this step, unless you are authenticating clients.
@rem The client keystore contains the key-pair for the client.
@rem You must protect the client keystore.
@rem If you then export the client certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the client keystore for the client key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the client certificate to the CA.
@rem When you now export the client certificate,
@rem the exported certificate includes the certificate chain.
set cltalias=cltalias
set cltdname="CN=mqttclient.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cltkeypass=%password%
@rem client key stores
set cltjkskeystore=%certpath%\cltkeystore.jks
set cltjkskeystorepass=%password%
set cltcertreq=%certpath%\cltcertreq.csr
set cltcertcasigned=%certpath%\cltcacertsigned.cer
set cltcertselfsigned=%certpath%\cltcertselfsigned.cer
```

```
@rem Set the paths to the client truststores signed by CA and signed by server key.
@rem You only need to define one of the trust stores.
@rem A trust store holds certificates that you trust,
@rem which are used to authenticate untrusted certificates.
```

```
@rem In this example, when the client authenticates the MQTT server it connects to,
@rem it authenticates the certificate it is sent by the server
@rem with the certificates in its trust store.
@rem For example, the MQTT server sends its server certificate,
@rem and the client authenticates it with either the same server certificate
@rem that you have stored in the cltsrvtruststore.jks trust store,
@rem or against the CA certificate, if the server certificate is signed by the CA.
set cltcajkstruststore=%certpath%\cltcatruststore.jks
set cltcajkstruststorepass=%password%
set cltsrvjkstruststore=%certpath%\cltsrvtruststore.jks
set cltsrvjkstruststorepass=%password%
```

```
@rem Set the paths to the client PKCS12 and PEM key and trust stores.
@rem Omit this step, unless you are configuring a C or iOS client.
@rem You only need to define either one of the trust stores for storing CA
@rem or server signed server certificates.
set cltp12keystore=%certpath%\cltkeystore.p12
set cltp12keystorepass=%password%
set cltpemkeystore=%certpath%\cltkeystore.pem
set cltpemkeystorepass=%password%
set cltcap12truststore=%certpath%\cltcatruststore.p12
set cltcap12truststorepass=%password%
set cltcapemtruststore=%certpath%\cltcatruststore.pem
set cltcapemtruststorepass=%password%
set cltsrvp12truststore=%certpath%\cltsrvtruststore.p12
set cltsrvp12truststorepass=%password%
set cltsrvpemtruststore=%certpath%\cltsrvtruststore.pem
set cltsrvpemtruststorepass=%password%
```

```
@rem set WMQ Variables
set authopt=NEVER
set authreq=REQUIRED
set qm=MQXR_SAMPLE_QM
set host=localhost
set mcauser='Guest'
set portsslopt=8884
set chlopt=SSLOPT
set portsslreq=8885
set chlreq=SSLREQ
V7.5.0.1 set portws=1886
set chlws=PLAINWS
set chlssloptws=SSLOPTWS
set portssloptws=8886
set chlsslreqws=SSLREQWS
set portsslreqws=8887
set mqlog=%certpath%\wmq.log
```

cleancert.bat

The commands in the `cleancert.bat` script delete the MQTT client queue manager to ensure that the server certificate store is not locked, and then delete all the keystores and certificates that are created by the sample security scripts.

```
@rem Delete the MQTT sample queue manager, MQXR_SAMPLE_QM
call "%MQ_FILE_PATH%\bin\setmqenv" -s
endmqm -i %qm%
dltmqm %qm%
```

```
@rem Erase all the certificates and key stores created by the sample scripts.
erase %cajkskeystore%
erase %cacert%
erase %srvjkskeystore%
erase %srvcertreq%
erase %srvcertcasigned%
erase %srvcertselfsigned%
erase %cltjkskeystore%
erase %cltp12keystore%
erase %cltpemkeystore%
erase %cltcertreq%
erase %cltcertcasigned%
erase %cltcertselfsigned%
erase %cltcajkstruststore%
erase %cltcap12truststore%
erase %cltcapemtruststore%
erase %cltsrvjkstruststore%
erase %cltsrvp12truststore%
```

```
erase %cltsrvpemtruststore%
erase %mqlog%
@echo Cleared all certificates
dir %certpath%\*.* /b
```

genkeys.bat

The commands in the `genkeys.bat` script create key-pairs for your private certificate authority, the server, and a client.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Generate %caalias%, %srvalias%, and %cltalias% key-pairs in %cajkskeystore%,
%srvjkskeystore%, and %cltjkskeystore%
@rem
@rem -- Generate a client certificate and a private key pair
@rem Omit this step, unless you are authenticating clients.
%keytool% -genkeypair -noprompt -alias %cltalias% -dname %cltdname% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass% -keypass %cltkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem -- Generate a server certificate and private key pair
%keytool% -genkeypair -noprompt -alias %srvalias% -dname %srvdname% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass% -keypass %srvkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem Create CA, client and server key-pairs
@rem  -- Generate a CA certificate and private key pair - The extension asserts this is a
certificate authority certificate, which is required to import into firefox
%keytool% -genkeypair -noprompt -ext bc=ca:true -alias %caalias%  -dname %cadname%
-keystore %cajkskeystore% -storepass %cajkskeystorepass%  -keypass %cakeypass% -keyalg
%algorithm% -validity %validity%
```

sscerts.bat

The commands in the `sscerts.bat` script export the client and server self-signed certificates from their keystores, and import the server certificate into the client truststore, and the client certificate into the server keystore. The server does not have a truststore. The commands create a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Export self-signed certificates:  %srvcertselfsigned% and %cltcertselfsigned%
@rem Export Server public certificate
%keytool% -exportcert -noprompt -rfc -alias %srvalias% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass% -file %srvcertselfsigned%
@rem Export Client public certificate
@rem Omit this step, unless you are authenticating clients.
%keytool% -exportcert -noprompt -rfc -alias %cltalias% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass% -file %cltcertselfsigned%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Add selfsigned server certificate %srvcertselfsigned% to client trust store:
%cltsrvjkstruststore%
@rem Import the server certificate into the client-server trust store (for server self-
signed authentication)
%keytool% -import -noprompt -alias %srvalias% -file %srvcertselfsigned% -keystore
%cltsrvjkstruststore% -storepass %cltsrvjkstruststorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Add selfsigned client certificate %cltcertselfsigned% to server trust store:
%srvjkskeystore%
@rem Import the client certificate into the server trust store (for client self-signed
authentication)
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %cltalias% -file %cltcertselfsigned% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
```

```
@echo Create a pem client-server trust store from the jks client-server trust store:
%cltsrvpemtruststore%
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltsrvjkstruststore% -destkeystore
%cltsrvp12truststore% -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltsrvjkstruststorepass% -deststorepass %cltsrvp12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltsrvp12truststore% -out %cltsrvpemtruststore% -passin
pass:%cltsrvp12truststorepass% -passout pass:%cltsrvpemtruststorepass%@rem
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

cacerts.bat

The script imports the certificate authority root certificate into the private keystores. The CA root certificate is needed to create the keychain between the root certificate and the signed certificate. The `cacerts.bat` script exports the client and server certificate requests from their keystores. The script signs the certificate requests with the key of the private certificate authority in the `cajkskeystore.jks` keystore, then imports the signed certificates back into the same keystores from which the requests came. The import creates the certificate chain with the CA root certificate. The script creates a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ____________________________________________________________________________
@echo Export self-signed certificates: %cacert%
@rem
@rem Export CA public certificate
%keytool% -exportcert -noprompt -rfc -alias %caalias%  -keystore %cajkskeystore% -storepass
%cajkskeystorepass%  -file %cacert%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add CA to server key and client key and trust stores: %srvjkskeystore%,
%cltjkskeystore%, %cltcajkstruststore%,
@rem The CA certificate is necessary to create key chains in the client and server key
stores,
@rem and to certify key chains in the server key store and the client trust store
@rem
@rem Import the CA root certificate into the server key store
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass%
@rem Import the CA root certificate into the client key store
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass%
@rem Import the CA root certificate into the client ca-trust store (for ca chained
authentication)
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltcajkstruststore%
-storepass %cltcajkstruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create certificate signing requests: %srvcertreq% and %cltcertreq%
@rem
@rem Create a certificate signing request (CSR) for the server key
%keytool% -certreq -alias %srvalias% -file %srvcertreq% -keypass %srvkeypass% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Create a certificate signing request (CSR) for the client key
%keytool% -certreq -alias %cltalias% -file %cltcertreq% -keypass %cltkeypass% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Sign certificate requests: %srvcertcasigned% and %cltcertcasigned%
@rem The requests are signed with the ca key in the cajkskeystore.jks keystore
@rem
@rem Sign server certificate request
%keytool% -gencert -infile %srvcertreq% -outfile %srvcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
```

```
@rem Sign client certificate request
@rem Omit this step, unless you are authenticating clients.
%keytool% -gencert -infile %cltcertreq% -outfile %cltcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Import the signed certificates back into the key stores to create the key chain:
%srvjkskeystore% and %cltjkskeystore%
@rem
@rem Import the signed server certificate
%keytool% -import -noprompt -alias %srvalias% -file %srvcertcasigned% -keypass %srvkeypass%
-keystore %srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Import the signed client certificate and key chain back into the client keystore
%keytool% -import -noprompt -alias %cltalias% -file %cltcertcasigned% -keypass %cltkeypass%
-keystore %cltjkskeystore% -storepass %cltjkskeystorepass%
@rem
@rem The CA certificate is needed in the server key store, and the client trust store
@rem to verify the key chain sent from the client or server
@echo Delete the CA certificate from %cltjkskeystore%: it causes a problem in converting
keystore to pem
@rem Omit this step, unless you are authenticating clients.
%keytool% -delete -alias %caalias% -keystore %cltjkskeystore% -storepass
%cltjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-ca trust store from the jks client-ca trust store:
%cltcapemtruststore%
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltcajkstruststore%  -destkeystore
%cltcap12truststore%  -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltcajkstruststorepass%  -deststorepass %cltcap12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltcap12truststore%  -out %cltcapemtruststore%  -passin
pass:%cltcap12truststorepass%  -passout pass:%cltpemtruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

mqcerts.bat

The script lists the keystores and certificates in the certificate directory. It then creates the MQTT sample queue manager and configures the secure telemetry channels.

```
@echo ____________________________________________________________________________
@echo List keystores and certificates
dir %certpath%\*.* /b
```

```
@rem
@echo Create queue manager and define mqtt channels and certificate stores
call "%MQ_FILE_PATH%\mqxr\Samples\SampleMQM" >> %mqlog%
echo DEFINE CHANNEL(%chlreq%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreq%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlopt%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslopt%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
V7.5.0.1
echo DEFINE CHANNEL(%chlsslreqws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreqws%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlssloptws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portssloptws%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlws%)       CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portws%)
```

```
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
@echo MQ logs saved in %mqlog%echo
```

Generating keys and certificates

Follow this procedure to generate keys and certificates for Java and C clients, including Android and iOS apps, and the IBM WebSphere MQ and IBM MessageSight servers.

1. You must have a copy of the **keytool** command. Not all versions of **keytool** support conversion of keystores from Java keystore (JKS) to Public Key Cryptographic System (PKCS), or the signing of certificate requests. The example uses the **keytool** command in JDK Version 7.0, which supports both of these capabilities.
2. If you intend to generate keys and certificates for the client for C, which are in Privacy-Enhanced Mail (PEM) format, you must have a copy of the **openSSL** command. Follow the steps in "Building the MQTT client for C libraries" on page 27 to build the openSSL package.
3. Change the parameter values in the `initcert.bat` script to meet your needs. In particular, you might choose to omit the password parameters to avoid writing passwords down. The **keytool** command prompts you for missing passwords.

You require keys and certificates to create secure SSL connections between MQTT clients and servers. This task shows you two different ways to create the keys and certificates you require: self-signed and signed by your own certificate authority. The method that you follow depends on how you plan to manage keystores and certificates.

To use certificates that are signed by an external certificate authority, replace the signing step in `cacerts.bat`, with sending the certificate requests to an external certificate authority. The certificate authority might return an intermediate and a root certificate in addition to the signed certificate. Follow the guidance that is provided by the external CA where to install the returned certificates.

The IBM WebSphere MQ server searches for certificates only in the certificate store you specify in the telemetry channel configuration parameters. It does not additionally search in the JSE `cacerts` store. A Java client searches for certificates in the truststore you specify. If you do not specify a truststore, it searches in the `cacerts` keystore in the JSE `jre\lib\security` directory. Android clients search for certificates in the predefined certificate store on the Android device. C client apps and iOS apps search only in the certificate stores the application specifies.

Android and Java clients search a pre-configured truststore for trusted certificates. CA root certificates are stored in the Android trusted certificate store, and in the JSE `jre\lib\security\cacerts` store. If the root certificate of the CA that certified the server certificate is already installed in the pre-configured truststore, do not define a client truststore. The only configuration that is required is to set the TCP/IP port for the secured MQTT server channel.

The tools to create keys and certificates, and manage all the different formats, are not simple to use. They have numerous parameters to manage, and **openSSL** requires a configuration file, `openssl.cnf`, and command-line parameters. No one tool provides all the functions that are required to manage keys and certificates for applications that run on both C and Java. Telemetry channels in IBM WebSphere MQ require a JKS keystore, and so the examples mainly use the Java certificate tools, **ikeyman** and **keytool**. However, the Java tools do not support the PEM format, which is required for C client apps. To create keystores in the PEM format, run the **openSSL** tool. The **openSSL** tool converts keystores from PKCS12 format to PEM format, and **keytool** converts keystores between JKS format and PKCS12 format. No `openssl.cnf` file is required for keystore conversion. You require **openSSL** only if you plan to build C client apps or iOS apps. If you prefer working with **openSSL**, you can use it to sign certificates instead of signing certificates with **keytool**.

1. Open a command window to run the following scripts.
2. Create and run the `initcert.bat` script to set the parameters that are required to run the MQTT secure sample clients.
3. Create and run the `cleancert.bat` script to clear the environment ready to create new keystores and certificates.
4. Create and run the `genkeys.bat` script to generate the key-pairs you require.

5. Do one of the following options:
   - Create and run the `sscerts.bat` script to create self-signed certificates.
   - Create and run the `cacerts.bat` script to create certificate authority signed certificate chains.
6. Create and run the `mqcerts.bat` script to create the `MQXR_SAMPLE_QM` queue manager and configure its telemetry channels.

Related tasks

"Building and running the secure MQTT client sample C app" on page 79
Based on a Windows example, you can get up and running with the secure sample C app on any operating system for which you can compile the C source. Verify that you can run the sample C app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server.

"Building and running the secure MQTT client sample Java app" on page 50
Based on a Windows example, you can get up and running with the secure sample Java app on either IBM MessageSight or IBM WebSphere MQ as the MQTT server. You can run an MQTT client for Java app on any platform with JSE 1.5 or above that is "Java Compatible"

Example scripts to configure SSL certificates for Windows

The example command files create the certificates and certificate stores as described in the steps in the task. In addition, the example sets up the MQTT client queue manager to use the server certificate store. The example deletes and re-creates the queue manager by calling the `SampleMQM.bat` script that is provided with IBM WebSphere MQ.

`initcert.bat`

`initcert.bat` sets the names and paths to certificates and other parameters that are required by the **keytool** and **openSSL** commands. The settings are described in comments in the script.

```
@echo off
@rem Set the path where you installed the MQTT SDK
@rem and short cuts to the samples directories.
set SDKRoot=C:\MQTT
set jsamppath=%SDKRoot%\sdk\clients\java\samples
set csamppath=%SDKRoot%\sdk\clients\c\samples
```

```
@rem Set the paths to Version 7 of the JDK
@rem and to the directory where you built the openSSL package.
@rem Set short cuts to the tools.
set javapath=C:\Program Files\IBM\Java70
set keytool="%javapath%\jre\bin\keytool.exe"
set ikeyman="%javapath%\jre\bin\ikeyman.exe"
set openssl=%SDKRoot%\openSSL
set runopenssl="%openssl%\bin\openssl"
```

```
@rem Set the path to where certificates are to be stored,
@rem and set global security parameters.
@rem Omit set password, and the security tools prompt you for passwords.
@rem Validity is the expiry time of the certificates in days.
set certpath=%SDKRoot%\Certificates
set password=password
set validity=5000
set algorithm=RSA
```

```
@rem Set the certificate authority (CA) jks keystore and certificate parameters.
@rem Omit this step, unless you are defining your own certificate authority.
@rem The CA keystore contains the key-pair for your own certificate authority.
@rem You must protect the CA keystore.
@rem The CA certificate is the self-signed certificate authority public certificate.
@rem It is commonly known as the CA root certificate.
set caalias=caalias
set cadname="CN=mqttca.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cakeypass=%password%
@rem ca key store
set cajkskeystore=%certpath%\cakeystore.jks
set cajkskeystorepass=%password%
@rem ca certificate (root certificate)
set cacert=%certpath%\cacert.cer
```

```
@rem Set the server jks keystore and certificate parameters.
@rem The server keystore contains the key-pair for the server.
@rem You must protect the server keystore.
@rem If you then export the server certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the server keystore for the server key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the server certificate to the CA.
@rem When you now export the server certificate,
@rem the exported certificate includes the certificate chain.
set srvalias=srvalias
set srvdname="CN=mqttserver.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set srvkeypass=%password%
@rem server key stores
set srvjkskeystore=%certpath%\srvkeystore.jks
set srvjkskeystorepass=%password%
@rem server certificates
set srvcertreq=%certpath%\srvcertreq.csr
set srvcertcasigned=%certpath%\srvcertcasigned.cer
set srvcertselfsigned=%certpath%\srvcertselfsigned.cer
```

```
@rem Set the client jks keystore and certificate parameters
@rem Omit this step, unless you are authenticating clients.
@rem The client keystore contains the key-pair for the client.
@rem You must protect the client keystore.
@rem If you then export the client certificate it is self-signed.
@rem Alternatively, if you export a certificate signing request (CSR)
@rem from the client keystore for the client key,
@rem and import the signed certificate back into the same keystore,
@rem it forms a certificate chain.
@rem The certificate chain links the client certificate to the CA.
@rem When you now export the client certificate,
@rem the exported certificate includes the certificate chain.
set cltalias=cltalias
set cltdname="CN=mqttclient.ibm.id.com, OU=ID, O=IBM, L=Hursley, S=Hants, C=GB"
set cltkeypass=%password%
@rem client key stores
set cltjkskeystore=%certpath%\cltkeystore.jks
set cltjkskeystorepass=%password%
set cltcertreq=%certpath%\cltcertreq.csr
set cltcertcasigned=%certpath%\cltcacertsigned.cer
set cltcertselfsigned=%certpath%\cltcertselfsigned.cer
```

```
@rem Set the paths to the client truststores signed by CA and signed by server key.
@rem You only need to define one of the trust stores.
@rem A trust store holds certificates that you trust,
@rem which are used to authenticate untrusted certificates.
@rem In this example, when the client authenticates the MQTT server it connects to,
@rem it authenticates the certificate it is sent by the server
@rem with the certificates in its trust store.
@rem For example, the MQTT server sends its server certificate,
@rem and the client authenticates it with either the same server certificate
@rem that you have stored in the cltsrvtruststore.jks trust store,
@rem or against the CA certificate, if the server certificate is signed by the CA.
set cltcajkstruststore=%certpath%\cltcatruststore.jks
set cltcajkstruststorepass=%password%
set cltsrvjkstruststore=%certpath%\cltsrvtruststore.jks
set cltsrvjkstruststorepass=%password%
```

```
@rem Set the paths to the client PKCS12 and PEM key and trust stores.
@rem Omit this step, unless you are configuring a C or iOS client.
@rem You only need to define either one of the trust stores for storing CA
@rem or server signed server certificates.
set cltp12keystore=%certpath%\cltkeystore.p12
set cltp12keystorepass=%password%
set cltpemkeystore=%certpath%\cltkeystore.pem
set cltpemkeystorepass=%password%
set cltcap12truststore=%certpath%\cltcatruststore.p12
set cltcap12truststorepass=%password%
set cltcapemtruststore=%certpath%\cltcatruststore.pem
set cltcapemtruststorepass=%password%
set cltsrvp12truststore=%certpath%\cltsrvtruststore.p12
set cltsrvp12truststorepass=%password%
set cltsrvpemtruststore=%certpath%\cltsrvtruststore.pem
set cltsrvpemtruststorepass=%password%
```

```
@rem set WMQ Variables
set authopt=NEVER
set authreq=REQUIRED
set qm=MQXR_SAMPLE_QM
set host=localhost
set mcauser='Guest'
set portsslopt=8884
set chlopt=SSLOPT
set portsslreq=8885
set chlreq=SSLREQ
V7.5.0.1 set portws=1886
set chlws=PLAINWS
set chlssloptws=SSLOPTWS
set portssloptws=8886
set chlsslreqws=SSLREQWS
set portsslreqws=8887
set mqlog=%certpath%\wmq.log
```

cleancert.bat

The commands in the `cleancert.bat` script delete the MQTT client queue manager to ensure that the server certificate store is not locked, and then delete all the keystores and certificates that are created by the sample security scripts.

```
@rem Delete the MQTT sample queue manager, MQXR_SAMPLE_QM
call "%MQ_FILE_PATH%\bin\setmqenv" -s
endmqm -i %qm%
dltmqm %qm%
```

```
@rem Erase all the certificates and key stores created by the sample scripts.
erase %cajkskeystore%
erase %cacert%
erase %srvjkskeystore%
erase %srvcertreq%
erase %srvcertcasigned%
erase %srvcertselfsigned%
erase %cltjkskeystore%
erase %cltp12keystore%
erase %cltpemkeystore%
erase %cltcertreq%
erase %cltcertcasigned%
erase %cltcertselfsigned%
erase %cltcajkstruststore%
erase %cltcap12truststore%
erase %cltcapemtruststore%
erase %cltsrvjkstruststore%
erase %cltsrvp12truststore%
erase %cltsrvpemtruststore%
erase %mqlog%
@echo Cleared all certificates
dir %certpath%\*.* /b
```

genkeys.bat

The commands in the `genkeys.bat` script create key-pairs for your private certificate authority, the server, and a client.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Generate %caalias%, %srvalias%, and %cltalias% key-pairs in %cajkskeystore%,
%srvjkskeystore%, and %cltjkskeystore%
@rem
@rem -- Generate a client certificate and a private key pair
@rem Omit this step, unless you are authenticating clients.
%keytool% -genkeypair -noprompt -alias %cltalias% -dname %cltdname% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass% -keypass %cltkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem -- Generate a server certificate and private key pair
%keytool% -genkeypair -noprompt -alias %srvalias% -dname %srvdname% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass% -keypass %srvkeypass% -keyalg %algorithm%
-validity %validity%
```

```
@rem Create CA, client and server key-pairs
@rem  -- Generate a CA certificate and private key pair - The extension asserts this is a
```

```
certificate authority certificate, which is required to import into firefox
%keytool% -genkeypair -noprompt -ext bc=ca:true -alias %caalias%  -dname %cadname%
-keystore %cajkskeystore% -storepass %cajkskeystorepass%  -keypass %cakeypass% -keyalg
%algorithm% -validity %validity%
```

sscerts.bat

The commands in the `sscerts.bat` script export the client and server self-signed certificates from their keystores, and import the server certificate into the client truststore, and the client certificate into the server keystore. The server does not have a truststore. The commands create a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ____________________________________________________________________________
@echo Export self-signed certificates:  %srvcertselfsigned% and %cltcertselfsigned%
@rem Export Server public certificate
%keytool% -exportcert -noprompt -rfc -alias %srvalias% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass% -file %srvcertselfsigned%
@rem Export Client public certificate
@rem Omit this step, unless you are authenticating clients.
%keytool% -exportcert -noprompt -rfc -alias %cltalias% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass% -file %cltcertselfsigned%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned server certificate %srvcertselfsigned% to client trust store:
%cltsrvjkstruststore%
@rem Import the server certificate into the client-server trust store (for server self-
signed authentication)
%keytool% -import -noprompt -alias %srvalias% -file %srvcertselfsigned% -keystore
%cltsrvjkstruststore% -storepass %cltsrvjkstruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Add selfsigned client certificate %cltcertselfsigned% to server trust store:
%srvjkskeystore%
@rem Import the client certificate into the server trust store (for client self-signed
authentication)
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %cltalias% -file %cltcertselfsigned% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-server trust store from the jks client-server trust store:
%cltsrvpemtruststore%
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltsrvjkstruststore% -destkeystore
%cltsrvp12truststore% -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltsrvjkstruststorepass% -deststorepass %cltsrvp12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltsrvp12truststore% -out %cltsrvpemtruststore% -passin
pass:%cltsrvp12truststorepass% -passout pass:%cltsrvpemtruststorepass%@rem
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

cacerts.bat

The script imports the certificate authority root certificate into the private keystores. The CA root certificate is needed to create the keychain between the root certificate and the signed certificate. The `cacerts.bat` script exports the client and server certificate requests from their keystores. The script signs the certificate requests with the key of the private certificate authority in the `cajkskeystore.jks` keystore, then imports the signed certificates back into the same keystores

from which the requests came. The import creates the certificate chain with the CA root certificate. The script creates a client truststore in PEM format from the client JKS truststore.

```
@rem
@echo ------------------------------------------------------------------------------
@echo Export self-signed certificates: %cacert%
@rem
@rem Export CA public certificate
%keytool% -exportcert -noprompt -rfc -alias %caalias%  -keystore %cajkskeystore% -storepass
%cajkskeystorepass%  -file %cacert%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Add CA to server key and client key and trust stores: %srvjkskeystore%,
%cltjkskeystore%, %cltcajkstruststore%,
@rem The CA certificate is necessary to create key chains in the client and server key
stores,
@rem and to certify key chains in the server key store and the client trust store
@rem
@rem Import the CA root certificate into the server key store
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %srvjkskeystore%
-storepass %srvjkskeystorepass%
@rem Import the CA root certificate into the client key store
@rem Omit this step, unless you are authenticating clients.
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltjkskeystore%
-storepass %cltjkskeystorepass%
@rem Import the CA root certificate into the client ca-trust store (for ca chained
authentication)
%keytool% -import -noprompt -alias %caalias% -file %cacert% -keystore %cltcajkstruststore%
-storepass %cltcajkstruststorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Create certificate signing requests: %srvcertreq% and %cltcertreq%
@rem
@rem Create a certificate signing request (CSR) for the server key
%keytool% -certreq -alias %srvalias% -file %srvcertreq% -keypass %srvkeypass% -keystore
%srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Create a certificate signing request (CSR) for the client key
%keytool% -certreq -alias %cltalias% -file %cltcertreq% -keypass %cltkeypass% -keystore
%cltjkskeystore% -storepass %cltjkskeystorepass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Sign certificate requests: %srvcertcasigned% and %cltcertcasigned%
@rem The requests are signed with the ca key in the cajkskeystore.jks keystore
@rem
@rem Sign server certificate request
%keytool% -gencert -infile %srvcertreq% -outfile %srvcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
@rem Sign client certificate request
@rem Omit this step, unless you are authenticating clients.
%keytool% -gencert -infile %cltcertreq% -outfile %cltcertcasigned% -alias %caalias%
-keystore %cajkskeystore% -storepass %cajkskeystorepass% -keypass %cakeypass%
```

```
@rem
@echo ------------------------------------------------------------------------------
@echo Import the signed certificates back into the key stores to create the key chain:
%srvjkskeystore% and %cltjkskeystore%
@rem
@rem Import the signed server certificate
%keytool% -import -noprompt -alias %srvalias% -file %srvcertcasigned% -keypass %srvkeypass%
-keystore %srvjkskeystore% -storepass %srvjkskeystorepass%
@rem Import the signed client certificate and key chain back into the client keystore
%keytool% -import -noprompt -alias %cltalias% -file %cltcertcasigned% -keypass %cltkeypass%
-keystore %cltjkskeystore% -storepass %cltjkskeystorepass%
@rem
@rem The CA certificate is needed in the server key store, and the client trust store
@rem to verify the key chain sent from the client or server
@echo Delete the CA certificate from %cltjkskeystore%: it causes a problem in converting
keystore to pem
@rem Omit this step, unless you are authenticating clients.
%keytool% -delete -alias %caalias% -keystore %cltjkskeystore% -storepass
%cltjkskeystorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client-ca trust store from the jks client-ca trust store:
%cltcapemtruststore%
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltcajkstruststore%  -destkeystore
%cltcap12truststore%  -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltcajkstruststorepass%  -deststorepass %cltcap12truststorepass%
%openssl%\bin\openssl pkcs12 -in %cltcap12truststore%  -out %cltcapemtruststore%  -passin
pass:%cltcap12truststorepass%  -passout pass:%cltpemtruststorepass%
```

```
@rem
@echo ____________________________________________________________________________
@echo Create a pem client key store from the jks client keystore
@rem Omit this step, unless you are configuring a C or iOS client.
@rem
%keytool% -importkeystore -noprompt -srckeystore %cltjkskeystore%      -destkeystore
%cltp12keystore%     -srcstoretype jks -deststoretype pkcs12 -srcstorepass
%cltjkskeystorepass%     -deststorepass %cltp12keystorepass%
%openssl%\bin\openssl pkcs12 -in %cltp12keystore%     -out %cltpemkeystore%      -passin
pass:%cltp12keystorepass% -passout pass:%cltpemkeystorepass%
```

mqcerts.bat

The script lists the keystores and certificates in the certificate directory. It then creates the MQTT sample queue manager and configures the secure telemetry channels.

