Getting Started Guide

Support Team

Version 2.1, September 3, 2021

Table of Contents

Chapter 1. Introduction

1.1. What Is MQTT.Cool?

MQTT (Message Queuing Telemetry Transport) is an ISO standard publish-subscribe-based messaging protocol. It works on top of the TCP/IP protocol. It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited. The publish/subscribe messaging pattern requires a message broker. MQTT has become the de facto messaging standard for IoT (Internet of Things) solutions and M2M (machine-to-machine) connectivity.

MQTT.Cool is a gateway designed for boosting existing MQTT brokers by extending their native functionalities with new out-of-the-box features.

MQTT.Cool provides architecture extensions, performance extensions, and security extensions to any third-party MQTT broker as detailed below.

1.1.1. Architecture Extensions

Connect to Any MQTT Broker from Anywhere

Connect to your MQTT broker from anywhere on the Internet, even behind the strictest corporate firewalls and proxies, without sacrificing security. No need to fight with firewall rules and change your security policies.

Your MQTT broker will be instantly available through standard Web protocols (HTTP and WebSockets). This means your MQTT broker does not need to support WebSockets because it’s up to MQTT.Cool to re-encode the MQTT protocol into a very efficient and firewall-friendly protocol (called the *Lightstreamer protocol*).

Even if WebSockets are not supported by the network infrastructure (for instance, they might be blocked by a client-side proxy), the connection will automatically work over HTTP, thanks to StreamSense, Lightstreamer’s state-of-the-art implementation of HTTP streaming and HTTP long polling.

What happens if a client gets disconnected, for any reason? No worries—the MQTT.Cool client library will automatically reconnect and re-establish the correct subscriptions.

Develop Web Clients with Our Eclipse Paho-like API

We provide you with a JavaScript library that works in any existing browser, including mobile browsers, as well as Node.js.

The library exposes an Eclipse Paho-like API. Any HTML page can easily become an MQTT client, able to publish and subscribe to/from MQTT topics, irrespective of which MQTT broker you are using. This way, web pages can exchange messages with IoT devices and existing MQTT applications as well as interact with other web pages in real time.

Similarly, any Node.js application will be able to access any MQTT broker and produce/consume

messages.

Access Multiple MQTT Brokers

A single MQTT.Cool instance can connect to different MQTT brokers. For example, it might connect to both a Mosquitto instance and a HiveMQ instance and make these brokers available to the clients. If the same client needs to access both the brokers, then it will be able to do it with a single physical connection to MQTT.Cool because all the traffic is multiplexed over a single link for each client.

1.1.2. Performance Extensions

Scale Up Your MQTT Broker with Massive Fan-Out

Scale to millions of MQTT clients by offloading the fan-out from your existing MQTT broker to MQTT.Cool.

Clients will physically connect to MQTT.Cool, which uses the world-class Lightstreamer engine to handle massively concurrent connections. MQTT.Cool scales horizontally by automatically employing all the available cores; it also scales vertically with multiple instances managed by any common load balancer.

Always Receive Fresh Data with Adaptive Throttling and Conflation

Imagine an IoT sensor that produces hundreds of measurements per second and publishes them on MQTT. You have a web page showing such data in real time.

Now, imagine a user watching that page on a desktop browser with a broadband connection and another user watching the same page on a mobile browser with a bad signal. MQTT.Cool will automatically throttle the data flow for each user, to adapt to any network congestion. It will resample the data on the fly while applying conflation, so that the two users will both see real-time and coherent data but with different update rates.

Get Full Control over Bandwidth and Frequency

In addition to adaptive throttling, each client can explicitly configure a maximum bandwidth for its downstream channel. For example, a client might request to never consume more than 10 kbps. Queuing, resampling, and conflation will be applied automatically to respect the allocated bandwidth.

Similarly, a maximum update frequency can be requested by each client for every fanout subscription. For example, a client might subscribe to a topic specifying that no more than 2 messages per second must be delivered.

1.1.3. Security Extensions

Authenticate Users with Total Flexibility

Typical MQTT authentication is based on username and password only. Furthermore, it can be a nightmare to integrate MQTT authentication offered by a typical MQTT broker with existing

enterprise authentication systems.

MQTT.Cool offers a pluggable authentication system, which is totally independent of the target MQTT broker. Users' authentication is managed by MQTT.Cool via your own integration code and based on the *Hook API*. Not only will your authentication code be able to receive a username and password, but it will be passed full connections details (including client remote IP, user agent, cookies, client-side certificates, etc.).

It is straightforward to develop a Hook that integrates MQTT.Cool with an existing user DB. You can also switch MQTT broker without losing your authentication logic.

Add Fine-grained Authorization

How do you make sure that user A cannot subscribe to topic X and user B cannot publish to topic Y? This is left to each MQTT broker proprietary authorization system (if available at all).

With MQTT.Cool, you can add very fine-grained authorization to any MQTT broker in a completely broker-agnostic way. With the Hook API, any action performed by a user is authorized via a specific callback to your own code. As for authentication, you have total flexibility in defining your security policies, based on your specific needs. Again, you can switch MQTT broker without losing your authorization logic.

Offload TLS/SSL Encryption

MQTT.Cool can take care of encrypting the traffic with the clients, based on TLS/SSL configurable cipher suites and certificates. This way, you can remove the burden of encryption from your MQTT broker and offload it to MQTT.Cool, which uses WSS and HTTPS for the client connections.

Increase Security by Avoiding Public Access to the Broker

You might want to hide or firewall-protect the connection details of your MQTT broker (address and port) and make it reachable from the Internet only through MQTT.Cool, which will reside in the DMZ. This way, you will add a layer of security, preventing the broker from dealing with external and potentially unauthorized connections.

1.2. MQTT.Cool Architecture

MQTT.Cool is a stand-alone server, which can be installed on any machine, be it physical or virtual, either in the cloud or on premises.

MQTT.Cool connects to any existing MQTT broker and acts as a **gateway**. Clients use the provided MQTT.Cool APIs, which are Eclipse Paho-like, to connect to the MQTT.Cool server.

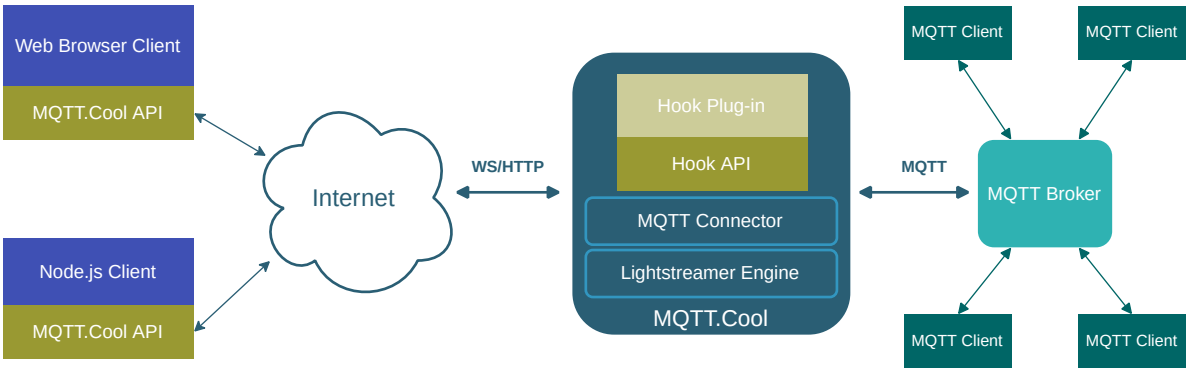

Figure 1. Architecture overview

MQTT.Cool embeds the high-performance Lightstreamer Engine to deliver real-time data through the Internet, achieving low latency and high scalability.

The MQTT Connector can connect to any MQTT broker to send and receive MQTT messages to and from topics and to encapsulate them into Lightstreamer's internal protocol.

The *Hook Plug-in* enables you to add custom authentication and authorization through the provided *Hook API*.

Base knowledge of the MQTT protocol is required to fully understand this guide.

1.3. Quick Start

This section will guide you through the installation of MQTT.Cool to get it up and running in very short time.

To install MQTT.Cool, you can either download it or use the Docker image. Please refer to Section 1.3.8, "Docker Image" if you go for Docker.

1.3.1. Check System Requirements

MQTT.Cool requires a Java Virtual Machine. You can use either OpenJDK or Oracle Java Platform, Standard Edition (either the Server JRE or the JDK). Minimum supported Java version is 8, but newer versions are recommended.

1.3.2. Download and Install

Download the latest MQTT.Cool package for your operating system.

On Mac or Unix-based systems, unpack the tarball:

```
$ tar -xvfz mqtt.cool_X_Y_Z_YYYYMMDD.tar.gz
```

The `mqtt.cool` directory will be created.

On Windows systems, extract the *mqtt.cool_X_Y_Z_YYYYMMDD.zip* archive using your preferred tool into a directory of your choice (a short base path is recommended, e.g., "C:\"). The `mqtt.cool` directory will be created.

1.3.3. Configure

Modify a few factory settings of the MQTT.Cool configuration:

Choose TCP ports

By default, MQTT.Cool listens on TCP ports 8080 and 8888, but you may change them by opening the `mqtt.cool/conf/configuration.xml` file and editing the following elements:

- <***port***> inside the <*http_server*> block
- <***port***> inside the <*rmi_connector*> block.

Set JAVA_HOME

By default, a `JAVA_HOME` variable pointing to your JDK installation is looked up in the environment. If missing, edit the launch script:

- on Mac or Unix-based systems, open the **mc.sh** file under the `mqtt.cool/bin/unix-like` directory
- on Windows systems, open the **mc.bat** file under the `mqtt.cool\bin\windows` directory

and set the value of the `JAVA_HOME` variable according to the Java SE installation available.

1.3.4. Attach an MQTT Broker

Before starting MQTT.Cool, a target MQTT broker to which to connect should be up and running, otherwise no client connections can be accepted.

MQTT.Cool comes with a set of predefined configurations for connecting with local MQTT server instances as well as with the most common publicly accessible brokers. To provide a new custom configuration, open the `mqtt.cool/conf/brokers_configuration.xml` file and add a set of entries similar to the following:

```
<!-- MQTT broker connection parameters for a local instance
     listening on port 1883, aliased by "mybroker". -->
<param name="mybroker.server_address">tcp://localhost:1883</param>
<param name="mybroker.connection_timeout">5</param>
<param name="mybroker.keep_alive">20</param>
```

As you can see, the connection parameters for a specific configuration share a common prefix, which is the *connection alias* to be used by the clients to specify the MQTT broker to use.

Now, start your MQTT broker or ensure it is already running.

1.3.5. Start

To launch MQTT.Cool:

- on Mac or Unix-based systems:

```
$ cd bin/unix-like
$ ./start.sh
```

- on Windows systems: go to the `bin\windows` directory and execute `start.bat`.

1.3.6. Check

To verify whether the server has started correctly, open your browser to `http://localhost:8080` (change the port according to Section 1.3.3.1, “Choose TCP ports”) and watch the Welcome page:

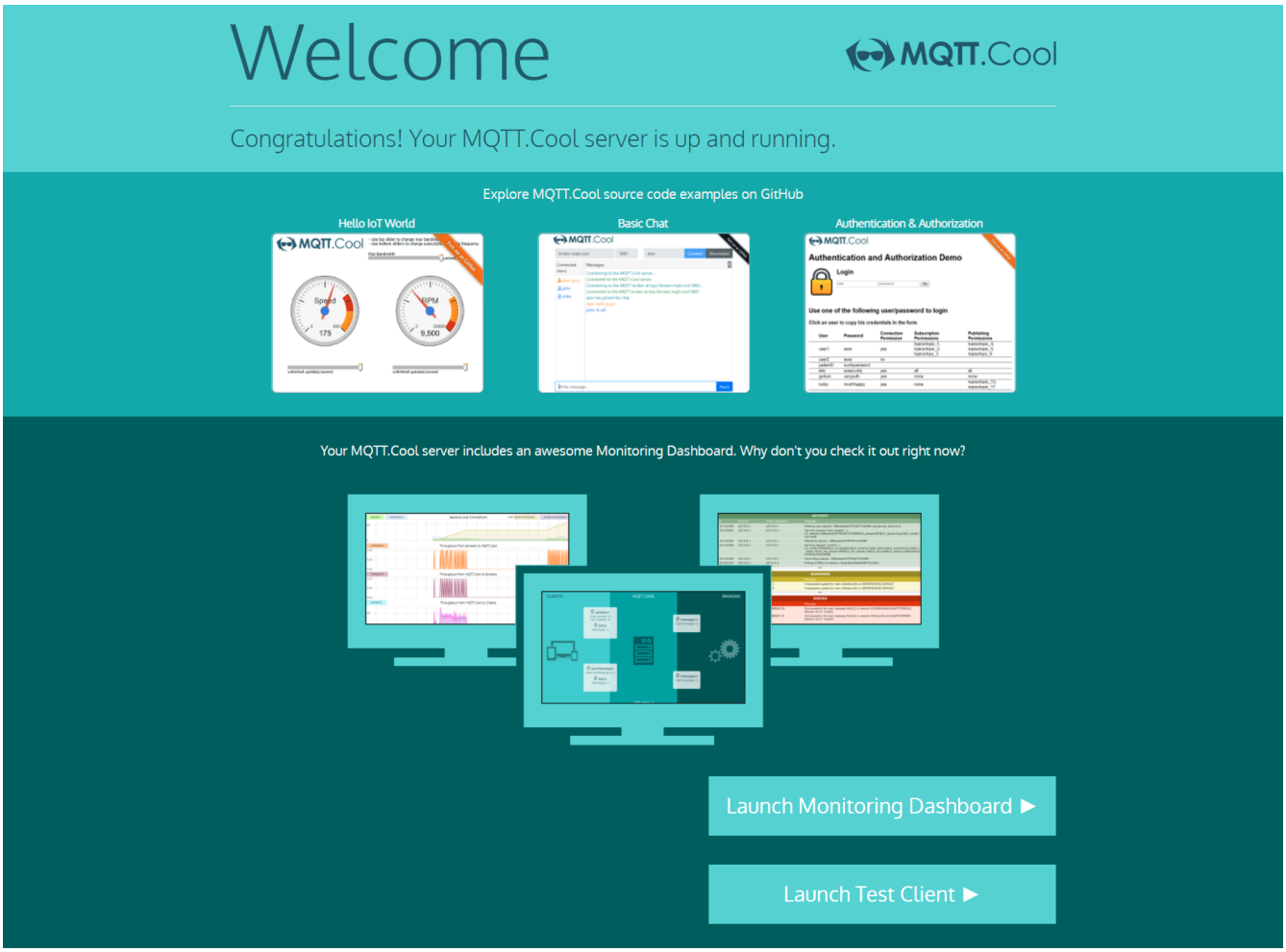

Figure 2. The Welcome page

1.3.7. Test Client

From the Welcome page, you can immediately start using MQTT.Cool by launching the *Test Client*, a very simple and linear GUI (based on the MQTT.Cool API) through which you can test the interaction between the MQTT.Cool server and MQTT brokers.

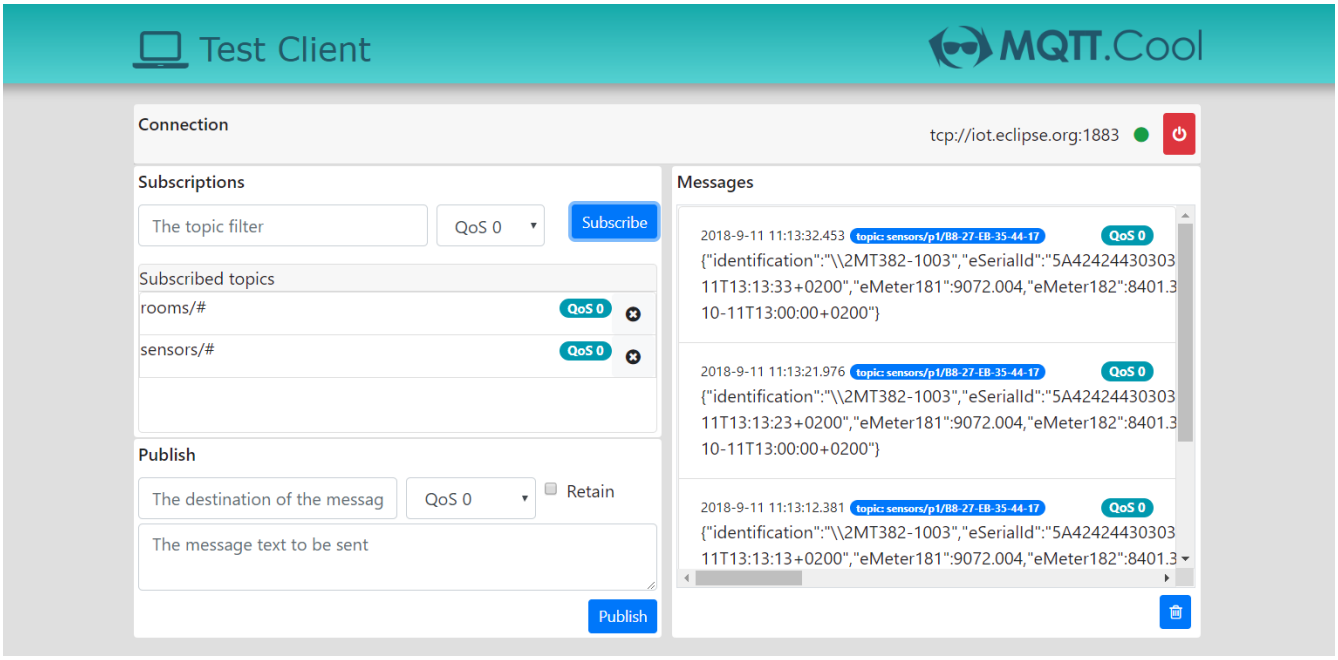

Figure 3. The Test Client

By providing clear and separate sections, aimed at offering an enjoyable user experience, the Test Client allows you to perform all the common operations you may expect from any MQTT client tool, such as:

- Connecting to a broker, either by choosing a connection from a preconfigured list or by creating a new one.
- Publishing a message to a topic name with a specific QoS.
- Subscribing to a topic filter at a specific QoS.
- Displaying the incoming messages relative to the submitted subscriptions.

Being embedded in the tool, the preconfigured connections are not related to the ones provided in the `brokers_configuration.xml` file. Nevertheless, you can change such connections by editing the `brokers.json` file under the `<MQTT.COOL_HOME>/pages/test_client` directory (see `README.txt` in the same path).

The Test Client is a really handy tool you may take benefit of every time you find yourself:

- Testing or debugging a new deployment of the MQTT.Cool server.
- Ensuring new MQTT brokers (either deployed in the same network infrastructure or reachable over the Internet) can be contacted correctly.

1.3.8. Docker Image

A very simple and fast way to play with MQTT.Cool is running the official Docker image:

```
$ docker run --name mc-server -d -p 8080:8080 mqttcool/mqtt.cool
```

Once launched the container, point the browser to http://localhost:8080 to check the Welcome Page.

