knowledge++;
By Eric J. Bruno
if (success == true) {
(index.html)

(http://twitter.com/progwithreason) (http://facebook.com/programmingwithreason)

MQTT Programming In Depth

Eric J. Bruno
March 2016

MQTT is an open message protocol for machine-to-machine (M2M) or Internet of Things (IoT) communications that enables the transfer of telemetry-style data (i.e. measurements collected in remote locations) in the form of messages from devices and sensors, along unreliable or constrained networks, to a server. Andy Stanford-Clark of IBM, and Arlen Nipper of Cirrus Link Solutions invented the protocol.

MQTT's strengths are simplicity, a compact binary packet payload (compressed headers, much less verbose than HTTP), and it makes a good fit for simple push messaging scenarios such as temperature updates, stock price tickers, oil pressure feeds or mobile notifications. It also works well connecting constrained or smaller devices and sensors to the enterprise, such as connecting an Arduino device to a web service, for example.

How It Works

MQTT defines a small set of message types, such as:

Message Type	***Value***	***Description***
CONNECT	1	Client request to connect to Server
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request

SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting

Note that the message type values 0 and 15 are reserved. The message type is set as part of the MQTT message fixed header, which we'll examine now.

Message Fixed Header

MQTT messages contain a fixed header that includes flags in the first byte indicating the message type (four bits), whether the message is being re-sent (one bit), a quality-of-service (QOS) flag (two bits), and a message retention flag (one bit). The remaining portion of the fixed header indicates the length of the rest of the message, which includes a header that varies by message type (hence it's called a variable header), and the message payload.

bit	7	6	5	4	3	2	1	0
byte 1	Message type				DUP	QoS level		RETAIN
byte 2	Message length (between one and four bytes)							
byte 3	… if needed to encode message length							
byte 4	… if needed to encode message length							
byte 5	… if needed to encode message length							

The message types were described in the previous section. The DUP flag is set to 1 when a message is re-sent. In this case, the variable header portion of the message will contain a message ID to indicate specifically which message has been duplicated. The QoS level will either be 0 (meaning no guarantee of delivery), 1 (meaning it will never be lost but duplicate delivery may occur), or the strictest level of 2 (meaning exactly once delivery: never lost, never duplicated).

The RETAIN flag applies only to PUBLISHed messages, and indicates that the server should hold on to that message even after it has been delivered. Then, when a new client subscribes on that message's topic, the message will be sent immediately. Subsequent messages on the same topic with the RETAIN flag set will replace the previously retained

message. This is useful for cases where you don't want a client to wait for a published message (say, due to a change in value) before it has an existing value. One example is a grid of stock prices. For stocks that trade slowly, or when the markets are closed, a freshly connected client would otherwise contain missing values until price changes occur. The use of retained messages eliminates this by sending the last retained published message for the topic as soon as that client connects.

The length of the fixed header can be between one and four bytes long depending upon the length of the rest of the message. Hence, the fixed header can be between two and five bytes in length. An algorithm is used, whereby the last bit in each byte of the message length is used as a flag to indicate if another byte follows that should be used to calculate the total length. For instance, any value between 0 and 127 fit into the first byte of the message length portion of the header. If the message is greater than 128, more than one byte will be used. Up to four bytes may be needed, and the maximum message length is 256MB. For more details on the algorithm, see the specification, or read the code in Listing 1 below:

```
public class VariableLengthEncoding {
    public static void main(String[] args) {
        int value = 321;

        // Encode it
        Vector<Integer> digits = encodeValue(value);
        System.out.println(value + " encodes to " + digits);

        // Derive original from encoded digits
        int value1 = decodeValue(digits);
        System.out.println("Original value was " + value1);
    }

    public static Vector encodeValue(int value) {
        Vector<Integer> digits = new Vector();

        do {
            int digit = value % 128;
            value = value / 128;
            // if there are more digits to encode, set the top bit of this digit
            if ( value > 0 ) {
              digit = digit | 0x80;
            }

            digits.add(digit);
        }
        while ( value > 0 );

        return digits;
    }

    public static int decodeValue(Vector<Integer> digits) {
        int value = 0;
        int multiplier = 1;
        Iterator<Integer> iter = digits.iterator();
        while ( iter.hasNext() ) {
            int digit = iter.next();
            value += (digit & 127) * multiplier;
            multiplier *= 128;
        }
        return value;
    }
}
```

Listing 1 - The fixed header length algorithm explain in Java

The length applies to the remaining portion of the message and does not include the length of the fixed header (which can be between 2 to 5 bytes). Let's examine the rest of the message now.

Message Variable Header

Each message type contains its own header with applicable flags, called a variable header because it varies depending on message type. For instance, the CONNECT message, used when a new client application connects to the MQTT server, contains the protocol name

and version it will use for communication, among other things a username and password, flags that indicate whether messages are retained if the client session is lost (disconnects), and a keep-alive (heartbeat ping) interval. Whereas a PUBLISH message variable header contains the topic name and message ID only.

More information on variable header components for each of the message types can be found in the MQTT specification.

Subscriptions

When an MQTT client application wishes to subscribe to a topic, it first connects, then sends a SUBSCRIBE message. This message contains a variable length list of one or more topics (each specified by name) along with a QoS value for each topic that the client is subscribing to. The server will send a SUBACK message in response to the SUBSCRIBE request message, but there's no guarantee PUBLISHed messages for one or more of the topics subscribed will not arrive first.

In terms of the QoS level for a topic, a client receives PUBLISH messages at less than or equal to this level, depending on the QoS level of the original message from the publisher. For example, if a client has a QoS level 1 subscription to a particular topic, then a QoS level 0 PUBLISH message to that topic is delivered to the client at QoS level 0. However, a QoS level 2 PUBLISH message to the same topic is downgraded to QoS level 1 for delivery to that client. The server may also downgrade a QoS level for a particular topic in some cases.

Publishing Messages

The MQTT PUBLISH message has two use cases: first, the PUBLISH message type is sent by a client when that client wishes to publish a message to a topic. Second, the MQTT server sends a PUBLISH message to each topic subscriber when a message is available (i.e. a message has been published by another client).