```
@echo ____________________________________________________________________________
@echo List keystores and certificates
dir %certpath%\*.* /b
```

```
@rem
@echo Create queue manager and define mqtt channels and certificate stores
call "%MQ_FILE_PATH%\mqxr\Samples\SampleMQM" >> %mqlog%
echo DEFINE CHANNEL(%chlreq%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreq%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlopt%)      CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslopt%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) | runmqsc %qm% >> %mqlog%
V 7.5.0.1
echo DEFINE CHANNEL(%chlsslreqws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portsslreqws%)
SSLCAUTH(%authreq%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlssloptws%) CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portssloptws%)
SSLCAUTH(%authopt%) SSLKEYR('%srvjkskeystore%') SSLKEYP('%srvjkskeystorepass%')
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
echo DEFINE CHANNEL(%chlws%)       CHLTYPE(MQTT) TRPTYPE(TCP) PORT(%portws%)
MCAUSER(%mcauser%) PROTOCOL(HTTP) | runmqsc %qm% >> %mqlog%
@echo MQ logs saved in %mqlog%echo
```

MQTT client identification, authorization, and authentication

The telemetry (MQXR) service publishes, or subscribes to, WebSphere MQ topics on behalf of MQTT clients, using MQTT channels. The WebSphere MQ administrator configures the MQTT channel identity that is used for WebSphere MQ authorization. The administrator can define a common identity for the channel, or use the `Username` or `ClientIdentifier` of a client connected to the channel.

The telemetry (MQXR) service can authenticate the client using the `Username` supplied by the client, or by using a client certificate. The `Username` is authenticated using a password provided by the client.

To summarize: Client identification is the selection of the client identity. Depending on the context, the client is identified by the `ClientIdentifier`, `Username`, a common client identity created by the administrator, or a client certificate. The client identifier used for authenticity checking does not have to be the same identifier that is used for authorization.

MQTT client programs set the `Username` and `Password` that are sent to the server using an MQTT channel. They can also set the SSL properties that are required to encrypt and authenticate

the connection. The administrator decides whether to authenticate the MQTT channel, and how to authenticate the channel.

To authorize an MQTT client to access IBM WebSphere MQ objects, authorize the `ClientIdentifier`, or `Username` of the client, or authorize a common client identity. To permit a client to connect to IBM WebSphere MQ, authenticate the `Username`, or use a client certificate. Configure JAAS to authenticate the `Username`, and configure SSL to authenticate a client certificate.

If you set a `Password` at the client, either encrypt the connection using VPN, or configure the MQTT channel to use SSL, to keep the password private.

It is difficult to manage client certificates. For this reason, if the risks associated with password authentication are acceptable, password authentication is often used to authenticate clients.

If there is a secure way to manage and store the client certificate it is possible to rely on certificate authentication. However, it is rarely the case that certificates can be managed securely in the types of environments that telemetry is used in. Instead, the authentication of devices using client certificates is complemented by authenticating client passwords at the server. Because of the additional complexity, the use of client certificates is restricted to highly sensitive applications. The use of two forms of authentication is called two-factor authentication. You must know one of the factors, such as a password, and have the other, such as a certificate.

In a highly sensitive application, such as a chip-and-pin device, the device is locked down during manufacture to prevent tampering with the internal hardware and software. A trusted, time-limited, client certificate is copied to the device. The device is deployed to the location where it is to be used. Further authentication is performed each time the device is used, either using a password, or another certificate from a smart card.

MQTT client identity and authorization

Use the `ClientIdentifier`, `Username`, or a common client identity for authorization to access WebSphere MQ objects.

The IBM WebSphere MQ administrator has three choices for selecting the identity of the MQTT channel. The administrator makes the choice when defining or modifying the MQTT channel used by the client. The identity is used to authorize access to IBM WebSphere MQ topics. The choices are:

1. The client identifier.
2. An identity the administrator provides for the channel.
3. The `Username` passed from the MQTT client.

`Username` is an attribute of the MqttConnectOptions class. It must be set before the client connects to the service. Its default value is null.

Use the IBM WebSphere MQ **`setmqaut`** command to select which objects, and which actions, are authorized to be used by the identity associated with the MQTT channel. For example, to authorize a channel identity, `MQTTClient`, provided by the administrator of queue manager, `QM1`:

```
setmqaut -m QM1 -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p MQTTClient -all +put
setmqaut -m QM1 -t topic -n SYSTEM.BASE.TOPIC -p MQTTClient -all +pub +sub
```

Related information

Authorizing MQTT clients to access WebSphere MQ objects

MQTT client authentication using a password

Authenticate the `Username` using the client password. You can authenticate the client using a different identity to the identity used to authorize the client to publish and subscribe to topics.

The telemetry (MQXR) service uses JAAS to authenticate the client `Username`. JAAS uses the `Password` supplied by the MQTT client.

The IBM WebSphere MQ administrator decides whether to authenticate the `Username`, or not to authenticate at all, by configuring the MQTT channel a client connects to. Clients can be assigned to different channels, and each channel can be configured to authenticate its clients in different ways. Using JAAS, you can configure which methods must authenticate the client, and which can optionally authenticate the client.

The choice of identity for authentication does not affect the choice of identity for authorization. You might want to set up a common identity for authorization for administrative convenience, but authenticate each user to use that identity. The following procedure outlines the steps to authenticate individual users to use a common identity:

1. The IBM WebSphere MQ administrator sets the MQTT channel identity to any name, such as `MQTTClientUser`, using IBM WebSphere MQ Explorer.
2. The IBM WebSphere MQ administrator authorizes `MQTTClient` to publish and subscribe to any topic:

```
setmqaut -m QM1 -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p MQTTClient -all +put
setmqaut -m QM1 -t topic -n SYSTEM.BASE.TOPIC -p MQTTClient -all +pub +sub
```

3. The MQTT client app developer creates an `MqttConnectOptions` object and sets `Username` and `Password` before connecting to the server.
4. The security developer creates a JAAS `LoginModule` to authenticate the `Username` with the `Password` and includes it in the JAAS configuration file.
5. The IBM WebSphere MQ administrator configures the MQTT channel to authenticate the `UserName` of the client using JAAS.

MQTT client authentication using SSL

Connections between the MQTT client and the queue manager are always initiated by the MQTT client. The MQTT client is always the SSL client. Client authentication of the server and server authentication of the MQTT client are both optional.

By providing the client with a private signed digital certificate, you can authenticate the MQTT client to IBM WebSphere MQ. The IBM WebSphere MQ Administrator can force MQTT clients to authenticate themselves to the queue manager using SSL. You can only request client authentication as part of mutual authentication.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

Client authentication using SSL relies upon the client having a secret. The secret is the private key of the client in the case of a self-signed certificate, or a key provided by a certificate authority. The key is used to sign the digital certificate of the client. Anyone in possession of the corresponding public key can verify the digital certificate. Certificates can be trusted, or if they are chained, traced back through a certificate chain to a trusted root certificate. Client verification sends all the certificates in the certificate chain provided by the client to the server. The server checks the certificate chain until it finds a certificate it trusts. The trusted certificate is either the public certificate generated from a self-signed certificate, or a root certificate typically issued by a certificate authority. As a final, optional, step the trusted certificate can be compared with a "live" certificate revocation list.

The trusted certificate might be issued by a certificate authority and already included in the JRE certificate store. It might be a self-signed certificate, or any certificate that has been added to the telemetry channel keystore as a trusted certificate.

Note: The telemetry channel has a combined keystore/truststore that holds both the private keys to one or more telemetry channels, and any public certificates needed to authenticate clients. Because an SSL channel must have a keystore, and it is the same file as the channel truststore, the JRE certificate store is never referenced. The implication is that if authentication of a client requires a CA root certificate, you must place the root certificate in the keystore for the channel, even if the CA root certificate is already in the JRE certificate store. The JRE certificate store is never referenced.

Think about the threats that client authentication is intended to counter, and the roles the client and server play in countering the threats. Authenticating the client certificate alone is insufficient to prevent unauthorized access to a system. If someone else has got hold of the client device, the client device is not necessarily acting with the authority of the certificate holder. Never rely on a single defense against unwanted attacks. At least use a two-factor authentication approach and supplement possession of a certificate with knowledge of private information. For example, use JAAS, and authenticate the client using a password issued by the server.

The primary threat to the client certificate is that it gets into the wrong hands. The certificate is held in a password protected keystore at the client. How does it get placed in the keystore? How does the MQTT client get the password to the keystore? How secure is the password protection? Telemetry devices are often easy to remove, and then can be hacked in private. Must the device hardware be tamper-proof? Distributing and protecting client-side certificates is recognized to be hard; it is called the key-management problem.

A secondary threat is that the device is misused to access servers in unintended ways. For example, if the MQTT application is tampered with, it might be possible to use a weakness in the server configuration using the authenticated client identity.

To authenticate an MQTT client using SSL, configure the telemetry channel, and the client.

•
•

MQTT client configuration for client authentication using SSL

To authenticate the MQTT client using SSL, the client connects to a telemetry channel using SSL. It must specify a TCP port that corresponds to a telemetry channel that is configured to authenticate SSL clients.

For example, at the client:

```
MQTTClient mqttClient = new MqttClient( "ssl://www.example.org:8884", "clientId1");
mqttClient.connect();
```

The client JVM must use the standard socket factory from JSSE. If you are using Java ME, you must ensure that the JSSE package is loaded. If you are using Java SE, JSSE has been included with the JRE since Java version 1.4.1.

The SSL connection requires a number of SSL properties to be set before connecting. You can set the properties either by passing them to the JVM using the `-D` switch, or you can set the properties using the `MqttConnectionOptions.setSSLProperties` method.

If you load a non-standard socket factory, by calling the method `MqttConnectOptions.setSocketFactory(javax.net.SocketFactory)`, then the way SSL settings are passed to the network socket is application defined.

Add the digital certificate of the client, signed either using the private key of the client, or by a CA, to the password protected keystore on the client. If the certificate has a key chain, you can add the certificates from the key chain to the store. When the server verifies the client certificate, it uses the certificates sent by the client to match against certificates in its keystore. It is looking for the first match in the key chain with a certificate that it has. The remainder of the key chain is ignored.

The MQTT client sends all the certificates in its keystore to the server. If the server authenticates any of the key chains the client sends, then the client is authenticated.

You can also use SSL cipher suites for client authentication. Here is an alphabetic list of the SSL cipher suites that are currently supported:

- `SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DH_anon_EXPORT_WITH_RC4_40_MD5`
- `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`
- `SSL_DH_anon_WITH_AES_128_CBC_SHA`
- `SSL_DH_anon_WITH_DES_CBC_SHA`

- SSL_DH_anon_WITH_RC4_128_MD5
- SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_DSS_WITH_AES_128_CBC_SHA
- SSL_DHE_DSS_WITH_DES_CBC_SHA
- SSL_DHE_DSS_WITH_RC4_128_SHA
- SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_DHE_RSA_WITH_AES_128_CBC_SHA
- SSL_DHE_RSA_WITH_DES_CBC_SHA
- SSL_KRB5_EXPORT_WITH_DES_CBC_40_MD5
- SSL_KRB5_EXPORT_WITH_DES_CBC_40_SHA
- SSL_KRB5_EXPORT_WITH_RC4_40_MD5
- SSL_KRB5_EXPORT_WITH_RC4_40_SHA
- SSL_KRB5_WITH_3DES_EDE_CBC_MD5
- SSL_KRB5_WITH_3DES_EDE_CBC_SHA
- SSL_KRB5_WITH_DES_CBC_MD5
- SSL_KRB5_WITH_DES_CBC_SHA
- SSL_KRB5_WITH_RC4_128_MD5
- SSL_KRB5_WITH_RC4_128_SHA
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
- V7.5.0.2 SSL_RSA_FIPS_WITH_AES_128_CBC_SHA256
- V7.5.0.2 SSL_RSA_FIPS_WITH_AES_256_CBC_SHA256
- SSL_RSA_FIPS_WITH_DES_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_AES_128_CBC_SHA
- V7.5.0.2 SSL_RSA_WITH_AES_128_CBC_SHA256
- V7.5.0.2 SSL_RSA_WITH_AES_256_CBC_SHA256
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_NULL_SHA
- V7.5.0.2 SSL_RSA_WITH_NULL_SHA256
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA

V7.5.0.2 If you plan to use SHA-2 cipher suites, see “System requirements for using SHA-2 cipher suites with MQTT clients” on page 162.

Related concepts

“MQTT client configuration for channel authentication using SSL” on page 100
To authenticate the telemetry channel using SSL, the client must connect to the telemetry channel using SSL. It must specify a port that corresponds to a telemetry channel that is configured for SSL. The

configuration must include a passphrase protected keystore that contains the privately signed digital certificate of the server.

Telemetry channel authentication using SSL

Connections between the MQTT client and the queue manager are always initiated by the MQTT client. The MQTT client is always the SSL client. Client authentication of the server and server authentication of the MQTT client are both optional.

The client always attempts to authenticate the server, unless the client is configured to use a CipherSpec that supports anonymous connection. If the authentication fails, then the connection is not established.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

Server authentication using SSL authenticates the server to which you are about to send confidential information to. The client performs the checks matching the certificates sent from the server, against certificates placed in its truststore, or in its JRE `cacerts` store.

The JRE certificate store is a JKS file, `cacerts`. It is located in `JRE InstallPath\lib\security\`. It is installed with the default password `changeit`. You can either store certificates you trust in the JRE certificate store, or in the client truststore. You cannot use both stores. Use the client truststore if you want to keep the public certificates the client trusts separate from certificates other Java applications use. Use the JRE certificate store if you want to use a common certificate store for all Java applications running on the client. If you decide to use the JRE certificate store review the certificates it contains, to make sure you trust them.

You can modify the JSSE configuration by supplying a different trust provider. You can customize a trust provider to perform different checks on a certificate. In some OGSi environments that have used the MQTT client, the environment provides a different trust provider.

To authenticate the telemetry channel using SSL, configure the server, and the client.

•

Related concepts

“MQTT client configuration for channel authentication using SSL” on page 100
To authenticate the telemetry channel using SSL, the client must connect to the telemetry channel using SSL. It must specify a port that corresponds to a telemetry channel that is configured for SSL. The configuration must include a passphrase protected keystore that contains the privately signed digital certificate of the server.

MQTT client configuration for channel authentication using SSL

To authenticate the telemetry channel using SSL, the client must connect to the telemetry channel using SSL. It must specify a port that corresponds to a telemetry channel that is configured for SSL. The configuration must include a passphrase protected keystore that contains the privately signed digital certificate of the server.

For example, at the client:

```
MQTTClient mqttClient = new MqttClient( "ssl://www.example.org:8884", "clientId1");
mqttClient.connect();
```

The client JVM must use the standard socket factory from JSSE. If you are using Java ME, you must ensure that the JSSE package is loaded. If you are using Java SE, JSSE has been included with the JRE since Java version 1.4.1.

The SSL connection requires a number of SSL properties to be set before connecting. You can set the properties either by passing them to the JVM using the `-D` switch, or you can set the properties using the `MqttConnectionOptions.setSSLProperties` method.

If you load a non-standard socket factory, by calling the method `MqttConnectOptions.setSocketFactory(javax.net.SocketFactory)`, then the way SSL settings are passed to the network socket is application defined.

Code the client to connect to the telemetry channel using SSL, and configure the client to trust a server certificate in one of three ways:

Using a server certificate signed by well-known certificate authority in the `cacerts` store.

No additional configuration, if the server sends all the intermediate keys in the certificate chain. You are advised to review the certificates in the `cacerts` store of the client JRE and change the password to the `cacerts` store

Other certificates

Store the certificates you trust in the truststore at the client. You must store at least one of the certificates in the certificate chain in the truststore, Set the truststore parameters in `MqttConnectionOptions.SSLProperty`.

- `com.ibm.ssl.trustStore`
- `com.ibm.ssl.trustStorePassword`

Using a custom trust manager

Implement a trust provider and pass it the name of the algorithm that is used. Set the name of the provider class and the algorithm to use in `MqttConnectionOptions.SSLProperty`.

- `com.ibm.ssl.trustStoreProvider`
- `com.ibm.ssl.trustStoreManager`

You can also use SSL cipher suites for channel authentication. Here is an alphabetic list of the SSL cipher suites that are currently supported:

- `SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DH_anon_EXPORT_WITH_RC4_40_MD5`
- `SSL_DH_anon_WITH_3DES_EDE_CBC_SHA`
- `SSL_DH_anon_WITH_AES_128_CBC_SHA`
- `SSL_DH_anon_WITH_DES_CBC_SHA`
- `SSL_DH_anon_WITH_RC4_128_MD5`
- `SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA`
- `SSL_DHE_DSS_WITH_AES_128_CBC_SHA`
- `SSL_DHE_DSS_WITH_DES_CBC_SHA`
- `SSL_DHE_DSS_WITH_RC4_128_SHA`
- `SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA`
- `SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA`
- `SSL_DHE_RSA_WITH_AES_128_CBC_SHA`
- `SSL_DHE_RSA_WITH_DES_CBC_SHA`
- `SSL_KRB5_EXPORT_WITH_DES_CBC_40_MD5`
- `SSL_KRB5_EXPORT_WITH_DES_CBC_40_SHA`
- `SSL_KRB5_EXPORT_WITH_RC4_40_MD5`
- `SSL_KRB5_EXPORT_WITH_RC4_40_SHA`
- `SSL_KRB5_WITH_3DES_EDE_CBC_MD5`
- `SSL_KRB5_WITH_3DES_EDE_CBC_SHA`
- `SSL_KRB5_WITH_DES_CBC_MD5`

- SSL_KRB5_WITH_DES_CBC_SHA
- SSL_KRB5_WITH_RC4_128_MD5
- SSL_KRB5_WITH_RC4_128_SHA
- SSL_RSA_EXPORT_WITH_DES40_CBC_SHA
- SSL_RSA_EXPORT_WITH_RC4_40_MD5
- SSL_RSA_FIPS_WITH_3DES_EDE_CBC_SHA
- V7.5.0.2 SSL_RSA_FIPS_WITH_AES_128_CBC_SHA256
- V7.5.0.2 SSL_RSA_FIPS_WITH_AES_256_CBC_SHA256
- SSL_RSA_FIPS_WITH_DES_CBC_SHA
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- SSL_RSA_WITH_AES_128_CBC_SHA
- V7.5.0.2 SSL_RSA_WITH_AES_128_CBC_SHA256
- V7.5.0.2 SSL_RSA_WITH_AES_256_CBC_SHA256
- SSL_RSA_WITH_DES_CBC_SHA
- SSL_RSA_WITH_NULL_MD5
- SSL_RSA_WITH_NULL_SHA
- V7.5.0.2 SSL_RSA_WITH_NULL_SHA256
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_RC4_128_SHA

V7.5.0.2 If you plan to use SHA-2 cipher suites, see "System requirements for using SHA-2 cipher suites with MQTT clients" on page 162.

Related concepts

"MQTT client configuration for client authentication using SSL" on page 98
To authenticate the MQTT client using SSL, the client connects to a telemetry channel using SSL. It must specify a TCP port that corresponds to a telemetry channel that is configured to authenticate SSL clients.

Publication privacy on telemetry channels

The privacy of MQTT publications sent in either direction across telemetry channels is secured by using SSL to encrypt transmissions over the connection.

MQTT clients that connect to telemetry channels use SSL to secure the privacy of publications transmitted on the channel using symmetric key cryptography. Because the endpoints are not authenticated, you cannot trust channel encryption alone. Combine securing privacy with server or mutual authentication.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

For a typical configuration, which encrypts the channel and authenticates the server, consult "Telemetry channel authentication using SSL" on page 100.

Encrypting SSL connections without authenticating the server exposes the connection to man-in-the-middle attacks. Although the information you exchange is protected against eavesdropping, you do not know who you are exchanging it with. Unless you control the network, you are exposed to someone intercepting your IP transmissions, and masquerading as the endpoint.

You can create an encrypted SSL connection, without authenticating the server, by using a Diffie-Hellman key exchange CipherSpec that supports anonymous SSL. The master secret, shared between the client

and server, and used to encrypt SSL transmissions, is established without exchanging a privately signed server certificate.

Because anonymous connections are insecure, most SSL implementations do not default to using anonymous CipherSpecs. If a client request for SSL connection is accepted by a telemetry channel, the channel must have a keystore protected by a passphrase. By default, since SSL implementations do not use anonymous CipherSpecs, the keystore must contain a privately signed certificate that the client can authenticate.

If you use anonymous CipherSpecs, the server keystore must exist, but it need not contain any privately signed certificates.

Another way to establish an encrypted connection is to replace the trust provider at the client with your own implementation. Your trust provider would not authenticate the server certificate, but the connection would be encrypted.

SSL configuration of MQTT clients and telemetry channels

MQTT clients and the WebSphere MQ Telemetry (MQXR) service use Java Secure Socket Extension (JSSE) to connect telemetry channels using SSL. MQTT C clients, and the WebSphere MQ Telemetry daemon for devices do not support SSL.

Configure SSL to authenticate the telemetry channel and the MQTT client, and encrypt the transfer of messages between clients and the telemetry channel.

As an alternative to using SSL, some kinds of Virtual Private Network (VPN), such as IPsec, authenticate the endpoints of a TCP/IP connection. VPN encrypts each IP packet that flows over the network. Once such a VPN connection is established, you have established a trusted network. You can connect MQTT clients to telemetry channels using TCP/IP over the VPN network.

You can configure the connection between a Java MQTT client and a telemetry channel to use the SSL protocol over TCP/IP. What is secured depends on how you configure SSL to use JSSE. Starting with the most secured configuration, you can configure three different levels of security:

1. Permit only trusted MQTT clients to connect. Connect an MQTT client only to a trusted telemetry channel. Encrypt messages between the client and the queue manager; see “MQTT client authentication using SSL” on page 97.
2. Connect an MQTT client only to a trusted telemetry channel. Encrypt messages between the client and the queue manager; see “Telemetry channel authentication using SSL” on page 100.
3. Encrypt messages between the client and the queue manager; see “Publication privacy on telemetry channels ” on page 102.

JSSE configuration parameters

Modify JSSE parameters to alter the way an SSL connection is configured. The JSSE configuration parameters are arranged into three sets:

1. IBM WebSphere MQ Telemetry channel
2. MQTT Java client
3. JRE

Configure the telemetry channel parameters using IBM WebSphere MQ Explorer. Set the MQTT Java Client parameters in the `MqttConnectionOptions.SSLProperties` attribute. Modify JRE security parameters by editing files in the JRE security directory on both the client and server.

IBM WebSphere MQ Telemetry channel

Set all the telemetry channel SSL parameters using WebSphere MQ Explorer.

`ChannelName`

`ChannelName` is a required parameter on all channels.

The channel name identifies the channel associated with a particular port number. Name channels to help you administer sets of MQTT clients.

PortNumber

PortNumber is an optional parameter on all channels. It defaults to 1883 for TCP channels, and 8883 for SSL channels.

The TCP/IP port number associated with this channel. MQTT clients are connected to a channel by specifying the port defined for the channel. If the channel has SSL properties, the client must connect using the SSL protocol; for example:

```
MQTTClient mqttClient = new MqttClient( "ssl://www.example.org:8884", "clientId1");
mqttClient.connect();
```

KeyFileName

KeyFileName is a required parameter for SSL channels. It must be omitted for TCP channels.

KeyFileName is the path to the Java keystore containing digital certificates that you provide. Use JKS, JCEKS or PKCS12 as the type of keystore on the server.

Identify the keystore type by using one of the following file extensions:

```
.jks
.jceks
.p12
.pkcs12
```

A keystore with any other file extension is assumed to be a JKS keystore.

You can combine one type of keystore at the server with other types of keystore at the client.

Place the private certificate of the server in the keystore. The certificate is known as the server certificate. The certificate can be self-signed, or part of a certificate chain that is signed by a signing authority.

If you are using a certificate chain, place the associated certificates in the server keystore.

The server certificate, and any certificates in its certificate chain, are sent to clients to authenticate the identity of the server.

If you have set ClientAuth to Required, the keystore must contain any certificates necessary to authenticate the client. The client sends a self-signed certificate, or a certificate chain, and the client is authenticated by the first verification of this material against a certificate in the keystore. Using a certificate chain, one certificate can verify many clients, even if they are issued with different client certificates.

PassPhrase

PassPhrase is a required parameter for SSL channels. It must be omitted for TCP channels.

The passphrase is used to protect the keystore.

ClientAuth

ClientAuth is an optional SSL parameter. It defaults to no client authentication. It must be omitted for TCP channels.

Set ClientAuth if you want the telemetry (MQXR) service to authenticate the client, before permitting the client to connect to the telemetry channel.

If you set ClientAuth, the client must connect to the server using SSL, and authenticate the server. In response to setting ClientAuth, the client sends its digital certificate to the server, and any other certificates in its keystore. Its digital certificate is known as the client certificate. These certificates are authenticated against certificates held in the channel keystore, and in the JRE cacerts store.

CipherSuite

`CipherSuite` is an optional SSL parameter. It defaults to try all the enabled CipherSpecs. It must be omitted for TCP channels.

If you want to use a particular CipherSpec, set `CipherSuite` to the name of the CipherSpec that must be used to establish the SSL connection.

The telemetry service and MQTT client negotiate a common CipherSpec from all the CipherSpecs that are enabled at each end. If a specific CipherSpec is specified at either or both ends of the connection, it must match the CipherSpec at the other end.

Install additional ciphers by adding additional providers to JSSE.

Federal Information Processing Standards (FIPS)

FIPS is an optional setting. By default it is not set.

Either in the properties panel of the queue manager, or using **runmqsc**, set SSLFIPS. SSLFIPS specifies whether only FIPS-certified algorithms are to be used.

Revocation namelist

Revocation namelist is an optional setting. By default it is not set.

Either in the properties panel of the queue manager, or using **runmqsc**, set SSLCRLNL. SSLCRLNL specifies a namelist of authentication information objects which are used to provide certificate revocation locations.

No other queue manager parameters that set SSL properties are used.

MQTT Java client

Set SSL properties for the Java client in `MqttConnectionOptions.SSLProperties`; for example:

```
java.util.Properties sslClientProperties = new Properties();
sslClientProperties.setProperty("com.ibm.ssl.keyStoreType", "JKS");
com.ibm.micro.client.mqttv3.MqttConnectOptions conOptions = new MqttConnectOptions();
conOptions.setSSLProperties(sslClientProperties);
```

The names and values of specific properties are described in the API documentation for `MqttConnectOptions`. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

Protocol

`Protocol` is optional.

The protocol is selected in negotiation with the telemetry server. If you require a specific protocol you can select one. If the telemetry server does not support the protocol the connection fails.

ContextProvider

`ContextProvider` is optional.

KeyStore

`KeyStore` is optional. Configure it if `ClientAuth` is set at the server to force authentication of the client.

Place the digital certificate of the client, signed using its private key, into the keystore. Specify the keystore path and password. The type and provider are optional. JKS is the default type, and IBMJCE is the default provider.

Specify a different keystore provider to reference a class that adds a new keystore provider. Pass the name of the algorithm used by the keystore provider to instantiate the `KeyManagerFactory` by setting the key manager name.

TrustStore

`TrustStore` is optional. You can place all the certificates you trust in the JRE `cacerts` store.

Configure the truststore if you want to have a different truststore for the client. You might not configure the truststore if the server is using a certificate issued by a well known CA that already has its root certificate stored in `cacerts`.

Add the publicly signed certificate of the server or the root certificate to the truststore, and specify the truststore path and password. JKS is the default type, and IBMJCE is the default provider.

Specify a different truststore provider to reference a class that adds a new truststore provider. Pass the name of the algorithm used by the truststore provider to instantiate the `TrustManagerFactory` by setting the trust manager name.

JRE

Other aspects of Java security that affect the behavior of SSL on both the client and server are configured in the JRE. The configuration files on Windows are in *`Java Installation Directory`*`\jre\lib\security`. If you are using the JRE shipped with IBM WebSphere MQ the path is as shown in the following table:

Table 3. Filepaths by platform for JRE SSL configuration files

Platform	Filepath
Windows	*`WMQ Installation Directory`*`\java\jre\lib\security`
Linux for System x 32 bit	*`WMQ Installation Directory`*`/java/jre/lib/security`
Other UNIX and Linux platforms	*`WMQ Installation Directory`*`/java/jre64/jre/lib/security`

Well-known certificate authorities

The `cacerts` file contains the root certificates of well-known certificate authorities. The `cacerts` is used by default, unless you specify a truststore. If you use the `cacerts` store, or do not provide a truststore, you must review and edit the list of signers in `cacerts` to meet your security requirements.

You can open `cacerts` using the WebSphere MQ command `strmqikm`.which runs the IBM Key Management utility. Open `cacerts` as a JKS file, using the password `changeit`. Modify the password to secure the file.

Configuring security classes

Use the `java.security` file to register additional security providers and other default security properties.

Permissions

Use the `java.policy` file to modify the permissions granted to resources. `javaws.policy` grants permissions to `javaws.jar`

Encryption strength

Some JREs ship with reduced strength encryption. If you cannot import keys into keystores, reduced strength encryption might be the cause. Either, try starting **ikeyman** using the **strmqikm** command, or download strong, but limited jurisdiction files from IBM developer kits, Security information.

Important: Your country of origin might have restrictions on the import, possession, use, or re-export to another country, of encryption software. Before downloading or using the unrestricted policy files, you must check the laws of your country. Check its regulations, and its policies concerning the import, possession, use, and re-export of encryption software, to determine if it is permitted.

Modify the trust provider to permit the client to connect to any server

The example illustrates how to add a trust provider and reference it from the MQTT client code. The example performs no authentication of the client or server. The resulting SSL connection is encrypted without being authenticated.

The code snippet in Figure 20 on page 107 sets the `AcceptAllProviders` trust provider and trust manager for the MQTT client.

```
java.security.Security.addProvider(new AcceptAllProvider());
java.util.Properties sslClientProperties = new Properties();
sslClientProperties.setProperty("com.ibm.ssl.trustManager","TrustAllCertificates");
sslClientProperties.setProperty("com.ibm.ssl.trustStoreProvider","AcceptAllProvider");
conOptions.setSSLProperties(sslClientProperties);
```

Figure 20. MQTT Client code snippet

```
package com.ibm.mq.id;
public class AcceptAllProvider extends java.security.Provider {
    private static final long serialVersionUID = 1L;
    public AcceptAllProvider() {
        super("AcceptAllProvider", 1.0, "Trust all X509 certificates");
        put("TrustManagerFactory.TrustAllCertificates",
                AcceptAllTrustManagerFactory.class.getName());
    }
```

Figure 21. AcceptAllProvider.java

```
    protected static class AcceptAllTrustManagerFactory extends
            javax.net.ssl.TrustManagerFactorySpi {
        public AcceptAllTrustManagerFactory() {}
        protected void engineInit(java.security.KeyStore keystore) {}
        protected void engineInit(
                javax.net.ssl.ManagerFactoryParameters parameters) {}
        protected javax.net.ssl.TrustManager[] engineGetTrustManagers() {
            return new javax.net.ssl.TrustManager[] { new AcceptAllX509TrustManager() };
        }
```

Figure 22. AcceptAllTrustManagerFactory.java

```
    protected static class AcceptAllX509TrustManager implements
            javax.net.ssl.X509TrustManager {
        public void checkClientTrusted(
                java.security.cert.X509Certificate[] certificateChain,
                String authType) throws java.security.cert.CertificateException {
            report("Client authtype=" + authType);
            for (java.security.cert.X509Certificate certificate : certificateChain) {
                report("Accepting:" + certificate);
            }
        }
        public void checkServerTrusted(
                java.security.cert.X509Certificate[] certificateChain,
                String authType) throws java.security.cert.CertificateException {
            report("Server authtype=" + authType);
            for (java.security.cert.X509Certificate certificate : certificateChain) {
                report("Accepting:" + certificate);
            }
        }
        public java.security.cert.X509Certificate[] getAcceptedIssuers() {
            return new java.security.cert.X509Certificate[0];
        }
    private static void report(String string) {
           System.out.println(string);
      }
    }
```

Figure 23. AcceptAllX509TrustManager.java

Telemetry channel JAAS configuration

Configure JAAS to authenticate the Username sent by the client.

The WebSphere MQ administrator configures which MQTT channels require client authentication using JAAS. Specify the name of a JAAS configuration for each channel that is to perform JAAS authentication. Channels can all use the same JAAS configuration, or they can use different JAAS configurations. The configurations are defined in *WMQData directory*\qmgrs*qMgrName*\mqxr\jaas.config.

The jaas.config file is organized by JAAS configuration name. Under each configuration name is a list of Login configurations; see Figure 24 on page 109.

JAAS provides four standard Login modules. The standard NT and UNIX Login modules are of limited value.

JndiLoginModule
Authenticates against a directory service configured under JNDI (Java Naming and Directory Interface).

Krb5LoginModule
Authenticates using Kerberos protocols.

NTLoginModule
Authenticates using the NT security information for the current user.

UnixLoginModule
Authenticates using the UNIX security information for the current user.

The problem with using NTLoginModule or UnixLoginModule is that the telemetry (MQXR) service runs with the mqm identity, and not the identity of the MQTT channel. mqm is the identity passed to NTLoginModule or UnixLoginModule for authentication, and not the identity of the client.

To overcome this problem, write your own Login module, or use the other standard Login modules. A sample JAASLoginModule.java is supplied with WebSphere MQ Telemetry. It is an implementation of the javax.security.auth.spi.LoginModule interface. Use it to develop your own authentication method.

Any new LoginModule classes you provide must be on the class path of the telemetry (MQXR) service. Do not place your classes in WebSphere MQ directories that are in the class path. Create your own directories, and define the whole class path for the telemetry (MQXR) service.

You can augment the class path used by the telemetry (MQXR) service by setting class path in the service.env file. CLASSPATH must be capitalized, and the class path statement can only contain literals. You cannot use variables in the CLASSPATH; for example CLASSPATH=%CLASSPATH% is incorrect. The telemetry (MQXR) service sets its own classpath. The CLASSPATH defined in service.env is added to it.

The telemetry (MQXR) service provides two callbacks that return the Username and the Password for a client connected to the MQTT channel. The Username and Password are set in the MqttConnectOptions object. See Figure 25 on page 109 for an example of how to access Username and Password.

Examples

An example of a JAAS configuration file with one named configuration, MQXRConfig.