It is possible to customize practically every aspect of the running instance. In the example below, a specific configuration file is provided to the container:

```
$ docker run --name mc-server -v
/path/to/my_connections.xml:/mqtt.cool/conf/brokers_configuration.xml -d -p 8080:8080
mqttcool/mqtt.cool
```

You may also want to create your own image by deriving the official one. The following is a very basic *Dockerfile* that simply overrides the configuration files with your customized versions (which must be placed in the same folder in this example).

```
FROM mqttcool/mqtt.cool

COPY my_connections.xml /mqtt.cool/conf/brokers_configuration.xml
COPY my_log_conf.xml /mqtt.cool/conf/log_configuration.xml
```

To build the image, launch the `docker build` command:

```
$ docker build -t my-mqttcool .
```

Then run it with:

```
$ docker run --name mc-server -d -p 8080:8080 my-mqttcool
```

See the official page on *Docker Hub* to get more details.

1.4. Hello IoT World Tutorial

Now it is time to "get your hands dirty" through a quick tutorial that lets you build a full end-to-end example.

In the following diagram, the general flow we are going to implement is:

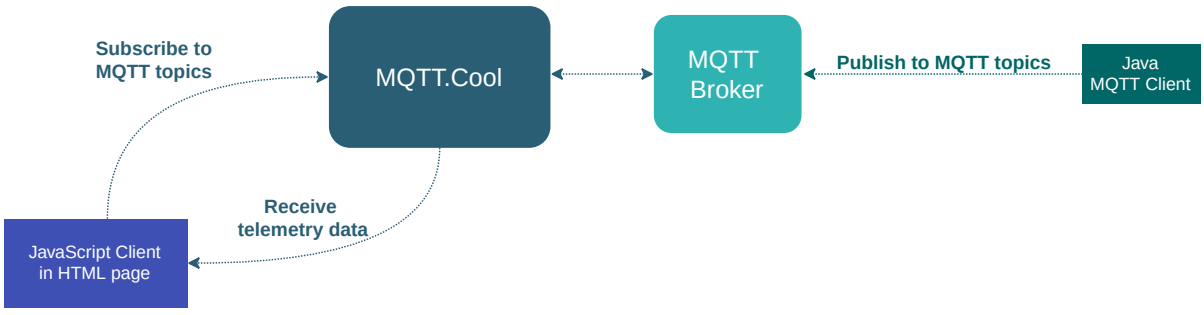

Figure 4. General flow for the example

A JavaScript application running inside the browser subscribes to two MQTT topics in order to *consume* real-time telemetry metrics like *speed* and *engine RPM* from a hypothetical car; upon receipt, data messages are displayed on the HTML page.

Real-time telemetry updates are delivered by a random feed simulator, which is a pure MQTT Java client that connects to the MQTT broker, generates simulated data, and *publishes* them to the target topics at a fixed interval (100 ms).

This example is a *light* version of the Hello IoT World example available on GitHub, where graphical gauges are also employed.

1.4.1. Step 1: Set up an MQTT Broker

First of all, we need an up and running MQTT broker to which clients can connect for messages exchange.

To keep things simple, let's configure a broker on the same machine on which MQTT.Cool has been already installed so that no particular network setup will be needed to make the two servers talk to each other.

We suggest using Mosquitto, as it is very easy to configure and run, besides being one of the most popular and widely used open source MQTT brokers.

For the purpose of the tutorial, there is no requirement to use a specific MQTT broker; therefore, feel free to use your preferred MQTT broker.

Download and install the binary package according to your system.

As the default configuration is OK for our goals (no authentication required and service will listen on default TCP port 1883), simply start the broker:

- on Mac or Linux systems:

```
$ mosquitto
```

- on Windows systems: go to the Mosquitto installation directory and execute `mosquitto.exe`

If you are familiar with Docker, then you might also want to install and run the official image. Please also consider that you could use one of the public test brokers available for development and testing purposes: here an up-to-date list.

1.4.2. Step 2: Attach the MQTT Broker

After having configured and started the broker, you need to ensure that MQTT.Cool can contact it upon a connection request coming from any client.

MQTT.Cool comes with a set of predefined configurations for connecting with local MQTT server instances as well as with the most common publicly accessible brokers. They are defined in the

`mqtt.cool/conf/brokers_configuration.xml` file through sets of entries similar to the following:

```
<!-- MQTT broker connection parameters for a local instance
     listening on port 1883, aliased by "mybroker". -->
<param name="mybroker.server_address">tcp://localhost:1883</param>
<param name="mybroker.connection_timeout">5</param>
<param name="mybroker.keep_alive">20</param>
```

Basically, the connection parameters for a specific configuration share a common prefix, which is the connection alias to be used by the clients to specify the MQTT broker to use More details on this will be provided in Section 2.1, “Static Lookup”.

For the exercise, the following predefined configuration allows MQTT.Cool to connect to the local Mosquitto server; therefore, there is no need to change default settings:

```
<!--
  ====================================
  CONFIGURATION EXAMPLE FOR "Mosquitto"
  ====================================
-->
<param name="mosquitto.server_address">tcp://localhost:1883</param>
<param name="mosquitto.clientid_prefix">mosquitto_client</param>
<param name="mosquitto.connection_timeout">30</param>
<param name="mosquitto.keep_alive">10</param>
```

1.4.3. Step 3: Write the Feed Simulator

As stated, to simulate and publish telemetry information, we will lean on a Java client application that is based on the Eclipse Paho Java Client, a widely used MQTT client library for the JVM.

Let’s start with the code snippet relative to the application entry point:

```
public static void main(String args[]) throws Exception {
    if (args.length == 0) {
      System.err.println("Please specifiy a validr broker url");
      System.exit(1);
    }

    String brokerUrl = args[0];
    MqttClient client = new MqttClient(brokerUrl, "telemetry-feed"); ①
    client.connect(); ②

    ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
    executor.scheduleAtFixedRate(new Feed(client), 0, 100, TimeUnit.MILLISECONDS); ③
}
```

① Create a new client instanced targeted to the MQTT broker running on the host passed via

command line argument; the client is identified as `telemetry-feed`.

② Connect and wait for response.

③ Once connected, schedule a task for generating and publishing telemetry data every 100 ms.

The task is implemented through an instance of the `Feed` class, which produces *fake* data and sends them to the designated MQTT topics. Below the code snippet for the `run` method:

```
public void run()
  long speedKmH = simulateSpeed(); ①
  long rpm = simulateRPM(speedKmH); ②

  try {
      client.publish("telemetry/speed", toBytes(speedKmH), 0, false); ③
      client.publish("telemetry/rpm", toBytes(rpm), 0, false); ④
  } catch (MqttException e) {
      e.printStackTrace();
      executor.shutdown();
  }
}
```

① Calculate the simulated car speed.

② Calculate the simulated car's engine RPM.

③ Publish the speed data to the `telemetry/speed` topic.

④ Publish the RMP data to the `telemetry/rpm` topic.

To recap, here is how the full MQTT client application code should look like:

```
import org.eclipse.paho.client.mqttv3.*;
import java.util.concurrent.*;

/** The feed task for publishing telemetry data at fixed rate to the MQTT broker. */
public class Feed implements Runnable {

  /** Speed limit in Km/h to determine speed variation range */
  private static final int SPEED_LIMIT_KMH = 135;

  /** Base speed limit to determine speed variation range */
  private static final int BASE_SPEED_KMH = 215;

  /** RPM base value */
  private static final int BASE_RPM = 2000;

  /** Fixed thresholds to determine RPM calculation */
  private static final int[] GEAR_THRESHOLDS = { 90, 150, 220, 300, 320 };

  /** Gear ratios to determine RPM calculation */
  private static final int[] GEAR_RATIOS = { 250, 400, 300, 250, 1000, 660 };
```

```
/** Initial simulated speed in Km/h, also to used to save last determined speed */
private long lastSpeedKmH = 130;

/** Reference to the MqttClient instance connected to the MQTT broker */
private MqttClient client;

/** Reference to the scheduler */
private ScheduledExecutorService executor;

Feed(ScheduledExecutorService executor, MqttClient client) {
  this.client = client;
  this.executor = executor;
}

/**
 * Utility method for converting a long value into a string representation to be
 * used as the message payload.
 *
 * @param value a long
 * @return The resultant byte array
 */
static byte[] toBytes(long value) {
  return String.valueOf(value).getBytes();
}

@Override
public void run() {
  long speedKmH = simulateSpeed();
  long rpm = simulateRPM(speedKmH);

  // Publish data to the MQTT broker.
  try {
    client.publish("telemetry/speed", toBytes(speedKmH), 0, false);
    client.publish("telemetry/rpm", toBytes(rpm), 0, false);
  } catch (MqttException e) {
    e.printStackTrace();
    // Stop schedulation in case of any issues.
    executor.shutdown();
  }
}

/**
 * Calculates and returns the simulated speed.
 *
 * @return the simulated speed
 */
long simulateSpeed() {
  // Simulate the speed variation.
  double ratio = (double) (lastSpeedKmH - BASE_SPEED_KMH) / SPEED_LIMIT_KMH;
  int direction = 1;
```

```
    if (ratio < 0) {
      direction = -1;
    }
    ratio = Math.min(Math.abs(ratio), 1);
    double weight = (ratio * ratio * ratio);
    double prob = (1 - weight) / 2;
    if (!(Math.random() < prob)) {
      direction = direction * -1;
    }
    long difference = Math.round(Math.random() * 3) * direction;

    // Get current speed and save it for next simulation.
    long speedKmH = lastSpeedKmH + difference;
    lastSpeedKmH = speedKmH;
    return speedKmH;
  }

  /**
   * Calculates and returns the simulated engine RPM.
   *
   * @param speedKmH the input speed
   * @return the simulated RPM
   */
  long simulateRPM(long speedKmH) {
    // Calculate current RPM.
    long diff = speedKmH;
    int i = 0;
    for (i = 0; i < GEAR_THRESHOLDS.length; i++) {
      if (speedKmH < GEAR_THRESHOLDS[i]) {
        break;
      }
    }
    if (i > 0) {
      diff = speedKmH - GEAR_THRESHOLDS[i - 1];
    }
    return BASE_RPM + GEAR_RATIOS[i] * diff;
  }

  public static void main(String[] args) throws MqttException {
    if (args.length == 0) {
      System.err.println("Please specifiy a validr broker url");
      System.exit(1);
    }

    // Create a client connection to the MQTT broker running at the specified url
    String brokerUrl = args[0];
    MqttClient client = new MqttClient(brokerUrl, "telemetry-feed");
    client.connect();

    // Once connected, generate and publish simulated telemetry data every 100 ms
    ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
```

```
    executor.scheduleAtFixedRate(new Feed(executor, client), 0, 100,
TimeUnit.MILLISECONDS);
  }
}
```

From the development folder under which you are going to build the feed example, execute the following easy steps (shown for a Linux system):

- Copy the code above into a new Java source file called `Feed.java`.
- Download the Paho Java Client JAR file into a new `lib` directory.
- Compile the source code:

```
$ javac -cp lib/org.eclipse.paho.client.mqttv3-1.2.1.jar Feed.java
```

- Then start the Feed:

```
$ java -cp .:lib/org.eclipse.paho.client.mqttv3-1.2.1.jar Feed tcp://localhost:1883
```

From now on, the messages are periodically published to the MQTT broker.

As shown by the full example on GitHub, using a dependency management tool like Maven is a more practical way to include the Paho Java Client in your projects.

1.4.4. Step 4: Make the JavaScript Client

Let's focus now on the most exciting part, which will lead us to run our MQTT.Cool JavaScript client application!

Once again, because we aim to make our life easier, we are going to embed the client code into a very simple HTML page as follows:

```
<html>
<head>
  <title>Hello IoT World Tutorial</title>
  <script src="https://unpkg.com/mqtt.cool-web-client/dist/mqtt.cool.js"></script> ①

  <script>
    mqttcool.openSession('http://localhost:8080', { ②

      onConnectionSuccess: function(mqttCoolSession) {
        var mqttClient = mqttCoolSession.createClient('mosquitto'); ③

        mqttClient.onMessageArrived = onMessageArrived; ④

        mqttClient.connect({ ⑤
          onSuccess: function() {
            mqttClient.subscribe('telemetry/speed');
            mqttClient.subscribe('telemetry/rpm'); ⑥
          }
        });

        function onMessageArrived(message) {
          var dest = message.destinationName;
          var rowId = dest.split('/', 2)[1]; ⑦
          document.getElementById(rowId).innerHTML = message.payloadString; ⑧
        }
      }
    });
  </script>
</head>
<body>
  <div>
    <div style="float: left;margin-right: 10px;">Speed</div><div id="speed"></div>
    <div style="float: left;margin-right: 10px;">RPM</div><div id="rpm"></div>
  </div>
</body>
</html>
```

① Load the MQTT.Cool JavaScript library file from the `unpkg` service.

② Connect to the local MQTT.Cool server, listening on TPC port `8080`.

③ Upon successful connection, make a new `MqttClient` instance, ready to be connected to the Mosquitto server as configured in `brokers_configuration.xml`.

④ Set the function callback to be triggered on messages arriving.

⑤ Connect to Mosquitto.

⑥ Upon successful `CONNACK`, subscribe to the designated MQTT topics.

⑦ Once a new message arrives, elaborate its topic name in order to retrieve the identifier relative to the `<div>` element to be updated.

⑧ Update the `<div>` element with the message payload.

The `body` element simply contains the two `div` objects referenced by the JavaScript code and periodically rendered with published telemetry data. Their identifiers match the relevant part of the topics (e.g., `telemetry/<metric>`), so that it is trivial to dispatch any message payload to its pertinent visual object.

Finally, load the HTML page into the web browser, which should display something similar to the following:

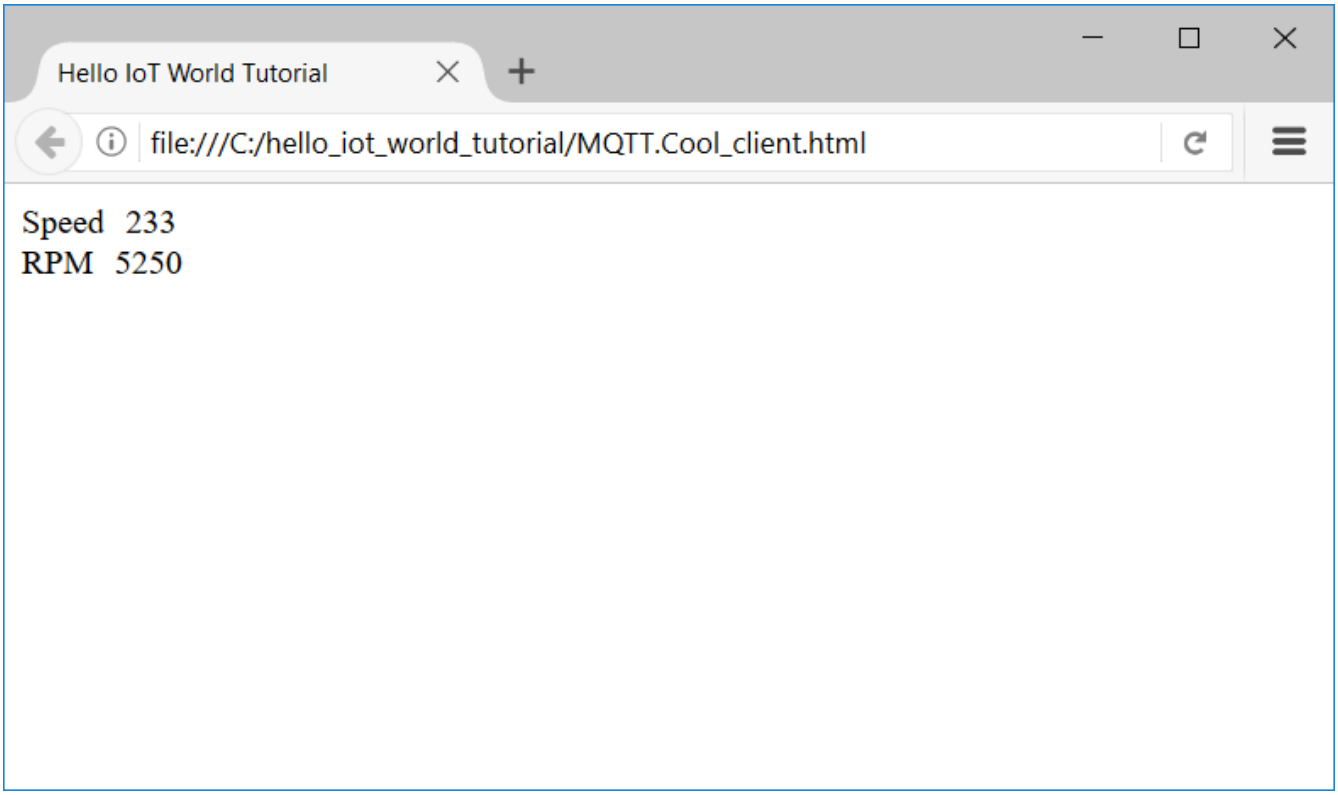

Figure 5. Example running into web browser

Just two last notes:

- The proposed tutorial is a gentle introduction to the entire MQTT.Cool ecosystem; hence, most of the features have not been covered here. In the next sections, you will be provided with full information about server side configuration and client side development.
- As already anticipated, the discussed end-to-end example is a simplified version of the Hello IoT World demo available on GitHub, which shows you a more sophisticated usage of the MQTT.Cool JavaScript library and other "cool" things…

Final recommendation: enjoy MQTT messaging with MQTT.Cool!

Chapter 2. Addressing MQTT Brokers

In this chapter, we will discuss two different approaches through which a client can connect to an MQTT broker:

1. **Static Lookup**, by providing a reference to a static configuration already supplied in MQTT.Cool.
2. **Dynamic Lookup**, by providing an explicit *URI* of the MQTT broker.

2.1. Static Lookup

To activate *Static Lookup*, a specific configuration for each addressable MQTT broker has to be provided in the `brokers_configuration.xml` file. A configuration is given by the set of connection parameters to be used for establishing the communication from the MQTT.Cool server process to the target broker, and is uniquely identified by a *connection alias*, which is the common prefix embedded in each parameter name.

More specifically, you will define a configuration by providing entries in the following form:

```
<param name="<connection_alias>.<connection_param_1>">.../param>
<param name="<connection_alias>.<connection_param_2>">.../param>
...
<param name="<connection_alias>.<connection_param_N>">.../param>
```

where:

- <**connection_alias**> is the configuration identifier
- <**connection_param_J**> is the J_th connection parameter to be set for this configuration and which can be one of the following:
  - `server_address`, the address of the MQTT broker to connect to
  - `clientid_prefix`, the Client Id prefix to be used for *shared connections* as will be detailed in Section 3.6.3, “Shared End-to-End Connection”
  - `connection_timeout`, the connection timeout
  - `keep_alive`, the keep alive interval
  - `username`, the username for authenticating with the MQTT broker
  - `password`, the password for authenticating with the MQTT broker
  - `will_message`, the Will Message payload
  - `will_message_format`, the format of the Will Message payload
  - `will_topic`, the Will Message topic name
  - `will_qos`, the Will Message Quality of Service
  - `will_retain`, the Will Message retain flag.