When a client sends a PUBLISH message (hence, publishing a message), the message must contain the applicable topic name along with the message payload, and a QoS level for that particular message. The response to the sender will depend on the QoS level for the message sent:

- QoS 0: Make the message available to all interested parties. No response.
- QoS 1: Store the message in persistent storage, send to all interested parties, and then send a PUBACK back to the sender (in that order).
- QoS 2: Store the message in persistent storage (but don't send to interested parties yet), and then send PUBREC to the sender (in that order).

Let's examine the message flow per QoS level in more detail.

QoS Level 0 Published Messages

For QoS level 0 messages, the server simply makes the published message to all available parties and never sends a message in response to the sender, as shown in Figure 1 below.

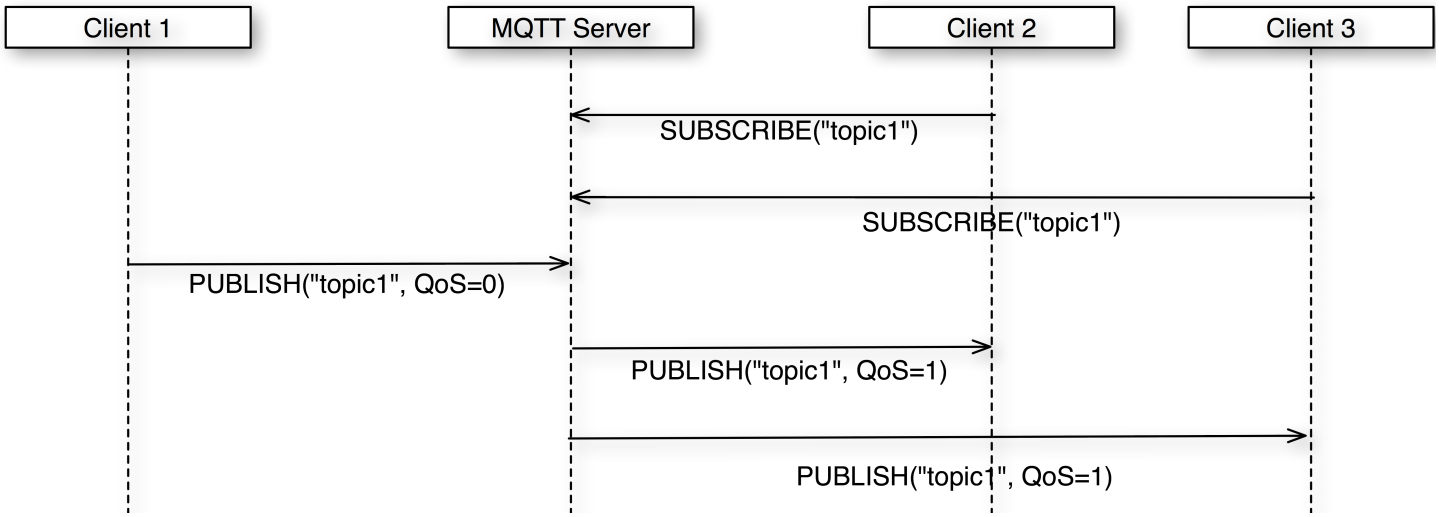

Figure 1 - Message flow for a QoS Level 0 published message.

Since there are no guarantees with QoS level 0 messages, neither the sender nor the server persist the messages.

QoS Level 1 Published Messages

For QoS level 1 messages, the message flow is very different than QoS level 0 messages, as shown in Figure 2 below.