```
MQXRConfig {
  samples.JAASLoginModule required debug=true;
  //com.ibm.security.auth.module.NTLoginModule required;
  //com.ibm.security.auth.module.Krb5LoginModule required
  //                 principal=principal@your_realm
  //                 useDefaultCcache=TRUE
  //                 renewTGT=true;
  //com.sun.security.auth.module.NTLoginModule required;
  //com.sun.security.auth.module.UnixLoginModule required;
  //com.sun.security.auth.module.Krb5LoginModule required
  //                 useTicketCache="true"
  //                 ticketCache="${user.home}${/}tickets";
};
```

Figure 24. Sample `jaas.config` file

An example of a JAAS Login module coded to receive the `Username` and `Password` provided by an MQTT client.

```
  public boolean login()
      throws javax.security.auth.login.LoginException {
    javax.security.auth.callback.Callback[] callbacks =
      new javax.security.auth.callback.Callback[2];
    callbacks[0] = new javax.security.auth.callback.NameCallback("NameCallback");
    callbacks[1] = new javax.security.auth.callback.PasswordCallback(
        "PasswordCallback", false);
    try {
      callbackHandler.handle(callbacks);
      String username = ((javax.security.auth.callback.NameCallback) callbacks[0])
          .getName();
      char[] password = ((javax.security.auth.callback.PasswordCallback) callbacks[1])
          .getPassword();
      // Accept everything.
      if (true) {
        loggedIn = true;
      } else
        throw new javax.security.auth.login.FailedLoginException("Login failed");

      principal= new JAASPrincipal(username);

    } catch (java.io.IOException exception) {
      throw new javax.security.auth.login.LoginException(exception.toString());
    } catch (javax.security.auth.callback.UnsupportedCallbackException exception) {
      throw new javax.security.auth.login.LoginException(exception.toString());
    }

    return loggedIn;
  }
```

Figure 25. Sample `JAASLoginModule.Login()` method

Client programming concepts

The concepts described in this section help you to understand the Java client for version 3.1 of the MQTT protocol. The concepts complement the API documentation accompanying the package `com.ibm.micro.client.mqttv3`.

`com.ibm.micro.client.mqttv3` contains the classes that provide the public methods for the Java implementations of the MQTT version 3.1 protocol. The `com.ibm.micro.client.mqttv3` package, and the accompanying packages that implement the protocol for Java SE and ME, are provided with the installation of IBM WebSphere MQ Telemetry.

To develop and run an MQTT client you need to copy or install these packages on the client device. You do not need to install a separate client runtime.

The licensing conditions for clients are associated with the server that you are connecting the clients to.

The Java client is a reference implementation of version 3.1 of the MQTT protocol. You can implement your own clients in different languages suitable for different device platforms. Refer to MQ Telemetry Transport format and protocol for details.

The client API documentation for the package `com.ibm.micro.client.mqttv3` makes no assumptions about which server the client is connected to. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference. The behavior of the client might differ slightly when connected to different servers. The descriptions that follow describe the behavior of the client when connected to the IBM WebSphere MQ telemetry (MQXR) service.

The MQTT messaging client for JavaScript and web apps

Until recently, programming web apps and creating messaging apps have been separate disciplines. No matter where your previous experience lies, there are significant benefits in using JavaScript and messaging together. When you code your messaging app as a web app, it can be pulled in and run on any up-to-date browser. If you change the app, the latest version is pulled in whenever the browser is refreshed. The browser also looks after security, and the reliable transmission of messages.

How using a web app eases application deployment

If you have experience of developing and deploying traditional messaging apps on (for example) IBM WebSphere MQ, you might be familiar with the following deployment process:

1. The system administrator installs or embeds the client library.
2. The system administrator arranges for the messaging app to be distributed to the end users and installed on their local systems.
3. When the code changes, the system administrator repeats the previous steps (so change management is complex).

If you code your messaging app as a web app, this is the deployment process:

1. The system administrator serves the web app and the client library at a URL.
2. The end user's browser pulls in the web app and client library together.
3. When the code changes, the updated version is picked up when the browser is refreshed (so change management is simple).

Why you might want to use messaging directly from the browser in your web apps

If you have experience of programming apps in JavaScript, you might be interested to know the benefits provided by messaging systems such as IBM WebSphere MQ:

- If you send and receive messages through a messaging system, that system is responsible for ensuring that the messages are delivered.
- Because the messaging system looks after delivery, your web app can "fire and forget". This greatly simplifies the programming logic. If messages are delivered for you, your code need not check they got there. Your app no longer needs to handle receipt acknowledgement, or save undelivered messages and retry them later.
- Messaging systems provide event-driven messaging. Your client app no longer needs to send a request then continuously poll for a response. Instead, the messaging server sends a message to your client app when an interesting event occurs. This also means that your client app is alerted as soon as the event happens, rather than waiting until the next time the app polls the server.
- Event-driven messaging also massively reduces the load on the device hosting your client app, the network traffic between the browser and the messaging server, and the load on the messaging server. This matters increasingly, as more and more systems are running on mobile devices, and connecting across wireless networks.

How the pieces fit together

The MQTT messaging client for JavaScript includes a client library, and an example web app that uses the library. You code your own web app that uses the library. The web app and client library are then made available at your chosen URL, for example by an MQ queue manager (as in the following diagram) or by an application server. The browser pulls in the web app and client library, and the web app then uses the browser to connect to, and exchange messages with, an MQTT server such as IBM WebSphere MQ Telemetry or IBM MessageSight.

These are the flows:

1. Each instance of the browser refreshes its connection to the URL on which the web app is available, and an up-to-date version of the web app and client library is loaded onto the browser.
2. The web app connects to a queue manager, using MQTT over the WebSocket protocol, and subscribes to a topic of interest.
3. The queue manager uses the same connection to send messages that match the subscription back to the web app.

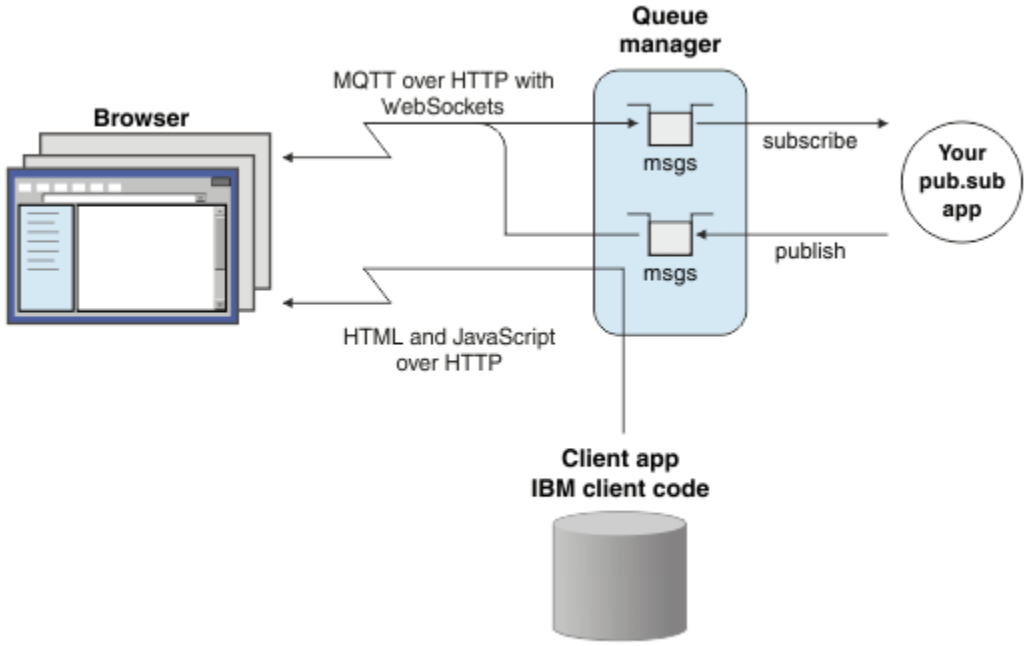


Figure 26. Using the MQTT messaging client for JavaScript with publish and subscribe messaging

The web app contains application logic, and the URL of the MQTT server. When opened in a browser, the app connects to the MQTT server, creates the subscriptions it needs, then waits to receive event-driven alerts and act on them.

The web app connects using MQTT as the transport protocol, running over WebSockets. Most modern browsers can make WebSockets connections. By using WebSockets, the web app can pass messages through firewalls that accept HTTP and the WebSocket protocol, and can send packets of data (known as "frames") just like using TCP over IP.

When a message sent by the web app arrives in the MQTT server, the server-side app just sees it as a message. It does not know the message has come from a browser.

Administering and controlling an MQTT server

The MQTT server handles the server-side complexity of the messaging. It ensures delivery of the messages it receives from the web app, and it hosts the publish and subscribe application that responds to the web app. For any MQTT server, you need to complete the following steps:

- Create a server.
- Pick a port.
- Define a new MQTT channel.
- Configure your client web app to connect to the chosen port across the new MQTT channel.

You also need to serve the web app executable JavaScript to the browser. If you are using IBM WebSphere MQ Telemetry, by default the MQTT server does this for you, using the same MQTT channel that the web app uses to connect to the MQTT server. If you are trying out MQTT, this can help you get up and running quickly. For production use, particularly in high throughput environments, you might prefer to serve the web app executable JavaScript on a separate channel, using a dedicated application server such as WebSphere Application Server.

Note: Because it is designed for high throughput environments, IBM MessageSight expects you to do this.

For example, if you are using IBM WebSphere MQ Telemetry, you use the IBM WebSphere MQ Explorer **New Telemetry Channel** wizard to complete the following steps:

1. Create a server.
2. Pick a port (1883 by default).
3. Define a new MQTT channel.
4. Configure your client web app to connect to the chosen port across the new MQTT channel.

The web app executable JavaScript is (optionally) also served through the queue manager on the same channel. To do this, the queue manager must support both MQTT and HTTP. If you already have a queue manager that is configured for MQTT, you can use the MQSC command line tool to alter the protocol in the channel definition to support both MQTT and HTTP. See ALTER CHANNEL.

The web app and the MQTT messaging client for JavaScript client library are stored on disk in a structure that is defined by your application server or queue manager. If you are using IBM WebSphere MQ Telemetry, the web app and the client library are stored in the following directory structure:

```
MQINSTALL
  |
  mqxr
  |  |
  |  SDK
  |    |
  |    WebContent
  |       sample web app (the "Web Messaging Utility" sample HTML pages)
  |       |
  |       WebSocket
  |          IBM client library (the messaging client JavaScript classes)
  |
  qmgrs
     |
     qmgr_name
        |
        mqxr
           |
           WebContent
              |
              your_client_app (your own JavaScript pages)
```

The sample web app and client library are stored in the *MQINSTALL*/mqxr/SDK/WebContent directory. Material in this directory is served by all queue managers. If you do not want your users to see and use all this material, you should create your own version of the app. To make this app, or your own replacement app, available on specific queue managers, you put the app in the *MQINSTALL*/qmgrs/*qmgr_name*/mqxr/WebContent directory. To select the app and associated JavaScript classes to serve at a URL, the queue manager looks first in its own WebContent directory, then in the global WebContent directory. In the previous example directory tree, the queue manager serves *your_client_app* and the global copy of the JavaScript classes.

To stop the queue manager serving the web app executable files, or modify where the queue manager looks for the executable files, you configure the **webcontentpath** property and add it to the mqxr.properties file. See MQXR properties.

Related concepts

“How to program messaging apps in JavaScript” on page 113

Related tasks

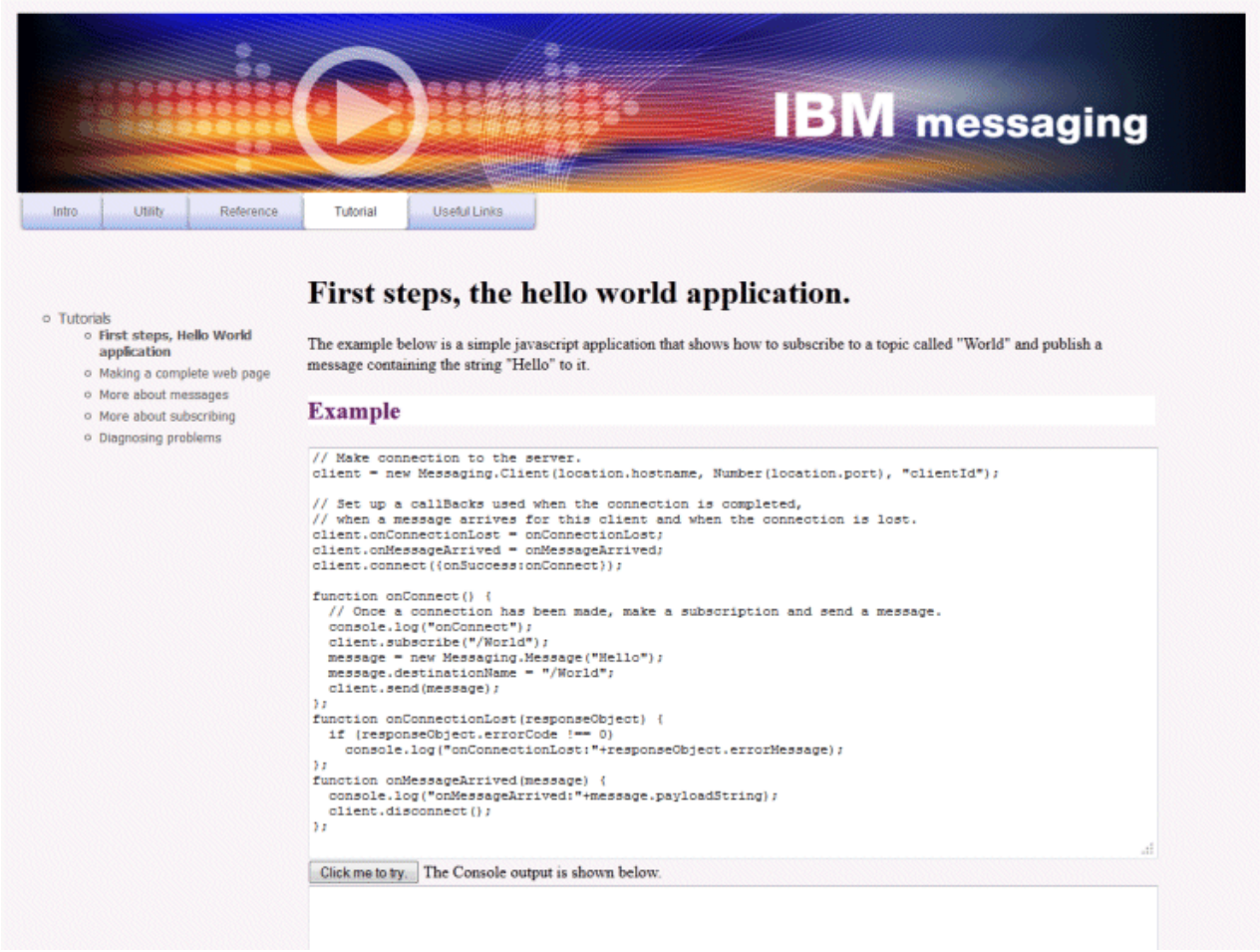
“Connecting the MQTT messaging client for JavaScript over SSL and WebSockets” on page 72
Connect your web app securely to IBM WebSphere MQ by using the MQTT messaging client for JavaScript sample HTML pages with SSL and the WebSocket protocol.

“Getting started with the MQTT messaging client for JavaScript” on page 21
You can get started with the MQTT messaging client for JavaScript by displaying the messaging client sample home page, and browsing the resources to which it links. To display this home page, you configure an MQTT server to accept connections from the MQTT messaging client sample JavaScript pages, then you type the URL that you have configured on the server into a web browser. The MQTT messaging client for JavaScript automatically starts on your device, and the messaging client sample home page is displayed. This page contains links to utilities, programming interface documentation, a tutorial, and other useful information.

How to program messaging apps in JavaScript

The MQTT messaging client for JavaScript includes a tutorial that demonstrates how to create a simple publish and subscribe web app. By exploring the "First steps, Hello world" application code, you can get a basic understanding of the mechanics of programming web apps for messaging.

If your experience to date has been mainly in developing and deploying traditional messaging applications, you might also find the “JavaScript coding tips” on page 114 section helpful. If you are an experienced JavaScript developer who is new to messaging, you'll find a brief introduction to key messaging concepts in the “Messaging basics” on page 116 section.



JavaScript coding tips

If you are used to developing messaging applications, but new to web apps, you might find the following tips helpful:

Wrapping the code for each event in an onSuccess callback

When you code a messaging application, you code the following events in the following order:

1. connect
2. subscribe
3. publish
4. receive message

The MQTT messaging client for JavaScript API is fully asynchronous, which means that your application thread does not block while waiting for calls like connect or subscribe to take effect. Instead these calls signal their completion by calling an `onSuccess` or `onFailure` callback. To be sure that each event has completed before the next event is triggered, you need to wrap the code for each event in an `onSuccess` callback. For example, the JavaScript application might return from making the connect call before the connection has been created. To be sure that connection has happened before you subscribe, you need to put the subscribe code in an `onSuccess` callback for the connect.

The "First steps, Hello world" application code uses this approach.

Embedding the application code within HTML mark-up

Here is an example JavaScript page:

Here is the source for the previous page, to show how the application code is embedded within HTML mark-up:

```
<!DOCTYPE html>


<head>
  <script type="text/javascript" src="../WebSocket/mqttws31.js"></script>

  <script type="text/javascript">

  var client;
  var form = document.getElementById("tutorial");

  function doConnect() {
      client = new Messaging.Client("whistler1.hursley.ibm.com", 1883, "ClientId");
      client.onConnect = onConnect;
      client.onMessageArrived = onMessageArrived;
      client.onConnectionLost = onConnectionLost;
      client.connect({onSuccess:onConnect});
```

```
  }

  function doSubscribe() {
      client.subscribe("/World");
  }

  function doSend() {
      message = new Messaging.Message("Hello");
      message.destinationName = "/World";
      client.send(message);
  }

  function doDisconnect() {
      client.disconnect();
  }

  // Web Messaging API callbacks

  function onConnect() {
      var form = document.getElementById("example");
      form.connected.checked= true;
  }

  function onConnectionLost(responseObject) {
      var form = document.getElementById("example");
      form.connected.checked= false;
      if (responseObject.errorCode !== 0)
          alert(client.clientId+"\n"+responseObject.errorCode);
  }

  function onMessageArrived(message) {
      var form = document.getElementById("example");
      form.receiveMsg.value = message.payloadString;
  }

  </script>
</head>

<body>
  <h1>Example Web Messaging web page.</h1>
  <form id="example">
  <fieldset>
  <legend id="Connect" > Connect </legend>
    Make a connection to the server, and set up a call back used if a
    message arrives for this client.
    <br>
    <input type="button" value="Connect" onClick="doConnect(this.form)" name="Connect"/>
    <input type="checkbox" name="connected" disabled="disabled"/>
  </fieldset>

  <fieldset>
  <legend id="Subscribe" > Subscribe </legend>
  Make a subscription to topic "/World".
  <br> <input type="button" value="Subscribe" onClick="doSubscribe(this.form)"/>
  </fieldset>

  <fieldset>
  <legend id="Send" > Send </legend>
    Create a Message object containing the word "Hello" and then publish it at
    the server.
    <br>
    <input type="button" value="Send" onClick="doSend(this.form)"/>
  </fieldset>

  <fieldset>
  <legend id="Receive" > Receive </legend>
    A copy of the published Message is received in the callback we created earlier.
  <textarea name="receiveMsg" rows="1" cols="40" disabled="disabled"></textarea>
  </fieldset>