See Appendix A, *Connection Parameters* for a more in-depth description of each parameter.

For example, a configuration for a local *Mosquitto* instance could be provided by defining all the related parameters under the connection alias `mosquitto` as follows:

```
<param name="mosquitto.server_address">tcp://localhost:1883</param>
<param name="mosquitto.connection_timeout">5</param>
<param name="mosquitto.keep_alive">10</param>
...
```

The clients that want to connect to an MQTT broker for which a static configuration has been provided on the server side, simply supply the corresponding connection alias. MQTT.Cool *resolves* such an alias by looking up the target configuration and then connects to the broker running on the host address, indicated by `server_address`, using all the other connection settings.

To get back to the previous example, when a client connects by providing the alias `mosquitto`, MQTT.Cool will connect to the MQTT broker running on localhost and listening on port 1883 (`mosquitto.server_address`), using a connection timeout of 5 seconds (`mosquitto.connection_timeout`) and a keep alive interval of 10 seconds (`mosquitto.keep_alive`).

2.1.1. The Hook Variant

Static Lookup may also be extended by a plugged Hook, which may be programmed to resolve the provided connection alias as will be further detailed in Section 4.2.5.1, “Providing Broker Configuration”: when no configurations exist in `brokers_configuration.xml`, MQTT.Cool will ask the Hook to supply a valid one.

2.1.2. Client Side Settings

The clients have the opportunity to customize the following connection settings, which will override the ones defined on the server side:

- username
- password
- Will Message and related settings

The Clean Session flag is **always** provided by clients.

2.1.3. Final Considerations

As you have seen so far, Static Lookup offers several major benefits:

- **Reduced complexity**: the clients do not have to deal with any MQTT connection settings (though they could provide specific overriding values).
- **Increased security**: URLs and ports are never specified.

- **Access restriction**: only configured MQTT brokers can be reached.
- **Increased flexibility**: MQTT broker host addresses and other parameters can change without affecting the client side deployed code.
- **Multiple settings**: several configurations could be defined for the same MQTT broker in order to meet different requirements coming from different kinds of client applications.

2.2. Dynamic Lookup

Dynamic Lookup is triggered when the clients connect to an MQTT broker whose connection parameters have not been statically configured on the server side, by supplying an explicit *URI* of the target host.

The provided URI has to be expressed in one of the following forms:

- `tcp://<mqtt_broker_address>:<mqtt_broker_port>`
- `mqtt://<mqtt_broker_address>:<mqtt_broker_port>`
- `mqtts://<mqtt_broker_address>:<mqtt_broker_port>` (for secure connections)
- `ssl://<mqtt_broker_address>:<mqtt_broker_port>` (for secure connections)

MQTT.Cool will bypass any provided static configuration and will connect to the MQTT broker running on the host at the supplied address; in addition, the following connection settings will be used:

- Connection timeout: 5 seconds
- Keep alive interval: 30 seconds

2.2.1. Client Side Settings

As for Static Lookup, the clients may specify other connection settings:

- username
- password
- Will Message along with relative settings

Once again, the Clean Session flag is provided only on the client side.

2.2.2. Final Considerations

By employing Dynamic Lookup, you will lose all the benefits described for Static Lookup; however, some use cases could take advantage of this feature, for example:

- To prototype a client application for which a backend infrastructure has not yet been consolidated.
- For client test tools that let users decide the MQTT broker with with wich to connect.
- To gather and compare performance metrics between an MQTT broker running inside your

own infrastructure (e.g., data center, cloud, etc…) and an external one.

The *Test Client* (seen in Section 1.3.7, “Test Client”) uses Dynamic Lookup to connect to brokers.

Chapter 3. Client Application Development

In this chapter, we will give a general overview on how to create client applications that interact with any MQTT broker through the mediation of MQTT.Cool.

3.1. The SDKs

You can use two different SDKs to develop JavaScript client applications:

- **SDK for Web Clients** for the development of clients running inside the web browser.
- **SDK for Node.js Clients** for the development of clients running on the Node.js runtime.

The SDKs provide a unified API that is designed to be formally equivalent to the *Eclipse Paho JavaScript Client* as much as possible, thus allowing you to seamlessly migrate existing client applications.

More SDKs for different client side technologies (including Android and iOS) are currently under development and will be available soon.

3.1.1. Deployment Architecture

Before starting to develop your web application, you should evaluate the most appropriate deployment architecture to employ.

Even though MQTT.Cool comes with an internal web server that can serve static content, normally this solution should be taken into consideration only for the purpose of test and demo, as it allows to easily run out-of-the-box prototypes and samples.

On the contrary, for production scenarios, it is highly recommended to deploy an architecture similar to the following:

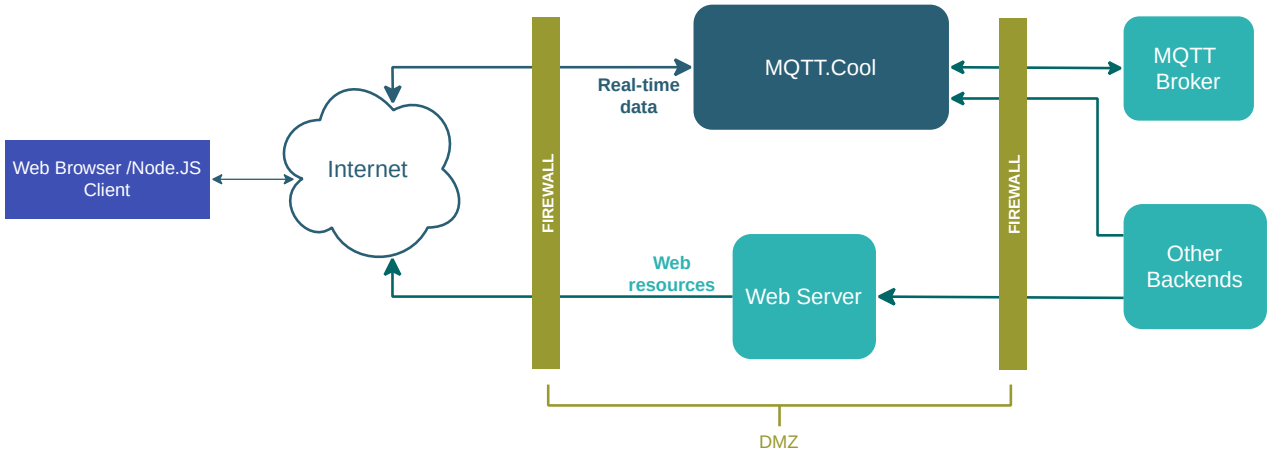

Figure 6. Web deployment architecture

where an external web server (e.g., Apache, nginx, IIS, or any other server of your choice) is used to supply web resources, whereas MQTT.Cool is in charge of supporting the end-to-end

communication between clients and the MQTT broker; moreover, both the MQTT.Cool server and the web server are deployed together in the *DMZ* network while the MQTT broker is protected by a second firewall. Of course, both servers can be clustered for fail-over and load balancing.

Let's explore now how to use the offered SDKs.

3.2. SDK for Web Clients

The SDK for Web Clients enables any HTML page to act as an *MQTT client*; that is, ready to send and receive real-time MQTT messages to/from any MQTT broker connected to the MQTT.Cool server.

3.2.1. Installation

You have different options to install the JavaScript library for Web Clients, with or without `npm`:

npm

The library is available as `npm` package; therefore, you can download and install it through the usual command:

```
$ npm install mqtt.cool-web-client
```

This installation method is strongly suggested when you are going to build web applications that depend on many different JavaScript libraries. In fact, this way you can leverage tools such as Webpack or Browserify to bundle all your modules together, as recommended by modern (and *cool*) JavaScript best practices.

Alternatively, you could simply include the downloaded library with a `<script>` tag by pointing to the installation folder:

```
<html>
<head>
  <script src="./node_modules/mqtt.cool-web-client/dist/mqtt.cool.js"></script>
  ...
</html>
```

Because including `node_modules` is most likely to be a non-efficient way to load the library, especially from a production perspective, you can manually copy the JavaScript file into a more appropriate location under your project layout (`./js/lib` in the example below):

```
<html>
<head>
  <script src="./js/lib/mqtt.cool.js"></script>
  ...
</html>
```

CDN

As soon the last version of the library is published to the `npm` repository, it will be immediately available through `unpkg`, to which you can point directly in the `script` tag as follows:

```
<html>
<head>
  <script src="https://unpkg.com/mqtt.cool-web-client/dist/mqtt.cool.js"></script>
  ...
</html>
```

If you want to get a specific library version, then include it in the url:

```
<script src="https://unpkg.com/mqtt.cool-web-client@1.2.4/dist/mqtt.cool.js"></script>
```

Have a look at the `unpkg` home page for more detailed information on other query parameters you may supply.

Alternatively, because `unpkg` offers a free CDN with no guarantee on the quality of service, you might want to host the library yourself or host it on another CDN. To do this, simply download the library from https://unpkg.com/mqtt.cool-web-client/dist/mqtt.cool.js.

3.2.2. The UMD Pattern

The library implements the UMD pattern, which means that its API objects can be exposed indifferently as:

- Global objects
- AMD modules
- CommonJS modules

This way, the same JavaScript file can be used according to your preferred development style, as will be shown in the following sub-sections.

Global Objects

When loading the library as illustrated above, the following two objects will be registered as *global*, attached to the `mqttcool` namespace:

- `openSession`: the library entry point, from which other local scoped objects can be created to interact with MQTT brokers; this function is always required.
- `Message`: the class to be used for encapsulating MQTT messages; normally it is only required when the clients need to send new messages.

Starting from Section 3.4, "The MQTT.Cool Connection Pattern", you will be provided with full details of these concepts.

```
<script src='mqtt.cool.js'></script>
<script>
  mqttcool.openSession(...);
  ...
  var message = new mqttcool.Message(...);
  ...
</script>
...
```

The library allows you to customize the namespace by using the custom attribute `data-mqttcool-ns` as follows:

```
<script src='mqtt.cool.js' data-mqttcool-ns='my.custom.namespace'></script>
<script>
  my.custom.namespace.openSession(...);
  ...
  var message = new my.custom.namespace.Message(...);
  ...
</script>
...
```

Finally, if you are an *old school* guy, you might decide to remove the namespace by setting `data-mqttcool-ns` to an empty string:

```
<script src='mqtt.cool.js' data-mqttcool-ns=''></script>
<script>
  openSession(...);
  ...
  var message = Message(...);
  ...
</script>
...
```

Consider that this approach leads to a notable drawback, as you could easily get name collisions in your application; therefore, you must ensure that no other code or library on your page declares a global object having the same name as the ones used in the library.

AMD

To use the API objects as an *AMD*-compliant module, you need an *AMD Loader*: you might want to use http://requirejs.org as do most modern JavaScript libraries.

The following example shows a basic usage where, after including `require.js` and mqtt.cool.js, the

`require` function is invoked, listing explicitly dependencies on the API objects.

```
<html>
<head>
  <script src="./js/lib/require.js"></script>
  <script src="./js/lib/mqtt.cool.js"></script>
  ...
  <script>
    require(['mqttcool/openSession', 'mqttcool/Message'], function(openSession,
Message) {
      openSession(...);
      ...
    });
  </script>
</head>
...
</html>
```

In this case, you may also use a different namespace and even remove it, this time through an explicit custom configuration of `require.js`:

```
<html>
<head>
  <script src="./js/lib/require.js"></script>
  <script>
    requirejs.config({
      // Provide a configuration object to the internal MQTT.Cool library configurator
      config: {
        'mqttcool': {
          'ns': 'my.namespace'  // Set the 'ns' custom attribute to your namespace (or
'' for no namespace)
        }
      }
    })
  </script>
  <script src="./js/lib/mqtt.cool.js"></script>
  <script>
    require(['my.namespace/openSession', 'my.namespace/Message'],
      function(openSession, Message) {
        ...
      }
    );
  </script>
</head>
...
</html>
```

3.2.3. CommonJS

Even if technically compatible with any CommonJS environment like Node.js, the library has been expressly designed to work in the context of a web browser. But don't worry: as already anticipated, a dedicated SDK is available!

3.3. SDK for Node.js Clients

The *SDK for Node.js Clients* allows you to develop and execute JavaScript applications acting as *MQTT* clients for Node.js, by using the same API already provided by the *SDK for Web Clients* (even if a few differences exist, as will be detailed in the next sections).

All you have to do is set up your development environment by installing the package through `npm`:

```
$ npm install mqtt.cool-node-client --save
```

The additional `--save` flag will update your `package.json` (if any) while installing the module.

Then you can start writing your application by loading `mqtt.cool-node-client` as follows:

```
// Access the library
const mqttcool = require('mqtt.cool-node-client');

// Start a connection
mqttcool.openSession(...);
...
// Create a new Message instance
var message = new mqttcool.Message(...);
```

As you can see, by leveraging the semantics of Node.js's `require()` function, you can load the only two concrete API objects needed to develop a client application.

3.4. The MQTT.Cool Connection Pattern

Any client application needs to perform a few basic operations in order to establish communication with an MQTT broker.

The main connection pattern is shown in the following code snippet:

```
mqttcool.openSession('http://my.server.company:8080', { ①

  onConnectionSuccess: function(mqttCoolSession) { ②

    var mqttClient = mqttCoolSession.createClient('mybroker', 'myclientid'); ③

    mqttClient.connect(); ④
  }

});
...
```

① Open a new session against the MQTT.Cool server running at the specified address (refer to the inline documentation of both `http_server` and `https_server` elements in `<MQTT.COOL_HOME>/conf/configuration.xml` for insights about TCP ports configuration).

② Upon successful connection, the `onConnectionSuccess` callback function is invoked.

③ Through the `mqttCoolSession` parameter (which implements the `MQTTCoolSession` interface), create a new `MqttClient` instance that is configured to connect to the MQTT broker aliased by *mybroker*, using *myclientid* as client identifier.

④ Connect to the MQTT broker.

Let's highlight some important variants of the main pattern:

- You can open a session to the MQTT.Cool server also specifying username and password, which will be checked by the plugged Hook, if any (as will be detailed in Section 4.2.4, "Authorizing Session Opening"). For example:

```
mqttcool.openSession('http://my.server.company:8080', 'my_cool_user',
'my_cool_password', {
  ...
  }
);
```

See the `openSession` API reference for a complete description of all allowed invocation forms.

- As already mentioned in Section 2.1, "Static Lookup", normally, you will use an alias to statically look-up the connection settings relative to the MQTT broker with wich to connect; but you could also lean on Dynamic Lookup (see Section 2.2, "Dynamic Lookup") to bypass any static configuration and connect to the supplied URI, as in the following example:

```
...
var mqttClient = mqttCoolSession.createClient('tcp://other.mqtt.broker:1883',
'myclientid');
...
```

- An `MqttClient` instance could also be created without a client identifier, thus enabling a *shared connection* (which will be discussed later) as follows:

```
...
var mqttClient = mqttCoolSession.createClient('tcp://other.mqtt.broker:1883');
...
```

- You could provide a set of connection options to be used for connecting to the target MQTT broker:

```
var connectOptions = { ... };
mqttClient.connect(connectOptions);
```

3.5. The MQTTCoolSession Interface

As you have seen so far, a fresh `MQTTCoolSession` instance is provided through the `onConnectionSuccess` callback function once a new connection to MQTT.Cool has been successfully established.

Such a connection, called the *MQTT.Cool connection*, is managed by `MQTTCoolSession` to handle the bidirectional communications between every `MqttClient` instance it can create and the target MQTT.Cool server process in order to support the end-to-end communications with MQTT brokers. Every `MQTTCoolSession` object is bound to the MQTT.Cool connection established through an `openSession` call.

Multiple invocations of `openSession` with the same server address will result in providing distinct `MQTTCoolSession` instances, which will handle their own underlying connections independently of each other, thus letting you manage as many separate sessions as you want.

In addition to the `onConnectionSuccess` callback, other handlers are provided to be notified of events related to the life cycle of an `MQTTCoolSession` object. These are defined through the `MQTTCoolListener` interface, which is formally required as the last argument to be passed to the `openSession` function.

A complete handling of all expected callbacks involves a slight extension of the main connection pattern:

```
mqttcool.openSession(..., {

  onLsClient: function(lsClient) { ... },

  onConnectionSuccess: function(mqttCoolSession) { ... },

  onConnectionFailure: function(errorType, errorCode, errorMessage) { ... }

});
```

where:

- `onLsClient` is the first fired event when the embedded *Lightstreamer Session* is initialized, but before actually opening the connection to MQTT.Cool (see Appendix B, *Access the Lightstreamer Client API*).
- `onConnectionFailure` is invoked if the connection to MQTT.Cool cannot be established and, as consequence of this, no `MQTTCoolSession` objects can be activated.

Connection failures may also include any authorization issue raised by the plugged Hook, as will be discussed in Section 4.2.4, "Authorizing Session Opening".

3.6. The MqttClient Interface

The `MqttClient` interface is used to manage the communication with any MQTT broker through a set of methods that allow you to:

- Open a connection.
- Send messages.
- Subscribe to topics for receiving messages.
- Unsubscribe from topics.
- Manage connection and authorization issues.
- Disconnect.

An instance of `MqttClient` enables an *end-to-end* virtual connection to the broker as if there was a direct link between the two endpoints.

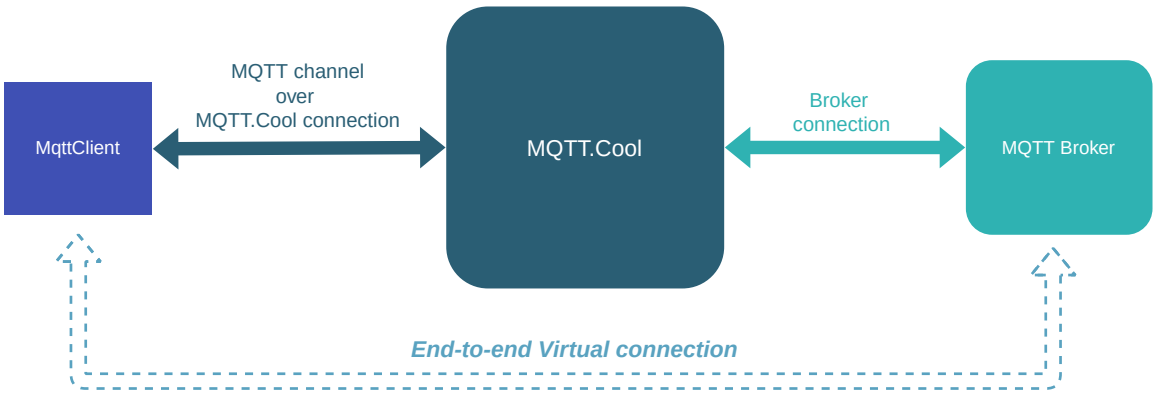

Figure 7. End-to-end virtual connection between MqttClient and MQTT Broker

As depicted in the above image, an end-to-end virtual connection comprises the following two distinct links:

1. **MQTT channel**, which is the logical channel established over the physical MQTT.Cool connection between `MqttClient` and the MQTT.Cool server.
2. **broker connection**, which is the physical MQTT connection between the MQTT.Cool server and

the MQTT broker.