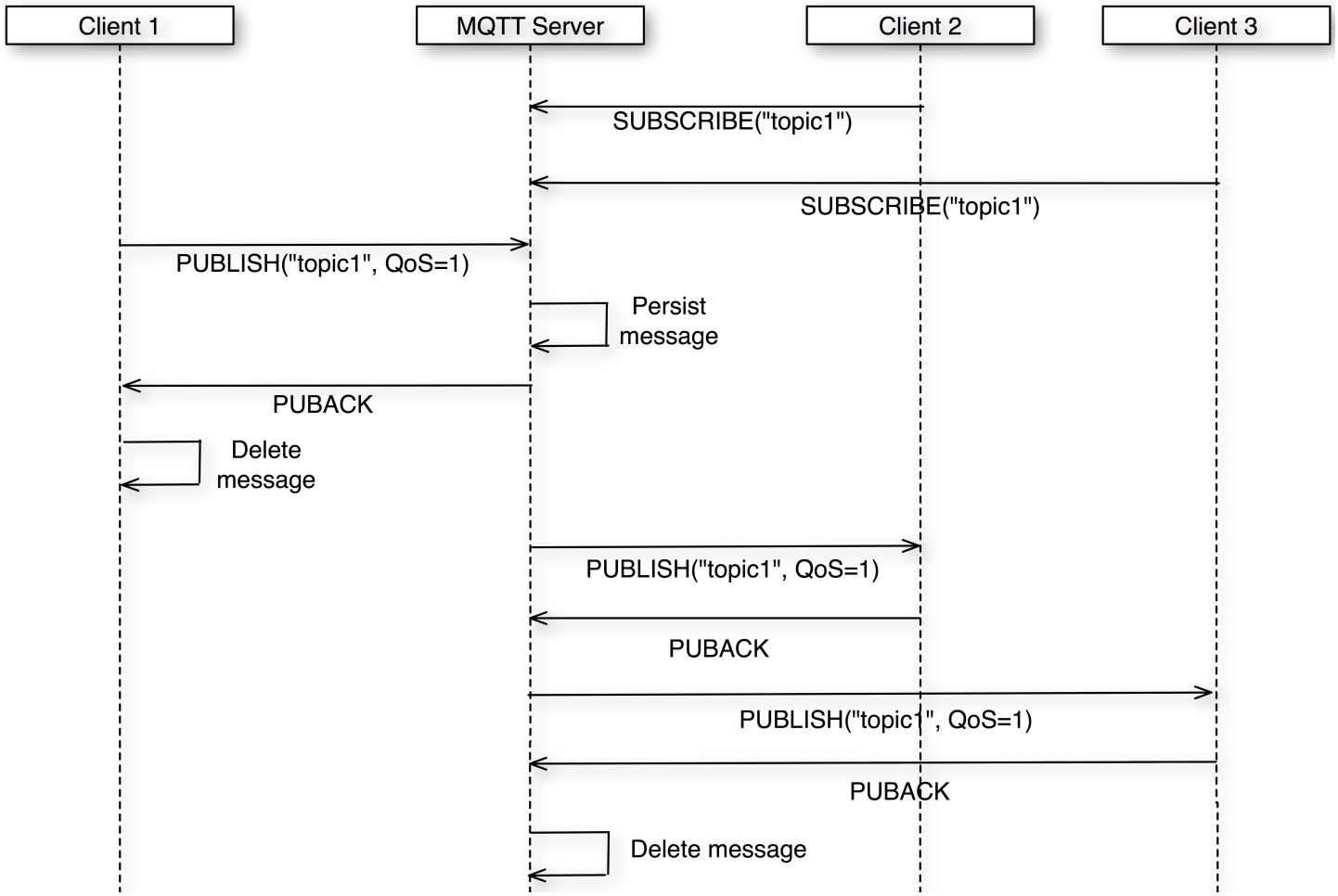

Figure 2 - Message flow for a QoS Level 1 published message.

When a client publishes a message, the MQTT server first persists the message to ensure delivery. Next, it delivers the message to each subscriber by sending them PUBLISH messages with the message payload. After each subscriber acknowledges the message, the server deletes the message from the persisted store, and sends the original sender an acknowledgement.

QoS Level 2 Published Messages

QoS Level 2 messages are the strictest in terms of message delivery, guaranteeing once-and-only-once message delivery per client. The message flow is as shown in Figure 3 below:

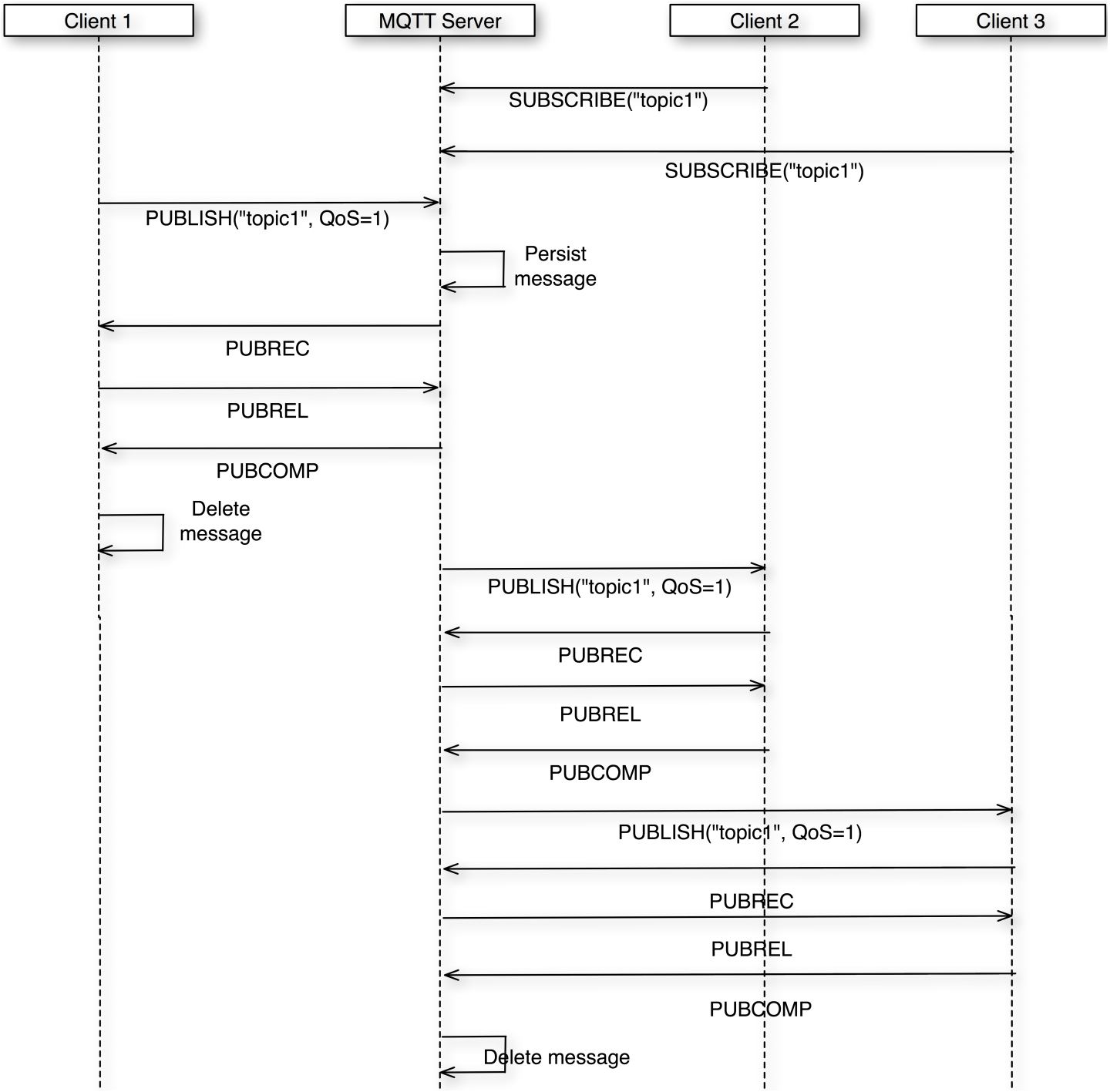

Figure 3 - Message flow for a QoS Level 2 published message.

When a client publishes a QoS level 2 message, the MQTT server first persists the message to ensure delivery. Next, it sends a PUBREC to the sender, who then replies with a PUBREL message. Next, when the server responds with a PUBCOMP message, the initial sender exchange is complete and the sender can delete the message from its persistent store. Next, the MQTT server delivers the message to each subscriber by sending them PUBLISH messages with the message payload. After message receipt, each subscriber sends a PUBREC message in acknowledgement, which the server responds to with a PUBREL (publish release message). After each subscriber acknowledges with a PUBCOMP message, the server deletes the message from the persisted store, and the QoS level 2 message send sequence is complete.

Sample Applications

Next, let's take a look at some sample MQTT applications written in Java, C++, Python, and JavaScript. To drive the sample, we'll use the Mosquitto open source MQTT broker (http://mosquito.org) running on a Raspberry Pi with a temperature sensor. A Java application will read the temperature sensor once per second and publish the current temperature reading on a queue named *currentTemp*.