  <fieldset>
  <legend id="Disconnect" > Disconnect </legend>
    Now disconnect this client from the server.
  <br> <input type="button" value="Disconnect" onClick="doDisconnect()"/>
  </fieldset>
  </form>
</body>
</html>
```

Messaging basics

Here is some background messaging information for web app developers who are new to messaging:

Asynchronous and fire-and-forget messaging.

The MQTT protocol supports assured delivery and fire-and-forget transfers. In the protocol, message delivery is asynchronous: the app passes the message to the client API, and takes no further action to ensure that the message is delivered. This is approach is known as *fire-and-forget*. When a response is available, it is automatically sent to the app.

Asynchronous delivery frees up the app from any server connection, and from waiting for messages. The interaction model is like email, but optimized for application programming.

See also the "MQTT protocol" section of “Introduction to MQTT” on page 5

An overview of publish and subscribe messaging.

The provider of information is called a *publisher*. A publisher supplies information about a subject, without needing to know anything about the applications that are interested in that information. A publisher chooses a *topic*, which is a container for messages on a specific subject. The publisher then generates each piece of information for that subject as a message, called a *publication*, and posts it to the associated topic.

The consumer of information is called a *subscriber*. A subscriber creates a *subscription* to a topic that it is interested in. When a new message is posted to topic, the message is forwarded to all subscribers to the topic. Subscribers can make multiple subscriptions and can receive information from many different publishers.

See also Introduction to IBM WebSphere MQ publish/subscribe messaging

How subscriptions and topics match up.

If you are using IBM WebSphere MQ as your MQTT server, you need to understand how IBM WebSphere MQ specifies topics. In IBM WebSphere MQ, a publisher creates a message, and publishes it with a topic string that best fits the subject of the publication. To receive publications, a subscriber creates a subscription with a pattern matching topic string to select publication topics. The queue manager delivers publications to subscribers that have subscriptions that match the publication topic, and are authorized to receive the publications.

Typically subjects are organized hierarchically, into topic trees, using the '/' character to create subtopics in the topic string. Topics are nodes in the topic tree. Topics can be leaf-nodes with no further subtopics, or intermediate nodes with subtopics. Subscribers can use wildcards to subscribe to more than one topic at a time. For example, a subscription to `/sport/tennis` only gets messages posted to the tennis subtopic, whereas a subscription to `/sport/#` gets messages posted to any subtopic of `/sport`.

See also Topics, Topic trees, and Wildcard schemes.

Related concepts

“The MQTT messaging client for JavaScript and web apps” on page 110

Related tasks

“Connecting the MQTT messaging client for JavaScript over SSL and WebSockets” on page 72
Connect your web app securely to IBM WebSphere MQ by using the MQTT messaging client for JavaScript sample HTML pages with SSL and the WebSocket protocol.

“Getting started with the MQTT messaging client for JavaScript” on page 21
You can get started with the MQTT messaging client for JavaScript by displaying the messaging client sample home page, and browsing the resources to which it links. To display this home page, you configure an MQTT server to accept connections from the MQTT messaging client sample JavaScript pages, then you type the URL that you have configured on the server into a web browser. The MQTT messaging client for JavaScript automatically starts on your device, and the messaging client sample home page is displayed. This page contains links to utilities, programming interface documentation, a tutorial, and other useful information.

Callbacks and synchronization in MQTT client apps

The MQTT client programming model uses threads extensively. The threads decouple an MQTT client app, as much as they can, from delays in transmitting messages to and from the server. Publications, delivery tokens, and connection lost events are delivered to the methods in a callback class that implements `MqttCallback`.

Callbacks

The `MqttCallback` interface has three callback methods; see an example implementation in Callback.java.

`connectionLost(java.lang.Throwable cause)`

`connectionLost` is called when a communications error leads to the connection dropping. It is also called if the server drops the connection as a result of an error on the server after the connection has been established. Server errors are logged to the queue manager error log. The server drops the connection to the client, and the client calls `MqttCallback.connectionLost`.

The only remote errors thrown as exceptions on the same thread as the client app are exceptions from `MqttClient.connect`. Errors detected by the server after the connection is established are reported back to the `MqttCallback.connectionLost` callback method as `throwables`.

Typical server errors that result in `connectionLost` are authorization errors. For example, the telemetry server tries to publish on a topic on behalf of a client that is not authorized to publish on the topic. Anything that results in a `MQCC_FAIL` condition code being returned to the telemetry server can result in the connection being dropped.

`deliveryComplete(MqttDeliveryToken token)`

`deliveryComplete` is called by the MQTT client to pass a delivery token back to the client app; see "Delivery tokens" on page 120. Using the delivery token, the callback can access the published message with the method `token.getMessage`.

When the application callback returns control to the MQTT client after being called by the `deliveryComplete` method, delivery is completed. Until delivery is completed, messages with QoS 1 or 2 are retained by the persistence class.

The call to `deliveryComplete` is a point of synchronization between the application and the persistence class. The `deliveryComplete` method is never called twice for the same message.

When the application callback returns from `deliveryComplete` to the MQTT client, the client calls `MqttClientPersistence.remove` for messages with QoS 1 or 2. `MqttClientPersistence.remove` deletes the locally stored copy of the published message.

From a transaction processing perspective, the call to `deliveryComplete` is a single phase transaction that commits the delivery. If processing fails during the callback, on restart of the client `MqttClientPersistence.remove` is called again to delete the local copy of the published message. The callback is not called again. If you are using the callback to store a log of delivered messages, you cannot synchronize the log with the MQTT client. If you want to store a log reliably, then update the log in the `MqttClientPersistence` class.

The delivery token and message are referenced by the main application thread and the MQTT client. The MQTT client dereferences the `MqttMessage` object when delivery is completed, and the delivery token object when the client disconnects. The `MqttMessage` object can be garbage collected after delivery is completed if the client app dereferences it. The delivery token can be garbage collected after the session is disconnected.

You can get `MqttDeliveryToken` and `MqttMessage` attributes after a message has been published. If you attempt to set any `MqttMessage` attributes after the message has been published the result is undefined.

The MQTT client continues to process delivery acknowledgments if the client reconnects to the previous session with the same `ClientIdentifier`; see "Clean sessions" on page 119. The MQTT client app must set `MqttClient.CleanSession` to `false` for the previous session, and set it to `false` in the new session. The MQTT client creates new delivery tokens and message objects in the new session for pending deliveries. It recovers the objects using the `MqttClientPersistence` class. If the application client still has references to the old delivery

tokens and messages, dereference them. The application callback is called in the new session for any deliveries initiated in the previous session and completed in this session.

The application callback is called after the application client connects, when a pending delivery is completed. Before the application client connects, it can retrieve pending deliveries using the `MqttClient.getPendingDeliveryTokens` method.

Notice that the client app originally created the message object that is published, and its payload byte array. The MQTT client references these objects. The message object returned by the delivery token in the method `token.getMessage` is not necessarily the same message object created by the client. If a new MQTT client instance recreates the delivery token, the `MqttClientPersistence` class recreates the `MqttMessage` object. For consistency `token.getMessage` returns `null` if `token.isCompleted` is `true`, regardless of whether the message object was created by the application client or the `MqttClientPersistence` class.

`messageArrived(MqttTopic topic, MqttMessage message)`

`messageArrived` is called when a publication arrives for the client that matched a subscription topic. `topic` is the publication topic, not the subscription filter. The two can be different if the filter contains wildcards.

If the topic matches multiple subscriptions created by the client, the client receives multiple copies of the publication. If a client publishes to a topic that it also subscribes to, it receives a copy of its own publication.

If a message is sent with a QoS of 1 or 2, the message is stored by the `MqttClientPersistence` class before the MQTT client calls `messageArrived`. `messageArrived` behaves like `deliveryComplete`: it is only called once for a publication, and the local copy of the publication is removed by `MqttClientPersistence.remove` when `messageArrived` returns to the MQTT client. The MQTT client drops its references to the topic and message when `messageArrived` returns to the MQTT client. The topic and message objects are garbage collected, if the application client has not held onto a reference to the objects.

Callbacks, threading, and client app synchronization

The MQTT client calls a callback method on a separate thread to the main application thread. The client app does not create a thread for the callback, it is created by the MQTT client.

The MQTT client synchronizes callback methods. Only one instance of the callback method runs at a time. The synchronization makes it easy to update an object that tallies which publications have been delivered. One instance of the `MqttCallback.deliveryComplete` runs at a time, and so it is safe to update the tally without further synchronization. It is also the case that only one publication arrives at a time. Your code in the `messageArrived` method can update an object without synchronizing it. If you are referring to the tally, or the object that is being updated, in another thread, synchronize the tally or object.

The delivery token provides a synchronization mechanism between the main application thread and delivery of a publication. The method `token.waitForCompletion` waits until delivery of a specific publication is completed, or until an optional timeout expires. You might use `token.waitForCompletion` in a couple of simple ways to process one publication at a time:

1. To pause the application client until delivery of the publication is completed; see PubSync.java.
2. To synchronize with the `MqttCallback.deliveryComplete` method. Only when `MqttCallback.deliveryComplete` returns to the MQTT Client does `token.waitForCompletion` resume. Using this mechanism you can synchronize running code in `MqttCallback.deliveryComplete` before code runs in the main application thread.

What if you wanted to publish without waiting for each publication to be delivered, but want confirmation when all the publications have been delivered? If you publish on a single thread, the last publication to be sent is also the last to be delivered.

Synchronization of requests sent to the server

Table 4. Synchronization behavior of methods that result in requests to the server.

This table lists the methods in the MQTT Java client that send a request to the server. For each method, the table describes the conditions under which the method either waits or returns, and how long the method waits.

Method	Synchronization	Timeout interval
`MqttClient.Connect`	Waits for a connection to be established with the server.	30 seconds by default, or as set by a parameter.
`MqttClient.Disconnect`	Waits for the MQTT client to finish any work it must do, and for the TCP/IP session to disconnect.	30 seconds by default, or as set by a parameter.
`MqttClient.Subscribe`	Waits for the subscribe request to complete.	30 seconds by default, or as set by a parameter.
`MqttClient.UnSubscribe`	Waits for the unsubscribe request to complete.	30 seconds by default, or as set by a parameter.
`MqttClient.Publish`	Returns immediately to the application thread after passing the request to the MQTT client.	None.
`MqttDeliveryToken.waitForCompletion`	Waits for the delivery token to be returned.	Indefinite by default, or as set by a parameter.

Clean sessions

The MQTT client, and the telemetry (MQXR) service maintain session state information. The state information is used to ensure "at least once" and "exactly once" delivery, and "exactly once" receipt of publications. Session state also includes subscriptions created by an MQTT client. You can choose to run an MQTT client with or without maintaining state information between sessions. Change the clean session mode by setting `MqttConnectOptions.cleanSession` before connecting.

When you connect an MQTT client app using the `MqttClient.connect` method, 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. 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.

Note: The WebSphere MQ Administrator can forcibly close an open session and delete all the session information. If the client reopens the session with `cleanSession=false`, a new session is started.

Publications

If you use the default `MqttConnectOptions`, or set `MqttConnectOptions.cleanSession` to `true` before connecting the client, all pending publication deliveries for the client are removed when the client connects.

The clean session setting has no effect on publications sent with `QoS=0`. For `QoS=1` and `QoS=2`, using `cleanSession=true` might result in losing a publication.

Subscriptions

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.

Another way of understanding the way the `cleanSession` attribute affects subscriptions is to think of it as a modal attribute. In its default mode, `cleanSession=true`, the client creates subscriptions and receives publications only within the scope of the session. In the alternative mode, `cleanSession=false`, subscriptions are durable. The client can connect and disconnect and its subscriptions remain active. When the client reconnects, it receives any undelivered publications. While it is connected, it can modify the set of subscriptions that are active on its behalf.

You must set the `cleanSession` mode before connecting; the mode lasts for the whole session. To change its setting, you must 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.

Client identifier

The client identifier is a 23 byte string that identifies an MQTT client. The client identifier must be unique across all clients that connect to the server, and must not be the same as the queue manager name on the server. Within these constraints, you are able to use any identification string. It is important to have a procedure for allocating client identifiers, and a means of configuring a client with its chosen identifier.

The client identifier is used in the administration of an MQTT system. With potentially hundreds of thousands of clients to administer, you need to be able to identify a particular client rapidly. Suppose for example, a device has malfunctioned, and you are notified, perhaps by a customer ringing a help desk. How does the customer identify the device, and how do you correlate that identification with the server that is typically connected to the client? Do you have to consult a database that maps each device to a client identifier and to a server? Does the name of the device identify which server it is attached to? When you browse through MQTT client connections, each connection is labeled with the client identifier. Do you need to look up a table to map a client identifier to a physical device?

Does the client identifier identify a particular device, a user, or an application running at the client? If a customer replaces a faulty device with a new one, does the new device have the same identifier as the old device? Do you allocate a new identifier? If you change a physical device, but keep the same identifier, outstanding publications and active subscriptions are automatically transferred to the new device.

How do you ensure that client identifiers are unique? As well as a system for generating unique identifiers, you must have a reliable process for setting the identifier on the client. Perhaps the client device is a "black-box", with no user interface. Do you manufacture the device with a client identifier - such as using its MAC address? Or do you have a software installation and configuration process that configures the device before it is activated?

You might create a client identifier from the 48 bit device MAC address, to keep the identifier short and unique. If transmission size is not a critical issue, you might use the remaining 17 bytes to make the address easier to administer.

Delivery tokens

When a client publishes on a topic a new delivery token is created. Use the delivery token to monitor the delivery of a publication, or to block the client app until delivery is complete.

The token is an `MqttDeliveryToken` object. It is created by calling the MqttTopic.publish() method and is retained by the MQTT client until the client session is disconnected and the delivery is completed.

The normal use of the token is to check whether delivery is complete. Block the client app until delivery is complete by using the returned token to call `token.waitForCompletion`. Alternatively, provide a

MqttCallBack handler. When the MQTT client has received all the acknowledgments it expects as part of delivering the publication, it calls MqttCallBack.deliveryComplete passing the delivery token as a parameter.

Until delivery is complete, you can inspect the publication using the returned delivery token by calling token.getMessage.

Completed deliveries

The completion of deliveries is asynchronous and depends on the quality of service associated with the publication.

At most once

QoS=0
Delivery is complete immediately on return from MqttTopic.publish. MqttCallback.deliveryComplete is called immediately.

At least once

QoS=1
Delivery is complete when an acknowledgment to the publication has been received from the queue manager. MqttCallback.deliveryComplete is called when the acknowledgment is received. The message might be delivered more than once before MqttCallback.deliveryComplete is called, if communications are slow or unreliable.

Exactly once

QoS=2
Delivery is complete when the client receives a completion message that the publication has been published to subscribers. MqttCallback.deliveryComplete is called as soon as the publication message is received. It does not wait for the completion message.

In rare circumstances, your client app might not return to the MQTT client from MqttCallback.deliveryComplete normally. You know that delivery has completed, because the MqttCallback.deliveryComplete was called. If the client restarts the same session, MqttCallback.deliveryComplete does not get called again.

Incomplete deliveries

If the delivery is not complete after the client session is disconnected you can connect the client again and complete the delivery. You can only complete the delivery of a message if the message was published in a session with the MqttConnectionOptions attribute set to false.

Create the client using the same client identifier and server address, and then connect, setting the cleanSession MqttConnectionOptions attribute to false again. If you set cleanSession to true, pending delivery tokens are thrown away.

You can check if there are any pending deliveries by calling MqttClient.getPendingDeliveryTokens. You can call MqttClient.getPendingDeliveryTokens before connecting the client.

Last will and testament publication

If an MQTT client connection unexpectedly ends, you can configure WebSphere MQ Telemetry to send a "last will and testament" publication. Predefine the content of the publication, and the topic to send it to. The "last will and testament" is a connection property. Create it before connecting the client.

Create a topic for the last will and testament. You might create a topic such as MQTTManagement/Connections/*server URI*/*client identifer*/Lost.

Set up a "last will and testament" using the MqttConnectionOptions.setWill(MqttTopic lastWillTopic, byte [] lastWillPayload, int lastWillQos, boolean lastWillRetained) method.

Consider creating a time stamp in the `lastWillPayload` message. Include other client information that assists in identifying the client and the circumstances of the connection. Pass the `MqttConnectionOptions` object to the `MqttClient` constructor.

Set `lastWillQos` to 1 or 2, to make the message persistent in IBM WebSphere MQ, and to ensure delivery. To retain the last lost connection information, set the `lastWillRetained` to `true`.

The "last will and testament" publication is sent to subscribers if the connection ends unexpectedly. It is sent if the connection ends without the client calling the `MqttClient.disconnect` method.

To monitor connections, complement the "last will and testament" publication with other publications to record connections and programmed disconnections.

Message persistence in MQTT clients

Publication messages are made persistent if they are sent with a quality of service of "at least once", or "exactly once". You can implement your own persistence mechanism on the client, or use the default persistence mechanism that is provided with the client. Persistence works in both directions, for publications that are sent to or from the client.

In MQTT, message persistence has two aspects; how the message is transferred, and whether it is queued in the MQTT server as a persistent message.

1. The MQTT client couples message persistence with quality of service. Depending on what quality of service you choose for a message, the message is made persistent. Message persistence is necessary to implement the required quality of service.

   If you specify "at most once", `QoS=0`, the client discards the message as soon as it is published. If there is any failure in the upstream processing of the message, the message is not sent again. Even if the client remains active the message is not sent again. The behavior of `QoS=0` messages is the same as IBM WebSphere MQ fast nonpersistent messages.

   If a message is published by a client with `QoS` of 1 or 2, it is made persistent. The message is stored locally, and only discarded from the client when it is no longer needed to ensure "at least once", `QoS=1`, or "exactly once", `QoS=2`, delivery.
2. If a message is marked as `QoS` 1 or 2, it is queued as a persistent message. If it is marked as `QoS=0`, then it is queued as a nonpersistent message. In IBM WebSphere MQ nonpersistent messages are transferred between queue managers "exactly once", unless the message channel has the `NPMSPEED` attribute that is set to FAST.

A persistent publication is stored on the client until it is received by a client app. For `QoS=2`, the publication is discarded from the client when the application callback returns control. For `QoS=1` the application might receive the publication again, if a failure occurs. For `QoS=0`, the callback receives the publication no more than once. It might not receive the publication if there is a failure, or if the client is disconnected at the time of publication.

When you subscribe to a topic, you can reduce the `QoS` with which the subscriber receives messages to match its persistence capabilities. Publications that are created at a higher `QoS` are sent with the highest `QoS` the subscriber requested.

Storing messages

The implementation of data storage on small devices varies a great deal. The model of temporarily saving persistent messages in storage that is managed by the MQTT client might be too slow, or demand too much storage. In mobile devices, the mobile operating system might provide a storage service that is ideal for MQTT messages.

To provide flexibility in meeting the constraints of small devices, the MQTT client has two persistence interfaces. The interfaces define the operations that are involved in storing persistent messages. The interfaces are described in the API documentation for the MQTT client for Java. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference. You can implement the interfaces to suit a device. The MQTT client that runs on Java SE has a default implementation of the

interfaces that store persistent messages in the file system. It uses the `java.io` package. The client also has a default implementation for Java ME, `MqttDefaultMIDPPersistence`.

Persistence classes

`MqttClientPersistence`
Pass an instance of your implementation of `MqttClientPersistence` to the MQTT client as a parameter of the `MqttClient` constructor. If you omit the `MqttClientPersistence` parameter from the `MqttClient` constructor, the MQTT client stores persistent messages using the class `MqttDefaultFilePersistence` or `MqttDefaultMIDPPersistence`.

`MqttPersistable`
`MqttClientPersistence` gets and puts `MqttPersistable` objects using a storage key. You must provide an implementation of `MqttPersistable` as well as the implementation of `MqttClientPersistence` if you are not using the `MqttDefaultFilePersistence` or `MqttDefaultMIDPPersistence`.

`MqttDefaultFilePersistence`

The MQTT client provides the `MqttDefaultFilePersistence` class. If you instantiate `MqttDefaultFilePersistence` in your client app, you can provide the directory to store persistent messages as a parameter of the `MqttDefaultFilePersistence` constructor.

Alternatively, the MQTT client can instantiate `MqttDefaultFilePersistence` and place files in a default directory. The name of the directory is *`client identifier`*`-tcp `*`hostname portnumber`*. `"\", "\\", "/", ":"` and `" "` are removed from the directory name string.

The path to the directory is the value of the system property `rcp.data`. If `rcp.data` is not set, the path is the value of the system property `usr.data`.

`rcp.data` is a property associated with installation of an OSGi or Eclipse Rich Client Platform (RCP).

`usr.data` is the directory in which the Java command that started the application was launched.

`MqttDefaultMIDPPersistence`
`MqttDefaultMIDPPersistence` has a default constructor and no parameters. It uses the `javax.microedition.rms.RecordStore` package to store messages.

Publications

Publications are instances of `MqttMessage` that are associated with a topic string. MQTT client can create publications to send to IBM WebSphere MQ, and subscribe to topics on IBM WebSphere MQ MQ to receive publications.

An `MqttMessage` has a byte array as its payload. Aim to keep messages as small as possible. The maximum length of message permitted by the MQTT protocol is 250 MB.

Typically, an MQTT client program uses `java.lang.String` or `java.lang.StringBuffer` to manipulate message contents. For convenience, the `MqttMessage` class has a `toString` method to convert its payload to a string. To create the byte array payload from a `java.lang.String` or `java.lang.StringBuffer`, use the `getBytes` method.

The `getBytes` method converts a string to the default character set for the platform. The default character set is generally `UTF-8`. MQTT publications that contain only text are usually encoded in `UTF-8`. Use the method `getBytes("UTF8")` to override the default character set.

In IBM WebSphere MQ, an MQTT publication is received as a `jms-bytes` message. The message includes an `MQRFH2` folder containing an `<mqtt>`, and an `<mqps>` folder. The `<mqtt>` folder contains the `clientId` and `qos`, but this content might change in the future.

An `MqttMessage` has three additional attributes: quality of service, whether it is retained, and whether it is a duplicate. The duplicate flag is only set if the quality of service is "at least once" or "exactly once". If the message was sent previously, and not acknowledged quickly enough by the MQTT client, the message is sent again, with the duplicate attribute set to `true`.

Publishing

To create a publication in an MQTT client app, create an `MqttMessage`. Set its payload, quality of service and whether it is retained, and call the `MqttTopic.publish(MqttMessage message)` method; `MqttDeliveryToken` is returned and the completion of the publication is asynchronous.

Alternatively, the MQTT client can create a temporary message object for you from the parameters on the `MqttTopic.publish(byte [] payload, int qos, boolean retained)` method when it creates a publication.

If the publication has an "at least once" or an "exactly once" quality of service, `QoS=1` or `QoS=2`, the MQTT client calls the `MqttClientPersistence` interface. It calls `MqttClientPersistence` to store the message before returning a delivery token to the application.

The application can choose to block until the message is delivered to the server, using the `MqttDeliveryToken.waitForCompletion` method. Alternatively, the application can continue without blocking. If you want to check if publications are delivered, without blocking, register an instance of a callback class that implements `MqttCallback` with the MQTT client. The MQTT client calls the `MqttCallback.deliveryComplete` method as soon as the publication has been delivered. Depending on the quality of service, the delivery might be almost immediate for `QoS=0`, or it might take some time for `QoS=2`.

Use the `MqttDeliveryToken.isComplete` method to poll if delivery is complete. While the value of `MqttDeliveryToken.isComplete` is `false`, you can call `MqttDeliveryToken.getMessage` to get the message contents. If the result of calling `MqttDeliveryToken.isComplete` is `true`, the message has been discarded and calling `MqttDeliveryToken.getMessage` would throw a null pointer exception. There is no built-in synchronization between `MqttDeliveryToken.getMessage` and `MqttDeliveryToken.isComplete`.

If the client disconnects before receiving all the pending delivery tokens, a new instance of the client can query pending delivery tokens before connecting. Until the client connects, no new deliveries are completed, and it is safe to call `MqttDeliveryToken.getMessage`. Use the `MqttDeliveryToken.getMessage` method to find out which publications have not been delivered. Pending delivery tokens are discarded if you connect with `MqttConnectOptions.cleanSession` set to its default value, `true`.

Subscribing

A queue manager or IBM MessageSight is responsible for creating publications to send to an MQTT subscriber. The queue manager checks if the topic filter in a subscription created by an MQTT client matches the topic string in a publication. The match can either be an exact match, or the match can include wildcards. Before the publication is forwarded to the subscriber by the queue manager, the queue manager checks the topic attributes associated with the publication. It follows the search procedure described in Subscribing using a topic string that contains wildcard characters to identify if an administrative topic object grants the user authority to subscribe.

When the MQTT client receives a publication with "at least once" quality of service, it calls the `MqttCallback.messageArrived` method to process the publication. If the quality of service of the publication is "exactly once", `QoS=2`, the MQTT client calls the `MqttClientPersistence` interface to store the message when it is received. It then calls `MqttCallback.messageArrived`.

Qualities of service provided by an MQTT client

An MQTT client provides three qualities of service for delivering publications to WebSphere MQ and to the MQTT client: "at most once", "at least once" and "exactly once". When an MQTT client sends a request to IBM WebSphere MQ to create a subscription, the request is sent with the "at least once" quality of service.

The quality of service of a publication is an attribute of `MqttMessage`. It is set by the method `MqttMessage.setQos`.

The method `MqttClient.subscribe` can reduce the quality of service applied to publications sent to a client on a topic. The quality of service of a publication forwarded to a subscriber might be different to the quality of service of the publication. The lower of the two values is used to forward a publication.

At most once

`QoS=0`

The message is delivered at most once, or it is not delivered at all. Its delivery across the network is not acknowledged.

The message is not stored. The message might be lost if the client is disconnected, or if the server fails.

`QoS=0` is the fastest mode of transfer. It is sometimes called "fire and forget".

The MQTT protocol does not require servers to forward publications at `QoS=0` to a client. If the client is disconnected at the time the server receives the publication, the publication might be discarded, depending on the server. The telemetry (MQXR) service does not discard messages sent with `QoS=0`. They are stored as nonpersistent messages, and are only discarded if the queue manager stops.

At least once

`QoS=1`

`QoS=1` is the default mode of transfer.

The message is always delivered at least once. If the sender does not receive an acknowledgment, the message is sent again with the DUP flag set until an acknowledgment is received. As a result receiver can be sent the same message multiple times, and might process it multiple times.

The message must be stored locally at the sender and the receiver until it is processed.

The message is deleted from the receiver after it has processed the message. If the receiver is a broker, the message is published to its subscribers. If the receiver is a client, the message is delivered to the subscriber application. After the message is deleted, the receiver sends an acknowledgment to the sender.

The message is deleted from the sender after it has received an acknowledgment from the receiver.

Exactly once

`QoS=2`

The message is always delivered exactly once.

The message must be stored locally at the sender and the receiver until it is processed.

`QoS=2` is the safest, but slowest mode of transfer. It takes at least two pairs of transmissions between the sender and receiver before the message is deleted from the sender. The message can be processed at the receiver after the first transmission.

In the first pair of transmissions, the sender transmits the message and gets acknowledgment from the receiver that it has stored the message. If the sender does not receive an acknowledgment, the message is sent again with the DUP flag set until an acknowledgment is received.

In the second pair of transmissions, the sender tells the receiver that it can complete processing the message, `"PUBREL"`. If the sender does not receive an acknowledgment of the `"PUBREL"` message, the `"PUBREL"` message is sent again until an acknowledgment is received. The sender deletes the message it saved when it receives the acknowledgment to the `"PUBREL"` message

The receiver can process the message in the first or second phases, provided that it does not reprocess the message. If the receiver is a broker, it publishes the message to subscribers. If the receiver is a client, it delivers the message to the subscriber application. The receiver sends a completion message back to the sender that it has finished processing the message.

Retained publications and MQTT clients

If you create a subscription to a topic that has a retained publication, the most recent retained publication on the topic is immediately forwarded to you.

Use the `MqttMessage.setRetained` method to specify whether a publication on a topic is retained, or not.

To delete a retained publication in IBM WebSphere MQ, run the CLEAR TOPICSTR**CLEAR TOPICSTR** MQSC command.

If you create a publication with a null payload, the empty publication is forwarded to subscribers. Other MQTT brokers might not forward an empty publication to subscribers.

If you publish a non-retained publication to a topic that has a retained publication, the retained publication is not affected. Current subscribers receive the new publication. New subscribers receive the retained publication first, then receive any new publications.

When you create or update a retained publication, send the publication with a QoS or 1 or 2. If you send it with a QoS of 0, IBM WebSphere MQ creates a nonpersistent retained publication. 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 are taken since the subscriber last subscribed to the publication topic, and if the subscriber subscribes again, the subscriber receives the most recent retained publication on the topic again.

Subscriptions

Create subscriptions to register an interest in publication topics using a topic filter. A client can create multiple subscriptions, or a subscription containing a topic filter that uses wildcards, to register an interest in multiple topics. Publications on topics matching the filters are sent to the client. Subscriptions can remain active while a client is disconnected. The publications are sent to the client when it reconnects.

Create subscriptions using the `MqttClient.subscribe` methods, passing one or more topic filters and quality of service parameters. The quality of service parameter sets the maximum quality of service that the subscriber is prepared to use to receive a message. Messages sent to this client cannot be delivered with a higher quality of service. The quality of service is set to the lower of the original value when the message was published and the level specified for the subscription. The default quality of service for receiving messages is `QoS=1`, at least once.

The subscription request itself is sent with `QoS=1`.

Publications are received by a subscriber when the MQTT client calls the `MqttCallback.messageArrived` method. The `messageArrived` method also passes the topic string with which the message was published to the subscriber.

You can remove a subscription, or a set or subscriptions, using the `MqttClient.unsubscribe` methods.

A WebSphere MQ command can remove a subscription. List subscriptions using WebSphere MQ Explorer, or by using **runmqsc** or PCF commands. All MQTT client subscriptions are named. They are given a name of the form: *`ClientIdentifier`*:*`Topic name`*

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.

Another way of understanding the way the `cleanSession` attribute affects subscriptions is to think of it as a modal attribute. In its default mode, `cleanSession=true`, the client creates

subscriptions and receives publications only within the scope of the session. In the alternative mode, `cleanSession=false`, subscriptions are durable. The client can connect and disconnect and its subscriptions remain active. When the client reconnects, it receives any undelivered publications. While it is connected, it can modify the set of subscriptions that are active on its behalf.

You must set the `cleanSession` mode before connecting; the mode lasts for the whole session. To change its setting, you must 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.

Publications that match active subscriptions are sent to the client as soon as they are published. If the client is disconnected, they are sent to the client if it reconnects to the same server with the same client identifier and `MqttConnectOptions.cleanSession` set to `false`.

Subscriptions for a particular client are identified by the client identifier. You can reconnect the client from a different client device to the same server, and continue with the same subscriptions and receive undelivered publications.

Topic strings and topic filters in MQTT clients

Topic strings and topic filters are used to publish and to subscribe. The syntax of topic strings and filters in MQTT clients is largely the same as topic strings in IBM WebSphere MQ.

Topics strings are used to send publications to subscribers. Create a topic string using the method, `MqttClient.getTopic(java.lang.String topicString)`.

Topic filters are used to subscribe to topics and receive publications. Topic filters can contain wildcards. With wildcards, you can subscribe to multiple topics. Create a topic filter by using a subscription method; for example, `MqttClient.subscribe(java.lang.String topicFilter)`.

Topic strings

The syntax of an IBM WebSphere MQ topic string is described in Topic Strings. The syntax of MQTT topic strings is described in the `MqttClient` class in the API documentation for the MQTT client for Java. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference..

The syntax of each type of topic string is almost identical. There are four minor differences:

1. Topic strings sent to IBM WebSphere MQ by MQTT clients must follow the convention for queue manager names. In particular, not topic strings cannot contain hyphens.
2. The maximum lengths differ. IBM WebSphere MQ topic strings are limited to 10,240 characters. An MQTT client can create topic strings of up to 65535 bytes.
3. A topic string created by an MQTT client cannot contain a null character.
4. In WebSphere Message Broker, a null topic level, `'...//...'` was invalid. Null topic levels are supported by IBM WebSphere MQ.

Unlike IBM WebSphere MQ publish/subscribe, the `mqttv3` protocol does not have a concept of an administrative topic object. You cannot construct a topic string from a topic object and a topic string. However, a topic string is mapped to an administrative topic in WebSphere MQ. The access control associated with the administrative topic determines whether a publication is published to the topic, or discarded. The attributes that are applied to a publication when it is forwarded to subscribers, are influenced by the attributes of the administrative topic.

Topic filters

The syntax of an IBM WebSphere MQ topic filter is described in Topic-based wildcard scheme. The syntax of the topic filters you can construct with an MQTT client are described in the `MqttClient` class in the API documentation for the MQTT client for Java. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

The syntax of each type of topic filter is almost identical. The only difference is in the way different MQTT brokers interpret a topic filter. In WebSphere Message Broker V6, a multilevel wildcard could only be used at the end of a topic filter. In WebSphere MQ, a multilevel wildcard can be used at any level in the topic tree; for example `USA/#/Dutchess County`.

MQTT client programming reference

Here are links to the Mobile Messaging and M2M Client Pack, and to the associated client API documentation.

In the Mobile Messaging and M2M Client Pack, the MQTT client libraries are bundled with their generated API documentation. You can download the client pack from IBM messaging community downloads.

You can see online copies of the most recent API documentation by following these links to the Eclipse Paho project:

- MQTT client for Java classes
- MQTT client library for C
- Asynchronous MQTT client library for C

Note:

1. Link MQTT Java applications to the `org.eclipse.paho.client.mqttv3` package rather than the `com.ibm.micro.client.mqttv3`. package. The `com.ibm.micro.client.mqttv3` package is provided to support existing MQTT Java applications.
2. V7.5.0.1 Link MQTT client apps for C to the `MQTTAsync` library rather the `MQTTClient` library. The `MQTTClient` is provided to support existing MQTT apps for C.
3. The MQTT messaging client for JavaScript requires an MQTT server that supports WebSockets. For example, IBM WebSphere MQ Version 7.5 and later versions do this.

Getting started with MQTT servers

Messaging servers that support the MQTT transport protocol are available from IBM and others. The most basic MQTT server enables mobile apps and devices, supported by MQTT client libraries, to exchange messages. IBM WebSphere MQ and IBM MessageSight are MQTT servers from IBM. In addition to acting as basic MQTT servers, they also exchange messages between MQTT client apps and enterprise apps. All MQTT servers from IBM support the MQTT version 3.1 protocol, and MQTT over the WebSocket protocol.

Current MQTT servers from IBM

IBM WebSphere MQ

- IBM WebSphere MQ provides enterprise-grade messaging. The telemetry component enables IBM WebSphere MQ also to act as an MQTT server.
- This supports your mobile, machine-to-machine (M2M) and device-based apps, and also allows them to exchange messages with enterprise messaging apps such as IBM WebSphere MQ and JMS apps.
- The IBM WebSphere MQ installation includes a copy of the MQTT SDK from IBM. This SDK provides sample MQTT client apps, and MQTT client libraries that support these apps.

  Note: To get the most up-to-date version of this SDK, download the Mobile Messaging and M2M Client Pack. For more information, see “Getting started with MQTT clients” on page 9.
- MQTT support was first included in IBM WebSphere MQ Version 7.0.1. For full information for each release of IBM WebSphere MQ, see the following product documentation:
  - WebSphere MQ Telemetry Version 7.5
  - WebSphere MQ Telemetry Version 7.1

  For a brief introduction to IBM WebSphere MQ, and the steps to get started with the IBM WebSphere MQ Telemetry component, see “IBM WebSphere MQ as the MQTT server” on page 130.

IBM MessageSight

- IBM MessageSight is an appliance-based MQTT server that can connect a massive number of MQTT clients at the same time, and deliver the performance and scalability needed to accommodate the ever growing multitude of mobile devices and sensors. It supports the MQTT version 3.1 protocol, and MQTT over the WebSocket protocol.

- The main features and benefits of IBM MessageSight as an MQTT server are as follows:
  - High-performance, reliability, and scalable messaging.
  - Designed specifically for machine-to-machine (M2M) and Internet of Things scenarios, by supporting massive communities for concurrently connected end points.
  - Ease of installation and use. It can be up and running in under 30 minutes.
  - Support for native mobile apps that include Android and iOS.
  - Integration with IBM WebSphere MQ as a publish/subscribe broker.
- For a quick introduction to IBM MessageSight, see the MessageSight introduction on YouTube and the MessageSight announcement. For detailed technical information, see the MessageSight product documentation.

IBM WebSphere MQ Telemetry daemon for devices

- This is also known as the IBM WebSphere MQ Telemetry advanced client for C. It is a small footprint MQTT server that typically runs in satellite locations or devices near the edge of the network; for example in set-top boxes, remote telemetry units, or point-of-sale terminals.
- A typical use for it is to concentrate lots of MQTT client connections, which are then connected to IBM WebSphere MQ over the internet in a single MQTT connection. For example, you might install a large number of sensors in a building, connect them to the IBM WebSphere MQ Telemetry daemon for devices, and connect the daemon to IBM WebSphere MQ.
- The IBM WebSphere MQ Telemetry daemon for devices is included with IBM WebSphere MQ. A separate license is required to connect it to IBM WebSphere MQ. See IBM United States Software Announcement 212-091.

Really Small Message Broker

- Really Small Message Broker (RSMB) is a version of the IBM WebSphere MQ Telemetry daemon for devices. The main difference is in usage. RSMB is a small test server, available from IBM alphaWorks®, and intended for use when evaluating or experimenting with MQTT-based solutions. RSMB supports MQTT on a number of Linux platforms, on Windows XP, on Apple Mac OS X Leopard, and on Unslung (Linksys NSLU12)

Previous MQTT servers from IBM

WebSphere Message Broker *(now known as IBM Integration Bus)*

- WebSphere Message Broker Version 6 provided its own MQTT server. The support was replaced in WebSphere Message Broker Version 7 by the telemetry component of IBM WebSphere MQ.

Other MQTT servers

MQTT.org maintains a list of MQTT servers and brokers on its Software page, including open source servers.

Related tasks

"Getting started with MQTT clients" on page 9
You can get started developing a mobile or machine-to-machine (M2M) app by building and running a sample MQTT client app that uses an MQTT client library. The sample apps, and associated client libraries, are available in the Mobile Messaging and M2M Client Pack from IBM. There are versions of the apps and client libraries written in Java, in JavaScript, and in C. You can run these apps on most platforms and devices, including Android devices and products from Apple.

IBM WebSphere MQ as the MQTT server

An introduction to using the MQTT server that is included in IBM WebSphere MQ.

To get started, follow the steps in the following articles:

- "Installing IBM WebSphere MQ" on page 130
- "Configuring the MQTT service from the command line" on page 132
- "Configuring the MQTT service with IBM WebSphere MQ Explorer" on page 134

Note: You can get started quickly by using the command-line interface example. However, if your configuration is significantly different to the example you need more knowledge and skill to use the command-line interface effectively. Use the IBM WebSphere MQ Explorer interface to both get started and do standard configuration tasks easily.

For key conceptual information about the IBM WebSphere MQ Telemetry component, see the following articles in the IBM WebSphere MQ product documentation:

- Connecting telemetry devices to a queue manager
- Telemetry (MQXR) service
- Telemetry channels

Related information

Configuring a queue manager for telemetry on Linux and AIX
Configuring a queue manager for telemetry on Windows
Configure distributed queuing to send messages to MQTT clients
Administering WebSphere MQ Telemetry

Installing IBM WebSphere MQ

Follow these instructions to obtain and install IBM WebSphere MQ and configure IBM WebSphere MQ Telemetry on Windows or Linux.

For the operating systems that are supported by the MQTT service running on IBM WebSphere MQ, see IBM WebSphere MQ Telemetry system requirements.

Get a copy of the IBM WebSphere MQ installation materials and a license in one of the following ways:

1. Ask your IBM WebSphere MQ administrator for the installation materials, and to confirm you can accept the license agreement.
2. Get a 90-day evaluation copy of IBM WebSphere MQ. See Evaluate: IBM WebSphere MQ.
3. Buy IBM WebSphere MQ. See IBM WebSphere MQ product page.

Install IBM WebSphere MQ as `root` on Linux, and as an administrator on Windows. At install time, select the additional options `Telemetry Service` and `Telemetry Clients` to install the IBM WebSphere

MQ Telemetry component. Create a user ID to administer IBM WebSphere MQ, and check that the guest user ID is defined. The guest user ID is used in the sample MQTT service configuration to authorize MQTT access to IBM WebSphere MQ.

After installing IBM WebSphere MQ, start the MQTT service by doing the steps in "Configuring the MQTT service from the command line" on page 132 or "Configuring the MQTT service with IBM WebSphere MQ Explorer" on page 134.

1. Log on as `root` on Linux, or as an administrator on Windows.
2. Install IBM WebSphere MQ.

   Follow the instructions in Installing WebSphere MQ server on Linux or Installing WebSphere MQ server on Windows. Select `Telemetry Service` and `Telemetry Clients` to install the IBM WebSphere MQ Telemetry component.

   On Linux, take note of the instruction in the "What to do next" section to make your installation primary. Even if this installation is the only IBM WebSphere MQ installation on your workstation, make it primary. See Single installation of WebSphere MQ Version 7.1, or later, configured as the primary installation.

   To follow the sample configuration instructions exactly, you must make the installation primary.

   Multiple installations: If you want to work with a non-primary installation, run the setmqenv command. It sets up the IBM WebSphere MQ environment in a command window on your workstation. See Multiple installations.

   Assuming you accepted the default installation location offered by the installation program, IBM WebSphere MQ is installed in the following directories:

   Linux 64-bit

   ```
   /opt/mqm
   ```

   Windows 32-bit

   ```
   C:\Program Files\IBM\WebSphere MQ
   ```

   Windows 64-bit

   ```
   C:\Program Files (x86)\IBM\WebSphere MQ
   ```

   The installation directory is shown as *MQ_INSTALLATION_PATH*
3. Optional: Add the user you are going to administer IBM WebSphere MQ with to the `mqm` group on this workstation.

   This step is optional on Windows because you can administer IBM WebSphere MQ as a Windows administrator. See Authority to administer WebSphere MQ on UNIX and Windows systems.

   If your Windows workstation is a member of a domain, see Windows 2000 domain with non-default, or Windows 2003 and Windows Server 2008 domain with default, security permissions.

   On Linux, the installation program creates a user `mqm`, as a member of the group `mqm`. Give this user a password, or create another user with `mqm` as its primary group.
4. Optional: Sign on with the user that you made a member of the `mqm` group.

   This step is optional on Windows because you can administer IBM WebSphere MQ as a Windows administrator.
5. Check that the guest user ID is defined on your workstation.

   The guest user ID is `"guest"` on Windows and `"nobody"` on Linux. The guest user ID does not require any operating system permissions or rights.

You installed IBM WebSphere MQ on your workstation as the primary IBM WebSphere MQ installation and created the group `mqm`. The installation gives permission to administer IBM WebSphere MQ to members

of the mqm group. Members of the administrators group on Windows also have authority to administer IBM WebSphere MQ.

1. Configure the MQTT service from the command line or IBM WebSphere MQ Explorer; see "Configuring the MQTT service with IBM WebSphere MQ Explorer" on page 134 or "Configuring the MQTT service from the command line" on page 132.
2. Test your Android, iOS, WebSockets, Java, and "C" MQTT clients.
3. When you finish testing, remove the queue manager and MQTT service by running the command *MQ_INSTALLATION_PATH*\mqxr\samples\CleanupMQM.bat on Windows and *MQ_INSTALLATION_PATH*/mqxr/samples/CleanupMQM.sh on Linux.

Related information

Installing WebSphere MQ Telemetry

Installing WebSphere MQ server on Linux

Installing WebSphere MQ server on Windows

Configuring the MQTT service from the command line

Follow these instructions configure IBM WebSphere MQ using the command line to run the sample IBM WebSphere MQ Telemetry applications. The steps show you how to run a script to create an MQTT service on a new queue manager called MQXR_SAMPLE_QM.

You must have administrative access to a IBM WebSphere MQ queue manager to set up the MQTT service. You have a number of ways to get access to a queue manager:

1. Get a copy of IBM WebSphere MQ and create a queue manager on your own Linux or Windows workstation. Follow the instructions in "Installing IBM WebSphere MQ" on page 130 to obtain and install IBM WebSphere MQ. Note that you need to also select Telemetry Service and Telemetry Clients when installing. You can also modify an existing installation to add these options.
2. Contact an IBM WebSphere MQ administrator and ask for administrative access to a queue manager on a server that has IBM WebSphere MQ Telemetry installed as an option. V 7.5.0.1 In addition to the name of the queue manager, you require at least two TCP/IP ports for MQTT and for MQTT over WebSockets. If you are planning to connect secure clients, you require at least two more ports.

To do the steps in the task exactly as they are described, you must be able to create a queue manager called MQXR_SAMPLE_QM, and TCP/IP port 1883 must be unused.

In this task you run a script that creates a queue manager, and then configures the MQTT service to listen for MQTT V3.1 client connections on port 1883. The configuration gives everyone permission to publish and subscribe to any topic. The configuration of security and access control is minimal and is intended only for a queue manager that is on a secure network with restricted access. To run IBM WebSphere MQ and MQTT in an insecure environment, you must configure security. To configure security for IBM WebSphere MQ and MQTT, see the related links at the end of this task.

1. Log on with a user ID that has administrative authority to IBM WebSphere MQ.

   To define a user ID with administrative authority to IBM WebSphere MQ, see step 3 in "Installing IBM WebSphere MQ" on page 130.
2. Open a command window and run the sample command script to create and start the sample queue manager called MQXR_SAMPLE_QM and the MQTT service.

   The path to the sample script is %MQ_FILE_PATH%\mqxr\samples\SampleMQM.bat on Windows and *MQ_INSTALLATION_PATH*/mqxr/samples/SampleMQM.sh on Linux.

   Type the following command to create and configure the queue manager:

   - Windows

     ```
     "%MQ_FILE_PATH%\mqxr\samples\SampleMQM.bat"
     ```

   - Linux