In an end-to-end connection, MQTT.Cool takes the role of a real MQTT server proxy, as it acts as an intermediary for the MQTT Control Packets wrapped into *MQTT.Cool Protocol* messages and transported over the MQTT channel, as well as for the MQTT Control Packets transported *as is* over the broker connection.

3.6.1. Multiplexed MQTT Channels

The client library transparently implements *multiplexed* MQTT channels as all the `MqttClient` instances created from the same `MQTTCoolSession` object establish a new channel over the same MQTT.Cool connection.

Though they are combined into a common physical connection, multiplexed channels enable separate and isolated end-to-end communications, thus allowing you to:

1. Reduce the overhead due to connection setup.
2. Optimize usage of physical connections.

By leveraging multiplexed channels, you can make `MqttClient` instances connect to single or multiple brokers. The following picture shows an example where three `MttClient` instances are establishing independent channels to set up their own end-to-end connection to the target MQTT broker.

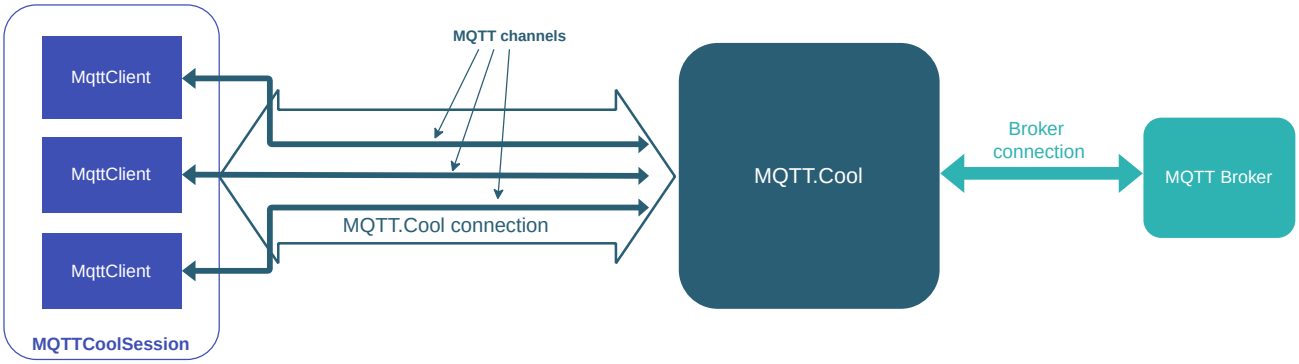

Figure 8. Multiplexed MQTT channels

Multiplexing is a very efficient way to manage the communications with the MQTT.Cool server and MQTT brokers. On the other hand, there could be circumstances where this is not the desired behavior; for example, if you want to avoid that an MQTT.Cool connection outage affects all engaged MQTT channels.

To prevent a channel from getting multiplexed, you have to initiate every single connection only from `MqttClient` instances with each one generated by separate `MQTTCoolSession` objects, thus enforcing each MQTT.Cool connection to host a single MQTT channel.

Let's focus now on broker connections. Based on the way they are managed on the server side, an *end-to-end* connection can be of two types, *dedicated* or *shared*, as will be detailed in the next two sections.

3.6.2. Dedicated End-to-End Connection

For each `MqttClient` that is provided with a valid client identifier at the time of creation through `MQTTCoolSession`, MQTT.Cool establishes on the server side a dedicated broker connection (as defined previously) devoted to carrying all the messages to be exchanged with the target MQTT server.

In the following deployment example, four dedicated broker connections are activated to support the same number of `MqttClient` instances (and relative MQTT channels).

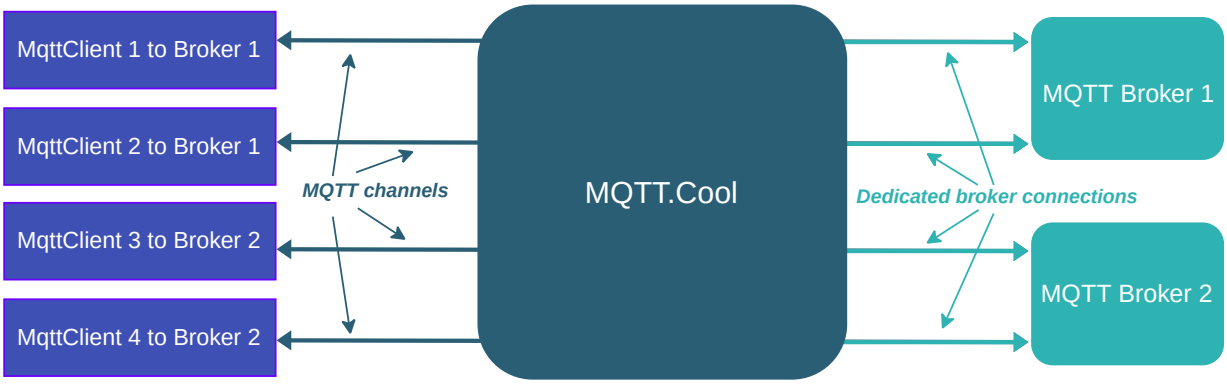

Figure 9. Dedicated end-to-end connections

Because each broker connection uses the client identifier as sent by the client, if another `MqttClient` object or any other generic external MQTT client connects to the same broker using an identical client identifier, the connection will be closed as per the *MQTT Protocol Specifications*.

By employing a dedicated end-to-end connection, MQTT.Cool guarantees full support of the following features:

- QoS levels 0, 1, and 2
- persistence of the Session state
- Will Message

The MQTT.Cool server does not take any active role in the management of the session state, which is, on the contrary, maintained exclusively by the two ends of the connection: the MQTT broker and `MqttClient`.

Life-Cycle Events

In a dedicated end-to-end connection, the life-cycle events relative to `MqttClient`, MQTT channel, and a dedicated broker connection are tied together, namely:

- A connection request initiated by `MqttClient` (which means a newly created MQTT channel) will be propagated up to the MQTT broker through the establishment of a new dedicated broker connection.
- A disconnection request initiated by `MqttClient` (which means the closure of the MQTT channel)

will be propagated up to the MQTT broker, which in turn will close the dedicated broker connection.

- An interruption of the dedicated broker connection (due to any network issue or to a problem in the MQTT broker process) will be propagated back to `MqttClient`, which in turn will disconnect from MQTT.Cool by closing the MQTT channel.
- An interruption of the MQTT.Cool connection on the client side (due to any network issue or to an explicit `MQTTCoolSession` closing) will be propagated up to the MQTT broker by shutting down the dedicated broker connection.
- Any MQTT.Cool server failure will cause the interruption of both the dedicated broker connection on the server side and the MQTT.Cool connection on the client side.

Any issue on the MQTT.Cool connection detected on the client side will cause the `MqttClient` instance to try a reconnection (as you will see in Section 3.12, “Reconnecting”).

3.6.3. Shared End-to-End Connection

A shared end-to-end connection is realized when MQTT.Cool holds a single broker connection that will be shared among all those `MqttClient` instances (even hosted on different network locations) with the following common characteristics:

1. No client identifier has been passed at the time of creation from `MQTTCoolSession`.
2. Connect to the same MQTT broker, along with identical username and password, if any (by providing a set of connection options, as will be detailed later).

From the following diagram, you can see how the MQTT.Cool server *logically* joins together separate MQTT channels targeted to the same MQTT broker into single broker connections, namely:

- The MQTT channels initiated by `MqttClient` 1 and 2 joined into the broker connection to "MQTT Broker A".
- The MQTT channels initiated by `MqttClient` 3 and 4 joined into the broker connection to "MQTT Broker B".

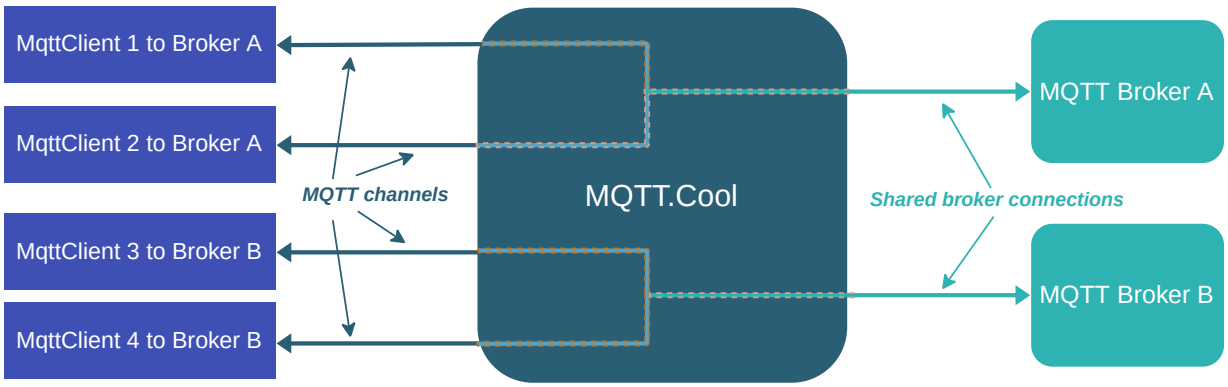

Figure 10. Shared end-to-end connections

By exploiting this mechanism, MQTT.Cool drastically reduces the number of newly created MQTT connections to the broker, with the purpose of optimizing the server side resources. From the above example, the number of physical MQTT connections required to serve all the `MqttClient` instances is halved with respect to the *dedicated* scenario.

While joining them into shared broker connections, the MQTT.Cool server takes into consideration only activated MQTT channels, regardless of the MQTT.Cool connections on the client side into which they have been multiplexed.

Furthermore, all the subscriptions to the same topic filter will be logically *merged* into one single MQTT subscription, which is actually activated on the shared broker connection. This enables offloading the fan-out to MQTT.Cool, as it will be responsible for propagating messages flowing on the MQTT channel up to all subscribing clients. We define such kind of subscriptions *fanout subscriptions*.

The picture below shows an example in which a message targeted to `cool/topic` is initially published by an external MQTT client to the MQTT broker.

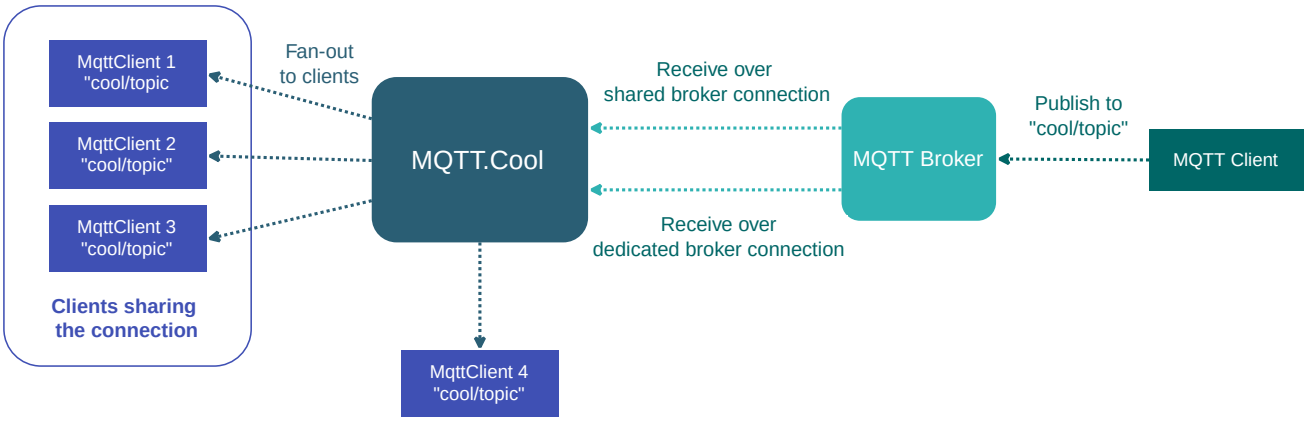

Figure 11. Fanout subscriptions

MQTT.Cool receives the message over the shared broker connection established to support the `MqttClient` instances 1, 2, and 3 (all connected to the same broker), and also over the dedicated one, which has been activated to allow communications to/from the `MqttClient` instance 4. The message is finally relayed to all clients. In the case of a shared connection, instead of sending a copy of the message to each connected client individually, the MQTT broker has delegated such heavy task to MQTT.Cool, thus releasing its own resources to serve other clients and dispatch other messages.

Lastly, before using a shared connection, take into consideration the following constraints:

- QoS levels 1 and 2 are only partially supported as will be detailed in the next sections.
- As stated before, the ClientId cannot be specified explicitly on the client side; on the contrary, it will be generated by appending a random string to the *clientid_prefix* parameter to be configured in `brokers_configuration.xml` as already mentioned in Section 2.1, “Static Lookup”.
- Session persistence is **NOT** supported.
- Will Message cannot be specified explicitly on the client side, but only globally and for all clients

sharing a connection, once again through the `brokers_configuration.xml` file.

Life Cycle Events

In a shared end-to-end connection, the life cycle events relative to `MqttClient`, MQTT channel, and the shared broker connection are only partially tied together, namely:

- A connection request initiated by `MqttClient` implies that the newly created MQTT channel will be logically joined to an already active shared broker connection; the shared MQTT connection will be established only the very first time that an `MqttClient` instance connects to the broker.
- A disconnection request initiated by `MqttClient` (which means closure of the MQTT channel) will cause only a *logical* detaching from the shared broker connection, which will be closed gracefully (that is, via an explicit `DISCONNECT` Control Packet) only the very last time that an `MqttClient` instance disconnects from the broker.
- An interruption of the shared broker connection (due to any network issue or to a problem in the MQTT broker process) will be propagated back to all sharing `MqttClient` instances, which in turn will disconnect from MQTT.Cool by closing the relative MQTT channels.
- An interruption of the MQTT.Cool connection on the client side (due to any network issue or to an explicit `MQTTCoolSession` closing) will be managed by MQTT.Cool as a logical detaching of all the engaged MQTT channels from the shared broker connection, which will be closed gracefully only when no more MQTT channels are attached.
- Any MQTT.Cool server failure will cause the interruption of the shared broker connection on the server side as well as all the engaged MQTT.Cool connections on the client side.

Any issue on the MQTT.Cool connection detected on the client side will cause the `MqttClient` instance to try a reconnection (as you will see in Section 3.12, “Reconnecting”).

3.7. Specifying Connection Options

As already mentioned before, you can specify a set of options on establishing a connection with any MQTT broker. For example, if required by the target server or even if you want to add a custom authentication functionality to MQTT.Cool through a plugged Hook, then you may provide the *username* and *password*:

```
var connectOptions = { username: 'my_mqtt_username', password: 'my_mqtt_password' };
```

The credentials used to connect to MQTT brokers are **not** related to the ones that may be specified to open new sessions against MQTT.Cool.

Very important: it is generally advisable to set the proper callback functions to handle the connection outcomes and any authorization issue you may have:

```
connectOptions.onSuccess = function() {
  console.log('Connection successful');
}

connectOptions.onFailure = function(responseObject) {
  console.log('Connection failure!');
}

connectOptions.onNotAuthorized = function(responseObject) {
  console.log('Connection NOT authorized!');
}
```

In particular:

- `onSuccess` is invoked in the case of a successful connection to the target MQTT broker; afterwards, the `MqttClient` instance switches to the *connected* status.
- `onFailure` is invoked in the case of any issue with either the target MQTT broker or the MQTT.Cool server.
- `onNotAuthorized` is invoked in the case of any authorization issue raised by the plugged Hook, if any (as will be shown in Section 4.2.5, “Authorizing Connection”).

To specify a Will Message, create and configure a `Message` instance and add it to the options:

```
var myWillMessage = new Message('my_will_message');
myWillMessage.destinationName = 'my_will/topic';
connectOptions.willMessage = myWillMessage;
```

Upon connection, such a message is stored to the target MQTT broker and will be published to the specified Will Topic if the end-to-end connection closes abruptly (or for any other event mentioned in the MQTT Protocol Specifications).

If you want to make the session persistent, then you have to set the Clean Session flag to `false`; otherwise, any previous session will be discarded upon successful connection:

```
connectOptions.cleanSession = false;
```

Will Message and Clean Session flag can only be set in the context of a **dedicated connection**.

3.7.1. Managing Persistent Session

If you start a dedicated connection requiring a persistent session, then its related data are normally saved on the default local storage, which is based on:

- the global `localStorage` property for web browsers

- the local file system for Node.js runtime

In addition, you also have the option to supply a custom persistence layer by setting the `storage` connection option with an implementation of the `MqttStorage` interface:

```
var myCustomStorage = {

  get: function(key) { ... },

  set: function(key, value) { ... }.

  remove: function(key) { ... },

  keys: function() { ... }
}
connectOptions.storage = myCustomStorage;
```

In the case of Node.js, the additional `storePath` property allows you to specify the directory that will host the files used to store the session data:

```
connectOptions.storePath = 'my/local/path';
```

3.7.2. Managing Connection Lost

Before opening a connection, you might want to provide the `onConnectionLost` callback function in order to properly manage the event of an end-to end connection lost. The callback has to be set on the `MqttClient` instance:

```
...
mqttClient.onConnectionLost = function(responseObject) {
  console.log('The
 Connection has been lost, due to ' + responseObject.errorCode);
};
```

An end-to-end connection could be lost for one of the following reasons:

- A regular disconnection initiated through invocation of the `disconnect` method.
- A disconnection from MQTT.Cool triggered through invocation of the `MQTTCoolSession.close` method (on the same `MQTTCoolSession` instance that originated the `MqttClient` object).
- An interruption of the broker connection.
- Stop of the reconnection attempts that follow an interruption of the MQTT.Cool connection, as will be discussed in Section 3.12.2, “Starting to Reconnect”.

Each type of disconnection generates its own error code and description, which are embedded in the `responseObject` passed to the callback.

After losing the connection, the `MqttClient` instance switches to the *disconnected* status from which it would be possible to start a new connection.

3.8. The Message Class

The `Message` class is a wrapper of the `PUBLISH` Control Packet, which lets you send and receive *Application Messages* to and from any MQTT broker through an end-to-end connection.

Generally, you need to explicitly make an instance of `Message` class when you have to publish messages to the broker; on the contrary, when receiving a message from a subscription, the library provides you with a pre-built instance.