The other sample applications will include a C++ listener, an Android listener (to view the temperature outside your home from your mobile), and a Python listener. All of the sample applications are available, just send an email with the article title to: eric@programmingwithreason.com (mailto:eric@programmingwithreason.com)

Setting Up the MQTT Broker

For this set of sample applications, I chose to use the Mosquitto open source MQTT broker. Although downloads are available for Windows, Linux (via `apt-get` if your distribution supports it) and Mac OS X (via Brew if you choose to use it). Simply download binaries for Windows and Mac OS X (via Brew), or install for Linux via the following command:

```
> sudo apt-get install mosquitto
```

This works on a Raspberry Pi running Raspian Linux as well. Using `apt-get`, Mosquitto will be installed in `/usr/sbin`, and you can start it directly from there. However, it's best to start Mosquitto on Linux as a System V init service (https://en.wikipedia.org/wiki/Init)

```
> sudo service mosquitto start
```

Subsequently you can stop Mosquitto with the following command:

```
> sudo service mosquitto stop
```

If you decide to explore the implementation and want to build Mosquitto from source, here are the steps that worked for me on both Linux and Mac OS X.

Building Mosquitto on Linux and Mac OS X

(Note: remember that you can always install Mosquitto the easy way on Debian-based Linux distributions via the `apt-get` command.) The steps to build Mosquitto on both Linux and Mac OS X are so similar they're described together here. Any differences are noted.

First, download the Mosquitto source code (http://mosquitto.org/download) To build from source, you'll need *CMake*, which is a cross-platform open source build system available at http://cmake.org (http://cmake.org). There are binary versions available for Windows, Mac OS X, and Linux (both 32-bit and 64-bit). You can also build from source for both Unix based and Windows systems, but the prebuilt binaries worked fine for me.

Next, you'll need the *c-ares* library, which is a C library for asynchronous DNS requests for hostname resolution.
Download c-ares at http://c-ares.haxx.se (http://c-ares.haxx.se), and then build it by executing the following:

```
> ./configure > sudo make install
```

After c-ares successfully builds, copy the entire c-ares folder to your Mosquitto source directory. Next, run the CMake application cmake-gui—just built—which opens a GUI window as shown in Figure 4. Set the source directory text field to your Mosquitto source directory, as well as the path where you want the binaries to go, in the text field below. I chose the Mosquitto directory as the base and appended `/bin` to the end (see Figure 4).

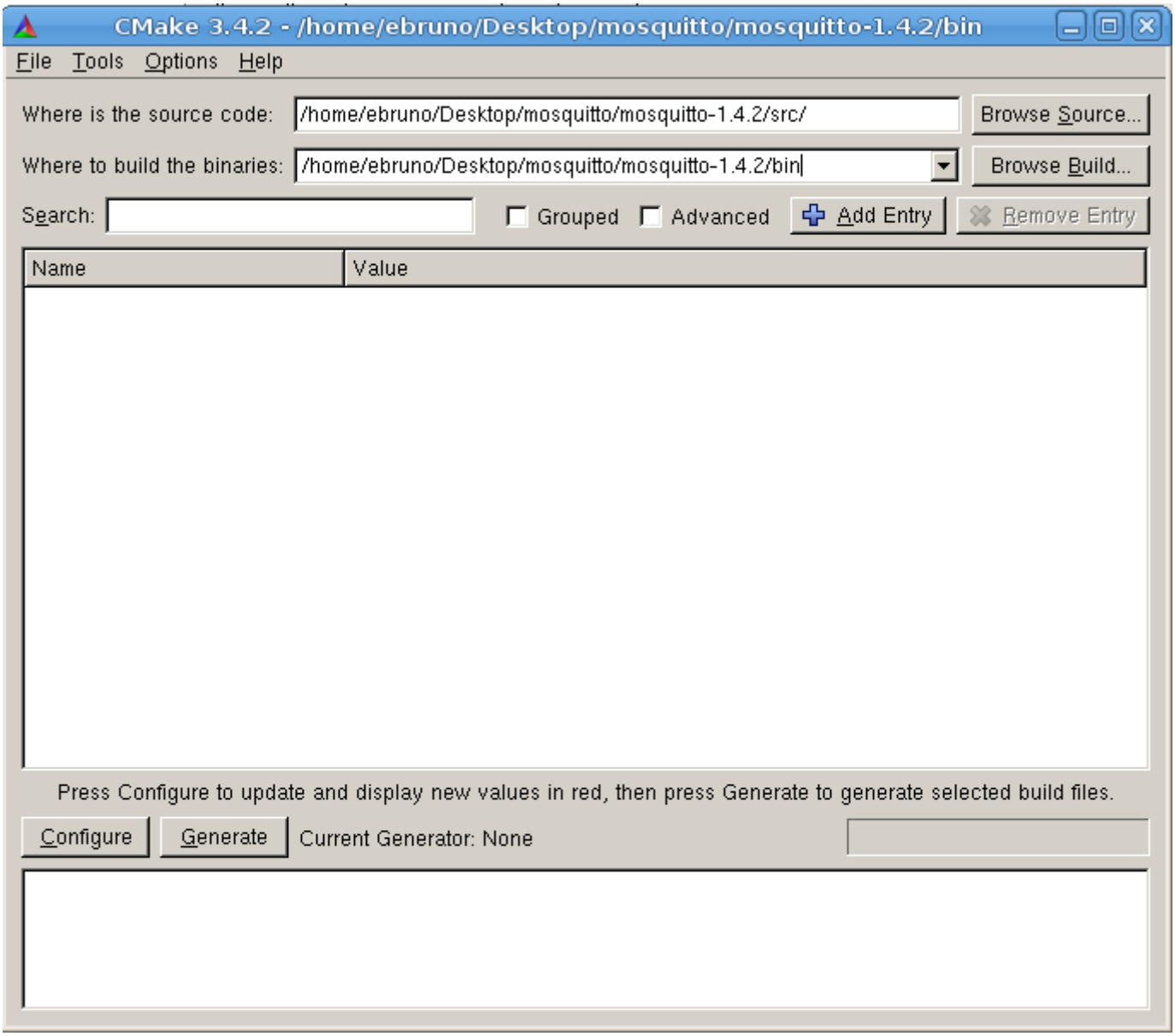

Figure 4 - The CMake build utility on Debian Linux.