```
MQ_INSTALLATION_PATH/mqxr/samples/SampleMQM.sh
```

The sample creates an MQTT channel called `PlainText` with these properties on Windows:

```
com.ibm.mq.MQXR.channel/PlainText: \
com.ibm.mq.MQXR.Protocol=MQTT;\
com.ibm.mq.MQXR.Port=1883;\
com.ibm.mq.MQXR.Backlog=4096;\
com.ibm.mq.MQXR.UserName=Guest;\
com.ibm.mq.MQXR.StartWithMQXRService=true
```

The channel properties on Linux are the same as Windows, except `com.ibm.mq.MQXR.UserName=nobody`.

MQTT V3.1 clients that connect to port 1883 access IBM WebSphere MQ with the user ID set in the variable *com.ibm.mq.MQXR.UserName*. The sample script authorizes the user ID with the following IBM WebSphere MQ commands:

```
setmqaut -m MQXR_SAMPLE_QM -t topic -n SYSTEM.BASE.TOPIC -p com.ibm.mq.MQXR.UserName -all +pub
+sub
setmqaut -m MQXR_SAMPLE_QM -t q -n SYSTEM.MQTT.TRANSMIT.QUEUE -p com.ibm.mq.MQXR.UserName -all
+put
```

The first command gives the user authority to publish and subscribe on topics that inherit their permissions from the base topic. The second command gives the user authority to put messages on the `SYSTEM.MQTT.TRANSMIT.QUEUE` transmission queue. The MQTT service sends messages on the `SYSTEM.MQTT.TRANSMIT.QUEUE` as publications to MQTT subscribers.

The script starts the MQTT service on the queue manage to listen for connections on port 1883.

Follow these steps to test the connection by running the sample MQTT V3.1 Java application.

The source for the sample Java application is in the `MQTTV3Sample.java` file.

Two command windows are required to run the sample. Run the sample as a subscriber in one window and as publisher in the other.

- Windows To start the subscriber, run the command

  ```
  "%MQ_FILE_PATH%\mqxr\samples\RunMQTTV3Sample.bat" -a subscriber
  ```

  To publish, run the command:

  ```
  "%MQ_FILE_PATH%\mqxr\samples\RunMQTTV3Sample.bat"
  ```

- Linux To start the subscriber, run the command

  ```
  MQ_INSTALLATION_PATH/mqxr/samples/RunMQTTV3Sample.sh -a subscriber
  ```

  To publish, run the command:

  ```
  MQ_INSTALLATION_PATH/mqxr/samples/RunMQTTV3Sample.sh
  ```

The publisher and subscriber write output to their command windows:

```
Connected to tcp://localhost:1883
Publishing to topic "MQTTV3Sample/Java/v3" qos 2
Disconnected
Press any key to continue . . .
```

Figure 27. Output from the publisher

```
Connected to tcp://localhost:1883
Subscribing to topic "MQTTV3Sample/#" qos 2
Press <Enter> to exit
Topic:          MQTTV3Sample/Java/v3
Message:        Message from MQTTv3 Java client
QoS:            2
```

Figure 28. Output from the subscriber

The server is now ready for you to test your MQTT V3.1 app.

Related tasks

Configuring the MQTT service with WebSphere MQ Explorer

Follow these instructions to configure IBM WebSphere MQ using IBM WebSphere MQ Explorer to run the sample IBM WebSphere MQ Telemetry clients. The steps show you how to create an MQTT service by running the `Define sample` configuration wizard.

Related information

WebSphere MQ Telemetry

Developing applications for WebSphere MQ Telemetry

Administering WebSphere MQ Telemetry

WebSphere MQ Telemetry security

Configuring the MQTT service with IBM WebSphere MQ Explorer

Follow these instructions to configure IBM WebSphere MQ using IBM WebSphere MQ Explorer to run the sample IBM WebSphere MQ Telemetry clients. The steps show you how to create an MQTT service by running the `Define sample` configuration wizard.

You must have administrative access to a IBM WebSphere MQ queue manager to set up the MQTT service. You have a number of ways to get access to a queue manager:

1. Get a copy of IBM WebSphere MQ and create a queue manager on your own Linux or Windows workstation. Follow the instructions in "Installing IBM WebSphere MQ" on page 130 to obtain and install IBM WebSphere MQ. Note that you need to also select `Telemetry Service` and `Telemetry Clients` when installing. You can also modify an existing installation to add these options.
2. Contact an IBM WebSphere MQ administrator and ask for administrative access to a queue manager on a server that has IBM WebSphere MQ Telemetry installed as an option. V7.5.0.1 In addition to the name of the queue manager, you require at least two TCP/IP ports for MQTT and for MQTT over WebSockets. If you are planning to connect secure clients, you require at least two more ports.

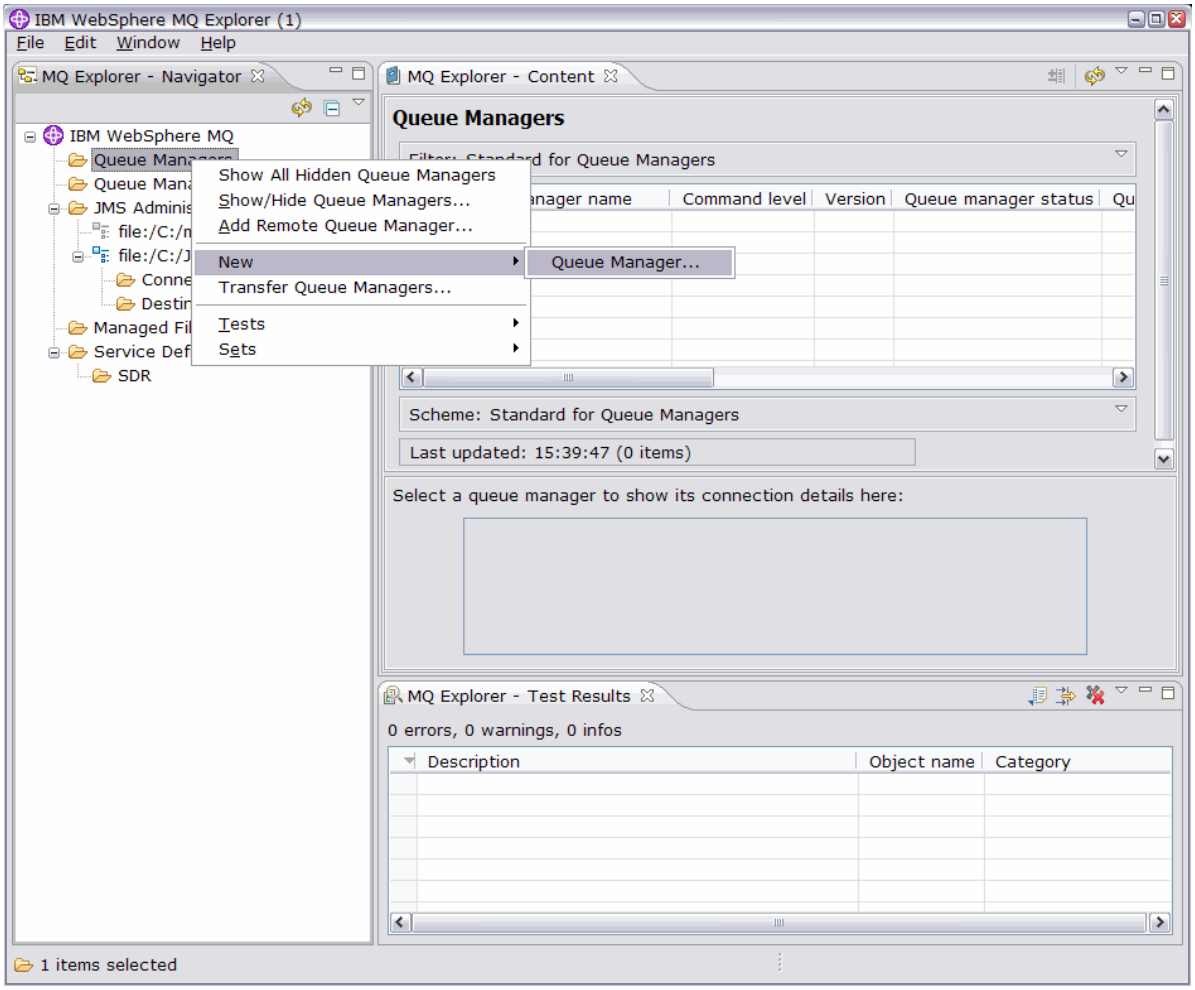
To do the steps in the task exactly as they are described, you must be able to create a queue manager called `MQXR_SAMPLE_QM`, and TCP/IP port 1883 must be unused.

In this task, you run the IBM WebSphere MQ Explorer `Define sample` configuration wizard to create a MQTT service to listen for MQTT V3.1 client connections on port 1883. The configuration gives everyone permission to publish and subscribe to any topic. The configuration of security and access control is minimal and is intended only for a queue manager that is on a secure network with restricted access. To run IBM WebSphere MQ and MQTT in an insecure environment, you must configure security. To configure security for IBM WebSphere MQ and MQTT, see the related links at the end of this task.

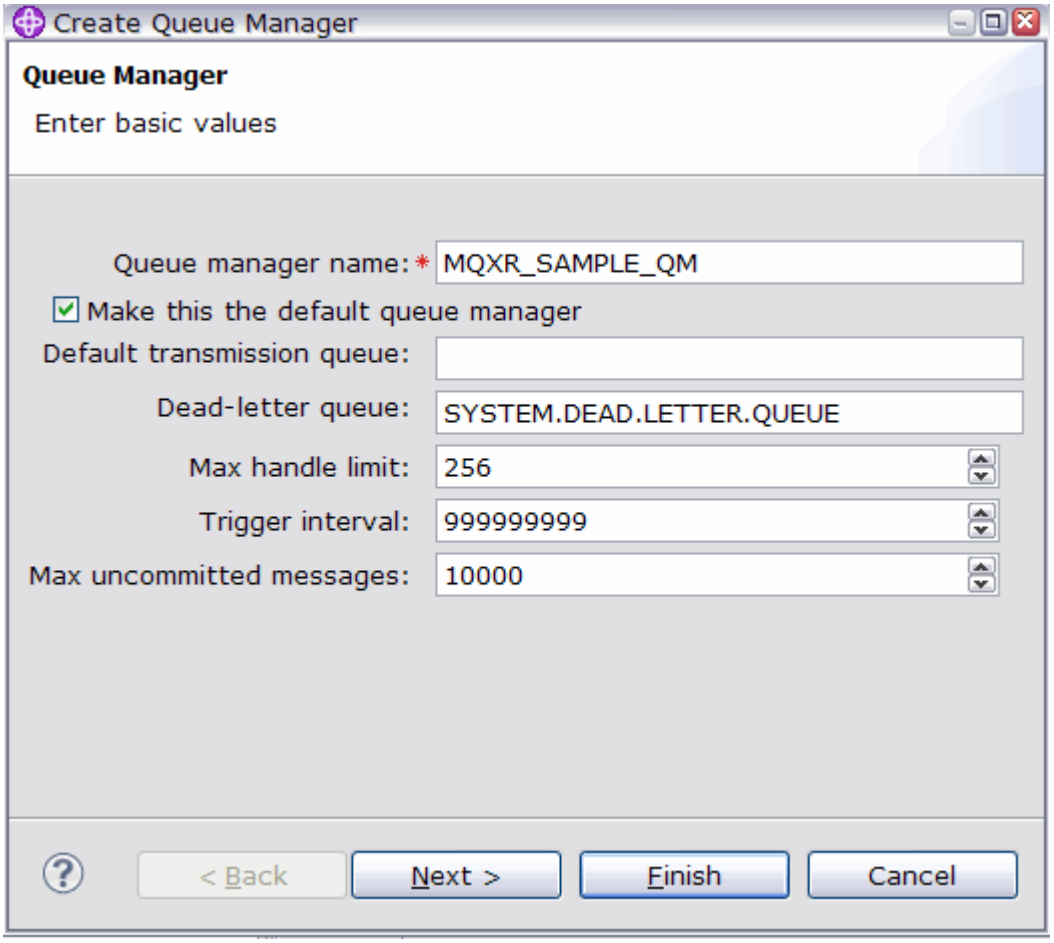
1. Log on with a user ID that has administrative authority to IBM WebSphere MQ.

   To define a user ID with administrative authority to IBM WebSphere MQ, see step 3 in "Installing IBM WebSphere MQ" on page 130.
2. Open a command window, and run the IBM WebSphere MQ Explorer command **`strmqcfg`** to start IBM WebSphere MQ Explorer.
3. Create a queue manager

   a) Start the **New Queue Manager** wizard

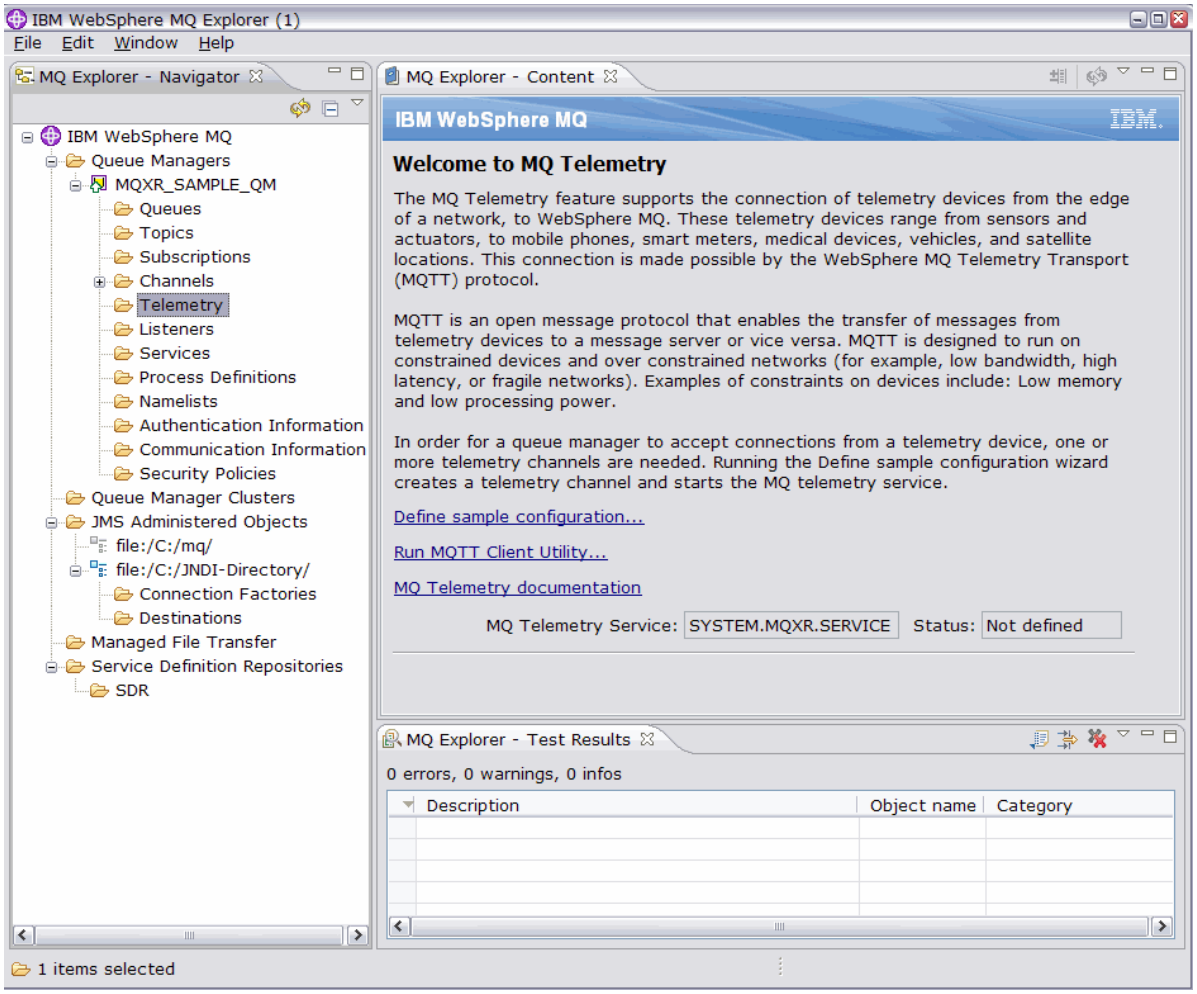


b) Type a **Queue manager name**, and the name of the **Dead-letter queue**. For convenience, make it the default queue manager. Click **Finish**.



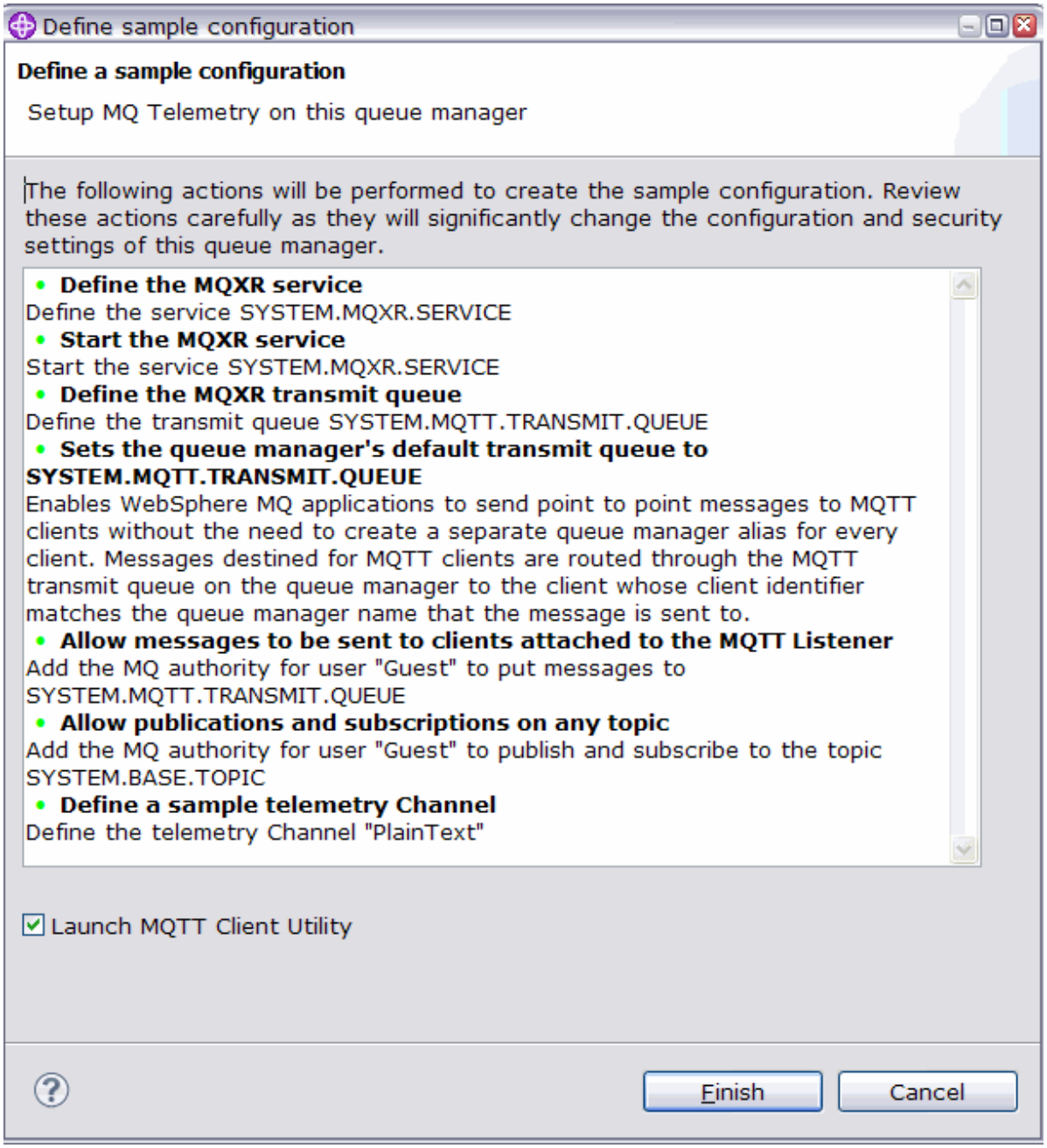
IBM WebSphere MQ Explorer creates the queue manager and starts it.

4. Run the Telemetry **Define sample configuration** wizard.
   a) Open the Telemetry folder for the queue manager.

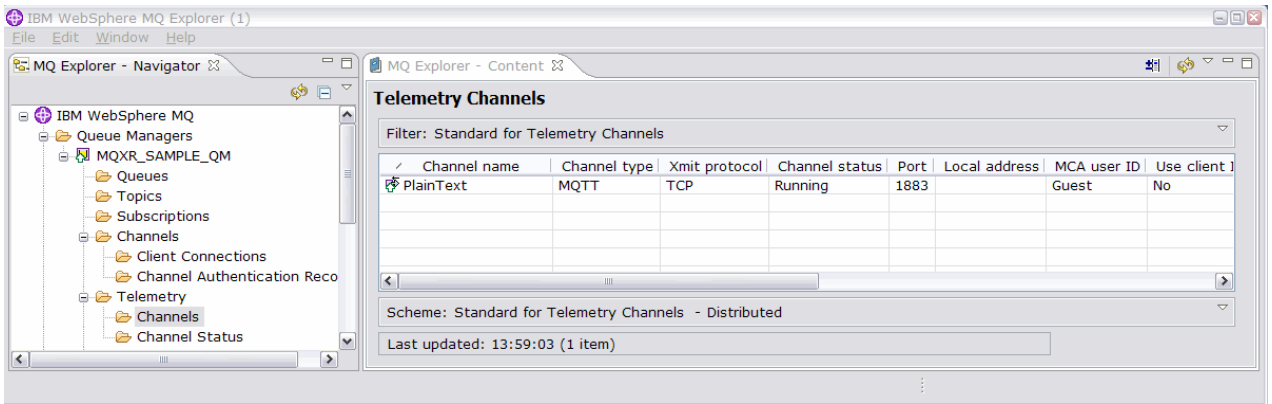


b) Click **Define sample configuration** to start the wizard.

c) Click **Finish** to create the telemetry service and run the MQTT Client Utility



Open the Telemetry Channels folder to list the sample channels.

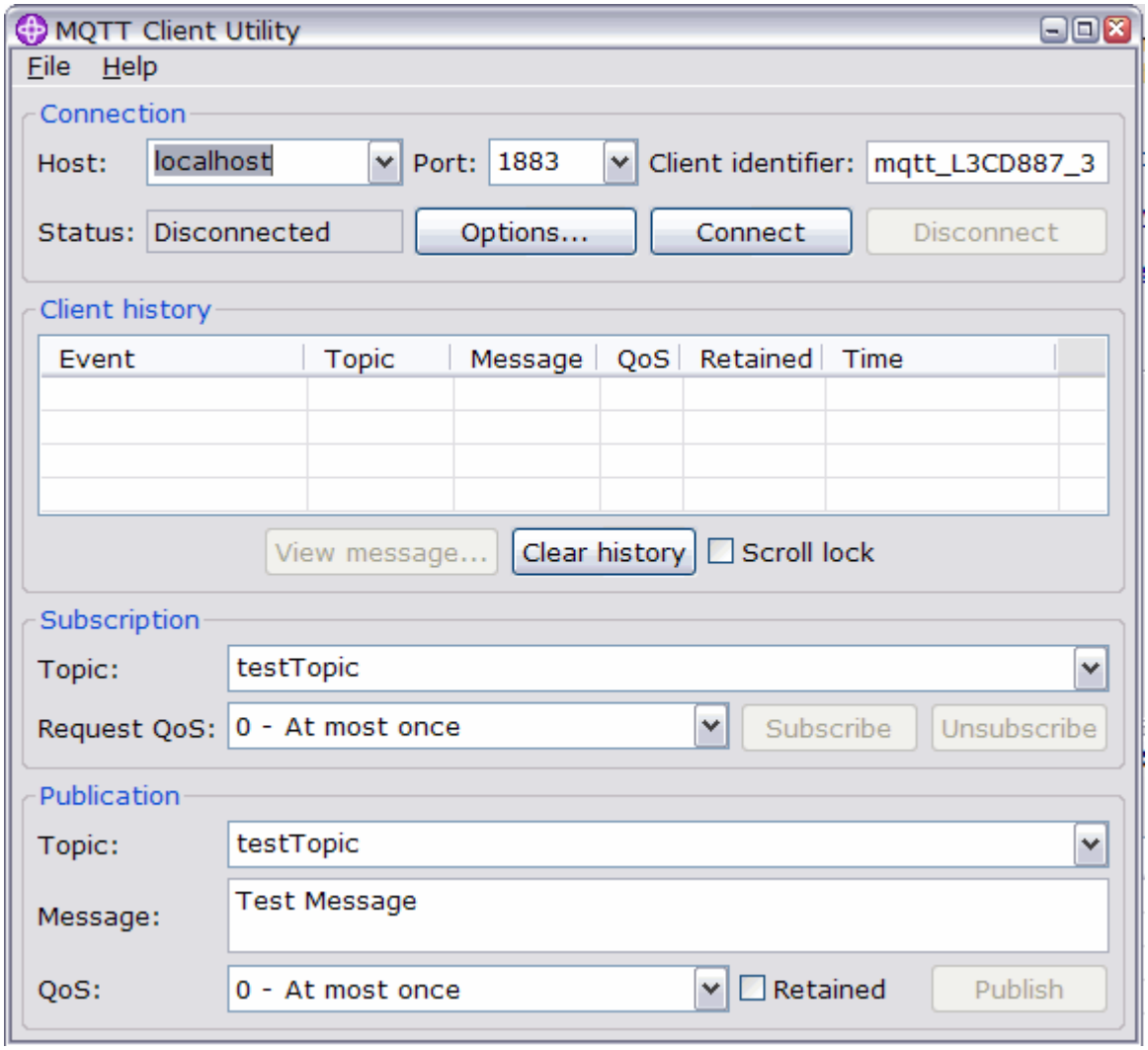


You can modify the properties of this channel, and add and delete channels in this window.

Test the connection by running the MQTT Client Utility.

1. To start the client utility, open the **Telemetry** folder and click **Run MQTT Client Utility** twice.

   Two **MQTT Client Utility** windows open, identical but for different client identifiers.



2. Click **Connect** in both windows.
3. Click **Subscribe** in both windows.
4. Click **Publish** in either window. The results are shown in Figure 29 on page 140

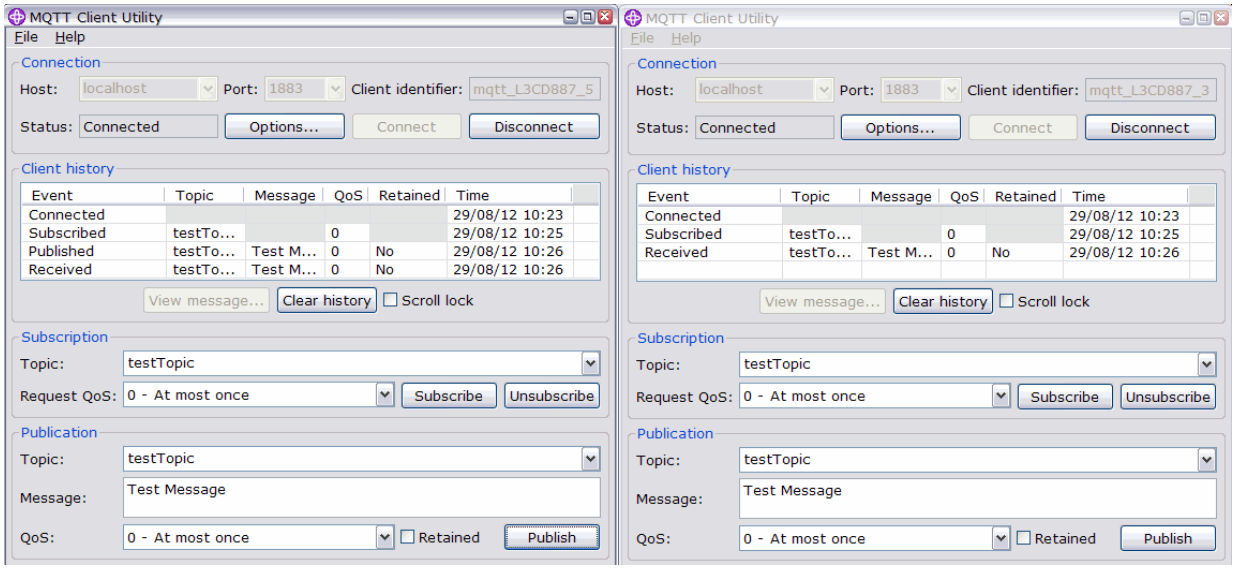


Figure 29. Results

5. Click **Disconnect** in both windows.

The server is now ready for you to test your MQTT V3.1 app.

Related tasks

Configuring the MQTT service from the command line

Follow these instructions configure IBM WebSphere MQ using the command line to run the sample IBM WebSphere MQ Telemetry applications. The steps show you how to run a script to create an MQTT service on a new queue manager called `MQXR_SAMPLE_QM`.

Administering WebSphere MQ Telemetry

Related information

WebSphere MQ Telemetry

Administering WebSphere MQ Telemetry with WebSphere MQ Explorer

Developing applications for WebSphere MQ Telemetry

Security

WebSphere MQ Telemetry security

IBM WebSphere MQ Telemetry daemon for devices concepts

The IBM WebSphere MQ Telemetry daemon for devices is an advanced MQTT V3 client app. Use it to store and forward messages from other MQTT clients. It connects to IBM WebSphere MQ like an MQTT client, but you can also connect other MQTT clients to it.

The daemon is a publish/subscribe broker. MQTT V3 clients connect to it to publish and subscribe to topics, using topic strings to publish, and topic filters to subscribe. The topic string is hierarchical, with topic levels divided by /. Topic filters are topic strings that can include single level + wildcards and a multilevel # wildcard as the last part of the topic string.

Note: Wildcards in the daemon follow the more restrictive rules of WebSphere Message Broker, v6. IBM WebSphere MQ is different. It supports mulitple multilevel wildcards; wildcards can stand in for any number of levels of the hierarchy, anywhere in the topic string.

Multiple MQTT v3 clients connect to the daemon using a listener port. The default listener port is modifiable. You can define multiple listener ports and allocate different namespaces to them, see “WebSphere MQ Telemetry daemon for devices listener ports” on page 148. The daemon is itself an MQTT v3 client. Configure a daemon bridge connection to connect the daemon to the listener port of another daemon, or to a WebSphere MQ Telemetry (MQXR) service.

You can configure multiple bridges for the WebSphere MQ Telemetry daemon for devices. Use the bridges to connect together a network of daemons that can exchange publications.

Each bridge can publish and subscribe to topics at its local daemon. It can also publish and subscribe to topics at another daemon, a WebSphere MQ publish/subscribe broker, or any other MQTT v3 broker it is connected to. Using a topic filter, you can select the publications to propagate from one broker to another. You can propagate publications in either direction. You can propagate publicaitons from the local daemon to each of its attached remote brokers, or from any of the attached brokers to the local daemon; see “IBM WebSphere MQ Telemetry daemon for devices bridges” on page 141.

IBM WebSphere MQ Telemetry daemon for devices bridges

An IBM WebSphere MQ Telemetry daemon for devices bridge connects two publish/subscribe brokers using the MQTT v3 protocol. The bridge propagates publications from one broker to the other, in either direction. At one end is a WebSphere MQ Telemetry daemon for devices bridge connection, and at the other might be a queue manager, or another daemon. A queue manager is connected to the bridge connection using a telemetry channel. A daemon is connected to the bridge connection using a daemon listener.

IBM WebSphere MQ Telemetry daemon for devices supports one or more simultaneous connections to other brokers. The connections from the daemon are called bridges and are defined by connection entries in the daemon configuration file. The connections to IBM WebSphere MQ are made using IBM WebSphere MQ telemetry channels, as shown in the following figure:

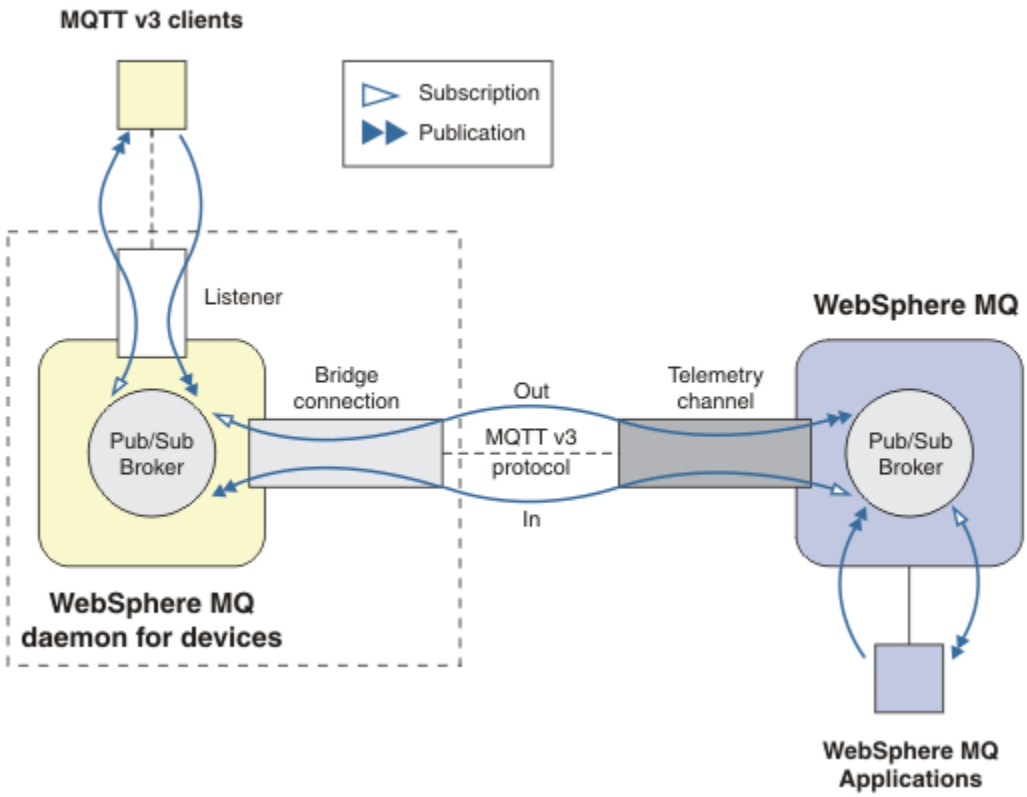


Figure 30. Connecting IBM WebSphere MQ Telemetry daemon for devices to IBM WebSphere MQ

A bridge connects the daemon to another broker as an MQTT v3 client. The bridge parameters mirror the attributes of an MQTT v3 client.

A bridge is more than a connection. It acts as a publish and subscribe agent situated between two publish/subscribe brokers. The local broker is the IBM WebSphere MQ Telemetry daemon for devices, and the remote broker is any publish/subscribe broker that supports the MQTT v3 protocol. Typically the remote broker is another daemon or IBM WebSphere MQ.

The job of the bridge is to propagate publications between the two brokers. The bridge is bidirectional. It propagates publications in either direction. Figure 30 on page 141 illustrates the way the bridge connects IBM WebSphere MQ Telemetry daemon for devices to IBM WebSphere MQ. "Example topic settings for the bridge" on page 142 uses examples to illustrate how to use the topic parameter to configure the bridge.

The In and Out arrows in Figure 30 on page 141 indicate the bidirectionality of the bridge. At one end of the arrow, a subscription is created. The publications that match the subscription are published to the broker at the opposite end of the arrow. The arrow is labeled according to the flow of publications. Publications flow In to the daemon and Out from the daemon. The importance of the labels is they are used in the command syntax. Remember that In and Out refer to where the publications flow, and not to where the subscription is sent.

Other clients, applications, or brokers might be connected either to IBM WebSphere MQ or to WebSphere MQ Telemetry daemon for devices. They publish and subscribe to topics at the broker they are connected to. If the broker is IBM WebSphere MQ, the topics might be clustered or distributed, and not explicitly defined at the local queue manager.

Uses of bridges

Connect daemons together using bridge connections and listeners. Connect daemons and queue managers together using bridge connections and telemetry channels. When you connect multiple brokers together it is possible to create loops. Be careful: Publications might circulate endlessly around a loop of brokers, undetected.

Some of the reasons for using daemons bridged to IBM WebSphere MQ are as follows:

Reduce the number of MQTT client connections to WebSphere MQ
Using a hierarchy of daemons you can connect many clients to WebSphere MQ; more clients than the number a single queue manager can connect at one time.

Store and forward messages between MQTT clients and WebSphere MQ
You might use store and forward to avoid maintaining continuous connections between clients and IBM WebSphere MQ, if the clients do not have their own storage. You might use multiple types of connections between the MQTT client and WebSphere MQ; see Telemetry concepts and scenarios for monitoring and control.

Filter the publications exchanged between MQTT clients and WebSphere MQ
Commonly, publications divide into messages that are processed locally and messages that involve other applications. Local publications might include control flows between sensors and actuators, and remote publications include requests for readings, status, and configuration commands.

Change the topic spaces of publications
Avoid topics strings from clients attached to different listener ports from colliding with one another. The example uses the daemon to label meter readings coming from different buildings; see Separating the topic spaces of different groups of clients.

Example topic settings for the bridge

Publish everything to the remote broker - using defaults

The default direction is called out, and the bridge publishes topics to the remote broker. The topic parameter controls what topics are propagated using topic filters.

The bridge uses the topic parameter in Figure 31 on page 143 to subscribe to everything published to the local daemon by MQTT clients, or by other brokers. The bridge publishes the topics to the remote broker connected by the bridge.