To prepare a new `Message` object, simply invoke the constructor passing a payload and set the required properties:

```
// Create a message starting from a string object
var message = new Message('A message');

// Set the target topic
message.destinationName = 'cool/topic';

// Set the Quality of Service
message.qos = 1;

// Set the retained flag
message.retained = true;
...
// Get the current duplicate flag
var dup = message.duplicate;
```

`destinationName`, `qos`, and `retained` are defined as read/write properties, whereas `duplicate` is a **read-only** property as it can only be set on messages received from the broker.

Besides string objects, you are free to supply even binary payloads, which can be expressed as `ArrayBuffer` or `TypedArray` objects:

```
// Create a message starting from an Uint8Array
var message = new Message(new Uint8Array([13, 73]));

// Create a message starting from a Float64Array
var message2 = new Message(new Float64Array([0.34, 0.122]));

// Create a message starting from an ArrayBuffer
var buffer = new ArrayBuffer(4);
var uint8Array = new Uint8Array(buffer);
uint8Array[0] = 1;
uint8Array[1] = 2;
uint8Array[2] = 3;
uint8Array[3] = 4;
var message3 = new Message(buffer);
```

String and binary payloads can be read by accessing their respective read-only properties:

```
function onReceiveMessage(message) {
  // Get the payload as a string
  var strPayload = message.payloadString;
  ...

  // Get the payload as an ArrayBuffer
  var binaryPayload = message.payloadBytes;
  ...

});
```

Of course, you could get a string from a source binary payload or vice-versa; however, you might want to access the right property based on the original content in order to avoid annoying conversions.

3.9. Publishing

To publish messages to the connected MQTT broker, you have to use the `send` method, which can be used in two different forms based on the number and the type of the provided arguments:

- *Single Argument Form*: the `send` method takes just one argument, which is the `Message` instance to be published. For example:

```
...
// Prepare the Message instance
var msg = new Message('My Message!')
msg.destinationName = 'my/topic';
msg.qos = 1;
msg.retained = false;

// Send the prepared message
client.send(msg);
```

- *Multiple Arguments Form*: the `send` method takes up to four arguments that specify the details of the message to be published. In this case, the arguments that are not explicitly provided will assume default values (see the API reference). For example:

```
// Send message by specifying topic and payload
client.send('my/cool/topic1', 'My Message!');

// Send message by specifying topic, payload and qos
client.send('my/cool/topic2', 'My Message!', 1);

// Send message by specifying topic, payload, qos and retained flag
client.send('my/cool/topic3', 'My Message!', 1, true);
```

You cannot provide arguments in a mixed form (for example, passing a `Message` instance as the first parameter and a topic as the second).

3.9.1. Message Delivery Notification

If you want to be notified about the outcome of a message delivery, then all you need is to set on the `MqttClient` instance a pair of callback functions.

In particular, `onMessageNotAuthorized` will be triggered in the case of any authorization issue raised by the plugged Hook once the message is received by MQTT.Cool (as will be described in Section 4.2.6, “Authorizing Message Publishing”):

```
...
mqttClient.onMessageNotAuthorized = function(message, responseObject) {
  console.log('Publishing of message targeted to topic ' + message.destinationName +
    ' NOT authorized!');

  // Check if responseObject has been actually provided
  if (responseObject) {
    var code = responseObject.errorCode;
    var msg =  responseObject.errorMessage
    console.log('code: ' + code + ', message: ' + msg);
  }
};
```

while `onMessageDelivered` will be invoked once the delivery process completes:

```
...
mqttClient.onMessageDelivered = function(message) {
  console.log('The Message has been delivered to topic ' + message.destinationName);
};
```

In the latter case, successful completion of the delivery process depends on several factors:

- The QoS level used to deliver the message.
- The current end-to-end connection type (shared or dedicated).
- Whether the session is persistent, which is allowed only in the case of a dedicated connection.

QoS 0 Message

`onMessageDelivered` will be triggered once the message has been delivered to the underlying MQTT channel (irrespective of the connection type, shared or dedicated), and before any possible authorization issue that could be raised when the message is actually received by MQTT.Cool.

QoS > 0 Message

`onMessageDelivered` will be invoked on the basis of the end-to-end connection type as follows:

- For a shared connection and a dedicated connection **without** session persistence, the invocation occurs when MQTT.Cool sends back the acknowledgment that the flow of the Control Packets exchanged on the server side and relative to the delivery protocol for the specified QoS level has been completed.
- For a dedicated connection **with** session persistence, the invocation occurs in accordance with the flow of the Control Packets exchanged between `MqttClient` and the MQTT broker and relative to the aforementioned delivery protocol. More specifically:
  - For QoS 1 message, upon receiving the `PUBACK` Control Packet, the callback is invoked and then the message is removed from the local storage.
  - For QoS 2 message, upon receiving the `PUBREC` Control Packet, the message state is updated

on the local storage and then the `PUBREL` Control Packet is sent back; upon receiving the `PUBCOMP` Control Packet, the callback is invoked and then the message is finally removed from the local storage.

The callback will never be invoked in the case of any authorization issue raised by the plugged Hook. Under this circumstance, the message is also removed from the local storage if session persistence is active.

3.10. Subscribing

To start receiving messages from the connected MQTT broker, you first have to provide the `MqttClient` instance with the `onMessageArrived` callback function, which will be invoked when messages arrive. Then, you must subscribe to the MQTT topics of interest by using the `subscribe` method, specifying a *topic filter* and an optional set of subscribe options:

```
...
// Set the callback function for receiving messages
mqttClient.onMessageArrived = function(message) {
  var topic = message.destinationName;
  var qos = message.qos;
  console.log('Received message from topic ' + topic + ' with QoS ' + qos);
};

// Prepare the set of subscription options
var subscribeOptions = { ... };

// Request the subscription to the MQTT broker
mqttClient.subscribe('my/cool/topic', subscribeOptions);
...
```

Let's now analyze in detail the subscription process.

3.10.1. Topic Filter

As allowed by the MQTT Protocol Specifications, the provided topic filter can also contain special wildcard characters so that you can subscribe to multiple topics at once. In particular, you can use *Multi-level* wildcard, *Single level* wildcard, or even mix them:

```
// Subscribe to topic filter with Multi-level wildcard
mqttClient.subscribe('my/#/topic', subscribeOptions);

// Subscribe to topic filter with Single level wildcard
mqttClient.subscribe('my/cool/+', subscribeOptions);

// Mix Multi-level and Single level wildcard in the same topic filter
mqttClient.subscribe('my/+/topic/#', subscribeOptions);
```

3.10.2. Subscribe Options

By means of subscribe options, you can specify several parameters that allow you to govern how the subscription has to be requested to the broker.

Requested QoS

Through the `qos` property, you can request the maximum *Quality Of Service* with which the target MQTT broker is allowed to send messages:

```
...
// Set the requested QoS value
var subscribeOptions = { qos: 1 };
```

Callbacks

To be notified about the outcome of the subscription request, you might want to provide two callback functions:

- `onSuccess`, which will be triggered upon successful acknowledgment of the request, also notifying of the maximum QoS level granted by the broker.
- `onFailure`, which will be invoked in the case of a request failure (with relative error code).

Below is an example:

```
...
subscribeOptions.onSuccess = function(grantedQoS) {
  console.log('Acknowledged subscription with granted QoS: ' + grantedQoS);
};

subscribeOptions.oFailure = function(responseObject) {
  console.log('Subscription request failed with error code: ' +
responseObject.errorCode);
};
```

In addition, by setting the `onNotAuthorized` callback function, you can also catch any authorization issue that could be raised by the plugged Hook (as you will see in Section 4.2.7, “Authorizing Subscription”):

```
...
subscribeOptions.onNotAuthorized = function(responseObject) {
  console.log('Subscription NOT authorized!');

  // Check if responseObject has been actually provided
  if (responseObject) {
    var code = responseObject.errorCode;
    var message = responseObject.errorMessage;
    console.log('code: ' + code + ', message: ' + msg);
  }

};
```

Frequency Management

In the case of a shared connection, you also have the opportunity to specify the maximum update frequency at which MQTT.Cool can route messages from the target MQTT broker to the client:

```
...
subscribeOptions.maxUpdateFrequency = 100;
```

The rate has to be expressed in *message/sec* and can be applied only for subscriptions made with QoS 0.

By leveraging this powerful feature, based on the native potentialities of the underlying Lightstreamer technology, you can take advantage of the optimized delivery employed by the Lightstreamer Kernel, which is able to allocate the desired frequency for each subscription requested by each client, also guaranteeing to never exceed the demanded maximum frequency.

You can dive deeper into this and other basic concepts by having a look at the General Concepts document from the Lightstreamer web site.

3.10.3. Message Arriving Notification

A message sent from the MQTT broker reaches `MqttClient` through the mediation of MQTT.Cool; once arrived, the invocation of `onMessageArrived` is based on the following factors, similarly to what happens for `onMessageDelivered`:

- The QoS level at which the MQTT broker sends the message.
- The current end-to-end connection type (shared or dedicated).
- Whether the session is persistent, which is allowed only in the case of a dedicated connection.

In particular:

- For a shared connection and a dedicated connection **without** session persistence, the callback is triggered when MQTT.Cool forwards the message to `MqttClient` as soon as the delivery of the Control Packets exchanged on the server side and relative to the delivery protocol for the

specified QoS level has been completed.

- For a dedicated connection **with** session persistence, `onMessageArrived` is called in accordance with the flow of the Control Packets exchanged between `MqttClient` and the MQTT broker and relative to the aforementioned delivery protocol. More specifically:
  - Upon receiving a QoS 0 message, the callback is invoked immediately without any further elaboration.
  - Upon receiving a QoS 1 message, the `PUBACK` Control Packet is sent back and then the callback is invoked.
  - Upon receiving a QoS 2 message, it is stored immediately and then the `PUBREC` Control Packet is sent back; upon receiving the `PUBREL` Control Packet, the message is finally removed from the local storage, the callback is invoked, and the `PUBCOMP` Control Packet is sent back.

QoS Downgrading

As already stated, for a shared connection the same message is forwarded to all the other `MqttClient` instances that share the same broker connection and have subscribed to the same topic filter, even specifying a different QoS level. As a consequence of this, it could be possible that the `qos` value of the received `Message` instance gets downgraded from the value actually granted by the MQTT broker because the sole existing subscription on the server side has been submitted with the maximum QoS level among the ones of the fanout subscriptions requested by the clients.

3.11. Unsubscribing

To stop receiving messages from previous subscribed topic(s), simply call the `unsubscribe` method passing the topic filter to unsubscribe from and a set of optional parameters:

```
...
var unsubscribeOptions = {

  onSuccess: function() {
    console.log('Successfully unsubscribed');
  },

  onFailure: function() {
    console.log('Successfully unsubscribed');

  }
};
mqttClient.unsubscribe('my/cool/topic', unsubscribeOptions);
...
```

Once again, a couple of callback functions allow you to be notified about the result of the unsubscription request, in particular:

- `onSuccess` will be triggered upon a successful acknowledgment of the unsubscription.
- `onFailure` will be invoked in the case of a request failure.

The `onFailure` callback function is not actually employed by the current implementation of the library as the MQTT Protocol Specifications does not define any behaviors in case of unsubscription failure. As a consequence, this function is only formally provided, but future use cannot be ruled out.

3.12. Reconnecting

While your `MqttClient` instance is connected, the related MQTT.Cool connection could be interrupted, for example, due to any network issue or to problems in the MQTT.Cool server process.

To help you deal with this condition, the library comes with an out-of-the-box *reconnection* feature that allows you to try to transparently re-establish connection to the MQTT.Cool server while also preserving the current *session state* of the involved `MqttClient` instance, even when the latter was connecting without session persistence: the state can then be resumed once the connection has been restored.

Let's elaborate a bit more on the session state.

3.12.1. The MqttClient Session State

The session state of an `MqttClient` instance obviously depends on the current end-to-end connection type:

- For a shared connection and a dedicated connection **without** session persistence, the session state consists of:
  1. All messages sent to the MQTT broker, but not yet acknowledged by the MQTT.Cool server.
  2. All the current active subscriptions.
- For a dedicated connection **with** session persistence, the session state is given by:
  1. QoS 0 messages sent to the MQTT broker, but not yet acknowledged by the MQTT.Cool server.
  2. QoS 1 and QoS 2 messages sent to the MQTT broker, but not completely acknowledged.
  3. QoS 2 messages received from the MQTT broker, but not completely acknowledged.

     Note that points 2 and 3 are compliant with the definition of *Session state in the Client* as stated in the MQTT Protocol Specifications; instead, point 1 goes beyond such definition as no recovery action is normally required for QoS 0 messages.

3.12.2. Starting to Reconnect

By setting the `onReconnectionStart` callback function on the `MqttClient` instance, you will be notified when the MQTT.Cool connection has been interrupted and an attempt to re-establish it has started:

```
mqttClient.onReconnectionStart = function() {
  console.log('Starting to resume the connection...');
};
```

The `onReconnectionStart` callback will be invoked indefinitely until one of the following events occurs:

- The MQTT.Cool connection has been restored.
- An explicit invocation of the `disconnect` method has been performed in the body of the provided callback as follows:

  ```
  mqttClient.onReconnectionStart = function() {
    // Issue in the underlying connection, stop any reconnection attempt
    mqttClient.disconnect();
  };
  ```

which is also the recommended pattern to stop definitively any reconnection attempt. In this case, there will not be any further chances to resume the state of an `MqttClient` instance that is without a persistent session.

The callback is invoked only if the `MqttClient` instance was previously in the connected status: if the MQTT.Cool connection was interrupted just before `MqttClient` connects through the `connect` method, then a connection failure will be immediately triggered. See the API reference relative to the OnConnectionFailure (the case with `errorCode=10`).

3.12.3. Completing Reconnection

As soon as the MQTT.Cool connection is restored, the `MqttClient` instance and the MQTT.Cool server cooperate silently in order to fully restore the whole end-to-end connection, whereupon the session state will be resumed as follows:

- For a shared connection and a dedicated connection **without** session persistence:
  - All the active subscriptions will be resubmitted silently.
  - Sent QoS 0 messages, not yet acknowledged by the MQTT.Cool server, will be redelivered, although the `onMessageDelivered` notification will no longer take place.
  - Sent QoS > 0 messages, not yet acknowledged by the MQTT.Cool server, will be redelivered; in this case, the flow of notification will prosecute as if the messages had been sent for the first time, which means that `onMessageDelivered` will be invoked as expected.

    Furthermore, message redelivery triggered on the client side could cause duplicates because on the server side, the same message may be sent and acknowledged completely before the end-to-end connection is interrupted: this specific condition represents a concern in the case of QoS 2 messages as the MQTT Protocol Specifications do not allow duplicates.

- For a dedicated connection **with** session persistence:
  - Sent QoS 0 messages, not yet acknowledged by MQTT.Cool, will be redelivered, although the `onMessageDelivered` notification will no longer take place.
  - Sent QoS 1 and 2 messages, not yet completely acknowledged, will be reprocessed as per the reached level of acknowledgment.
  - Received QoS 2 messages, not yet completely acknowledged to the MQTT broker, will be reprocessed as per the reached level of acknowledgment.

In order to be notified when the reconnection process successfully completes, provide the `MqttClient` instance with the `onReconnectionComplete` callback as follows:

```
mqttClient.onReconnectionComplete = function() {
  console.log('The connection has been successfully resumed');
};
```

3.13. Disconnecting

To explicitly disconnect from the MQTT broker, only a straight invocation of the `disconnect` method is required:

```
...
mqttClient.disconnect();
...
```

After sending the `DISCONNECT` Control Packet, `MqttClient` immediately switches to the disconnected status and closes the underlying MQTT channel, which provokes `onConnectionLost` to be fired (as already anticipated in Section 3.7.2, “Managing Connection Lost”). However, the MQTT.Cool connection remains up as it may serve the active MQTT channels owned by the other connected `MqttClient` objects that share the same parent `MQTTCoolSession` instance.

From the disconnected status, an `MqttClient` object is allowed to open a new connection through the `connect` method.

Chapter 4. The MQTT.Cool Hook

In this chapter, we will describe the Hook and its role in the MQTT.Cool architecture; in addition, a developer's guide will help you learn how to make and use custom Hooks.

4.1. Basics

The MQTT.Cool Hook is a custom pluggable component that provides a powerful extension mechanism to integrate your own authentication and authorization functionalities into the MQTT.Cool server.

By using the *MQTT.Cool SDK for Java Hooks* you can develop and package your Hook, which will be then deployed into the `hook` folder of the MQTT.Cool installation; once loaded inside the running server process, it will be able to intercept specific events originated from the client side in order to apply fine-grained custom authorization checks as per your own security needs. In addition, it can also extend the default Static Lookup mechanism provided by MQTT.Cool.

4.2. Hook Development

The following sections will guide you through the core concepts of Hook development.

4.2.1. Setting Up the Development Environment

The MQTT.Cool SDK for Java Hooks provides an open source API through which you can easily develop any Hook.

Because the API is available from the Maven Central Repository, add the following dependency to your `pom.xml` to set up your development environment :

```
<dependency>
    <groupId>cool.mqtt.hook</groupId>
    <artifactId>mqtt.cool-hook-api</artifactId>
    <version>1.3.0</version>
</dependency>
```

Please note that other dependency management tools, like *Gradle* or *Ivy*, may be used as well. Take a look at the online SDK section for more information.

4.2.2. The MQTTCoolHook Interface

Basically, an MQTT.Cool Hook is implemented through a user defined Java class, which implements the `MQTTCoolHook` interface.

Such an interface exposes a set of methods that may be categorized as follows:

- *Authorization Requests*, which are expected to return a Boolean value or throw an exception:

    - `canOpenSession`, to check the authorization for opening a new session against the MQTT.Cool server
    - `canConnect`, to check the authorization for creating a connection to a specified MQTT broker
    - `canPublish`, to check the authorization for message publishing
    - `canSubscribe`, to check the authorization for subscribing to topics
- *Simple Notifications*, which simply inform about specific events:
    - `init`, called during the MQTT.Cool initialization
    - `onSessionClose`, called to notify that a session opened against the MQTT.Cool server has been closed
    - `onDisconnection`, called to notify that a client has been disconnected from an MQTT broker
    - `onUnsubscribe`, called to notify that a client has been unsubscribed from a given topic filter
- *Factories*, which dynamically provide further MQTT connection settings:
    - `resolveAlias`, returns a valid broker configuration if no static entries exist for a given connection alias.

As will be shown in the subsequent sections, you might want to supply a specific behavior only to those methods in which you are actually interested, whereas you may provide a fake implementation for all others. Furthermore, SDK includes the `SimpleCoolHook` base class, which is a skeletal implementation of the interface meant to be extended to make it easier to provide a full implementation of a custom Hook.

The `SimpleCoolHook` class provides the following default behaviors for each specific method category:

- Authorization Requests: always permit.
- Simple Notifications: do nothing.
- Factories: return **null**.

Let's start writing a simple Hook, which will help you to grasp the core concepts needed to implement your own extensions.