Next, execute the CMake Configure and Generate steps by clicking on the respective buttons towards the bottom of the window, in that order. Once that completes without error, open a terminal and change to the directory you placed the Mosquitto source code to finish the build. On the Mac, you need Xcode installed to build the source. On Linux you may need the GNU Compiler, make utility and other libraries installed (described below). Then, enter the following commands:

```
> make
```

```
> sudo make install
```

Note that the installation directory will be different on Unix than if you used `apt-get` as described earlier. After building Mosquitto, it will be installed in the `/usr/local/sbin` directory. You can then start Mosquitto on a Unix-based system via the following command:

```
> /usr/local/sbin/mosquito
```

The default listen port is 1883, but you can specify a different port via the –p command line parameter when you start Mosquitto. You can also supply an optional configuration file via the –c command line parameter. Use this to specify alternative ports, certificate based encryption, an access control list file, logging directives, and much more. You can learn more about the configuration file here (http://mosquitto.org/man/mosquitto-conf-5.html) .

Linux Compiler, Tools, and Libraries

The following are some miscellaneous steps you may need to perform on your Linux system depending on how it's configured:

To install the GNU Compiler Collection:

```
> apt-get install gcc
```

```
> apt-get install g++
```

To install the make utility:

```
> apt-get install make
```

To install SSL libraries:

```
> apt-get install libssl
```

```
> apt-get install libssl-dev
```

You may also need to copy the download libraries to `/usr/local/lib` and set the `LD_LIBRARY_PATH` environment variable accordingly.

Sending and Receiving MQTT Messages from Java

As a starting point, we'll begin with a sample application that simulates reading the current temperature from a temperature sensor, which is then published as an MQTT message. To begin, create a Java project and add the Paho client library JAR file (needed to connect and communicate with an MQTT broker) to the classpath. If you don't already have it, you'll need to download or build Paho now.

Download or Build the Paho Client Libraries for Java

The Eclipse Paho project contains MQTT libraries for Java, Python, Android (not the same as Java), C/C++, JavaScript, and .Net. For Java, you can download the JAR files at `https://repo.eclipse.org/content/repositories/paho-releases`, or you can build from source from `https://repo.eclipse.org/content/repositories/paho-releases`.

To build the Paho JAR files, you'll need Apache Maven (http://maven.apache.org) . Next, define your `JAVA_HOME` environment variable or Maven will fail. For instance, on Mac OS X, you would define and export it similar to the following:

```
export
JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/Home
```

The command would be the same on Linux; make sure you use the correct path to the JDK on your local system. Once you've done this, execute the following command to build Paho:

```
> mvn package –DskipTests
```

Connect to the MQTT Broker

Next, connect to the MQTT message broker using the following code:

```
MqttClient mqttClient= null;
// ...

try {
     mqttClient =
      new MqttClient(
        "tcp://localhost:1883",    // broker URL
        "temp_sender",
        new MemoryPersistence() ); // or file-based

    MqttConnectOptions connOpts = new MqttConnectOptions();
    connOpts.setCleanSession(true);
    mqttClient.connect(connOpts);
    System.out.println("Connected");
}
catch ( MqttException me ) {
    // ...
}
```

Listing 2 - Java code to connect to an MQTT broker

You can connect to any MQTT broker by changing the address in the URL parameter of the `MqttClient` class constructor. For instance, instead of `tcp://localhost:1883` , you can specify a different address and even the port, such as `tcp://192.168.1.10:1880` . Either way, the URL must begin with `tcp://` .

The second parameter is the client ID that identifies the client connecting. This must be unique for all clients connecting to a specific broker. With the third parameter, you specify the way you want messages persisted at the client for guaranteed delivery. Keep in mind this does not affect message persistence at the broker, only messages in-flight to and from the client application.

For instance, this example uses `MemoryPersistence`, meaning messages will be stored in memory (which is fast) while awaiting notification from the broker that the message was received and/or delivered—depending on the QOS specified. If the message needs to be resent to the broker for some reason, it's retrieved from the memory store. In this case, message delivery is guaranteed only if the client application isn't shut down before the broker receives and stores the message. For additional reliability, choose FilePersistence instead (which can be slower). This guarantees message delivery even in cases where the client is shutdown or crashes while messages are in-flight, and subsequently restarted.

Finally, call `MqttClient.connect` to connect to the specified broker. You can provide connection options via a parameter, such as minimum MQTT protocol version requirements (via `setMqttVersion`), whether or not to maintain state across restarts of the client (via `setCleanSession`), authentication details, timeout intervals, and more.

Publishing MQTT Messages

Once your application has connected to a broker, it can begin to send and receive messages. MQTT messages payloads are sent as byte arrays, making it easier to avoid cross-platform or cross-language issues. The following code (see Listing 3) sends the current temperature, originally formatted as a `String`, to the topic `currentTemp`.

```
private void publishTemp(Long temp) {
    try {
        String content =
            "Current temp: " + temp.toString();

        MqttMessage message =
            new MqttMessage( content.getBytes() );
        message.setQos(2); // once and only once delivery

        mqttClient.publish("currentTemp", message);
    }
    catch ( MqttException me ) {
        // ...
    }
}
```

Listing 3 - Java code to publish an MQTT message

Here, you can see how the message is formatted as a `Byte` array, and then sent with the highest level of message quality of service (once-and-only-once delivery).

Receiving MQTT Messages

To receive messages, first the application needs to connect to an MQTT broker as shown in Listing 2 in the previous section. Next, subscribe to a topic (or multiple topics) by calling the method `subscribe()` on the `MqttClient` class, providing the topic and requested quality of service level as parameters:

```
mqttClient.subscribe(“currentTemp”, 2);
```

Messages published at a lower quality of service (QoS) will be received at the published QoS. Messages published at a higher quality of service will be received using the QoS specified in the call to subscribe. Next, provide the callback class to handle new messages as they’re received on this topic:

```
mqttClient.setCallback(this);
```

The Java class supplied must implement the `MqttCallback` interface and define its three notification methods: `messageArrived()`, `deliveryComplete()`, and `connectionLost()` as shown below:

```
@Override
public void messageArrived(String topic, MqttMessage mqttMessage) throws Exception {
    //
    // Called when a message arrives for a topic subscribed to
    //
    System.out.println("Topic: " + topic + ", Message: " + mqttMessage.toString() );
    if ( msgTextView != null ) {
        final String msg = mqttMessage.toString();
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                msgTextView.setText( msg );
            }
        });

    }
}

@Override
public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
    // Called when the message is known to have been delivered successfully...
}

@Override
public void connectionLost(Throwable throwable) {
    System.out.println( "Connection lost: " + throwable.getMessage() );
    throwable.printStackTrace();
}
```

Listing 4 - Implement the MqttCallback interface to receive messages and other notifications.