```
connection Daemon1
topic #
```

Figure 31. Publish everything to the remote broker

Publish everything to the remote broker - explicit

The topic setting in the following code fragment gives the same result as using the defaults. The only difference is that the **direction** parameter is explicit. Use the `out` direction to subscribe to the local broker, the daemon, and publish to the remote broker. Publications created on the local daemon that the bridge has subscribed to, are published at the remote broker.

```
connection Daemon1
topic # out
```

Figure 32. Publish everything to the remote broker - explicit

Publish everything to the local broker

Instead of using the direction `out`, you can set the opposite direction, `in`. The following code fragment configures the bridge to subscribe to everything published at the remote broker connected by the bridge. The bridge publishes the topics to the local broker, the daemon.

```
connection Daemon1
topic # in
```

Figure 33. Publish everything to the local broker

Publish everything from the export topic at the local broker to the import topic at the remote broker

Use two additional topic parameters, **`local_prefix`** and **`remote_prefix`**, to modify the topic filter, `#` in the previous examples. One parameter is used to modify the topic filter used in the subscription, and the other parameter is used to modify the topic the publication is published to. The effect is to replace the beginning of the topic string used in one broker with another topic string on the other broker.

Depending on the direction of the topic command the meaning of **`local_prefix`** and **`remote_prefix`** reverses. If the direction is `out`, the default, **`local_prefix`** is used as part of the topic subscription, and **`remote_prefix`** replaces the **`local_prefix`** part of the topic string in the remote publication. If the direction is `in`, **`remote_prefix`** becomes part of the remote subscription, and **`local_prefix`** replaces the **`remote_prefix`** part of the topic string.

The first part of a topic string is often thought of as defining a topic space. Use the additional parameters to change the topic space a topic is published to. You might do this to avoid the topic being propagated colliding with another the topic on the target broker, or to remove a mount point topic string.

As an example, in the following code fragment, all the publications to the topic string `export/#` at the daemon are republished to `import/#` at the remote broker.

```
topic # out export/ import/
```

Figure 34. Publish everything from the export topic at the local broker to the import topic at the remote broker

Publish everything to the import topic at the local broker from the export topic at the remote broker

The following code fragment shows the configuration reversed; the bridge subscribes to everything published with the `export/#` topic string at the remote broker and publishes it to `import/#` at the local broker.

```
connection Daemon1
topic # in import/ export/
```

Figure 35. Publish everything to the import topic at the local broker from the export topic at the remote broker

Publish everything from the 1884/ mount point to the remote broker with the original topic strings

In the following code fragment, the bridge subscribes to everything published by clients connected to the mount point `1884/` at the local daemon. The bridge publishes everything published to the mount point to the remote broker. The mount point string `1884/` is removed from the topics published to the remote broker. The *local_prefix* is the same as the mount point string `1884/`, and the *remote_prefix* is a blank string.

```
listener 1884
mount_point 1884/
connection Daemon1
topic # out 1884/ ""
```

Figure 36. Publish everything from the 1884/ mount point to the remote broker with the original topic strings.

Separating the topic spaces of different clients connected to different daemons

Suppose that an application is written for electrical power meters to publish meter readings for a building. The readings are published using MQTT clients to a daemon hosted in the same building. The topic selected for the publications is `power`. The same application is deployed to a number of buildings in a complex. For site monitoring and data storage, readings from all buildings are aggregated using bridge connections. The connections link the building daemons to WebSphere MQ at a central location.

An identical client app is used in all the buildings. This app publishes to the topic `power`. However the data must be differentiated by building. This is done by the daemon for each building, which adds the building number as a prefix to the topic name. The bridge from the first building in the complex uses the prefix `meters/building01/`, from building two the prefix is `meters/building02/`. The readings from the other buildings follow the same pattern. WebSphere MQ therefore receives the readings with topics like `meters/building01/power`.

The configuration file for each daemon has a topic statement that follows the pattern in the following code fragment:

```
connection Daemon1
topic power out "" meters/building01/
```

Figure 37. Separate the topic spaces of clients connected to different daemons

In the previous code fragment, the empty string is a placeholder for the unused local_prefix parameter.

Note: This example is somewhat artificial, and intended only as an illustration. In practice, the topic space the application publishes to is likely to be configurable.

Separate the topic spaces of clients connected to the same daemon

Suppose that a single daemon is used to connect all the power meters. Assuming that in the application can be configured to connect to different ports, you might distinguish the buildings by attaching the meters from different buildings to different listener ports, as in the following code fragment. Again, the example is contrived; it illustrates how mount points might be used.

```
listener 1884
mount_point meters/building01/
listener 1885
mount_point meters/building02/
connection Daemon1
topic meters/+/power out
```

Figure 38. Separate the topic spaces of clients connected to the same daemon

Remap different topics for publications flowing in both directions

In the configuration in the following code fragment, the bridge subscribes to the single topic b at the remote broker and forwards publications about b to the local daemon, changing the topic to a. The bridge also subscribes to the single topic x at the local broker and forwards publications about x to the remote broker, changing the topic to y.

```
connection Daemon1
topic "" in a b
topic "" out x y
```

Figure 39. Remap different topics for publications flowing in both directions

An important point about this example is that different topics are subscribed to and published to at both brokers. The topics spaces at both brokers are disjoint.

Remap the same topics for publications flowing in both directions (looping)

Unlike the previous example, the configuration in Figure 40 on page 145, in general, results in a loop. In the topic statement `topic "" in a b`, the bridge subscribes to b remotely, and publishes to a locally. In the other topic statement, the bridge subscribes to a locally, and publishes to b remotely. The same configuration can be written as shown in Figure 41 on page 146.

The general result is that if a client publishes to b remotely, the publication is transferred to the local daemon as a publication on topic a. However, on being published by the bridge to the local daemon on the topic a, the publication matches the subscription made by the bridge to local topic a. The subscription is `topic "" out a b`. As a result, the publication is transferred back to the remote broker as a publication on topic b. The bridge is now subscribed to the remote topic b, and the cycle begins again.

Some brokers implement loop detection to prevent the loop happening. But the loop detection mechanism must work when different types of brokers are bridged together. Loop detection does not work if WebSphere MQ is bridged to the WebSphere MQ Telemetry daemon for devices. It does work if two IBM WebSphere MQ Telemetry daemon for devices are bridged together. By default loop detection is turned on; see try_private.

```
connection Daemon1
topic "" in a b
topic "" out a b
```

*Figure 40. **!**Remap the same topics for publications flowing in both directions*

```
connection Daemon1
topic "" both a b
```

Figure 41. !Remap the same topics for publications flowing in both directions, using `both`.

The configuration in Figure 39 on page 145 is the same as Figure 40 on page 145.

Availability of IBM WebSphere MQ Telemetry daemon for devices bridge connections

Configure multiple IBM WebSphere MQ Telemetry daemon for devices bridge connection addresses to connect to the first available remote broker. If the broker is a multi-instance queue manager, provide both of its TCP/IP addresses. Configure a primary connection to connect, or reconnect, to the primary server, when it is available.

The connection bridge parameter, addresses, is a list of TCP/IP socket addresses. The bridge attempts to connect to each address in turn, until it makes a successful connection. The round_robin and start_type connection parameters control how the addresses are used once a successful connection has been made.

If start_type is auto, manual, or lazy, then if the connection fails, the bridge attempts to reconnect. It uses each address in turn, with about a 20 second delay between each connection attempt. If `start_type` is once, then if the connection fails, the bridge does not attempt to reconnect automatically.

If round_robin is true, the bridge connection attempts start at the first address in the list and tries each address in the list in turn. It starts at the first address again, when the list is exhausted. If there is only one address in the list, it tries it again every 20 seconds.

If round_robin is false, the first address in the list, which is called the primary server, is given preference. If the first attempt to connect to the primary server fails, the bridge continues to try to reconnect to the primary server in the background. At the same time, the bridge tries to connect using the other addresses in the list. When the background attempts to connect to the primary server succeed, the bridge disconnects from the current connection, and switches to the primary server connection.

If a connection is disconnected voluntarily, for example by issuing a **`connection_stop`** command, then if the connection is restarted, it tries to use the same address again. If the connection is disconnected due to a failure to connect, or to the remote broker dropping the connection, the bridge waits 20 seconds. It then tries to connect to the next address in the list, or the same address, if there is only one address in the list.

Connecting to a multi-instance queue manager

In a multi-instance queue manager configuration, the queue manager runs on two different servers with different IP addresses. Typically telemetry channels are configured without a specific IP address. They are configured only with a port number. When the telemetry channel is started, by default it selects the first available network address on the local server.

Configure the addresses parameter of the bridge connection with the two IP addresses used by the queue manager. Set round_robin to true.

If the active queue manager instance fails, the queue manager switches over to the standby instance. The daemon detects that the connection to the active instance has broken and tries to reconnect to the standby instance. It uses the other IP address in the list of addresses configured for the bridge connection.

The queue manager to which the bridge connects is still the same queue manager. The queue manager recovers its own state. If cleansession is set to false, the bridge connection session is restored to the same state as before the failover. The connection resumes after a delay. Messages with "at least once" or "at most once" quality of service are not lost, and subscriptions continue to work.

The reconnection time depends on the number of channels and clients that restart when the standby instance starts, and how many messages were inflight. The bridge connection might try to reconnect to both IP addresses a number of times before the connection is reestablished.

Do not configure a multi-instance queue manager telemetry channel with a specific IP address. The IP address is only valid on one server.

If you are using an alternative high-availability solution, that manages the IP address, then it might be correct to configure a telemetry channel with a specific IP address.

cleansession

A bridge connection is an MQTT v3 client session. You can control whether a connection starts a new session, or whether it restores an existing session. If it restores an existing session, the bridge connection preserves the subscriptions and retained publications from the previous session.

Do not set cleansession to false if addresses lists multiple IP addresses, and the IP addresses connect to telemetry channels hosted by different queue managers, or to different telemetry daemons. Session state is not transferred between queue managers or daemons. Trying to restart an existing session on a different queue manager or daemon results in a new session being started. In-doubt messages are lost, and subscriptions might not behave as expected.

notifications

An application can keep track of whether the bridge connection is running by using notifications. A notification is a publication that has the value 1, connected, or 0, disconnected. It is published to *topicString* defined by the `notification_topic` parameter. The default value of *topicString* is $SYS/`broker/connection/` *clientIdentifier*`/state`. The default *topicString* contains the prefix $SYS. Subscribe to topics beginning with $SYS by defining a topic filter beginning with $SYS. The topic filter *#*, subscribe to everything, does not subscribe to topics beginning with $SYS on the daemon. Think of $SYS as defining a special system topic space distinct from the application topic space.

Notifications enable IBM WebSphere MQ Telemetry daemon for devices to notify MQTT clients when a bridge is connected or disconnected.

keepalive_interval

The keepalive_interval bridge connection parameter sets the interval between the bridge sending a TCP/IP ping to the remote server. The default interval is 60 seconds. The ping prevents the TCP/IP session being closed by the remote server, or by a firewall, that detects a period of inactivity on the connection.

clientid

A bridge connection is an MQTT v3 client session and has a `clientIdentifier` that is set by the bridge connection parameter clientid. If you intend reconnections to resume a previous session by setting the cleansession parameter to false, the `clientIdentifier` used in each session must be the same. The default value of clientid is *hostname.connectionName*, which remains the same.

Installation, verification, configuration, and control of the WebSphere MQ Telemetry daemon for devices

Installation, configuration, and control of the daemon is file-based.

Install the daemon by copying the Software Development Kit to the device where you are going to run the daemon.

As an example, run the MQTT client utility and connect to the WebSphere MQ Telemetry daemon for devices as the publish/subscribe broker; see Use the WebSphere MQ Telemetry daemon for devices as the publish/subscribe broker.

Configure the daemon by creating a configuration file; see WebSphere MQ Telemetry daemon for devices configuration file.

Control a running daemon by creating commands in the file, `amqtdd.upd`. Every 5 seconds the daemon reads the file, runs the commands, and deletes the file; see WebSphere MQ Telemetry daemon for devices command file.

WebSphere MQ Telemetry daemon for devices listener ports

Connect MQTT V3 clients to the WebSphere MQ Telemetry daemon for devices using listener ports. You can qualify a listener port with a mount point and a maximum number of connections.

A listener port must correspond to the port number specified on the MQTT client `connect(serverURI)` method of a client connecting to this port. It defaults on both the client and the daemon to `1883`.

You can change the default port for the daemon by setting the global definition `port` in the daemon configuration file. You can set specific ports by adding a `listener` definition to the daemon configuration file.

For each listener port, other than the default port, you can specify a mount point to isolate clients. Clients connected to a port with a mount point are isolated from other clients; see “WebSphere MQ Telemetry daemon for devices mount points” on page 148.

You can limit the number of clients that can connect to any port. Set the global definition `max_connections` to limit connections to the default port, or qualify each listener port with `max_connections`.

Example

An example of a configuration file that changes the default port from `1883` to `1880`, and limits connections to port `1880` to `10000`. Connections to port `1884` are limited to `1000`. Clients attached to port `1884` are isolated from clients attached to other ports.

```
port 1880
max_connections 10000
listener 1884
mount_point 1884/
max_connections 1000
```

WebSphere MQ Telemetry daemon for devices mount points

You can associate a mount point with a listener port used by MQTT clients to connect to a WebSphere MQ Telemetry daemon for devices. A mount point isolates the publications and subscriptions exchanged by MQTT clients using one listener port from MQTT clients connected to a different listener port.

Clients attached to a listener port with a mount point can never directly exchange topics with clients attached to any other listener ports. Clients attached to a listener port without a mount point can publish or subscribe to topics of any client. Clients are not aware of whether they are attached through a mount point or not; it makes no difference to the topics strings created by clients.

A mount point is a string of text that is prefixed to the topic string of publications and subscriptions. It is prefixed to all the topic strings created by clients attached to listener port with a mount point. The string of text is removed from all topic strings sent to clients attached to the listener port.

If a listener port has no mount point, the topic strings of publications and subscriptions created and received by clients attached to the port are not altered.

Create mount point strings with a trailing `/`. That way the mount point is the parent topic of the topic tree for the mount point.

Example

A configuration file contains the following listener ports:

```
listener 1883
mount_point 1883/
```

```
listener 1884 127.0.0.1
mount_point 1884/
listener 1885
```

A client, attached to port 1883, creates a subscription to `MyTopic`. The daemon registers the subscription as `1883/MyTopic`. Another client attached to port 1883 publishes a message on the topic, `MyTopic`. The daemon changes the topic string to `1883/MyTopic` and searches for matching subscriptions. The subscriber on port 1883 receives the publication with the original topic string `MyTopic`. The daemon has removed the mount point prefix from the topic string.

Another client, attached to port 1884, also publishes on the topic `MyTopic`. This time the daemon registers the topic as `1884/MyTopic`. The subscriber on port 1883 does not receive the publication, because the different mount point results in a subscription with a different topic string.

A client, attached to port 1885, publishes on the topic, `1883/MyTopic`. The daemon does not change the topic string. The subscriber on port 1883 receives the publication to `MyTopic`.

WebSphere MQ Telemetry daemon for devices quality of service, durable subscriptions and retained publications

Quality of service settings apply only to a running daemon. If a daemon stops, whether in a controlled manner, or because of a failure, the state of inflight messages is lost. The delivery of a message at least once, or at most once, cannot be guaranteed if the daemon stops. WebSphere MQ Telemetry daemon for devices supports limited persistence. Set the **`retained_persistence`** configuration parameter to save retained publications and subscriptions when the daemon is shut down.

Unlike WebSphere MQ, the WebSphere MQ Telemetry daemon for devices does not journal persistent data. Session state, message state, and retained publications are not saved transactionally. By default, the daemon discards all data when it stops. You can set an option to periodically checkpoint subscriptions and retained publications. Message status is always lost when the daemon stops. All non-retained publications are lost.

Set the daemon configuration option, `Retained_persistence` to `true`, to save retained publications periodically to a file. When the daemon restarts, the retained publications that were last autosaved are reinstated. By default, retained messages created by clients are not reinstated when the daemon restarts.

Set the daemon configuration option, `Retained_persistence` to `true`, to save subscriptions created in a persistent session periodically to a file. If `Retained_persistence` is set to `true`, subscriptions that clients create in a session with `CleanSession` set to `false`, a "persistent session", are restored. The daemon restores the subscriptions when it restarts, which start receiving publications. The client receives the publications when it restarts with `CleanSession` to `false`. By default, client session state is not saved when a daemon stops, and so subscriptions are not restored, even if the client sets `CleanSession` to `false`.

`Retained_persistence` is an autosave mechanism. It might not save the most recent retained publications or subscriptions. You can change how often retained publications and subscriptions are saved. Set the interval between saves, or the number of changes between saves, using the configuration options `autosave_on_changes` and `autosave_interval`.

Example configuration for setting persistence

```
# Sample configuration
# Daemon listens on port 1882 with persistence in /tmp
# Autosave every minute
port 1882
persistence_location /tmp/
retained_persistence true
autosave_on_changes false
autosave_interval 60
```

WebSphere MQ Telemetry daemon for devices security

The WebSphere MQ Telemetry daemon for devices can authenticate clients that connect to it, use credentials to connect to other brokers, and control access to topics. The security the daemon provides is limited by being built using the WebSphere MQ Telemetry C client, which does not provide SSL support. Consequently, connections to and from the daemon are not encrypted, and cannot be authenticated using certificates.

By default, no security is switched on.

Authentication of clients

MQTT clients can set a username and password using the methods `MqttConnectOptions.setUserName` and `MqttConnectOptions.setPassword`.

Authenticate a client that connects to the daemon by checking the username and password provided by a client against entries in the password file. To enable authentication, create a password file and set the `password_file` parameter in the daemon configuration file; see password_file.

Set the `allow_anonymous` parameter in the daemon configuration file to allow clients connecting without usernames or passwords to connect to a daemon that is checking authentication; see allow_anonymous. If a client does provide a username or password it is always checked against the password file, if the `password_file` parameter is set.

Set the `clientid_prefixes` parameter in the daemon configuration file to limit connections to specific clients. The clients must have `clientIdentifiers` that start with one of the prefixes listed in the `clientid_prefixes` parameter; see clientid_prefixes.

Bridge connection security

Each WebSphere MQ Telemetry daemon for devices bridge connection is an MQTT V3 client. You can set the username and password for each bridge connection as a bridge connection parameter in the daemon configuration file; see username and password. A bridge can then authenticate itself to a broker.

Access control of topics

If clients are being authenticated, the daemon can also provide control access to topics for each user. The daemon grants access control based on matching the topic to which a client is either publishing or subscribing with an access topic string in the access control file; see acl_file.

The access control list has two parts. The first part controls access for all clients, including anonymous clients. The second part has a section for any user in the password file. It lists specific access control for each user.

Example

The security parameters are shown in the following example.