4.2.3. Notifying of Hook Initialization

A Hook is loaded during the MQTT.Cool initialization process. If you need to run some setup logic, then you have to override the `init` method:

```
package my.cool.hook;

import java.io.File;

import cool.mqtt.hooks.HookException;
import cool.mqtt.hooks.SimpleCoolHook;

public class MyCoolHook extends SimpleCoolHook {

  @Override
  public void init(File configDir) throws HookException {
    // Put your setup logic here
  }
}
```

The `configDir` parameter is the `<MQTT.COOL_HOME>/hook` directory, where you can put any resource you may need to initialize the Hook (for example, configuration files).

Any exception thrown from the method will cause the failure of the initialization phase and, as a consequence of this, the MQTT.Cool server process will abort.

4.2.4. Authorizing Session Opening

If you want to intercept a new session started by the client (as detailed in Section 3.4, “The MQTT.Cool Connection Pattern”), then you have to override the `canOpenSession` method:

```
@Override
public boolean canOpenSession(String sessionId, String user, String password,
  Map clientContext, String clientPrincipal) throws HookException {

  boolean canOpen = false;

  // Put your authorization logic here
  ...

  return canOpen;
}
```

On the basis of the parameter values, you could apply your own security policies to authorize the session. You could also contact an external service (like *databases* or *LDAP* servers) to authenticate the user who is requesting to connect to MQTT.Cool. The following example shows a straight user ID/password-based authentication:

```
@Override
public boolean canOpenSession(String sessionId, String user, String password,
  Map clientContext, String clientPrincipal) throws HookException {

  if (user == null || user.trim().isEmpty()) {
    return false;
  }

  // Lookup the password from an external service (for example, a database)
  String storedPassword = lookupPassword(user);

  if (storedPassword == null) {
    return false;
  }

  boolean authenticated = storedPassword.equals(password);

  return authenticated;
}
```

As you have seen in Section 3.5, "The MQTTCoolSession Interface", the `onConnectionFailure` callback is invoked in the case of any authorization issue:

```
mqttcool.openSession('http://my.server.company:8080', 'my_cool_user',
'my_wrong_password', {

  onConnectionFailure: function(errorType, errorCode, errorMessage) {
    // errorType is 'UNAUTHORIZED_SESSION'
    // errorCode and errorMessage are undefined
    if ('UNAUTHORIZED_SESSION' == errorType) {
      console.log('Authorization issue')
    } else {
      // Other connection failure conditions
      ...
    }
  }
  ...
});
```

In particular, the `errorType` parameter is set to the string `UNAUTHORIZED_SESSION`, which is the reserved value for any authorization issue, whereas the remaining parameters are `undefined`. This is what happens when the Hook denies authorization to open a new session by returning the `false` value (as in the example above). Nevertheless, you are allowed to throw a `HookException` with detailed code and message, which will be forwarded, respectively, as `errorCode` and `errorMessage` parameters to the client, thus enabling the latter to distinguish the kind of problem by looking at them.

More generally, from any Authorization Request method you have two options for reporting an

issue to the client:

1. Return the `false` value, which is the way to go if the client simply needs to know whether its request has not been authorized.
2. Throw a `HookException` with detailed information, which is the suggested pattern if you want the client to undertake specific actions based on the reported issue.

The following code snippet modifies the previous example; this time, a `HookException` is thrown to signal specific failure conditions, although a Boolean value is still returned to indicate whether the provided password matches the stored one.

```
@Override
public boolean canOpenSession(String sessionId, String user, String password,
  Map clientContext, String clientPrincipal) throws HookException {

  // Use errorCode 101 to signal missing username
  if (user == null || user.trim().isEmpty()) {
    throw new HookException(101, "No user provided");
  }

  // Lookup the password from an external service (for example, a database)
  String storedPassword = lookupPassword(user);

  // Use errorCode 102 to signal not existing account
  if (storedPassword == null) {
    throw new HookException(102, "User \"" + user + "\" not found");
  }

  // Return a Boolean value for the validation outcome
  boolean authenticated = storedPassword.equals(password);

  return authenticated;
}
```

`HookException` accepts only **non-negative** error codes.

On the client side, now you can decide how to react on the basis of the error details, sent as specified inside the raised HookException:

```
mqttcool.openSession('http://my.server.company:8080', 'my_cool_user',
'my_wrong_password', {

  onConnectionFailure: function(errorType, errorCode, errorMessage) {
    // errorType is 'UNAUTHORIZED_SESSION'
    // errorCode is:
    //  - undefined in case of simple password mismatch
    //  - 101 in case of no user provided
    //  - 102 in case of no user found
    // errorMessage is provided according to errorCode
    if ('UNAUTHORIZED_SESSION' == errorType) {
      if (errorCode) {
        console.log('Authorization issue: ' + errorMessage)
        // Here the actions to be undertaken for handling authorization issues
        // reported through a HookException
        ...
      } else {
        console.log('Authentication failure')
        // Here the actions to be undertaken for handling the authentication failure
        // due to password mismatch
        ...
      }
    } else {
      // Other connection failure conditions
      ...
    }
  }

  ...
});
```

As you will see for the next operations, `sessionId` is always included in the signature of all Hook methods as the first parameter (except `init` and `resolveAlias`): it will take on the meaning of **already trusted session**; hence, there will be no chance that any subsequent Authorization Request or *Simple Notification* may be invoked as part of an *untrusted session*.

The following diagram shows the sequence of messages and events on both the client side and the server side during the creation of a session.

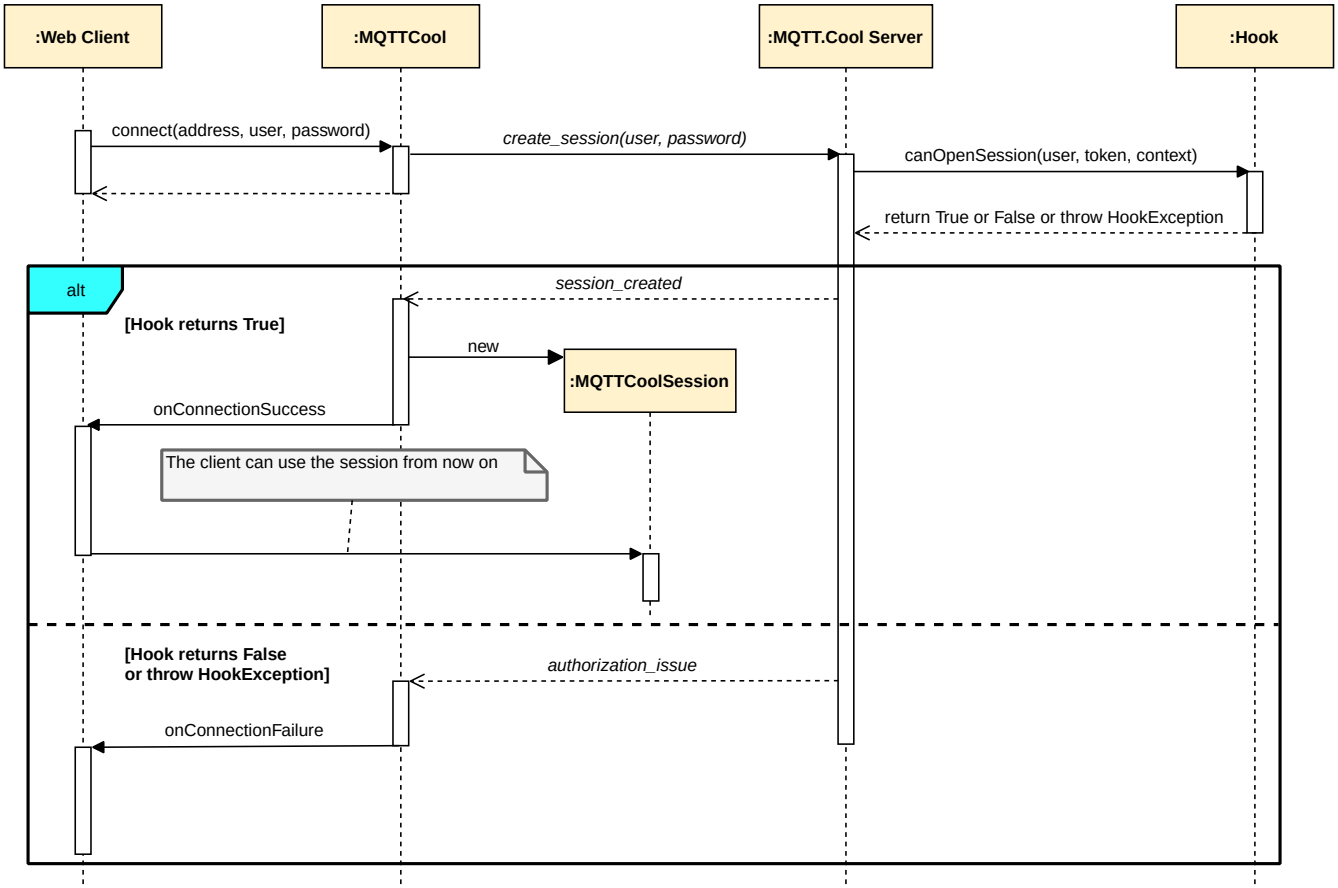

Figure 12. Sequence diagram for session opening

4.2.5. Authorizing Connection

As shown in Section 3.4, "The MQTT.Cool Connection Pattern", after having established a connection to MQTT.Cool, a client might want to connect to a specific MQTT broker. By overriding the `canConnect` method, you can intercept and authorize the connection request:

```
@Override
public boolean canConnect(String sessionId, String clientId, String brokerAddress,
  MqttConnectOptions connectOptions) throws HookException {

  boolean canConnect = false;

  // Put your authorization logic here
  ...

  return canConnect;
}
```

For example, you could check the credentials (provided in the connection options) before being passed to the broker, thus enabling you to extend the latter with a further layer of authentication logic if the native one does not fully satisfy your security needs (or even in the case where the broker does not implement any authentication itself).

In the case of a shared connection, the physical MQTT connection between the MQTT.Cool server and the target MQTT broker is realized only the very first time that a client requests to connect; however, the `canConnect` method is invoked on each connection request issued by every *joining* client.

The following code snippet shows you an example of a very specific authorization policy, which allows unauthenticated users to access to the MQTT broker on the condition that they establish a dedicated connection without managing session persistence:

```
@Override
public boolean canConnect(String sessionId, String clientId, String brokerAddress,
  MqttConnectOptions connectOptions) throws HookException {

  String mqttUser = connectOptions.getUsername();

  // If no credentials are provided, allow only NOT persistent sessions
  if (mqttUser == null) {
    boolean dedicatedConnection = !clientId.trim().isEmpty();
    boolean sessionPersistent = !connectOptions.isCleanSession();
    if (dedicatedConnection && sessionPersistent) {
      throw new HookException(201, "Cannot establish a connection with persistent
session for NOT authenticated user");
    }
    return true;
  }

  // Otherwise, simply check the provided credentials
  String mqttPassword = connectOptions.getPassword();
  String storedPassword = lookupPasswordForMQTT(mqttUser);
  if (storedPassword == null) {
    throw new HookException(202, "User \"" + mqttUser + "\" not found");
  }

  return storedPassword.equals(mqttPassword);
}
```

Inspecting the `connectOptions` parameter helps you to easily accomplish the task as it contains the options actually being used by the MQTT.Cool server to connect to the target MQTT broker. See the `MqttConnectOptions` API reference for more detailed information.

On the client side, you will be notified about the raised authorization issue by means of the related callback function as described in Section 3.7, “Specifying Connection Options”:

```
connectOptions.onNotAuthorized = function(responseObject) {
  // responseObject is:
  //  - undefined in case of simple password mismatch
  //  - populated with errorCode and errorMessage according to
  //    the thrown HookException
  console.log('Connection NOT authorized');

  if (!responseObject) {
    console.log('Authentication failure');
    // Here the actions to be undertaken for handling authentication failure
  } else {
    switch(responseObject.errorCode) {
      case 201:
        // Here, the actions to be undertaken for handling the authorization issue due
        // to unauthenticated user trying to connect with a persistent session
        ...
        break;

      case 202:
        // Here, the actions to be undertaken for handling the authorization issue due
        // to not found user
        ...
        break;
    }
  }
};
```

Similarly to the `onConnectionFailure` callback, here the `responseObject` parameter is passed only when a `HookException` has been thrown on the server side, thus allowing you to differentiate the kind of problem by looking at the `errorCode` and `errorMessage` properties, which reflects, respectively, the code and the message as embedded into the exception.

The following sequence diagram shows the interactions required to establish an authorized end-to-end connection between the client and the target MQTT broker.

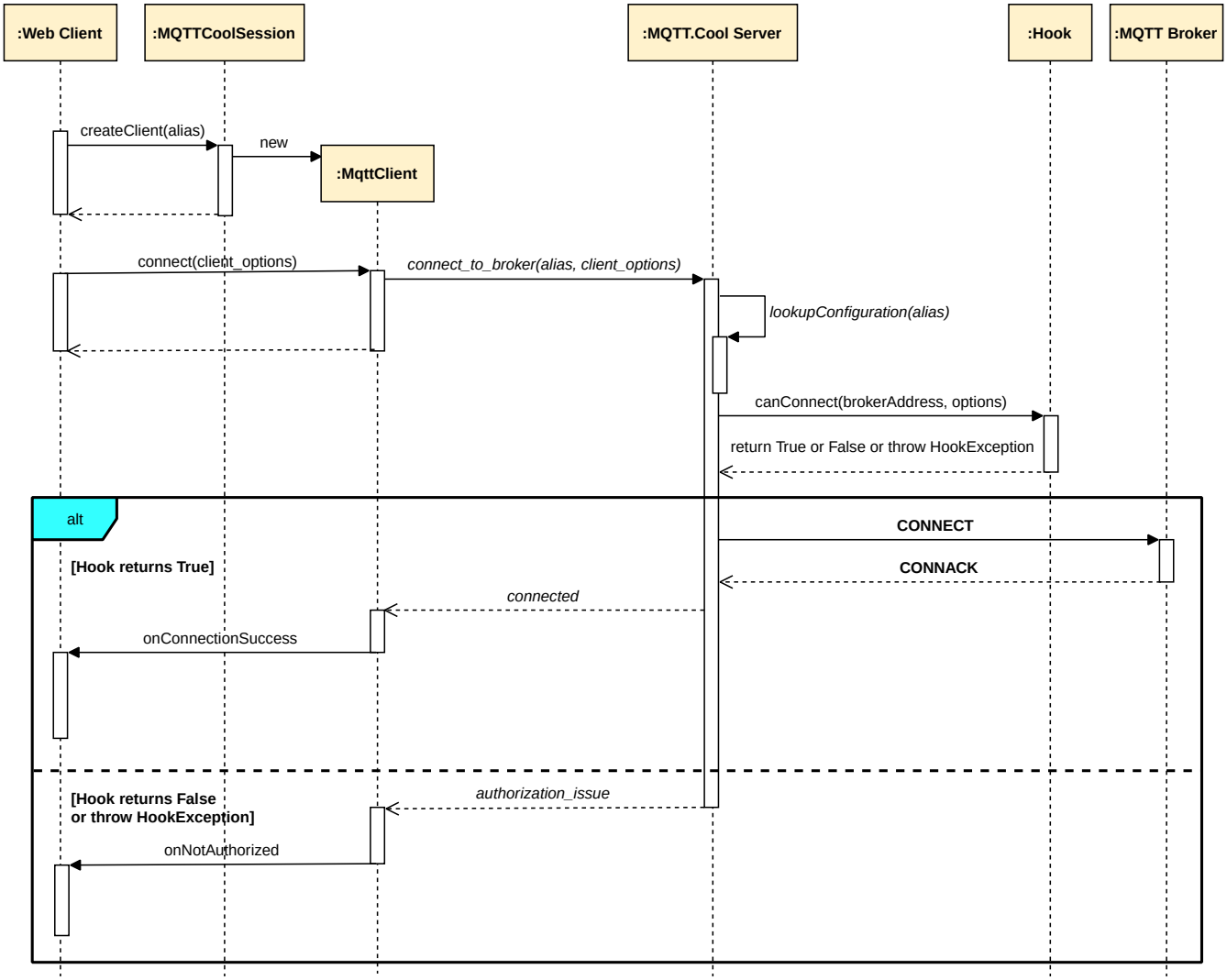

Figure 13. Sequence diagram for connection creation

Providing Broker Configuration

As already anticipated in Section 2.1.1, “The Hook Variant”, you can program your custom Hook to provide a valid MQTT broker configuration for a specific connection alias when no other static entries have been supplied in the `brokers_configuration.xml` file for the same alias. All you need is to override the `resolveAlias` method, where you will provide an instance of the `MqttBrokerConfig` interface, which will act as an MQTT broker configuration as it was provided from the static configuration:

```
import cool.mqtt.hooks.utils.MqttBrokerConfigBuilder;

...

@Override
public MqttBrokerConfig resolveAlias(String alias) throws HookException {
  /*
   * If a client wants to connect to the MQTT broker aliased
   * by "my_cool_broker", provide a valid configuration.
   */
   if ("my_cool_broker".equals(alias)) {
     MqttBrokerConfig config =
       new MqttBrokerConfigBuilder("tcp://my.cool.broker:1883")
         .connectionTimeout(10)
         .keepAlive(20)
         .clientIdPrefix("COOL_BROKER")
         .build();
     return config;
   }

   return null;
}
```

As shown above, you can also take advantage of the `MqttBrokerConfigBuilder` utility class, which is a *builder* designed to simplify the making of an `MqttBrokerConfig` instance.

Through `MqttBrokerConfig`, you can set all the same configuration parameters already discussed in Section 2.1, “Static Lookup”.

On the other hand, if no configuration is available for the provided connection alias (neither from `brokers_configuration.xml` nor from the Hook), then on the client side the `onFailure` callback function is invoked with the relative `responseObject.errorCode` value:

```
...
var mqttClient = mqttCoolSession.createClient('not_existing_connection_alias');

mqttClient.connect({

  onFailure: function(responseObject) {
    switch(responseObject.errorCode) {
      ...

      case 9:
        console.log('No valid configuration found');
        ...
      break
      ...
    }
  }

});
...
```

The following diagram illustrates how the Hook is involved in providing a valid configuration.