The method `messageArrived()` is called when a message is available from the broker for the supplied topic name. Providing the topic name is helpful in cases where you subscribe to multiple topics with a single callback class. Optionally, you can set a separate callback class for each topic, or some other arbitrary grouping. The sample code in Listing 4 is part of an Android app, which simply updates a text field with the latest message text for the applicable topic.

The `deliveryComplete()` method is called when a message your application has sent has been delivered and received, as well as all associated message acknowledgements according to the quality of service level specified. If the connection to the server is lost for some reason (i.e. a network failure, or if the server is shut down) the `connectionLost()` method will be called with some information about the issue provided as a `Throwable`.

Sending and Receiving MQTT Messages with Python

To send and receive MQTT messages from a Python application, download and setup the Paho client for Python (https://www.eclipse.org/paho/clients/python) . You can either use the pip tool to install Paho via the command below, or build the client it from source (which works on any platform).

To install via pip, execute the following command:

```
> pip install paho-mqtt
```

To build and install the Paho Python client from source instead, download and expand the source archive, and change to the directory you downloaded it to. Next, execute the following command:

```
> python setup.py install
```

On a Unix-based system (Linux, Mac OS X, Raspberry Pi) you may need to insert `sudo` before the python command shown above, supplying the root password when prompted.

Sending messages from a Python application can be very simple. For example, import the `paho.mqtt.publish` class, and any other classes you may require for your app, and then publish a message to the specified broker as shown in Listing 5 below.

```
import paho.mqtt.publish as publish
publish.single("currentTemp", "70", hostname="localhost")
```

Listing 5 – Simple publishing of an MQTT message from a Python application.

This simple application connects and published to the specified broker all in one statement. However, as with the Java example in Listing 3, you can connect to the MQTT broker once in Python, then subsequently publish messages (see Listing 6).

```
import paho.mqtt.client as mqtt
client = mqtt.Client()
client.connect("localhost", 1883, 60)
client.publish("temp/outside", "70")
```

Listing 6 - Python code to connect to an MQTT broker first.

Subscribing to topics and receiving MQTT messages in Python is also straightforward, as shown in Listing 7.

```
import sys
import paho.mqtt.client as mqtt

# The callback when the client connects to the broker
def on_connect(client, userdata, rc):
    client.subscribe("currentTemp")

# The callback when a message is received from the server
def on_message(client, userdata, msg):
    print(msg.topic+": "+str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(sys.argv[1], 1883, 60)

# Blocking call that processes network traffic, dispatches callbacks
# and handles reconnecting to the broker
client.loop_forever()
```

Listing 7 - Python code to subscribe and receive MQTT messages.

First, the application defines the two callback methods, `on_connect` and `on_message` which are called when the application connects to the MQTT broker and when a message is received, respectively. The application subscribes to the `currentTemp` topic when on_connect is called. Next, an instance of `paho.mqtt.client` is created, the callbacks are set, and `connect` is called to connect to the broker. In this sample, the MQTT broker URL is passed in as a command line argument, with the port and connection timeout values hardcoded.

Subsequently, the `on_message` callback will be called when a message is received for a topic the application has subscribed to. The message, a reference to the associated `paho.mqtt.client` object, and the private user data (if provided) are all sent as parameters.

Sending and Receiving MQTT Messages with C/C++

To build C/C++ MQTT applications, you can use the Paho C/C++ client or you can use the header and library files that come with the MQTT broker you're using (such as Mosquitto in this case). Let's explore an example of this now, and then we'll look at how to use the Paho client afterwards.

Sample Mosquitto C++ Sender Application

The first steps in building a Mosquitto C++ application is to extend the `mosqpp::mosquittopp` base class. This class defines methods your class should override to be notified when the broker connection succeeds, a message is received, and so on (see Listing 8).

```
class TempSender: public mosqpp::mosquittopp {
  bool connected;

  public:
    TempSender(const char* id, const char* host, int port);
    ~TempSender();

    void send(void);
    void outputError(int rc);

    void on_connect(int rc);
    void on_disconnect(int rc);
    void on_message(const struct mosquitto_message* message);
    void on_subscribe(int mid, int qos_count, const int* granted_qos);
};
```

Listing 8 - Implementing Mosquitto C++ callback methods.

You'll also need to include the `mosquito.h` and `mosquittopp.h` header files (include paths explained later) along with any others that are required. Before connecting to the broker or executing any MQTT-related code, you need to initialize the Mosquitto library via a call to `mosqpp::lib_init` (see Listing 9). To ensure all resources are cleaned up, call `mosqpp::lib_cleanup` before the application terminates.

```
#include <cstdio>
#include <cstring>
#include <mosquitto.h>
#include <mosquittopp.h>
#include <unistd.h>
#include "tempsender.h"

class TempSender* sender = null;

int main(int argc, char *argv[]) {
        mosqpp::lib_init();

        // Connect to the Mosquitto MQTT Broker
        sender = new TempSender("mqtt_sender", "localhost", 1883);

        // Send the current temperature
        long temp = sender.getCurrentTemp();
        sender->publish(temp);

        mosqpp::lib_cleanup();
        return 0;
}

TempSender::TempSender(const char *id,
                       const char *host,
                       int port ) : mosquittopp(id) {
    int keepalive = 60;

    connected = false; // Gets set in on_connected()

    connect(host, port, keepalive);

    loop_start();
};
```

Listing 9 – Connecting to a Mosquitto MQTT Broker from C++.

The code to publish over MQTT is shown in Listing 10. Here, you provide an optional message ID (to be used to track the message acknowledgement), the topic name, the length of the message payload in bytes, and the message payload byte array.

```
void TempSender::publish(long temp) {
    if ( ! connected )
        return;

    // Format the message
    char buf[51];
    memset(buf, 0, 51*sizeof(char));
    snprintf( buf, 50, "Temperature %d", temp );

    // Publish the message over MQTT
    int rc = publish( NULL, "currentTemp", strlen(buf), buf );

    // Check for success
    if ( rc == MOSQ_ERR_SUCCESS ) {
        printf("%sn", buf);
    }
    else {
        printf("PUBLISH Failed: ");
        outputError(rc);
    }
}
```

Listing 10 - Publishing a message from a Mosquitto C++ application.

Next, let's see how to configure the C++ build environment both from the command line and from within an IDE such as Eclipse.

Compiling and Linking a C/C++ Mosquitto Project

To build a Mosquitto C++ application, you need to include header files from the `lib` directory and `lib/cpp` directory, found in the directory where you built Mosquitto. Also, link both `mosquito.`[ver]`.so` (i.e. `mosquittopp.1.4.2.so`) and `mosquittopp.`[ver]`.so` library files, found in the directory where you built (or installed) Mosquitto under the `lib` directory and `lib/cpp` directory respectively.

For instance, for the code in Listing 9 and Listing 10, the compile statement should be similar to below (all on one line):

```
> g++ -I/Users/ericjbruno/mosquitto/mosquitto-1.4.2/lib
-I/Users/ericjbruno/mosquitto/mosquitto-1.4.2/lib/cpp
```

To link the project, the linker statement should be similar to below (again, all in one line):

```
> g++ -L/Users/ericjbruno/mosquitto/mosquitto-1.4.2/lib
-L/Users/ericjbruno/mosquitto/mosquitto-1.4.2/lib/cpp
-o "mqtt_receiver" ./src/mqtt_receiver.o
-lmosquittopp.1.4.2 -lmosquitto.1.4.2
```

To set the environment within Eclipse, create a new C++ project (make sure you have the C/C++ plugin installed), then go to *Project Settings* by right-mouse clicking on the project name. Next, expand the "*C/C++ Build*" entry in the tree on the left and locate your

compiler in the Settings area on the right (see Figure 5). This example uses the GCC compilers. Make sure you enter the paths to the Mosquitto include files, as described above.

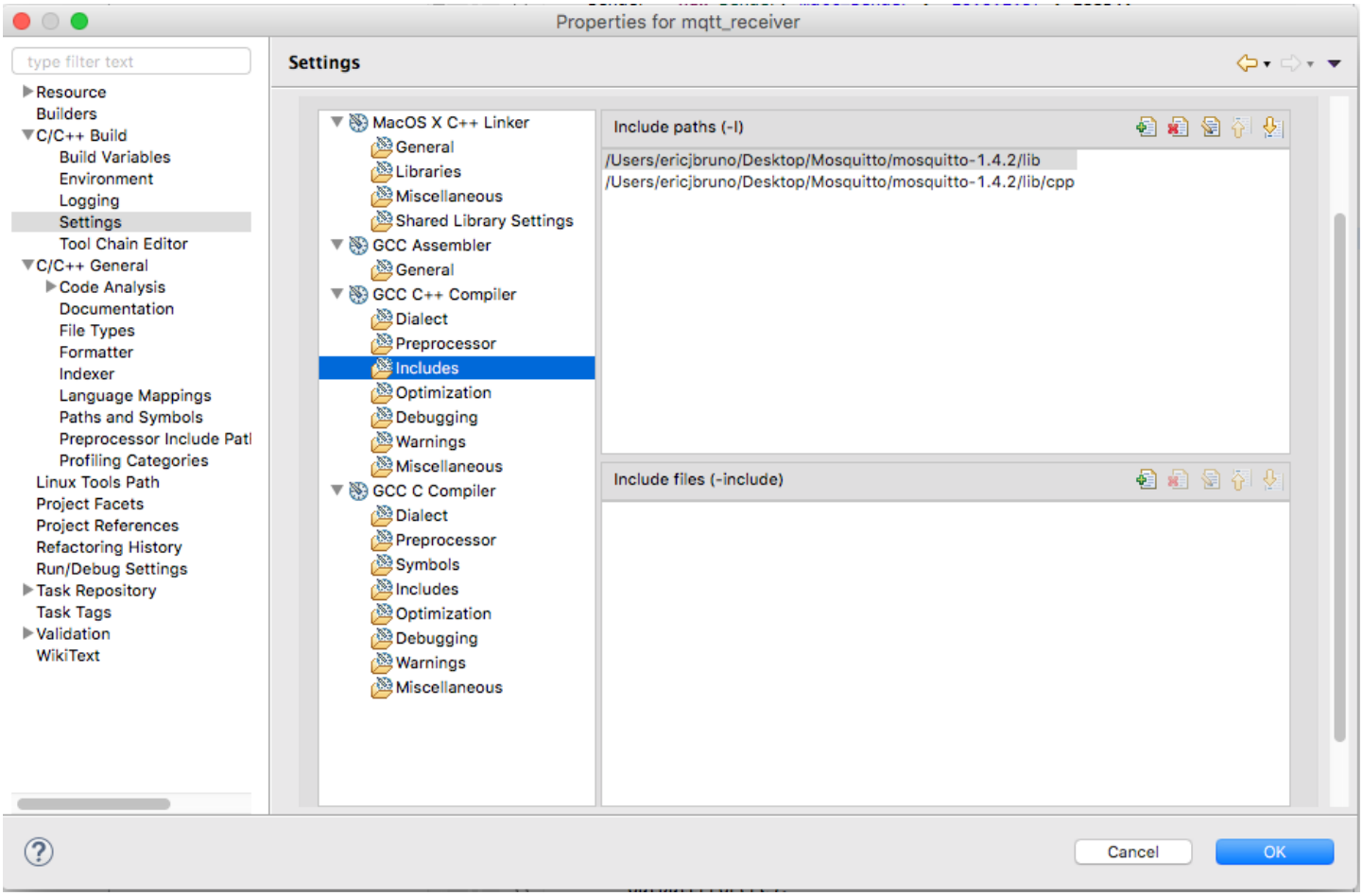

Figure 5 - Including the Mosquitto C++ header files in Eclipse.

Now you need to configure the linker settings, which you can do within the same window. However, this time you need to select the linker section within the right-hand panel of the window (see Figure 6).

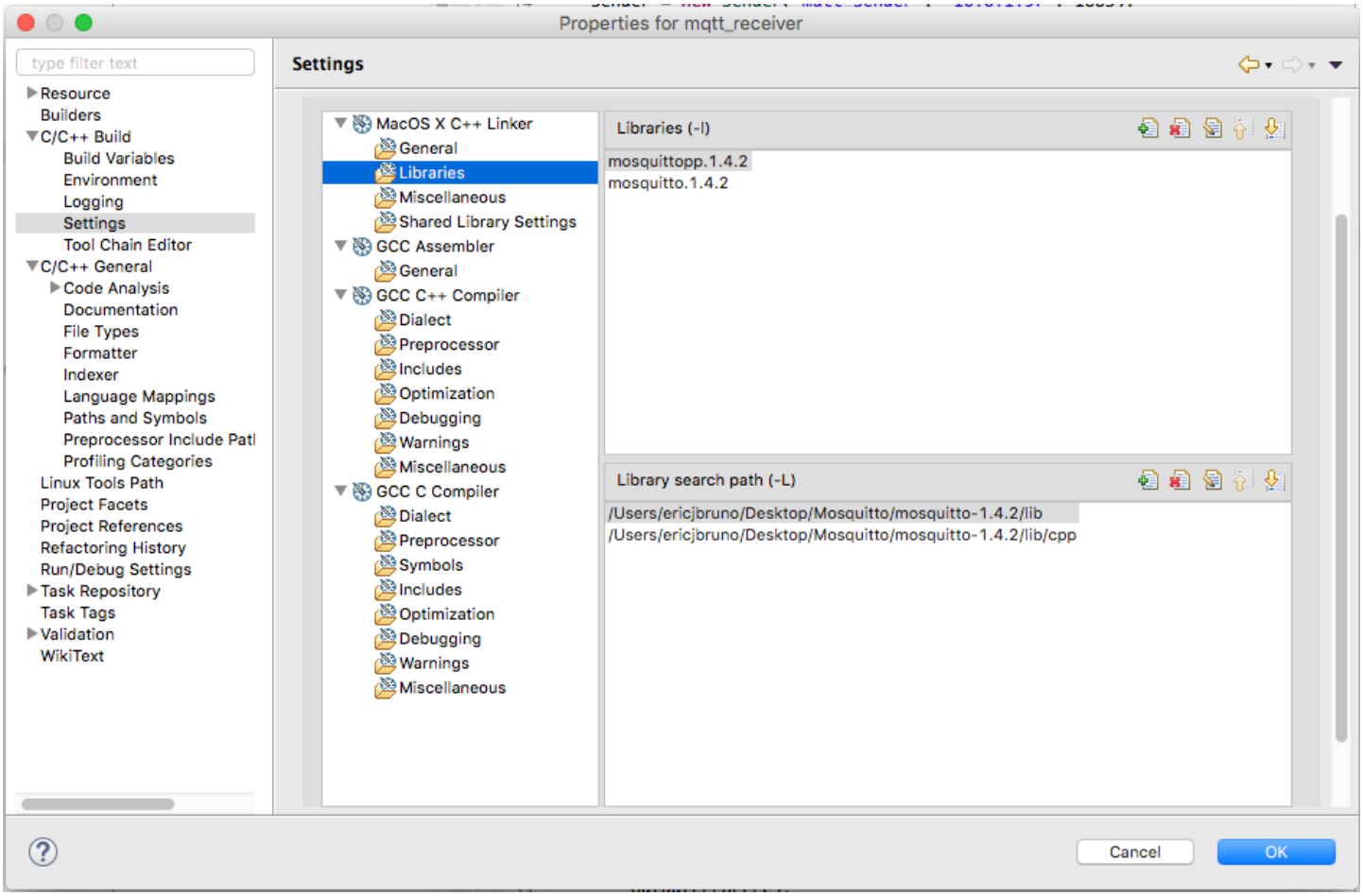

Figure 6 - Setting the linker environment for Mosquitto in Eclipse.

This example uses the Mac OS X Linker that comes with XTools, but the process should be similar for the OS or tools that you use. Add the two Mosquitto libraries in the “Libraries” section at top, and then the path(s) to the libraries in the “Library search path” section below. Once complete, your MQTT projects should compile, link, and execute all from within Eclipse.

Using the Paho C/C++ Client Libraries

If you decide to write code that’s independent of any MQTT broker, or you want use MQTT for embedded devices such as Arduino, you can use the Paho Embedded C/C++ library (https://www.eclipse.org/paho/clients/c/embedded) or C Client for Windows, Linux, and Mac OS X (https://www.eclipse.org/paho/clients/c) . Although you can use C API from within a C++ application, there’s also a convenient C++ wrapper (https://www.eclipse.org/paho/clients/cpp) as well.

Using the generic C API and the C++ wrapper is straightforward and similar to the Mosquitto API. First, implement appropriate callback classes, such as `mqtt::callback` and `mqtt::iaction_listener` to receive new message delivery notifications and failure notifications, respectively (see Listing 11).

```
class callback : public virtual mqtt::callback,
                 public virtual mqtt::iaction_listener {

    // Re-connect to the broker here
    void reconnect()
        // ...
    }

    virtual void on_failure(const mqtt::itoken& tok) {
        // ...
    }

    // connection success
    virtual void on_success(const mqtt::itoken& tok) {
        // ...
    }

    virtual void connection_lost(const std::string& cause) {
        reconnect();
    }

    virtual void message_arrived(const std::string& topic,
                                 mqtt::message_ptr msg) {
        std::cout << "Message arrived" << std::endl;
        std::cout << "  topic: '" << topic << "'" << std::endl;
        std::cout << "  msg: '" << msg->to_str() << "'n" << std::endl;
    }

    virtual void delivery_complete(mqtt::idelivery_token_ptr token) {
        // ...
    }

public:
    callback() { }
};
```

Listing 11 - Using the Paho MQTT C++ library to create the callback.

Connecting to the broker, setting up the callback, and subscribing to topics is similar to the previous C++ examples (see Listing 12).