```
acl_file c:\WMQTDaemon\config\acl.txt
password_file c:\WMQTDaemon\config\passwords.txt
allow_anonymous true
connection Daemon1
username daemon1
password deamonpassword
```

Figure 42. Daemon configuration file

```
Fred:Fredpassword
Barney:Barneypassword
```

Figure 43. Password file, passwords.txt

```
topic home/public/#
topic read meters/#
user Fred
topic write meters/fred
topic home/fred/#
user Barney
topic write meters/barney
topic home/barney/#
```

Figure 44. Access control file, acl.txt

Troubleshooting MQTT clients

Look for a troubleshooting task to help you solve a problem with running MQTT clients.

Related tasks

"Tracing and debugging the MQTT (Paho) Java client" on page 159
The default logger uses the standard Java logging facility which is known as `java.util.logging` (JSR47). You can configure it either by the use of a configuration file or programmatically.

"Tracing the MQTT JavaScript client" on page 161
You can use the JavaScript client to collect trace by altering the client web application to call methods on the connected client object.

"Tracing the telemetry (MQXR) service" on page 155
Follow these instructions to start a trace of the telemetry service, set the parameters that control the trace, and find the trace output.

"Tracing the MQTT v3 Java client " on page 156
Follow these instructions to create an MQTT Java client trace and control its output.

"Tracing the MQTT client for C" on page 158
Set the environment variable `MQTT_C_CLIENT_TRACE` to trace an MQTT client C app

"Resolving problem: MQTT client does not connect" on page 167
Resolve the problem of an MQTT client program failing to connect to the telemetry (MQXR) service.

"Resolving problem: MQTT client connection dropped" on page 169
Find out what is causing a client to throw unexpected `ConnectionLost` exceptions after successfully connecting and running for either a short or long while.

"Resolving problem: Lost messages in an MQTT application" on page 170
Resolve the problem of losing a message. Is the message non-persistent, sent to the wrong place, or never sent? A wrongly coded client program might lose messages.

"Resolving problem: Telemetry (MQXR) service does not start" on page 171
Resolve the problem of the telemetry (MQXR) service failing to start. Check the WebSphere MQ Telemetry installation and no files are missing, moved, or have the wrong permissions. Check the paths used by the telemetry (MQXR) service locate the telemetry (MQXR) service programs.

"Resolving problem: JAAS login module not called by the telemetry service" on page 172
Find out if your JAAS login module is not being called by the telemetry (MQXR) service, and configure JAAS to correct the problem.

"Resolving problem: Starting or running the daemon" on page 175

Consult the IBM WebSphere MQ Telemetry daemon for devices console log, turn on tracing, or use the symptom table in this topic to troubleshoot problems with the daemon.

“Resolving problem: MQTT clients not connecting to the daemon” on page 176
Clients are not connecting to the daemon, or the daemon is not connecting to other daemons or to a WebSphere MQ telemetry channel.

Related reference

“Location of telemetry logs, error logs, and configuration files ” on page 152
Find the logs, error logs, and configuration files used by IBM WebSphere MQ Telemetry.

“MQTT v3 Java client reason codes” on page 154
Look up the causes of reason codes in an MQTT v3 Java client exception or throwable.

“System requirements for using SHA-2 cipher suites with MQTT clients” on page 162
For Java 6 from IBM, SR13 onwards, you can use SHA-2 cipher suites to secure your MQTT channels and client apps. However, SHA-2 cipher suites are not enabled by default until Java 7 from IBM, SR4 onwards, so in earlier versions you must specify the required suite. If you are running an MQTT client with your own JRE, you need to ensure that it supports the SHA-2 cipher suites. For your client apps to use SHA-2 cipher suites, the client must also set the SSL context to a value that supports Transport Layer Security (TLS) version 1.2.

“Restrictions in browser support for mobile messaging web apps over SSL” on page 163
There are differences in capability between different browsers, on different platforms. Understanding these differences helps you configure your apps, certificate authorities (CAs) and client certificates to connect using the MQTT messaging client for JavaScript over SSL and WebSockets.

Location of telemetry logs, error logs, and configuration files

Find the logs, error logs, and configuration files used by IBM WebSphere MQ Telemetry.

Note: The examples are coded for Windows. Change the syntax to run the examples on Linux

Server-side logs

The installation wizard for IBM WebSphere MQ Telemetry writes messages to its installation log:

```
WMQ program directory\mqxr
```

The telemetry (MQXR) service writes messages to the WebSphere MQ queue manager error log, and FDC files to the IBM WebSphere MQ error directory:

```
WMQ data directory\Qmgrs\qMgrName\errors\AMQERR01.LOG
WMQ data directory\errors\AMQnnn.n.FDC
```

It also writes a log for the telemetry (MQXR) service. The log displays the properties the service started with, and errors it has found acting as a proxy for an MQTT client. For example, unsubscribing from a subscription that the client did not create. The log path is:

```
WMQ data directory\Qmgrs\qMgrName\errors\mqxr.log
```

The IBM WebSphere MQ telemetry sample configuration created by IBM WebSphere MQ Explorer starts the telemetry service using the command **runMQXRService**. **runMQXRService** is in *WMQ Telemetry install directory*\bin. It writes to:

```
WMQ data directory\Qmgrs\qMgrName\mqxr.stdout
WMQ data directory\Qmgrs\qMgrName\mqxr.stderr
```

Modify **runMQXRService** to display the paths configured for the telemetry (MQXR) service, or to echo the initialization before starting the telemetry (MQXR) service.

Server-side configuration files

Telemetry channels and telemetry (MQXR) service

Restriction: The format, location, content, and interpretation of the telemetry channel configuration file might change in future releases. You must use IBM WebSphere MQ Explorer to configure telemetry channels.

IBM WebSphere MQ Explorer saves telemetry configurations in the `mqxr_win.properties` file on Windows, and the `mqxr_unix.properties` file on Linux. The properties files are saved in the telemetry configuration directory:

```
WMQ data directory\Qmgrs\qMgrName\mqxr
```

Figure 45. Telemetry configuration directory on Windows

```
/var/mqm/qmgrs/qMgrName/mqxr
```

Figure 46. Telemetry configuration directory on Linux

JVM

Set Java properties that are passed as arguments to the telemetry (MQXR) service in the file, `java.properties`. The properties in the file are passed directly to the JVM running the telemetry (MQXR) service. They are passed as additional JVM properties on the Java command line. Properties set on the command line take precedence over properties added to the command line from the `java.properties` file.

Find the `java.properties` file in the same folder as the telemetry configurations, see Figure 45 on page 153 and Figure 46 on page 153.

Modify `java.properties` by specifying each property as a separate line. Format each property exactly as you would to pass the property to the JVM as an argument; for example:

```
-Xmx1024m
-Xms1024m
```

JAAS

The JAAS configuration file is described in Telemetry channel JAAS configuration , which includes the sample JAAS configuration file, JAAS.config, shipped with IBM WebSphere MQ Telemetry.

If you configure JAAS, you are almost certainly going to write a class to authenticate users to replace the standard JAAS authentication procedures.

To include your `Login` class in the class path used by the telemetry (MQXR) service class path, provide a WebSphere MQ `service.env` configuration file.

Set the class path for your JAAS `LoginModule` in `service.env`. You cannot use the variable, `%classpath%` in `service.env`. The class path in `service.env` is added to the class path already set in the telemetry (MQXR) service definition.

Display the class paths that are being used by the telemetry (MQXR) service by adding `echo set classpath` to `runMQXRService.bat`. The output is sent to `mqxr.stdout`.

The default location for the `service.env` file is:

```
WMQ data directory\service.env
```

Override these settings with a `service.env` file for each queue manager in:

```
WMQ data directory\Qmgrs\qMgrName\service.env
```

Figure 47 on page 154 shows a sample `service.env` file to use the sample `LoginModule.class`.

```
CLASSPATH=WMQ Install Directory\mqxr\samples
```

Note: `service.env` must not contain any variables. Substitute the actual value of *WMQ Install Directory*.

Figure 47. Sample `service.env` for Windows

Trace

An IBM service engineer might ask you to configure trace; see "Tracing the telemetry (MQXR) service" on page 155. the parameters to configured trace are stored in two files:

```
WMQ data directory\Qmgrs\qMgrName\mqxr\trace.config
WMQ data directory\Qmgrs\qMgrName\mqxr\mqxrtrace.properties
```

Client-side log files

The default file persistence class in the Java SE MQTT client supplied with IBM WebSphere MQ Telemetry creates a folder with the name: *clientIdentifier*-tcp*hostNameport* or *clientIdentifier*-ssl*hostNameport* in the client working directory. The folder name tells you the `hostName` and `port` used in the connection attempt. The folder contains messages that have been stored by the persistence class. The messages are deleted when they have been delivered successfully.

The folder is deleted when a client, with a clean session, ends.

If client trace is turned on, the unformatted log is, by default, stored in the client working directory. The trace file is called `mqtt-n.trc`

Client-side configuration files

Set trace and SSL properties for the MQTT Java client using Java property files, or set the properties programmatically. Pass the properties to the MQTT Java client using the JVM `-D` switch: for example,

```
Java -Dcom.ibm.micro.client.mqttv3.trace=c:\\MqttTrace.properties
     -Dcom.ibm.ssl.keyStore=C:\\MyKeyStore.jks
```

See "Tracing the MQTT v3 Java client " on page 156. For links to client API documentation for the MQTT client libraries, see MQTT client programming reference.

MQTT v3 Java client reason codes

Look up the causes of reason codes in an MQTT v3 Java client exception or throwable.

Table 5. MQTT v3 Java client reason codes

Reason code	Value	Cause
REASON_CODE_BROKER_UNAVAILABLE	3	
REASON_CODE_CLIENT_ALREADY_CONNECTED	32100	The client is already connected.
REASON_CODE_CLIENT_ALREADY_DISCONNECTED	32101	The client is already disconnected.
REASON_CODE_CLIENT_DISCONNECT_PROHIBITED	32107	Thrown when an attempt to call `MqttClient.disconnect` has been made from within a method on `MqttCallback`.
REASON_CODE_CLIENT_DISCONNECTING	32102	The client is currently disconnecting and cannot accept any new work.

Table 5. MQTT v3 Java client reason codes (continued)		
Reason code	**Value**	**Cause**
REASON_CODE_CLIENT_EXCEPTION	0	Client encountered an exception.
REASON_CODE_CLIENT_NOT_CONNECTED	32104	The client is not connected to the server.
REASON_CODE_CLIENT_TIMEOUT	32000	Client timed out while waiting for a response from the server.
REASON_CODE_FAILED_AUTHENTICATION	4	Authentication with the server has failed, due to a bad user name or password.
REASON_CODE_INVALID_CLIENT_ID	2	The server has rejected the supplied client ID.
REASON_CODE_INVALID_PROTOCOL_VERSION	1	The protocol version requested is not supported by the server.
REASON_CODE_NO_MESSAGE_IDS_AVAILABLE	32001	Internal error, caused by no new message IDs being available.
REASON_CODE_NOT_AUTHORIZED	5	Not authorized to perform the requested operation.
REASON_CODE_SERVER_CONNECT_ERROR	32103	Unable to connect to server.
REASON_CODE_SOCKET_FACTORY_MISMATCH	32105	Server URI and supplied SocketFactory do not match.
REASON_CODE_SSL_CONFIG_ERROR	32106	SSL configuration error.
REASON_CODE_UNEXPECTED_ERROR	6	An unexpected error has occurred.

Tracing the telemetry (MQXR) service

Follow these instructions to start a trace of the telemetry service, set the parameters that control the trace, and find the trace output.

Tracing is a support function. Follow these instructions if an IBM service engineer asks you to trace your telemetry (MQXR) service. The product documentation does not document the format of the trace file, or how to use it to debug a client.

You can use the IBM WebSphere MQ **strmqtrc** and **endmqtrc** commands to start and stop IBM WebSphere MQ trace. **strmqtrc** captures trace for the telemetry (MQXR) service. When using **strmqtrc** , there is a delay of up to a couple of seconds before the telemetry service trace is started. For further information about IBM WebSphere MQ trace, see Using trace. Alternatively, you can trace the telemetry (MQXR) service by using the following procedure:

1. Set the trace options to control the amount of detail and the size of the trace. The options apply to a trace started with either the **strmqtrc** or the **controlMQXRChannel** command.

   Set the trace options in the following files:

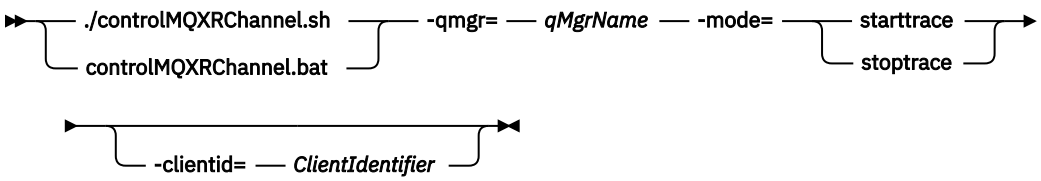
   ```
   mqxrtrace.properties
   trace.config
   ```

   The files are in the directory:

   - On Windows, *WebSphere MQ data directory*\qmgrs*qMgrName* \mqxr.

- On Linux, `var/mqm/qmgrs/ qMgrName/mqxr`.

2. Open a command window in the following directory:
   - On Windows systems, *WebSphere MQ installation directory*`\mqxr\bin` .
   - On Linux systems `/opt/mqm/mqxr/bin`.
3. Run the following command to start an `SYSTEM.MQXR.SERVICE` trace:



Mandatory parameters

qmgr=*qmgrName*
Set *qmgrName* to the queue manager name

mode=starttrace| stoptrace
Set starttrace to begin tracing or to stoptrace to end tracing

Optional parameters

clientid=*ClientIdentifier*
Set *ClientIdentifier* to the `ClientIdentifier` of a client. clientid filters trace to a single client. Run the trace command multiple times to trace multiple clients.

For example:

```
/opt/mqm/mqxr/bin/controlMQXRChannel.sh -qmgr=QM1 -mode=starttrace -clientid=
problemclient
```

To view the trace output, go to the following directory:

- On Windows , *WebSphere MQ data directory*`\trace`.
- On Linux, `/var/mqm/trace` .

Trace files are named `mqxr_PPPPP.trc`, where PPPPP is the process ID.

Related reference
strmqtrc

Tracing the MQTT v3 Java client

Follow these instructions to create an MQTT Java client trace and control its output.

This topic is only applicable to IBM WebSphere MQ version 7.5.0.0. For information describing tracing the Java client for later versions, see “Tracing and debugging the MQTT (Paho) Java client” on page 159.

Tracing is a support function. Follow these instructions if an IBM service engineer asks you to trace your MQTT Java client. The product documentation does not document the format of the trace file, or how to use it to debug a client.

Trace only works for the WebSphere MQ Telemetry Java client.

Note: The examples are coded for Windows. Change the syntax to run the examples on Linux[2].

1. Create a Java properties file containing the trace configuration.

   In the properties file specify the following optional properties. If a property key is specified more than once, the last occurrence sets the property.

[2] Java uses the correct path delimiter. You can code the delimiter in a property file as `'/'` or `'\\'`; `'\'` is the escape character

a) `com.ibm.micro.client.mqttv3.trace.outputName`

The directory to write the trace file to. It defaults to the client working directory. The trace file is called `mqtt-`*`n`*`.trc`.

```
java com.ibm.micro.client.mqttv3.trace.outputName=c:\\MQTT_Trace
```

b) `com.ibm.micro.client.mqttv3.trace.count`

The number of trace files to write. The default is one file, of unlimited size.

```
java com.ibm.micro.client.mqttv3.trace.count=5
```

c) `com.ibm.micro.client.mqttv3.trace.limit`

The maximum size of file to write, the default is 500000. The limit only applies if more than one trace file is requested.

```
java com.ibm.micro.client.mqttv3.trace.limit=100000
```

d) `com.ibm.micro.client.mqttv3.trace.client.`*`clientIdentifier`*`.status`

Turn trace on or off, per client. If *`clientIdentifier`*`=*`, trace is turned on or off for all clients. By default, trace is turned off for all clients.

```
java com.ibm.micro.client.mqttv3.trace.client.*.status=on
```

```
java com.ibm.micro.client.mqttv3.trace.client.Client10.status=on
```

2. Pass the trace properties file to the JVM using a system property.

```
java -Dcom.ibm.micro.client.mqttv3.trace=c:\\MqttTrace.properties
```

3. Run the client.
4. Convert the trace file from binary encoding to text or `.html`. Use the following command:

```
java com.ibm.micro.client.mqttv3.internal.trace.TraceFormatter [-i traceFile] [-o
outputFile] [-h] [-d
time]
```

where the arguments are:

`-?`
Displays help

`-i traceFile`
Required. Passes in the input file (for example, `mqtt-0.trc`).

`-o outputFile`
Required. Defines the output file (for example, `mqtt-0.trc.html` or `mqtt-0.trc.txt`).

`-h`
Output as HTML. The output files extension must be `.html`. If not specified, the output is plain text.

`-d time`
Indents a line with `*` if the time difference in milliseconds is greater than or equal to (>=) time. Not applicable for HTML output.

The following example will output the trace file in HTML format

```
java com.ibm.micro.client.mqttv3.internal.trace.TraceFormatter -i mqtt-0.trc -o
mqtt-0.trc.html -h
```

The second example will output the trace file as plain text, with any consecutive timestamps that have milliseconds with a difference of 50 or greater indented with an asterisk (*).

```
java com.ibm.micro.client.mqttv3.internal.trace.TraceFormatter -i mqtt-0.trc -o
mqtt-0.trc.txt -d 50
```

The final example will output the trace file as plain text:

```
java com.ibm.micro.client.mqttv3.internal.trace.TraceFormatter -i mqtt-0.trc -o
mqtt-0.trc.txt
```

Tracing the MQTT client for C

Set the environment variable `MQTT_C_CLIENT_TRACE` to trace an MQTT client C app

MQTT client for C trace is available for both the pre-built Windows and Linux MQTT client for C libraries, and for the iOS libraries you build yourself.

Set the environment variable `MQTT_C_CLIENT_TRACE` to a path to a file that is to contain the trace output. Trace output is written to the file.

Set `MQTT_C_CLIENT_TRACE=mqttcclient.log` before running your MQTT client C app.

a) For example, modify the sample script in "Getting started with the MQTT client for C" on page 23:

```
@echo off
setlocal
set MQTT_C_CLIENT_TRACE=mqttcclient.log
call "C:\Program Files\Microsoft Visual Studio 10.0\VC\vcvarsall.bat" x86
cl  /nologo /D "WIN32" /I "..\include" "MQTTV3Sample.c" /link /
nologo ..\windows_ia32\mqttv3c.lib
set path=%path%;..\windows_ia32;
start "MQTT Subscriber" MQTTV3Sample -a subscribe -b localhost -p 1883
@rem Sleep for 2 seconds
ping -n 2 127.0.0.1 > NUL 2>&1
MQTTV3Sample -b localhost -p 1883
pause
endlocal
```

b) Run the script from the `%sdkroot%/sdk/client/c/samples` directory.

The trace output files starts with the following lines:

```
============================================================
                   Trace Output
============================================================
19700101 000000.000 (8084)  (1)> Socket_outInitialize:113
19700101 000000.000 (8084)   (2)> SocketBuffer_initialize:81
19700101 000000.000 (8084)   (2)< SocketBuffer_initialize:85
19700101 000000.000 (8084)  (1)< Socket_outInitialize:129
19700101 000000.000 (8084)  (1)> Thread_create_sem:189
19700101 000000.000 (8084)  (1)< Thread_create_sem:222 (0)
19700101 000000.000 (8084)  (1)> Thread_create_sem:189
19700101 000000.000 (8084)  (1)< Thread_create_sem:222 (0)
19700101 000000.000 (8084)  (1)> Thread_create_sem:189
19700101 000000.000 (8084)  (1)< Thread_create_sem:222 (0)
19700101 000000.000 (8084)  (1)> Thread_create_sem:189
19700101 000000.000 (8084)  (1)< Thread_create_sem:222 (0)
19700101 000000.000 (8084)  (1)> MQTTPersistence_create:43
19700101 000000.000 (8084)  (1)< MQTTPersistence_create:89 (0)
19700101 000000.000 (8084)  (1)> MQTTPersistence_initialize:104
19700101 000000.000 (8084)  (1)< MQTTPersistence_initialize:112 (0)
19700101 000000.000 (8084) (0)< MQTTClient_create:267 (0)
19700101 000000.000 (8084) (0)> MQTTClient_connect:701
19700101 000000.000 Connecting to serverURI localhost:1883
20130201 125912.234 (8084)  (1)> MQTTProtocol_connect:93
```

```
20130201 125912.234 (8084)   (2)> MQTTProtocol_addressPort:43
20130201 125912.234 (8084)   (2)< MQTTProtocol_addressPort:68
20130201 125912.234 (8084)   (2)> Socket_new:594
20130201 125912.234 New socket 1860 for localhost, port 1883
```

Related tasks

"Getting started with the MQTT client for C" on page 23
Get up and running with the sample MQTT client for C on any platform on which you can compile the C source. Verify that you can run the sample MQTT client for C with either IBM MessageSight or IBM WebSphere MQ as the MQTT server.

Tracing and debugging the MQTT (Paho) Java client

The default logger uses the standard Java logging facility which is known as `java.util.logging` (JSR47). You can configure it either by the use of a configuration file or programmatically.

Note: The Paho Java client is applicable only to versions of IBM WebSphere MQ versions 7.5.0.1 and later. For information describing tracing the Java client in IBM WebSphere MQ version 7.5.0.0, see "Tracing the MQTT v3 Java client " on page 156.

Note: Tracing is a support function. Follow these instructions if an IBM service engineer asks you to trace your MQTT Java client. The product documentation does not document the format of the trace file, or how to use it to debug a client. Trace only works for the IBM WebSphere MQ Telemetry Java client.

The simplest method to use a configuration file is to specify its name in the property `java.util.logging.config.file`.

A working property file `jsr47min.properties` is provided in package `org.eclipse.paho.client.mqttv3.logging`

The JSR47 logging facility can be used in a number of ways:

- To collect messages from a selected set of packages
- To collect messages from at and below a log level
- To choose multiple destinations for the log messages
- By providing a built-in logger that writes to a file and controls the size and number of files used
- By providing a built-in logger that writes to memory and enables the in memory messages to be written out based on a trigger
- If the application that uses the MQTT client library is also instrumented by using JSR47, then messages from the application and client library are intermingled

A utility class is provided to help collect debug information. This class includes the log and trace messages that are described earlier, but can collect information such as Java system properties and the value of variables from inside the Paho client.

The debug facility is provided in the public class `Debug`, that is part of the package `org.eclipse.paho.client.mqttv3.util`. An instance of Debug can be obtained by using the method getDebug() on both the asynchronous and synchronous MQTT client objects.

For example:

```
MqttClient cl = new MqttClient();
Debug d = cl.getDebug();
```

The method dumpClientDebug() dumps the maximum amount of debug information. The log facility must be enabled to capture the full debug information, which is written to the it. To capture the full debug information, call a dump method when the problem is known to occur, for example after a particular exception occurs.

1. Create a configuration file or use the jsr47min.properties that is supplied.

If you are using the provided properties file, check that the push trigger is set to correct error level. By default this is set to a Severe level error, but it might be necessary to continually write the trace to the file rather than holding it in memory until an error occurs. To do this, change:

```
java.util.logging.MemoryHandler.push=SEVERE
```

to

```
java.util.logging.MemoryHandler.push=ALL
```

2. Pass the trace configuration file to the JVM using a system property.

   If are you are using the jsr4min.properties this is:

   ```
   java -Djava.util.logging.config.file=C:\temp\jsr47min.properties
   ```

3. Run the client.

When an exception or problem occurs the Paho Debug class writes the in memory trace to the configured file target.

Trace is not automatically written to the file as it is generated, this occurs only when either the push trigger is hit or the debug class causes the trace to be written. The latter might require changes to the application code.

Each line is written to the file handler as it is created. You can control the format in which messages are written out by configuring a FileHandler. A custom file handler is provided with Paho that writes out more than the SimpleHandler and less than the XMLHandler provided with the JRE. Trace records that use the Paho log formatter are of the following form:

```
Level   Data and Time   Class   Method   Thread   clientID   Message
```

A working property file `jsr47min.properties` is provided. This file contains a suggested configuration for collecting trace that helps solve problems that are related to the Paho MQTT client. It configures trace to be continuously collected in memory with minimal impact on performance. When the push trigger occurs or a specific request is made to push, the in memory trace is pushed to the configured target handler. The default push trigger is a Severe level message, which is a broken connection. By default, the trace that is collected in memory is written to the specified file at this point. By default this file is the standard `java.util.logging.FileHandler`. You can use the Paho Debug class to push the memory trace to its target.

Full details of JSR47 can be found in the Javadoc for package java.util.logging.

```
# Loggers
# ----------------------------------------
# A memory handler is attached to the Paho packages
# and the level specified to collect all trace related
# to Paho packages.  This will override any root/global
# level handlers if set.
org.eclipse.paho.client.mqttv3.handlers=java.util.logging.MemoryHandler
org.eclipse.paho.client.mqttv3.level=ALL
# It is possible to set more granular trace on a per class basis e.g.
#org.eclipse.paho.client.mqttv3.internal.ClientComms.level=ALL

# Handlers
# ----------------------------------------
# Note: the target handler that is associated with the Memory Handler is not a root handler
# and hence not returned when getting the handlers from root. It appears accessing
# target handler programmatically is not possible as target is a private variable in
# class MemoryHandler
java.util.logging.MemoryHandler.level=FINEST
java.util.logging.MemoryHandler.size=10000
java.util.logging.MemoryHandler.push=SEVERE
java.util.logging.MemoryHandler.target=java.util.logging.FileHandler
#java.util.logging.MemoryHandler.target=java.util.logging.ConsoleHandler

# --- FileHandler ---
# Override of global logging level
java.util.logging.FileHandler.level=ALL
```

```
# Naming style for the output file:
# (The output file is placed in the directory
# defined by the "user.home" System property.)
# See java.util.logging for more options
java.util.logging.FileHandler.pattern=%h/paho%u.log

# Limiting size of output file in bytes:
java.util.logging.FileHandler.limit=200000

# Number of output files to cycle through, by appending an
# integer to the base file name:
java.util.logging.FileHandler.count=3

# Style of output (Simple or XML):
java.util.logging.FileHandler.formatter=org.eclipse.paho.client.mqttv3.logging.SimpleLogFormatte
r
```

In order to collect trace programmatically a utility class is provided to help collect debug information. This class includes the log and trace messages that are described earlier, but can collect information such as Java system properties and the value of variables from inside the Paho client.

The debug facility is provided in the public class Debug, that is part of the package `org.eclipse.paho.client.mqttv3.util`. An instance of Debug can be obtained by using the method getDebug() on both the asynchronous and synchronous MQTT client objects.

For example:

```
MqttClient cl = new MqttClient();
Debug d = cl.getDebug();
```

The method dumpClientDebug() dumps the maximum amount of debug information. The log facility must be enabled to capture the full debug information, which is written to the it. To capture the full debug information, call a dump method when the problem is known to occur, for example after a particular exception occurs.

Tracing the MQTT JavaScript client

You can use the JavaScript client to collect trace by altering the client web application to call methods on the connected client object.

To collect trace you can use the following methods:

- `client.startTrace()` starts tracing for the client.
- `client.stopTrace()` stops trace for the client.
- `client.getTraceLog()` returns the current trace buffer.

You can output the trace buffer to send to IBM Software Support. A number of ways exist to do this. The example shows trace being started, then the output sent to both the console and a specified email address, and finally the trace being stopped.