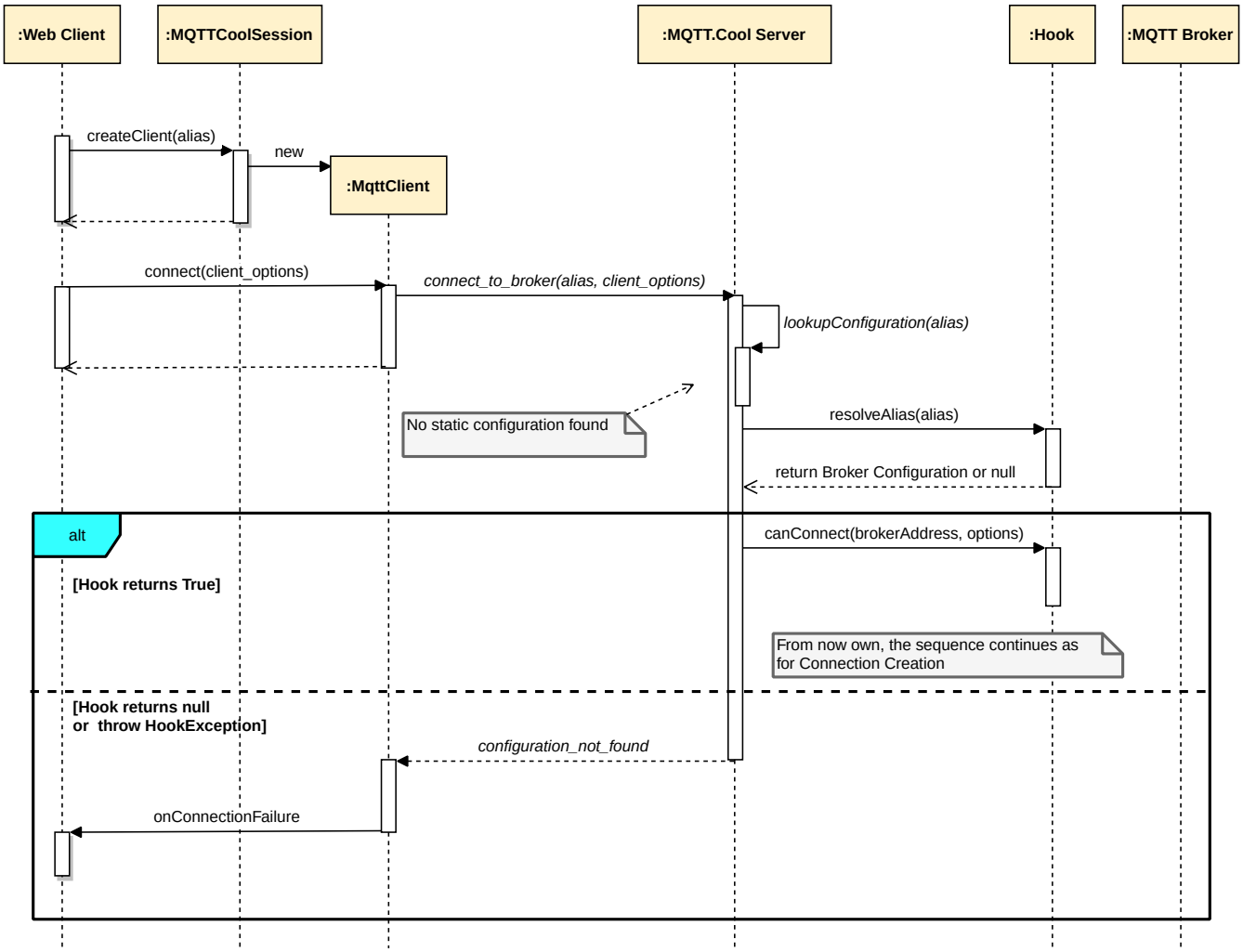

Figure 14. Sequence diagram for a broker configuration provided by the Hook

4.2.6. Authorizing Message Publishing

Any MQTT message sent from the client can be checked through the `canPublish` method before being delivered to the target MQTT broker:

```
@Override
public boolean canPublish(String sessionId, String clientId, String brokerAddress,
  MqttMessage message) throws HookException {

  boolean canPublish = false;

  // Put your authorization logic here
    ...

  return canPublish;
}
```

The message contents can be inspected by looking at the `message` parameter, which you could use to apply your own authorization logic. See the `MqttMessage` API reference for more detailed information.

As an example, if your broker does not support QoS 2 messages, then you might prevent them from being sent; likewise, you could block all the messages targeted to some specific topic names:

```
@Override
public boolean canPublish(String sessionId, String clientId, String brokerAddress,
  MqttMessage message) throws HookException {

  // Block QoS 2 messages
  if (QoS.EXACTLY_ONCE.equals(message.getQos())) {
    throw new HookException(301, "QoS 2 messages are NOT allowed");
  }

  // Block messages targeted to topic name starting with "$"
  if (message.getTopicName().startsWith("$")) {
    throw new HookException(302, "Message
 with topic name beginning with \"$\" are NOT allowed");
  }

  return true;
}
```

As anticipated in Section 3.9.1, “Message Delivery Notification”, on the client side you can leverage the `onMessageNotAuthorized` callback function to be informed about any authorization issue:

```
mqttClient.onMessageNotAuthorized = function(message, responseObject) {
  console.log('Publishing of message targeted to topic ' + message.destinationName +
    ' NOT authorized!');

  if (responseObject) {
    switch (responseObject.errorCode) {
      case 301:
        // Here the actions to be undertaken for handling the authorization issue due
        // to unsupported QoS 2 messages
        ...
        break;

      case 302:
        // Here the actions to be undertaken for handling the authorization issue due
        // to unsupported topic name
        ...
        break;
    }
  }
};
```

Once again, the `responseObject` parameter is provided only in the case of the `HookException` thrown on the server side.

The diagram below shows an example of QoS 1 message publishing checked by the Hook, where the activated connection type (shared or dedicated) does not involve session persistence; therefore, on the client side, the message delivery notification occurs once the delivery protocol has been completed on the server side.

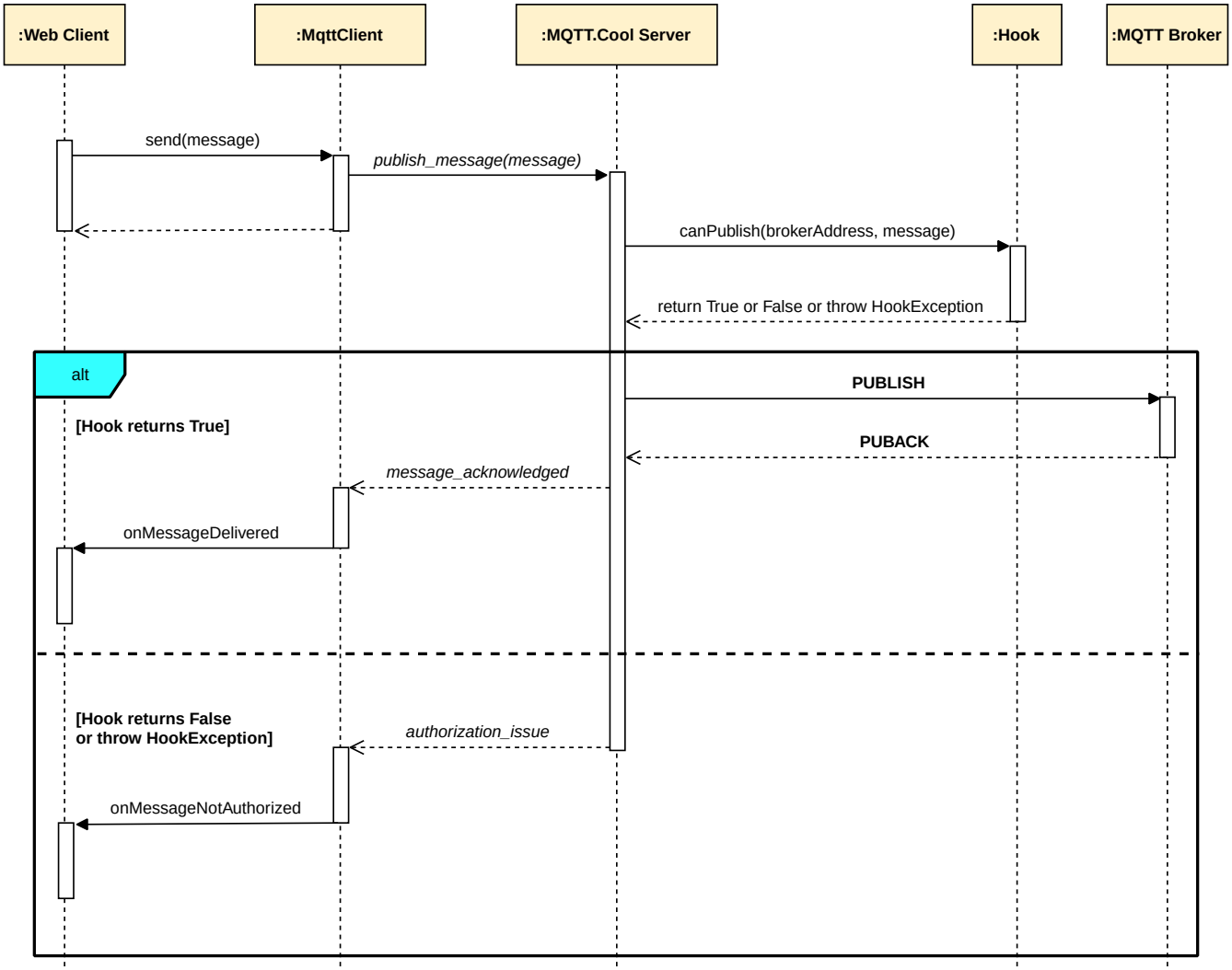

Figure 15. Sequence diagram for a QoS 1 message publishing

4.2.7. Authorizing Subscription

Similarly to the message publishing, any MQTT subscription can be analyzed as well before being actually forwarded to the target MQTT broker by overriding the `canSubscribe` method:

```
@Override
public boolean canSubscribe(String sessionId, String clientId, String brokerAddress,
  MqttSubscription subscription) throws HookException {

  boolean canSubscribe = false;

  // Put your authorization logic here
  ...

  return canSubscribe;
}
```

In this case, the `subscription` parameter comes in handy to assist you in deciding whether to proceed with the request. See the `MqttSubscription` API reference for a complete description.

As a complementary example of the one proposed for the publishing case, the following

implementation disallows any subscription that requires QoS 2, or that specifies a topic filter starting with "$":

```
@Override
public boolean canSubscribe(String sessionId, String clientId, String brokerAddress,
  MqttSubscription subscription) throws HookException {

  // Reject subscriptions with QoS 2
  if (QoS.EXACTLY_ONCE.equals(subscription.getQos())) {
    throw new HookException(401, "Subscription at QoS 2 are NOT allowed");
  }

  // Reject subscriptions to topic filter starting with "$"
  if (subscription.getTopicFilter().startsWith("$")) {
     throw new HookException(402, "Subscription to topic filters beginning with \"$\"
are NOT allowed");
  }

  return true;
}
```

On the client side, the unauthorized subscription request will be signaled through the `onNotAuthorized` callback function as introduced in Section 3.10.2, “Subscribe Options”:

```
subscribeOptions.onNotAuthorized = function(responseObject) {
  console.log('Subscription NOT authorized!');

  if (responseObject) {
    switch (responseObject.errorCode) {
      case 401:
        // Here, the actions to be undertaken for handling the authorization issue due
        // to rejected QoS 2 level
        ...
        break;

      case 402:
        // Here, the actions to be undertaken for handling the authorization issue due
        // to rejected topic filter
        ...
        break;
    }
  }
};
```

As always, the `responseObject` parameter follows the same common pattern for all Authorization Requests as illustrated in the previous sections.

In the diagram below, the sequence related to a subscription request that goes through the Hook

before being conveyed to the MQTT broker.

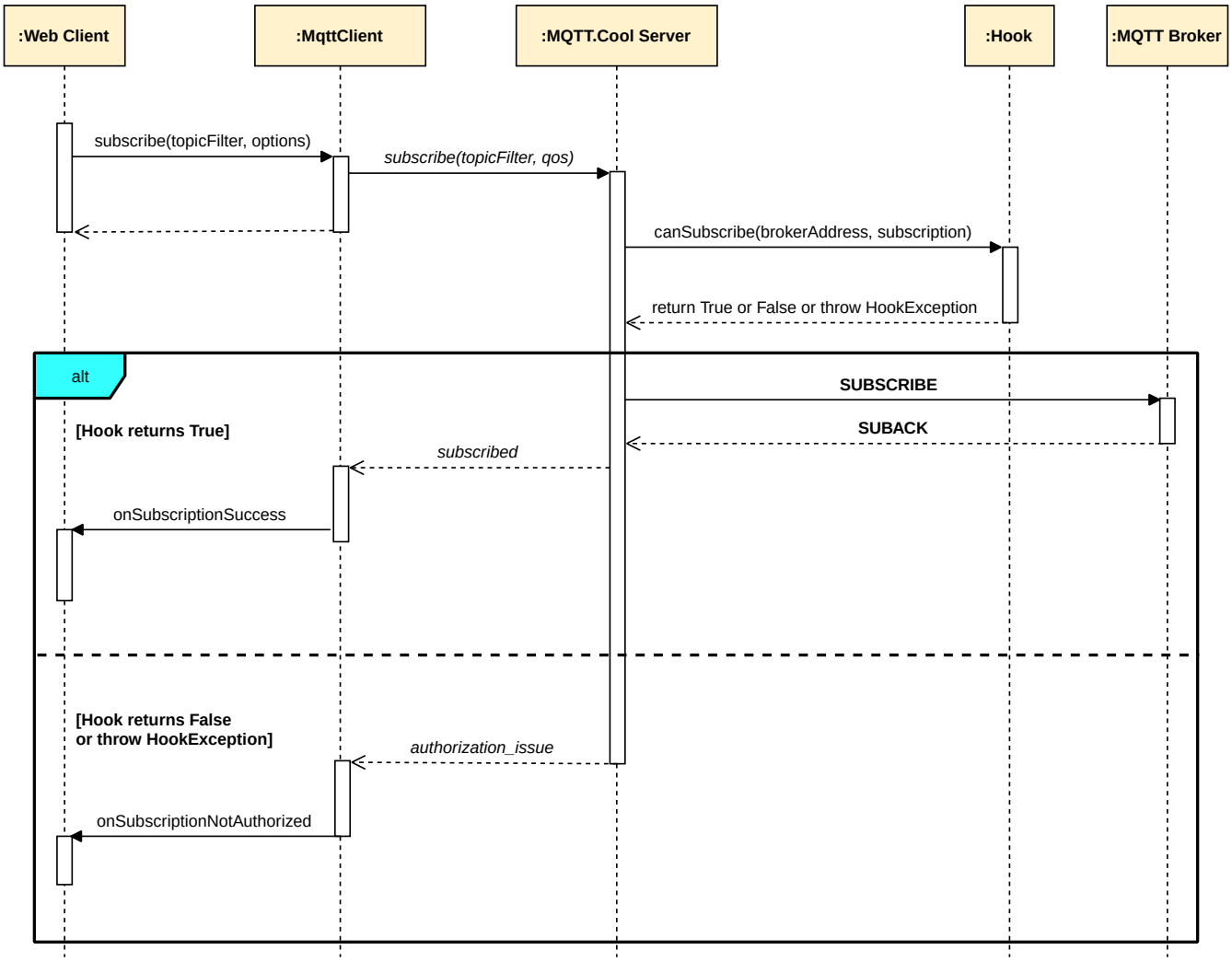

Figure 16. Sequence diagram for a subscription

4.2.8. Notifying of Session Closing, Disconnection, and Unsubscription

Let's finish this chapter on the Hook development with a quick overview of the so-called *Simple Notifications* methods, which are invoked when long-term activities like sessions, connections, and subscriptions complete their own life cycle.

For example, a session opened against MQTT.Cool could end due to either an explicit request made by the client or any network issue; in such a case, the `onSessionClose` method is invoked; therefore, you could override it to apply your own cleanup logic, if any:

```
@Override
public void onSessionClose(String sessionId) {
  // Put your cleanup logic here
  ...
}
```

Similarly, a connection established with any MQTT broker may be explicitly closed by the originating client (see Section 3.13, "Disconnecting"): you will be informed of this event through `onDisconnection`:

```
@Override
public void onDisconnection(String sessionId, String clientId, String brokerAddress) {
  // Put your actions here
  ...
}
```

`onDisconnection` is also invoked because of session interruption while the client is currently connected: if this is the case, then both `onSessionClose` and `onDisconnection` will be triggered.

Lastly, a currently active subscription may be terminated by an unsubscription request (see Section 3.11, “Unsubscribing”), which will be reported by the `onUnsubscribe` method:

```
@Override
public void onUnsubscribe(String sessionId, String clientId, String brokerAddress,
  String topicFilter) {

  // Put your actions here
  ...
}
```

4.3. Packaging, Configuration, and Deployment

To package and deploy your custom Hook into the MQTT.Cool installation, you need to perform the following steps:

1. Build the Hook by executing the Maven `package` goal from your project folder:

   ```
   $ mvn package
   ```

2. Locate the JAR file (which should be in the `target` folder) and drop it into the `<MQTT.COOL_HOME>/hook/lib` folder.

   

   Make sure to populate `lib` with all other JAR files the Hook may depend on.

3. Edit `<MQTT.COOL_HOME>/conf/brokers_configuration.xml`, by specifying the fully qualified class name of the Hook in the `<hook_class>` tag:

   ```
   <mqttcool_brokers_config>

     <hook_class>my.cool.hook.MyCoolHook</hook_class>

     <configurations>
     ...
   ```

Alternatively, you might want to uncomment the provided factory example and replace the tag value:

```
<!-- Optional. Fully qualified class name of a hook with purpose of
     authentication and authorization of users. Must implement the
     MqttCoolHook interface.
     See docs for more information. -->
<!--
<hook_class>my.package.Hook</hook_class>
-->
```

4. Start MQTT.Cool and look into the `<MQTT.COOL_HOME>/logs/mqtt.cool.log` file for any possible issue.

   If deployment went well, then you should get a log line like this at startup:

```
...
04-apr-17 12:06:34,136|INFO|MQTTCoolLogger.init     |Init for MQTT      |Hook
initialized with FQN my.cool.hook.MyCoolHook
...
```

Appendix A: Connection Parameters

The following is the complete list of the supported connection parameters that can be configured in the `<MQTT.COOL_HOME>/conf/brokers_configuration.xml` file.

The details below are similar to the inline documentation provided in the same file.

<connection_alias>.server_address

(Mandatory). The address of the MQTT broker to connect to. Accepted URI forms are:

- `tcp://<host>:<port>`
- `mqtt://<host>:<port>`
- `mqtts://<host>:<port>`
- `ssl://<host>:<port>`

Note that the use of `mqtts` and `ssl` triggers encrypted communication with the MQTT broker. Therefore, further parameters might be required in order to properly support the secure channel. See below `security_protocol`, `truststore_path`, `truststore_password`, `keystore_path`, `keystore_password`, and `private_key_password`.

Example:

```
<param name="mosquitto.server_address">tcp://localhost:1883</param>
```

<connection_alias>.clientid_prefix

(Optional). Client Id prefix to be used for shared connections. If the clients want to share a single connection (see Section 3.6.3, “Shared End-to-End Connection”), then a randomly generated suffix will be appended to generate a unique ClientId for opening the physical connection toward the MQTT broker. Uniqueness of the ClientId is mandatory as multiple shared connections may exist for the same broker.

Default value: `MQTT_Cool`.

Example:

```
<param name="mosquitto.clientid_prefix">mosquitto_client</param>
```

<connection_alias>.connection_timeout

(Optional). The connection timeout expressed in seconds.

Default value: `5`.

Example:

```
<param name="mosquitto.connection_timeout">30</param>
```

<connection_alias>.keep_alive

(Optional). The keep alive interval expressed in seconds.

Default value: `30`

Example:

```
<param name="mosquitto.keep_alive">10</param>
```

<connection_alias>.username

(Optional). The username for authenticating with the MQTT broker. The value that may be provided by the client will take precedence over this setting.

Example:

```
<param name="mosquitto.username">username</param>
```

<connection_alias>.password

(Optional). The password for authenticating with the MQTT broker. The value that may be provided by the client will take precedence over this setting.

Example:

```
<param name="mosquitto.username">username</param>
```

<connection_alias>.will_message

(Optional). The payload of the Will Message, which is interpreted on the basis of the `will_message_format` parameter. The Will Message and related parameters (as defined below) are processed only in the case of dedicated connections. The Will Message and related parameters that may be provided by the client will take precedence over the corresponding settings.

Example:

```
<param name="mosquitto.will_message">will_message</param>
```

<connection_alias>.will_message_format

(Optional). As the `CONNECT` Control Packet will contain just a sequence of raw bytes for the Will Message, this setting specifies how to interpret the payload provided through the `will_message` parameter in order to give the opportunity to supply either a string or a binary sequence:

Specify:

- `UTF-8` to interpret the text as a regular string, which will then be encoded into the final sequence of bytes using the UTF-8 character set.
- `base64` to interpret the text as a *Base64* encoded string, which will then be decoded accordingly to make the final sequence of bytes.

Default value: `UTF-8`.

Example:

```
<param name="mosquitto.will_message_format">base64</param>
```

<connection_alias>.will_topic

(Mandatory if `will_message` is defined). The topic name of the Will Message. Must be at least 1 character in length.

Example:

```
<param name="mosquitto.will_topic">will_topic</param>
```

<connection_alias>.will_qos

(Optional, but considered only if `will_message` is defined). The QoS integer value of the Will Message.

Default value: `0`.

Example:

```
<param name="mosquitto.will_qos">1</param>
```

<connection_alias>.will_retain

(Optional, but considered only if `will_message` is defined). The retain flag of the Will Message.

Default value: `false`.

Example:

```
<param name="mosquitto.will_retain">true</param>
```

<connection_alias>.basic_statistics_log_interval

(Optional). Sets the interval (expressed in milliseconds) at which basic statistics (e.g., number of brokers, number of client connections, current upstream and downstream message frequency, etc.) are logged through the `MQTTCoolLogger.statistics` logger defined in `<MQTT.COOL_HOME>/conf/log_configuration.xml`. The same statistics are also accessible through the JMX interface. If `0`, basic statistics are not logged.

Default value: `0`.

Example:

```
<param name="mosquitto.basic_statistics_log_interval">5</param>
```

<connection_alias>.security_protocol

(Optional, but considered only if a secure schema is used in `server_address`). The protocol to be used in the case of encrypted connection to the broker.

Default: `TLSv1.2`.

Example:

```
<param name="mosquitto.security_protocol">TLSv1.1</param>
```

<connection_alias>.truststore_path

(Optional, but considered only if a secure schema is used in the `server_address`). The path to the *JKS* truststore relative to the `conf` folder. The truststore contains the MQTT broker certificate to trust. If no truststore is specified, then a default one is determined according to the JSSE Reference Guide.

Example:

```
<param name="mosquitto.truststore_path">trust-store.jks</param>
```

<connection_alias>.truststore_password

(Mandatory if `truststore_path` is defined). The password for the truststore.

Example:

```
<param name="mosquitto.truststore_password">mytruststorepassword</param>
```

<connection_alias>.keystore_path

(Optional, but considered only if a secure schema is used in the `server_address`). The path to the *JKS* keystore relative to the `conf` folder. The keystore contains the client certificate and the private key to be used in the case the target MQTT broker requires client authentication. If no keystore is specified, then a default one is determined according to the JSSE Reference Guide.

Example:

```
<param name="mosquitto.keystore_path">keystore.jks</param>
```

<connection_alias>.keystore_password

(Mandatory if `keystore_path` is defined). The password for the keystore.

Example:

```
<param name="mosquitto.keystore_password">mykeystorepassword</param>
```

<connection_alias>.private_key_password

(Mandatory if `keystore_path` is defined). The password for the private key stored into the keystore.

Example:

```
<param name="mosquitto.private_key_password">myprivatekeypassword</param>
```

Appendix B: Access the Lightstreamer Client API

In this appendix, we will give you a short introduction on how to access the Lightstreamer Client API (on which the MQTT.Cool library is based) in order to exploit the features offered by the embedded Lightstreamer Engine.

For more in-depth details on the API and Lightstreamer client development, see:

- Web Client Unified API Reference
- Web Client Development

B.1. The LightstreamerClient Object

In Section 3.5, "The MQTTCoolSession Interface", we have briefly mentioned the `onLsClient` event, which is invoked when a new *Lightstreamer Session* is available and just before connecting to the MQTT.Cool server.

The Session is provided through the `lsClient` parameter, which represents an instance of `LightstreamerClient`. Being the entry point of the Lightstreamer JavaScript Client API, this object could be considered as the *bridge* between the two APIs.

In the subsequent sections, you will be provided with some tips on several potential uses of this object and related entities.

B.1.1. Managing the Connection Options

The provided `LightstreamerClient` instance is already configured and initialized to be used by the MQTT.Cool library for establishing the connection toward the target MQTT.Cool server.

However, you have the opportunity to manage the attached `ConnectionOptions` instance to set specific connection properties as per your needs.

Changes made to the `ConnectionOptions` object will affect only the connection established between the client and the MQTT.Cool server. The connection parameters relative to MQTT brokers have been already discussed in Chapter 2, *Addressing MQTT Brokers* and Appendix A, *Connection Parameters*.

For example, you might want to change default values for properties related to possible timeouts, like the following:

- `connectTimeout`
- `keepaliveInterval`
- `idleTimeout`
- `reconnectTimeout`

and so on.

Similarly, through setForcedTransport method you could *force* a fixed transport (for example, HTTP only), disabling the native *Stream-Sense* algorithm.

See the `ConnectionOptions` API reference for the complete list of modifiable properties.

As already stated, `onLsClient` is the expected place where to apply any change to the connection just before it is actually established. We are now ready to complete the main connection pattern extension started here:

```
mqttcool.openSession(..., {

  onLsClient: function(lsClient) {
    // Get the attached ConnectOptions object
    var connectOptions = lsClient.connectionOptions;

    // Apply the required changes
    connectOptions.setConnectTimeout(10);
    connectOptions.setKeepaliveInterval(30);

    ...
  },

  ...
});
```

B.2. Bandwidth Throttling

In Section 3.10.2.3, “Frequency Management”, we have seen how MQTT.Cool allows specifying the rate at which messages can arrive for a specific MQTT topic. In addition, the underlying Lightstreamer Engine permits to go beyond that—you can also specify the maximum bandwidth allowed for all messages routed by the MQTT.Cool server for this specific session.

Once more, `ConnectionOptions` exposes the needed setMaxBandwidth method that allows you to do the magic:

```
mqttcool.openSession(..., {

  onLsClient: function(lsClient) {
    // Get the attached ConnectOptions object
    var connectOptions = lsClient.connectionOptions;

    // Request to consume up to 300 kbps
    connectOptions.setMaxBandwidth(300);
    ...
  },

  ...
});
```

The Hello IoT World example available on GitHub employs the same technique to issue a new maximum bandwidth request according to the movements on the slider as shown in app.js (line 116).

B.3. Attaching a Listener

If you are are interested in being notified about low-level connection activities and errors, then you could add a `ClientListener` instance to `LightstreamerClient`:

```
mqttcool.openSession(..., {

  onLsClient: function(lsClient) {
    // Add a listener to the LightstreamerClient object
    lsClient.addListener({

      onListenerStart: function { },

      onListenerEnd: function { },

      onPropertyChange: function { },

      onServerError: function { },

      onShareAbort: function { },

      onStatusChange: function { }
    });
    ...
  },

  ...
});
```

See the `ClientListener` API reference to get detailed documentation for every event.

B.4. Private Operations

Through the `LightstreamerClient` instance, you gain instant access to all the power and complexity of Lightstreamer's world. But, as "with great power comes great responsibility," you must take care while using this object; in fact, you might be unaware of triggering some action that could mislead the normal functioning of the overlying MQTT.Cool library.

The following operations are considered *private* from the MQTT.Cool perspective because they could negatively impact normal operation if not properly managed. Therefore, **DO NOT USE** them in any circumstance:

- LightstreamerClient.connect
- LightstreamerClient.disconnect
- LightstreamerClient.enableSharing
- LightstreamerClient.subscribe
- LightstreamerClient.unsubscribe
- LightstreamerClient.sendMessage
- any use of LightstreamerClient.connectionDetails

Appendix C: Configuring TLS/SSL Connections

In this appendix, we will show you how to configure connection parameters to allow encrypted connections between MQTT.Cool and MQTT brokers.

To make things easy, let's walk through a simple scenario in which you want to establish a secure channel to the publicly available Mosquitto server hosted at test.mosquitto.org, which listens on several TCP ports:

- 1883, for plain connections.
- 8883, for encrypted connections (over TLS v1.2, v1.1, or v1.1).
- 8884, for encrypted connections (over TLS v1.2, v1.1, or v1.1) with client authentication.

As your goal is to secure the traffic through a TLS/SSL connection, you are interested in using ports *8883* (server authentication only) and *8884* (server and client authentication).

Let's first illustrate the case where only server authentication is required.

C.1. Server Authentication

An MQTT client that wants to authenticate any MQTT broker must trust its certificate. For this purpose, `test.mosquitto.org` provides a downloadable self-signed certificate file, which will be used on the client side to verify the server connection.

Therefore, you must first download the certificate and then make it suitable for MQTT.Cool for the authentication process through the following easy steps:

1. Generate the truststore and import the certificate:

   Run the command:

   ```
   $ keytool -importcert -alias "test.mosquitto.org" -keystore truststore.jks -file
   mosquitto.org.crt
   ```

   where:

   - `test.mosquitto.org` is the chosen alias, but feel free to use any identifier you prefer.
   - `truststore.jks` is the truststore file to generate.
   - `mosquitto.org.crt` is the server certificate file downloaded from test.mosquitto.org.

   At the prompt, set and confirm the password (obviously, for a production environment, use a **strong** password):

```
$ Enter keystore password: password
$ Re-enter keystore password: password
```

Next, trust the certificate when prompted for confirmation by answering `yes`:

```
Owner: EMAILADDRESS=roger@atchoo.org, CN=mosquitto.org, OU=CA, O=Mosquitto,
L=Derby, ST=United Kingdom, C=GB
Issuer: EMAILADDRESS=roger@atchoo.org, CN=mosquitto.org, OU=CA, O=Mosquitto,
L=Derby, ST=United Kingdom, C=GB
Serial number: e0fadcf9578c98bc

...

Trust this certificate? [no]: yes
Certificate was added to keystore
```

You end with the `truststore.jks` file.

2. Copy the truststore file under the `<MQTT.COOL_HOME>/conf` folder:

```
$ cp truststore.jks <MQTT.COOL_HOME>/conf
```

3. Supply a new configuration in `brokers_configuration.xml`; let's use `tls-test-mosquitto` as a *connection alias*:

```
...
<param name="tls-test-
mosquitto.server_address">ssl://test.mosquitto.org:8883</param> ①
<param name="tls-test-mosquitto.truststore_path">truststore.jks</param> ②
<param name="tls-test-mosquitto.truststore_password">password</param> ③
...
<param name="tls-test-mosquitto.clientid_prefix">...</param>
<param name="tls-test-mosquitto.connection_timeout">...</param>
<param name="tls-test-mosquitto.keep_alive">...</param>
...
```

① The address of the MQTT broker now specifies **ssl** as the URI schema, which enables MQTT.Cool to establish a secure connection.

② The truststore file.

③ The truststore password.

See Appendix A, *Connection Parameters* for a full explanation of each parameter.

Now you have properly set up a broker configuration that when invoked through Static Lookup,

will trigger an encrypted connection to the public Mosquitto server.

The `brokers_configuration.xml` file comes with an example of configuration (named `public-mosquitto`) similar to the one proposed in this explanation. In addition, the `<MQTT.COOL_HOME>/conf` folder contains the referenced truststore file so that you can immediately test it from your client application.

In the next section, we will describe how to connect to the service also using client authentication.

C.2. Server and Client Authentication

As already anticipated, port `8884` of test.mosquitto.org requires clients to provide a certificate to authenticate their connection. You can find instructions on how to generate a client certificate file at url; however, for the sake of completeness, you need to perform the following steps:

The example requires the OpenSSL toolkit.

1. Generate a CSR (Certificate Signing Request):

   Run the command to generate a private key file (`client.key`):

   ```
   $ openssl genrsa -out client.key
   ```

   Then create the CSR file (`client.csr`):

   ```
   $ openssl req -out client.csr -key client.key -new
   ```

   When prompted, provide the required fields (below is an example with details from our own organization):

```
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a □Distinguished Name□ or □DN.□
There are quite a few fields, but you can leave some blank.
For some fields, there will be a default value.
If you enter '.', then the field will be left blank.
-----
Country Name (2 letter code) [AU]: IT
State or Province Name (full name) [Some-State]: Italy
Locality Name (e.g., city) []: Milan
Organization Name (e.g., company) [Internet Widgits Pty Ltd]: Lightstreamer Srl
Organizational Unit Name (e.g., section) []: MQTT.Cool
Common Name (e.g., server FQDN or YOUR name) []: mqtt.cool
Email Address []: support@lightstreamer.com

Please enter the following 'extra' attributes
to be sent with your certificate request:
A challenge password []:
An optional company name []:
```

2. Paste the contents of the generated `client.csr` file into the form you find on the page and submit.

Now you have to make both the downloaded certificate and the private key suitable for MQTT.Cool, this time providing a *keystore*:

1. Convert to PKCS12 format

   As the client certificate and the private key cannot be imported directly into a JKS keystore, you first must convert and combine them into a PKCS12 file, which can then be imported into the keystore.

   ```
   $ openssl pkcs12 -export -out clientcert.p12 -in client.crt -inkey client.key
   ```

   where:

   - `clientcert.p12` is the PKCS12 file to generate.
   - `client.crt` is the client certificate file downloaded from test.mosquitto.org.
   - `client.key` is the private key generated previously.

   At the prompt, set and confirm the password (once more, use a **strong** password):

   ```
   Enter Export Password: clientpass
   Verifying - Enter Export Password: clientpass
   ```

2. Import the PKCS12 file into a new keystore:

```
$ keytool -importkeystore -destkeystore keystore.jks -srckeystore clientcert.p12
-srcstoretype PKCS12
```

At the prompt, set and confirm the password for the new keystore; then enter the password set when `clientcert.p12` was created:

```
Enter destination keystore password: password
Re-enter new password: password
Enter source keystore password:  clientpass
Entry for alias 1 successfully imported.
Import command completed: 1 entries successfully imported, 0 entries failed or
cancelled
```

You end with the `keystore.jks` file.

1. Copy the keystore file under the `<MQTT.COOL_HOME>/conf` folder:

```
$ cp keystore.jks <MQTT.COOL_HOME>/conf
```

2. Supply a new configuration in `brokers_configuration.xml`, let's use `pub-tls-client-auth-mosquitto` as a *connection alias*:

```
...
<param name="pub-tls-client-auth-
mosquitto.server_address">ssl://test.mosquitto.org:8884</param> ①
<param name="pub-tls-client-auth-mosquitto.truststore_path">truststore.jks</param>
②
<param name="pub-tls-client-auth-mosquitto.truststore_password">password</param> ②
<param name="pub-tls-client-auth-mosquitto.keystore_path">keystore.jks</param> ③
<param name="pub-tls-client-auth-mosquitto.keystore_password">password</param> ④
<param name="pub-tls-client-auth-
mosquitto.keystore_private_password">clientpass</param> ⑤

...
<param name="pub-tls-client-auth-mosquitto.clientid_prefix">...</param>
<param name="pub-tls-client-auth-mosquitto.connection_timeout">...</param>
<param name="pub-tls-client-auth-mosquitto.keep_alive">...</param>
...
```

① The port of the address is now set to `8884`, which supports client authentication.

② The truststore file and its password remain unchanged.

③ The keystore file.

④ The keystore password.

⑤ The private key password, which is the one set when `clientcert.p12` was created.

In this case, `brokers_configuration.xml` also comes with an example of configuration (named `public-tls-client-auth-mosquitto`) similar to the one proposed in this explanation. In addition, the `<MQTT.COOL_HOME>/conf` folder contains the referenced keystore file so that you can immediately test the configuration from you client application.

Now this new broker configuration will be able to authenticate to test.mosquitto.org listening at port 8884.

As a side note, TLS/SSL settings can be provided even through a custom Hook that returns a properly configured instance of the `MqttBrokerConfig` class. See the `getSecurityParams` method for more information.

C.3. Secure Channel with Dynamic Lookup

Establishing secure channels is even possible when the target MQTT broker is accessed through Dynamic Lookup, although such a mechanism limits the number of options you may use to customize a connection (as mentioned in Section 2.2, “Dynamic Lookup”).

You can set up an encrypted connection toward TLS/SSL-enabled MQTT brokers by explicitly providing a secure schema in the URI (similarly to what you have to do for the `server_address` parameter in the Static Lookup case) when invoking the MQTTCoolSession.createClient method on the client side:

```
...
var mqttClient = mqttCoolSession.createClient('ssl://secure.mqtt.broker:8883',
'myclientid');
...
```

The TLS/SSL connection will be successfully established if the JVM on which the MQTT.Cool server is running includes the trusted root certificates that can validate the identity of the contacted MQTT broker.

If this is not the case, then you have to import into your JVM’s default truststore the root certificate of the *CA* that issued the certificate for the MQTT broker. Please refer to the official Oracle documentation.

Client authentication is not currently supported by Dynamic Lookup.

C.4. Encrypted End-to-End Connection

Configuring a TLS/SSL connection is just one step toward a complete protection of data flowing to/from MQTT servers. Indeed, in order to realize a **fully encrypted** end-to-end connection between the client and the target MQTT broker, you must first make secure the communications between the client and the MQTT.Cool server:

1. Configure MQTT.Cool in *HTTPS/WSS* mode:

Follow the instructions of `<MQTT.COOL_HOME>/doc/MQTT.Cool SSL Certificates.html` to properly set up TLS/SSL certificates. Then, enable HTTPS listening as detailed by the inline comments of the `https_server` element in `<MQTT.COOL_HOME>/conf/configuration.xml`.

2. Open a secure session:

   From your client application, open an encrypted session against the secured MQTT.Cool server:

```
...
mqttcool.openSession('https://my.secure.server.company:443', {
   ...
}
...
```