```
client = new Messaging.Client(location.hostname, Number(location.port), "clientId");
// Start the client tracing, the trace records capture the method calls and network
//flows from now on.
client.startTrace();

client.onConnectionLost = onConnectionLost;
client.connect({onSuccess:onConnect});

function onConnect() {
  console.log("onConnect, will now disconnect then email Trace");
  client.disconnect();
};
function onConnectionLost(responseObject) {
  if (responseObject.errorCode !== 0)
    console.log("ConnectionLost:"+responseObject.errorMessage);
    console.log(client.getTraceLog());
    window.location="mailto:helpdesk@"+location.hostname+
        "?Subject=Web%20Messaging%20Utility%20Trace&body="+
        client.getTraceLog().join("%0A");
    client.stopTrace();
 };
```

Sample output:

```
Client.startTrace, "2013-10-03T10:58:10.531Z", "0.0.0.0",
Client.connect, {"keepAliveInterval":60,"cleanSession":true},, false,
Client._socket_send, {"type":1,"keepAliveInterval":60,"cleanSession":true,
    "clientId":"clientId"},
Client._on_socket_message, {},
Client._on_socket_message, {"type":2,"topicNameCompressionResponse":0,"returnCode":0},
Client.disconnect,Client._socket_send, {"type":14},
Client.getTraceLog, "2013-10-03T10:58:10.548Z",
Client.getTraceLog in flight messages,
```

V 7.5.0.2 System requirements for using SHA-2 cipher suites with MQTT clients

For Java 6 from IBM, SR13 onwards, you can use SHA-2 cipher suites to secure your MQTT channels and client apps. However, SHA-2 cipher suites are not enabled by default until Java 7 from IBM, SR4 onwards, so in earlier versions you must specify the required suite. If you are running an MQTT client with your own JRE, you need to ensure that it supports the SHA-2 cipher suites. For your client apps to use SHA-2 cipher suites, the client must also set the SSL context to a value that supports Transport Layer Security (TLS) version 1.2.

For Java 7 from IBM, SR4 onwards, SHA-2 cipher suites are enabled by default. For Java 6 from IBM, SR13 and later service releases, if you define an MQTT channel without specifying a cipher suite, the channel will not accept connections from a client using a SHA-2 cipher suite. To use SHA-2 cipher suites, you must specify the required suite in the channel definition. This makes the MQTT server enable the suite before making connections. It also means that only client apps using the specified suite can connect to this channel.

There is a similar limitation for the MQTT client for Java. If the client code is running on a Java 1.6 JRE from IBM, the required SHA-2 cipher suites must be explicitly enabled. In order to use these suites, the client must also set the SSL context to a value that supports Version 1.2 of the Transport Layer Security (TLS) protocol. For example:

```
MqttConnectOptions mqttConnectOptions = new MqttConnectOptions();
java.util.Properties sslClientProps = new java.util.Properties();
sslClientProps.setProperty("com.ibm.ssl.keyStore", sslKeys.clientKeyStore);
sslClientProps.setProperty("com.ibm.ssl.keyStorePassword", sslKeys.clientStorePassword);
sslClientProps.setProperty("com.ibm.ssl.trustStore", sslKeys.clientKeyStore);
sslClientProps.setProperty("com.ibm.ssl.trustStorePassword", sslKeys.clientStorePassword);
sslClientProps.setProperty("com.ibm.ssl.protocol", "TLSv1.2");
sslClientProps.setProperty("com.ibm.ssl.enabledCipherSuites",
"SSL_RSA_WITH_AES_256_CBC_SHA256" );
mqttConnectOptions.setSSLProperties(sslClientProps);
```

As at June 2013, Internet Explorer 10 is the only browser that works with the MQTT messaging client for JavaScript and also supports the TLS 1.2 protocol, so it is the only browser you can use if you want to make SHA-2 connections with the JavaScript client.

For a list of the cipher suites that are currently supported, see the related links.

Related concepts

“MQTT client configuration for client authentication using SSL” on page 98
To authenticate the MQTT client using SSL, the client connects to a telemetry channel using SSL. It must specify a TCP port that corresponds to a telemetry channel that is configured to authenticate SSL clients.

“MQTT client configuration for channel authentication using SSL” on page 100
To authenticate the telemetry channel using SSL, the client must connect to the telemetry channel using SSL. It must specify a port that corresponds to a telemetry channel that is configured for SSL. The

configuration must include a passphrase protected keystore that contains the privately signed digital certificate of the server.

V 7.5.0.1 Restrictions in browser support for mobile messaging web apps over SSL

There are differences in capability between different browsers, on different platforms. Understanding these differences helps you configure your apps, certificate authorities (CAs) and client certificates to connect using the MQTT messaging client for JavaScript over SSL and WebSockets.

Mobile messaging using JavaScript over SSL is fairly new, so it is not surprising that different browser and platform combinations have implemented the capability in slightly different ways, and to different extents. The following table gives an overview of what currently works and does not work for each combination of browser (Firefox, Chrome, Internet Explorer, and Safari) and platform (Windows, Linux, Mac, iOS, and Android).

Table 6. SSL support by platform and browser. For each browser and platform combination, the table specifies whether SSL Anonymous and Non-Anonymous connections are supported, and the extent to which the browser works with all Certificate Authorities (CAs) and client certificates.

Browser	SSL Support (Y/N)	SSL works with any CA (Y/N)	More information
Firefox desktop.	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - Yes	Add the CA and client certificate to the browser. Firefox uses its own certificate store. To import a CA certificate, click **Tools** > **Options** > **Advanced** > **Encryption** > **View Certificates** > **Authorities** > **Import** To import a client certificate, click **Tools** > **Options** > **Advanced** > **Encryption** > **View Certificates** > **Your Certificates** > **Import** To enable a secure connection, specify `https://` in the URL. Firefox gives you the option of selecting a certificate automatically or by asking you every time. Firefox also gives you the option of using SSL 3.0 or TLS 1.0; make sure that both are selected.

Table 6. SSL support by platform and browser. For each browser and platform combination, the table specifies whether SSL Anonymous and Non-Anonymous connections are supported, and the extent to which the browser works with all Certificate Authorities (CAs) and client certificates. *(continued)*

Browser	**SSL Support (Y/N)**	**SSL works with any CA (Y/N)**	**More information**
Chrome desktop.	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - Yes	Use the browser to add the CA and client certificate to the operating system certificate store, which is shared with other software. To import a CA certificate, click **Settings** > **Show Advanced Settings** > **Manage Certificates** > **Trusted Root Certification Authorities** > **Import** To import a client certificate, click **Settings** > **Show Advanced Settings** > **Manage Certificates** > **Personal** > **Import** To enable a secure connection, specify `https://` in the URL. Chrome prompts you with several choices; select the correct one, depending on whether you are configuring an Anonymous or Non-Anonymous connection.

Table 6. SSL support by platform and browser. For each browser and platform combination, the table specifies whether SSL Anonymous and Non-Anonymous connections are supported, and the extent to which the browser works with all Certificate Authorities (CAs) and client certificates. *(continued)*

Browser	**SSL Support (Y/N)**	**SSL works with any CA (Y/N)**	**More information**
Internet Explorer.	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - Yes	When you make a Non-Anonymous SSL connection, you are prompted to choose the correct client certificate. Internet Explorer uses the Windows certificate store, which is shared with other software. To import a CA certificate, click **Tools** > **Internet Options** > **Content** > **Certificates** > **Trusted Root Certification Authorities** > **Import** To import a client certificate, click **Tools** > **Internet Options** > **Content** > **Certificates** > **Personal** > **Import**
Safari desktop.	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - Yes	Use the browser to add the CA and client certificate to the operating system certificate store, which is shared with other software.
Firefox on Android	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - No	Non-Anonymous: Client certificates do not work, because you cannot meet the requirement to add your CA to the list in Firefox. To import a client certificate, click **Settings** > **Security** > **Credential Storage**. If your certificate is signed by a trusted CA in the list, you can make a secure connection.

Table 6. SSL support by platform and browser. For each browser and platform combination, the table specifies whether SSL Anonymous and Non-Anonymous connections are supported, and the extent to which the browser works with all Certificate Authorities (CAs) and client certificates. *(continued)*

Browser	SSL Support (Y/N)	SSL works with any CA (Y/N)	More information
Chrome on Android	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - No	Non-Anonymous: Client certificates do not work, because you cannot meet the requirement to add your CA to the list in Chrome. **Note:** Google plan to support this in Version 27 of Chrome. This has been an open defect since Version 18. To import a client certificate, click **Settings** > **Security** > **Credential Storage**. If your certificate is signed by a trusted CA in the list, you can make a secure connection.
Safari on iOS	SSL Anonymous - Yes SSL Non-Anonymous - Yes	SSL Anonymous - Yes SSL Non-Anonymous - No	Non-Anonymous: The device does not trust the client certificate, even when the CA certificate is installed at the same time. Safari uses the device certificate store. To import into this store, click **Settings** > **General** > **Profile**, and serve the CA or client certificate from a web page, or email it to yourself.

Table 6. SSL support by platform and browser. For each browser and platform combination, the table specifies whether SSL Anonymous and Non-Anonymous connections are supported, and the extent to which the browser works with all Certificate Authorities (CAs) and client certificates. *(continued)*

Browser	SSL Support (Y/N)	SSL works with any CA (Y/N)	More information
Chrome on iOS	SSL Anonymous - Yes SSL Non-Anonymous - No	SSL Anonymous - No SSL Non-Anonymous - No	Anonymous: Only Apple apps can access the iOS system root store. Therefore Chrome must use its own CA list, which you cannot add to. Non-Anonymous: Client certificates do not work, because you cannot meet the requirement to add your CA to the list.

Related tasks

"Connecting the MQTT messaging client for JavaScript over SSL and WebSockets" on page 72
Connect your web app securely to IBM WebSphere MQ by using the MQTT messaging client for JavaScript sample HTML pages with SSL and the WebSocket protocol.

Related information

Mozilla: (SSL) Does Firefox use the Android CA storage or its own?
Chromium: Issue 134418 - Implement client certificate support
Unable to open https site with not trusted certificate on ie10

Resolving problem: MQTT client does not connect

Resolve the problem of an MQTT client program failing to connect to the telemetry (MQXR) service.

Is the problem at the server, at the client, or with the connection? Have you have written your own MQTT v3 protocol handling client, or an MQTT client app using the C or Java WebSphere MQTT clients?

Run the verification application supplied with WebSphere MQ Telemetry on the server, and check that the telemetry channel and telemetry (MQXR) service are running correctly. Then transfer the verification application to the client, and run the verification application there.

There are a number of reasons why an MQTT client might not connect, or you might conclude it has not connected, to the telemetry server.

1. Consider what inferences can be drawn from the reason code that the telemetry (MQXR) service returned to `MqttClient.Connect`. What type of connection failure is it?

Option	Description
REASON_CODE_INVALID_PROTOCOL_VERSION	Make sure that the socket address corresponds to a telemetry channel, and you have not used the same socket address for another broker.
REASON_CODE_INVALID_CLIENT_ID	Check that the client identifier is no longer than 23 bytes, and contains only characters from the range: `A-Z, a-z, 0-9, './_%`

Option	Description
REASON_CODE_INVALID_DESTINATION	Check that the client identifier is not the same as the queue manager name.
REASON_CODE_SERVER_CONNECT_ERROR	Check that the telemetry (MQXR) service and the queue manager are running normally. Use **netstat** to check that the socket address is not allocated to another application.

If you have written an MQTT client library rather than use one of the libraries provided by IBM WebSphere MQ Telemetry, look at the CONNACK return code.

From these three errors you can infer that the client has connected to the telemetry (MQXR) service, but the service has found an error.

2. Consider what inferences can be drawn from the reason codes that the client produces when the telemetry (MQXR) service does not respond:

Option	Description
REASON_CODE_CLIENT_EXCEPTION **REASON_CODE_CLIENT_TIMEOUT**	Look for an FDC file at the server; see "Server-side logs" on page 152. When the telemetry (MQXR) service detects the client has timed out, it writes a first-failure data capture (FDC) file. It writes an FDC file whenever the connection is unexpectedly broken.

The telemetry (MQXR) service might not have responded to the client, and the timeout at the client expires. The WebSphere MQ Telemetry Java client only hangs if the application has set an indefinite timeout. The client throws one of these exceptions after the timeout set for `MqttClient.Connect` expires with an undiagnosed connection problem.

Unless you find an FDC file that correlates with the connection failure you cannot infer that the client tried to connect to the server:

a) Confirm that the client sent a connection request.

Check the TCPIP request with a tool such as **tcpmon**, available from https://java.net/projects/tcpmon

b) Does the remote socket address used by the client match the socket address defined for the telemetry channel?

The default file persistence class in the Java SE MQTT client supplied with IBM WebSphere MQ Telemetry creates a folder with the name: *clientIdentifier*`-tcp`*hostNameport* or *clientIdentifier*`-ssl`*hostNameport* in the client working directory. The folder name tells you the `hostName` and `port` used in the connection attempt; see "Client-side log files" on page 154.

c) Can you ping the remote server address?

d) Does **netstat** on the server show the telemetry channel is running on the port the client is connecting too?

3. Check whether the telemetry (MQXR) service found a problem in the client request.

The telemetry (MQXR) service writes errors it detects into `mqxr.log`, and the queue manager writes errors into `AMQERR01.LOG`; see

4. Attempt to isolate the problem by running another client.

- Run the MQTT sample application using the same telemetry channel.
- Run the **wmqttSample** GUI client to verify the connection. Get **wmqttSample** by downloading SupportPac IA92.

Note: Older versions of IA92 do not include the MQTT v3 Java client library.

Run the sample programs on the server platform to eliminate uncertainties about the network connection, then run the samples on the client platform.

5. Other things to check:
   a) Are tens of thousands of MQTT clients trying to connect at the same time?

      Telemetry channels have a queue to buffer a backlog of incoming connections. Connections are processed in excess of 10,000 a second. The size of the backlog buffer is configurable using the telemetry channel wizard in IBM WebSphere MQ Explorer. Its default size is 4096. Check that the backlog has not been configured to a low value.

   b) Are the telemetry (MQXR) service and queue manager still running?

   c) Has the client connected to a high availability queue manager that has switched its TCPIP address?

   d) Is a firewall selectively filtering outbound or return data packets?

Resolving problem: MQTT client connection dropped

Find out what is causing a client to throw unexpected `ConnectionLost` exceptions after successfully connecting and running for either a short or long while.

The MQTT client has connected successfully. The client might be up for a long while. If clients are starting with only a short interval between them, the time between connecting successfully and the connection being dropped might be short.

It is not hard to distinguish a dropped connection from a connection that was successfully made, and then later dropped. A dropped connection is defined by the MQTT client calling the `MqttCallback.ConnectionLost` method. The method is only called after the connection has been successfully established. The symptom is different to `MqttClient.Connect` throwing an exception after receiving a negative acknowledgment or timing out.

If the MQTT client app is not using the MQTT client libraries supplied by IBM WebSphere MQ, the symptom depends on the client. In the MQTT v3 protocol, the symptom is a lack of timely response to a request to the server, or the failure of the TCP/IP connection.

The MQTT client calls `MqttCallback.ConnectionLost` with a throwable exception in response to any server-side problems encountered after receiving a positive connection acknowledgment. When an MQTT client returns from `MqttTopic.publish` and `MqttClient.subscribe` the request is transferred to an MQTT client thread that is responsible for sending and receiving messages. Server-side errors are reported asynchronously by passing a throwable exception to the `ConnectionLost` callback method.

The telemetry (MQXR) service always writes a first-failure data capture file if it drops the connection.

1. Has another client started that used the same `ClientIdentifier`?

   If a second client is started, or the same client is restarted, using the same `ClientIdentifier`, the first connection to the first client is dropped.

2. Has the client accessed a topic that it is not authorized to publish or subscribe to?

   Any actions the telemetry service takes on behalf of a client that return `MQCC_FAIL` result in the service dropping the client connection.

   The reason code is not returned to the client.

   - Look for log messages in the `mqxr.log` and `AMQERR01.LOG` files for the queue manager the client is connected to; see “Server-side logs” on page 152.

3. Has the TCP/IP connection dropped?

   A firewall might have a low timeout setting for marking a TCPIP connection as inactive, and dropped the connection.

   - Shorten the inactive TCPIP connection time using `MqttConnectOptions.setKeepAliveInterval`.

Resolving problem: Lost messages in an MQTT application

Resolve the problem of losing a message. Is the message non-persistent, sent to the wrong place, or never sent? A wrongly coded client program might lose messages.

How certain are you that the message you sent, was lost? Can you infer that a message is lost because the message was not received? If message is a publication, which message is lost: the message sent by the publisher, or the message sent to the subscriber? Or did the subscription get lost, and the broker is not sending publications for that subscription to the subscriber?

If the solution involves distributed publish/subscribe, using clusters or publish/subscribe hierarchies, there are numerous configuration issues that might result in the appearance of a lost message.

If you sent a message with "At least once" or "At most once" quality of service, it is likely that the message you think is lost was not delivered in the way you expected. It is unlikely that the message has been wrongly deleted from the system. It might have failed to create the publication or the subscription you expected.

The most important step you take in doing problem determination of lost messages is to confirm the message is lost. Recreate the scenario and lose more messages. Use the "At least once" or "At most once" quality of service to eliminate all cases of the system discarding messages.

There are four legs to diagnosing a lost message.

1. "Fire and forget" messages working as-designed. "Fire and forget" messages are sometimes discarded by the system.
2. Configuration: setting up publish/subscribe with the correct authorities in a distributed environment is not straightforward.
3. Client programming errors: the responsibility for message delivery is not solely the responsibility of code written by IBM.
4. If you have exhausted all these possibilities, you might decide to involve IBM service.

1. If the lost message had the "Fire and forget" quality of service, set the "At least once" or "At most once" quality of service. Attempt to lose the message again.
   - Messages sent with "Fire and forget" quality of service are thrown away by IBM WebSphere MQ in a number of circumstances:
     - Communications loss and channel stopped.
     - Queue manager shut down.
     - Excessive number of messages.
   - The delivery of "Fire and forget" messages depends upon the reliability of TCP/IP. TCP/IP continues to send data packets again until their delivery is acknowledged. If the TCP/IP session is broken, messages with the "Fire and forget" quality of service are lost. The session might be broken by the client or server closing down, a communications problem, or a firewall disconnecting the session.
2. Check that client is restarting the previous session, in order to send undelivered messages with "At least once" or "At most once" quality of service again.
   a) If the client app is using the Java SE MQTT client, check that it sets `MqttClient.CleanSession` to `false`
   b) If you are using different client libraries, check that a session is being restarted correctly.
3. Check that the client app is restarting the same session, and not starting a different session by mistake.

   To start the same session again, `cleanSession = false`, and the `Mqttclient.clientIdentifier` and the `MqttClient.serverURI` must be the same as the previous session.
4. If a session closes prematurely, check that the message is available in the persistence store at the client to send again.

a) If the client app is using the Java SE MQTT client, check that the message is being saved in the persistence folder; see “Client-side log files” on page 154

b) If you are using different client libraries, or you have implemented your own persistence mechanism, check that it is working correctly.

5. Check that no one has deleted the message before it was delivered.

   Undelivered messages awaiting delivery to MQTT clients are stored in SYSTEM.MQTT.TRANSMIT.QUEUE. Messages awaiting delivery to the telemetry server are stored by the client persistence mechanism; see Message persistence in MQTT clients.

6. Check that the client has a subscription for the publication it expects to receive.

   List subscriptions using WebSphere MQ Explorer, or by using **runmqsc** or PCF commands. All MQTT client subscriptions are named. They are given a name of the form: *ClientIdentifier*:*Topic name*

7. Check that the publisher has authority to publish, and the subscriber to subscribe to the publication topic.

   ```
   dspmqaut -m qMgr -n topicName -t topic -p user ID
   ```

   In a clustered publish/subscribe system, the subscriber must be authorized to the topic on the queue manager to which the subscriber is connected. It is not necessary for the subscriber to be authorized to subscribe to the topic on the queue manager where the publication is published. The channels between the queue managers must be correctly authorized to pass on the proxy subscription and forward the publication.

   Create the same subscription and publish to it using IBM WebSphere MQ Explorer. Simulate your application client publishing and subscribing by using the client utility. Start the utility from IBM WebSphere MQ Explorer and change its user ID to match the one adopted by your client app.

8. Check that the subscriber has permission to put the publication on the SYSTEM.MQTT.TRANSMIT.QUEUE.

   ```
   dspmqaut -m qMgr -n queueName -t queue -p user ID
   ```

9. Check that the IBM WebSphere MQ point-to-point application has authority to put its message on the SYSTEM.MQTT.TRANSMIT.QUEUE.

   ```
   dspmqaut -m qMgr -n queueName -t queue -p user ID
   ```

   See "Sending a message to a client directly" in Configure distributed queuing to send messages to MQTT clients.

Resolving problem: Telemetry (MQXR) service does not start

Resolve the problem of the telemetry (MQXR) service failing to start. Check the WebSphere MQ Telemetry installation and no files are missing, moved, or have the wrong permissions. Check the paths used by the telemetry (MQXR) service locate the telemetry (MQXR) service programs.

The WebSphere MQ Telemetry feature is installed. The IBM WebSphere MQ Explorer has a Telemetry folder in **IBM WebSphere MQ > Queue Managers > *qMgrName* > Telemetry**. If the folder does not exist, the installation has failed.

The Telemetry (MQXR) service must have been created for it to start. If the telemetry (MQXR) service has not been created, then run the **Define sample configuration...** wizard in the Telemetry folder.

If the telemetry (MQXR) service has been started before, then additional **Channels** and **Channel Status** folders are created under the Telemetry folder. The Telemetry service, SYSTEM.MQXR.SERVICE, is in the **Services** folder. It is visible if the Explorer radio button to show System Objects is clicked.

Right click SYSTEM.MQXR.SERVICE to start and stop the service, show its status, and display whether your user ID has authority to start the service.

The SYSTEM.MQXR.SERVICE telemetry (MQXR) service fails to start. A failure to start manifests itself in two different ways:

1. The start command fails immediately.
2. The start command succeeds, and is immediately followed by the service stopping.

1. Start the service

   Result
   The service stops immediately. A window displays an error message; for example:

   ```
   WebSphere MQ cannot process the request because the
   executable specified cannot be started. (AMQ4160)
   ```

   Reason

   Files are missing from the installation, or the permissions on installed files are set wrongly.
   The IBM WebSphere MQ Telemetry feature is installed only on one of a pair of highly available queue managers. If the queue manager instance switches over to a standby, it tries to start SYSTEM.MQXR.SERVICE. The command to start the service fails because the telemetry (MQXR) service is not installed on the standby.

   Investigation
   Look in error logs; see "Server-side logs" on page 152.

   Actions
   Install, or uninstall and reinstall the WebSphere MQ Telemetry feature.
2. Start the service; wait for 30 seconds; refresh the Explorer and check the service status.

   Result
   The service starts and then stops.

   Reason
   SYSTEM.MQXR.SERVICE started the **runMQXRService** command, but the command failed.

   Investigation

   Look in error logs; see "Server-side logs" on page 152.
   See if the problem occurs with only the sample channel defined. Backup and the clear the contents of the *WMQ data directory*\Qmgrs*qMgrName*\mqxr\ directory. Run the sample configuration wizard and try to start the service.

   Actions
   Look for permission and path problems.

Resolving problem: JAAS login module not called by the telemetry service

Find out if your JAAS login module is not being called by the telemetry (MQXR) service, and configure JAAS to correct the problem.

You have modified *WMQ installation directory*\mqxr\samples\LoginModule.java to create your own authentication class *WMQ installation directory*\mqxr\samples\samples\LoginModule.class. Alternatively, you have written your own JAAS authentication classes and placed them in a directory of your choosing. After some initial testing with the telemetry (MQXR) service, you suspect that your authentication class is not being called by the telemetry (MQXR) service.

Note: Guard against the possibility that your authentication classes might be overwritten by maintenance being applied to WebSphere MQ. Use your own path for authentication classes, rather than a path within the WebSphere MQ directory tree.

The task uses a scenario to illustrate how to resolve the problem. In the scenario, a package called security.jaas contains a JAAS authentication class called JAASLogin.class. It is stored in the path C:\WMQTelemetryApps\security\jaas. Refer to Telemetry channel JAAS configuration for help in

configuring JAAS for IBM WebSphere MQ Telemetry. The example, “Example JAAS configuration” on page 173 is a sample configuration.

1. Look in `mqxr.log` for an exception thrown by `javax.security.auth.login.LoginException`.

   See “Server-side logs” on page 152 for the path to `mqxr.log`, and Figure 54 on page 175 for an example of the exception listed in the log.
2. Correct your JAAS configuration by comparing it with the worked example in “Example JAAS configuration” on page 173.
3. Replace your login class by the sample `JAASLoginModule`, after refactoring it into your authentication package and deploy it using the same path. Switch the value of `loggedIn` between `true` and `false`.

   If the problem goes away when `loggedIn` is `true`, and appears the same when `loggedIn` is `false`, the problem lies in your login class.
4. Check whether the problem is with authorization rather than authentication.
   a) Change the telemetry channel definition to perform authorization checking using a fixed user ID. Select a user ID that is a member of the `mqm` group.
   b) Rerun the client app.

      If the problem disappears, the solution lies with the user ID being passed for authorization. What is the user name being passed? Print it to file from your login module. Check its access permissions using IBM WebSphere MQ Explorer, or **dspmqauth**.

Example JAAS configuration

Use the **New telemetry channel** wizard, in WebSphere MQ Explorer, to configure a telemetry channel. The client connects on port 1884, and connects to the `JAASMCAUser` telemetry channel. Figure 48 on page 173 shows an example of the telemetry properties file created by the telemetry wizard. Do not edit this file directly. The channel authenticates using JAAS, using the configuration called `JAASConfig`. Once the client has authenticated, it uses the user ID `Admin` to authorize its access to IBM WebSphere MQ objects.

```
com.ibm.mq.MQXR.channel/JAASMCAUser: \
com.ibm.mq.MQXR.Port=1884;\
com.ibm.mq.MQXR.JAASConfig=JAASConfig;\
com.ibm.mq.MQXR.UserName=Admin;\
com.ibm.mq.MQXR.StartWithMQXRService=true
```

Figure 48. WMQ Installation directory\data\qmgrs\qMgrName\mqxr\mqxr_win.properties

The JAAS configuration file has a stanza named `JAASConfig` that names the Java class `security.jaas.JAASLogin`, which JAAS is to use to authenticate clients.

```
JAASConfig {
  security.jaas.JAASLogin required debug=true;
};
```

Figure 49. WMQ Installation directory\data\qmgrs\qMgrName\mqxr\jaas.config

When `SYSTEM.MQTT.SERVICE` starts, it adds the path in Figure 50 on page 173 to its classpath.

```
CLASSPATH=C:\WMQTelemtryApps;
```

Figure 50. WMQ Installation directory\data\qmgrs\qMgrName\service.env

Figure 51 on page 174 shows the additional path in Figure 50 on page 173 added to the classpath that is set up for the telemetry (MQXR) service.

```
CLASSPATH=;C:\IBM\MQ\Program\mqxr\bin\\..\lib\MQXRListener.jar;
C:\IBM\MQ\Program\mqxr\bin\\..\lib\WMQCommonServices.jar;
C:\IBM\MQ\Program\mqxr\bin\\..\lib\objectManager.utils.jar;
C:\IBM\MQ\Program\mqxr\bin\\..\lib\com.ibm.micro.xr.jar;
C:\IBM\MQ\Program\mqxr\bin\\..\..\java\lib\com.ibm.mq.jmqi.jar;
C:\IBM\MQ\Program\mqxr\bin\\..\..\java\lib\com.ibm.mqjms.jar;
C:\IBM\MQ\Program\mqxr\bin\\..\..\java\lib\com.ibm.mq.jar;
C:\WMQTelemtryApps;
```

Figure 51. Classpath output from runMQXRService.bat

The output in Figure 52 on page 174 shows that the telemetry (MQXR) service has started with the channel definition shown in Figure 48 on page 173.

```
21/05/2010 15:32:12 [main] com.ibm.mq.MQXRService.MQXRPropertiesFile
AMQXR2011I: Property com.ibm.mq.MQXR.channel/JAASMCAUser value
com.ibm.mq.MQXR.Port=1884;
com.ibm.mq.MQXR.JAASConfig=JAASConfig;
com.ibm.mq.MQXR.UserName=Admin;
com.ibm.mq.MQXR.StartWithMQXRService=true
```

Figure 52. WMQ Installation directory\data\qmgrs\qMgrName\errors\mqxr.log

When the client app connects to the JAAS channel, if `com.ibm.mq.MQXR.JAASConfig=JAASWrongConfig` does not match the name of a JAAS stanza in the `jaas.config` file, the connection fails, and the client throws an exception with a return code of `0`; see Figure 53 on page 174. The second exception, `Client is not connected (32104)`, was thrown because the client attempted to disconnect when it was not connected.

```
C:\WMQTelemetryApps>java com.ibm.mq.id.PubAsyncRestartable
Starting a clean session for instance "Admin_PubAsyncRestartab"
Publishing "Hello World Fri May 21 17:23:23 BST 2010" on topic "MQTT Example"
for client instance: "Admin_PubAsyncRestartab" using QoS=1 on address tcp://localhost:1884"
Userid: "Admin", Password: "Password"
Delivery token "528752516" has been received: false
Connection lost on instance "Admin_PubAsyncRestartab" with cause "MqttException"
MqttException (0) - java.io.EOFException
        at com.ibm.micro.client.mqttv3.internal.CommsReceiver.run(CommsReceiver.java:118)
        at java.lang.Thread.run(Thread.java:801)
Caused by: java.io.EOFException
        at java.io.DataInputStream.readByte(DataInputStream.java:269)
        at
com.ibm.micro.client.mqttv3.internal.wire.MqttInputStream.readMqttWireMessage(MqttInputStream.ja
va:56)
        at com.ibm.micro.client.mqttv3.internal.CommsReceiver.run(CommsReceiver.java:90)
        ... 1 more
Client is not connected (32104)
        at
com.ibm.micro.client.mqttv3.internal.ExceptionHelper.createMqttException(ExceptionHelper.java:33
)
        at com.ibm.micro.client.mqttv3.internal.ClientComms.internalSend(ClientComms.java:100)
        at com.ibm.micro.client.mqttv3.internal.ClientComms.sendNoWait(ClientComms.java:117)
        at com.ibm.micro.client.mqttv3.internal.ClientComms.disconnect(ClientComms.java:229)
        at com.ibm.micro.client.mqttv3.MqttClient.disconnect(MqttClient.java:385)
        at com.ibm.mq.id.PubAsyncRestartable.main(PubAsyncRestartable.java:49)
```

Figure 53. Exception thrown connecting com.ibm.mq.id.PubAsyncRestartable

`mqxr.log` contains additional output shown in Figure 53 on page 174.

The error is detected by JAAS which throws `javax.security.auth.login.LoginException` with the cause `No LoginModules configured for JAAS`. It could be caused, as in Figure 54 on page 175, by a bad configuration name. It might also be the result of other problems JAAS has encountered loading the JAAS configuration.

If no exception is reported by JAAS, JAAS has successfully loaded the `security.jaas.JAASLogin` class named in the `JAASConfig` stanza.

```
21/05/2010 12:06:12 [ServerWorker0] com.ibm.mq.MQXRService.MQTTCommunications
AMQXR2050E: Unable to load JAAS config: JAASWrongConfig.
The following exception occurred javax.security.auth.login.LoginException:
No LoginModules configured for JAAS
```

Figure 54. `mqxr.log` - error loading JAAS configuration

Resolving problem: Starting or running the daemon

Consult the IBM WebSphere MQ Telemetry daemon for devices console log, turn on tracing, or use the symptom table in this topic to troubleshoot problems with the daemon.

1. Check the console log.

   If the daemon is running in the foreground, the console messages are written to the terminal window. If the daemon has been started in the background, the console is where you have redirected `stdout` to.
2. Restart the daemon.

   Changes to the configuration file are not activated until the daemon is restarted.
3. Consult Table 7 on page 175:

Table 7. Symptom table

Problem	Suggested solution

The following message is displayed when you start the daemon on Windows:
`The system cannot execute the specified program`
or
`The application has failed to start because its side-by-side configuration is incorrect.`
	Install Microsoft Visual C++ 2008 Redistributable Package.

Two or more daemons or MQTT-capable servers are inter-connected by a bridge or bridges, and the processor is showing excessive load.	There is possibly a message loop, with one or more messages being repeatedly passed from one server to another. Examine the topic parameters in the configuration files. Use more specific topics where possible. Broad wildcard characters in both directions are the most common cause of connection loops.
The bridge is unable to connect to a remote MQTT-capable server that other MQTT clients can connect to.	The remote server might be incompatible with attempts to determine if the remote server is also WebSphere MQ Telemetry daemon for devices. Try setting **`try_private`** to `off` to disable special processing to eliminate message loops.

Table 7. Symptom table (continued)	
Problem	**Suggested solution**
This message is printed when a bridge is configured: `Warning: Connect was not first packet on socket 1888, got CONNACK.`	You have probably configured a bridge to loop back to the local daemon. Loopback is not supported.

Resolving problem: MQTT clients not connecting to the daemon

Clients are not connecting to the daemon, or the daemon is not connecting to other daemons or to a WebSphere MQ telemetry channel.

Trace each MQTT packet sent and received by the daemon.

Set the **`trace_output`** parameter to `protocol` in the daemon configuration file or send a command to the daemon using the `amqtdd.upd` file.

See Transfer messages between the IBM WebSphere MQ Telemetry daemon for devices and IBM WebSphere MQ , for an example of using the `amqtdd.upd` file.

Using the protocol setting, the daemon prints a message to the console describing each MQTT packet it sends and receives.

Notices

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

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

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

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

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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

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

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

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

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

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

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

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

COPYRIGHT LICENSE:

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

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software for use with this program.

This book contains information on intended programming interfaces that allow the customer to write programs to obtain the services of IBM WebSphere MQ.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Important: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks

IBM, the IBM logo, ibm.com®, are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at "Copyright and trademark information"www.ibm.com/legal/copytrade.shtml. Other product and service names might be trademarks of IBM or other companies.

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

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

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

This product includes software developed by the Eclipse Project (http://www.eclipse.org/).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Part Number:

(1P) P/N: