Session: M253

Fun with IoT Messaging

Lab instructions

Author:

Leo Davison
Software Engineer
IBM Internet of Things
<leo.davison@uk.ibm.com>

Minor Updates:

Jon Levell
Messaging Engine Tech Lead
IBM IoT MessageSight & IBM Watson IoT Platform
<levell@uk.ibm.com>

Introduction

In this lab session you will enable the multiplayer features of a simple browser based game by adding the messaging infrastructure necessary for the game clients to communicate indirectly with each other. You will be able to achieve this simply, and quickly, by making use of the open-source JavaScript MQTT client from the Eclipse Paho project.

What is MQTT

MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging “machine-to-machine” (M2M) or “Internet of Things” world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.

Because MQTT is a protocol that outlines the structure of message data which is sent over the network, rather that the specific interface that client applications use, client and server implementations do not have to be supplied from the same project or vendor. Provided a given implementation correctly conforms to the protocol, you can mix and match clients and servers.

What is Publish/Subscribe

Publish/Subscribe is a pattern for messaging in which message senders (*publishers*) and message receivers (*subscribers*) do not have any direct communication, or knowledge of each other. Instead of publishers sending messages to a specific known subscriber, they send (*publish*) messages to *topics*, which group/filter messages into named, logical, channels on an MQTT server.

Published messages may be delivered to zero, one, or more subscribers who have registered an interest in the topic to which the message was published. Importantly, the publisher has no knowledge of if, or when, any given message is delivered to any subscribers of the topic. Note that unlike message queuing, this means that a published message may be delivered to more than one subscriber.

What is the Eclipse Paho project

The Eclipse Paho project is a collection of open-source MQTT client implementations. There are clients available for a wide range of programming languages including, but not limited to, C, Java, JavaScript and C#.

As you will explore today, utilising these clients enables you to quickly add publish/subscribe messaging capabilities to your applications when used alongside an MQTT compatible messaging server.

Which MQTT Server will be used

In this lab session we will be making use of IBM IoT MessageSight 2.0

The IBM IoT MessageSight offering is a server product specifically designed to act as the messaging backbone for Internet of Things (IoT) and mobile environments. It provides a secure, high-performance, simple to configure, messaging infrastructure capable of supporting large numbers of concurrently connected client devices.

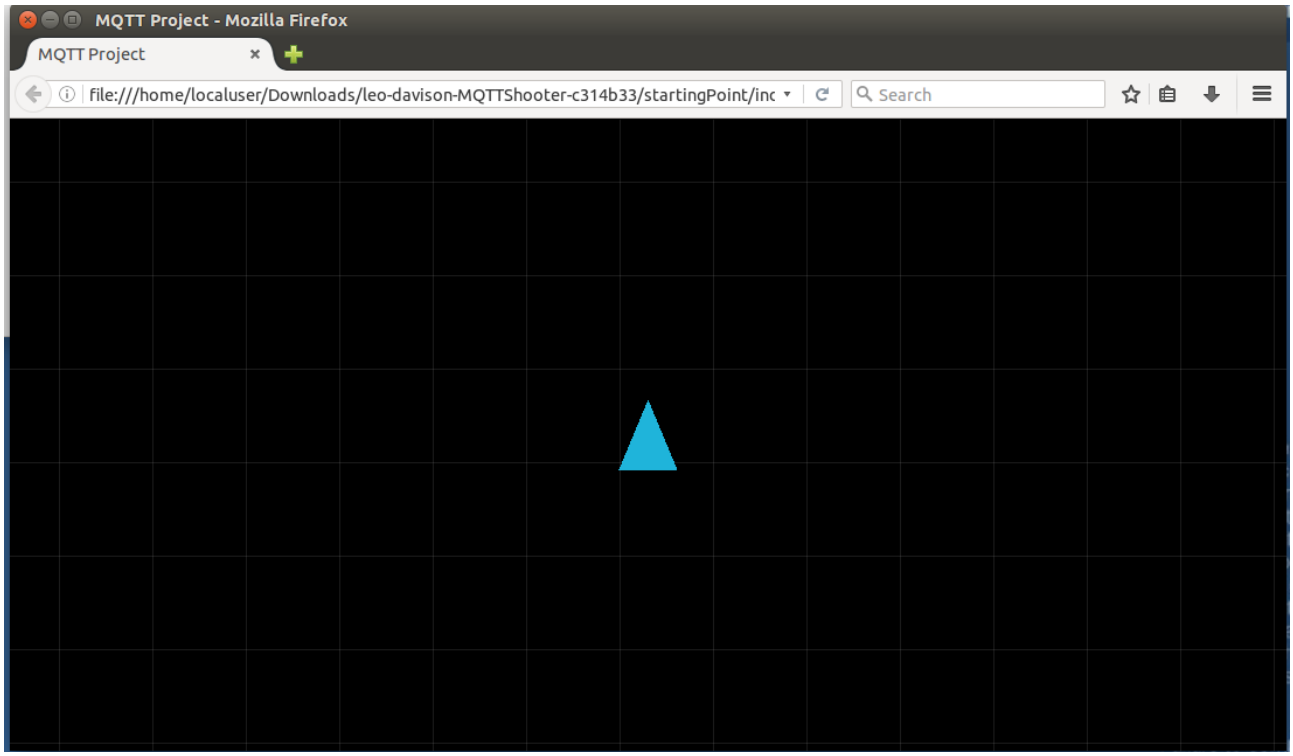
The lab host will run a single instance of the MessageSight Server and provide you with the details necessary to connect the MQTT client you will use.

Overview of the game and the primary task

The application to which you will be adding messaging is a simple JavaScript game that runs in the browser.

Initially, the game consists of a single spaceship (represented by blue triangle) which can be controlled using the arrow keys and space bar. The ship can be flown around an otherwise empty scene and can fire projectiles (represented by green dots). The game area is bounded by a green outline, which the ship will bounce off when flown in to.

In its initial state, the game is not particularly interesting. It would be much better if there were other ships, controlled by other players, which you could battle against.



In fact, the game code does include the logic to handle multiple remote players via functions which accept remote player key-strokes and ship updates. What's missing however, is a means to send/receive this data to/from other players.

This is where MQTT can help. Your task will be to complete the implementation of a JavaScript object which the game networking logic uses to send local player data, and to receive remote player data. The initial implementation consists of functions which are empty except for logging statements which print a “todo” message to the console.

Before providing instructions on how to proceed with the implementation, an overview of the high level architecture of the game, and the design of the multiplayer logic is provided.

High Level Game Architecture

At its simplest, a video game is an application which presents a rendered image which is updated over time based on some simulation and/or player input. This is achieved by creating an application loop that performs the following tasks over and over:

1. gather player input
2. update the simulation (using input and/or some other calculations eg. physics)
3. render the current view of the game world to the screen

This *game loop* occurs many times per second (eg 60 times per second), and creates the illusion of an interactive moving image.

The two most important elements of the simulation in this game are the **controllable ships** and the **projectiles**.

The **controllable ships** are JavaScript objects represented by an on-screen triangle, containing logic to rotate and move based on whether particular keys are pressed.

The **projectiles** are JavaScript objects represented by an on-screen dot, and contain simple logic to move based on an initial starting position and a constant velocity. The object also has the concept of a 'lifetime', which is a number of seconds that the object will remain active for, before disappearing.

The instances of **controllable ships** and **projectiles** that exist in the game are contained within a **game scene** object. This object provides the orchestration of **step 2** in the *game loop* – it ensures that all of the elements in the simulation (namely the **ships** and **projectiles**) are updated before their on-screen representations are displayed.

Another important function fulfilled by the **game scene** object is collision detection and collision response between the **ships** and the **projectiles**. Each time through the *game loop*, after updating the positions of all the **ships** and **projectiles**, the **game scene** checks whether any of the ships have been hit by any of the **projectiles**.

If a collision is detected, then the response issued by the **game scene** is to remove the **projectile** from the simulation and to reset (zero out) the **ship's** position and velocity and to remove it from the simulation for a small number of seconds (i.e. a cool-down after being hit by a **projectile**).

Game Multiplayer Networking Design

The basic principle of a multiplayer game is that several instances (clients) of a game agree on the state of a simulation, and changes caused by a local player in one instance are reflected in the other, remote, instances of the game and vice versa.

This game adopts non-authoritative model, in that there isn't a central multiplayer server which updates the simulation based on input from multiple clients. Instead, each client (instance of the game) performs a local simulation of all the objects in the game.

In order to ensure that the simulations are kept in synchronisation across all instances of the game, certain local events are published as messages to topics. Each game client subscribes to these topics in order to receive updates from other game instances. This remote data is used to affect the local simulation.

The local events/data which each game client publishes can be grouped into three main categories:

1. ship information
2. projectile information
3. collision information

Ship information messages primarily consist of changes to the movement input, eg. left arrow pressed or released. This information is used to update the local simulation of remote ships. In addition to this, each client also publishes a periodic message containing additional state for it's local ship. This additional information contains the position and velocity of the ship. This data, when received by another game instance, is used to override the local simulation of a remote player ship. This is done to ensure that differences in timing/floating point rounding don't cause the simulations to become too far out of sync.

Projectile information messages are sent each time a local ship fires a projectile. They contain the starting position and velocity for a projectile, along with the name of the player which launched it. When a game client receives these messages from a remote client, it adds an equivalent projectile to it's own local simulation.

Collision information messages are sent by a game client when it's local simulation has determined there has been a collision between the local player ship and a projectile which was launched by a remote player ship. Note that each local simulation only checks for collisions that occur against the local player – which means that each client is the authority on whether or not a projectile hit the local player ship in it's simulation. This is to ensure that there is no potential discrepancy between the multiple simulations (eg. where some clients believe that a ship was hit, but all the other clients do not).

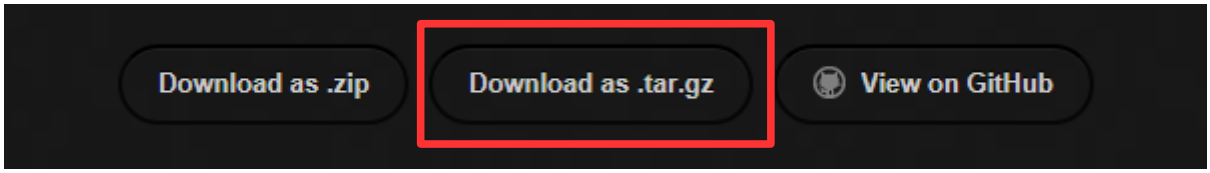
Lab Instructions

1: Download and extract the project

i. Start Firefox (left-click on the relevant icon on the toolbar on the left of the screen).

   The home page should be set to http://leo-davison.github.io/MQTTShooter/, if not, navigate there.

ii. Click on the "**Download as .tar.gz**" link, and select "S**ave File**" in the dialogue that appears and let it save it in the Downloads directory (default location).

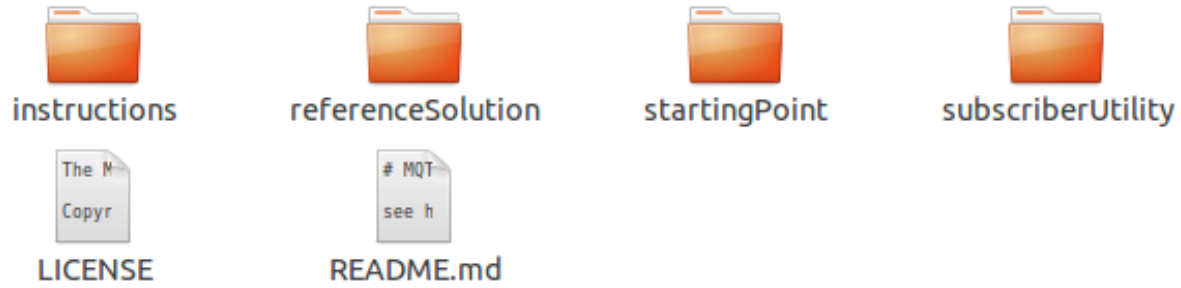


iii. When the download has finished, load the file browser and navigate to the Downloads.

iv. Right click on the downloaded .tar.gz file and click on "**Extract Here**" to unpack it.

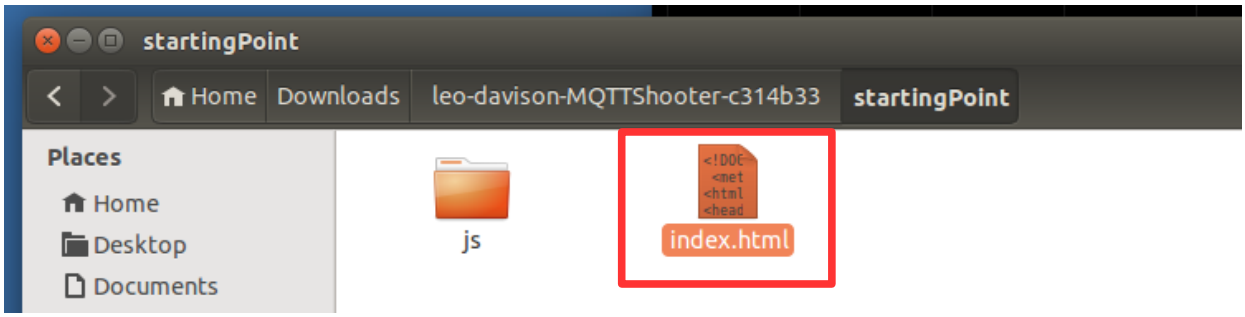
2: Verify the starting project works

Inside the extracted directory, you should see the following contents:

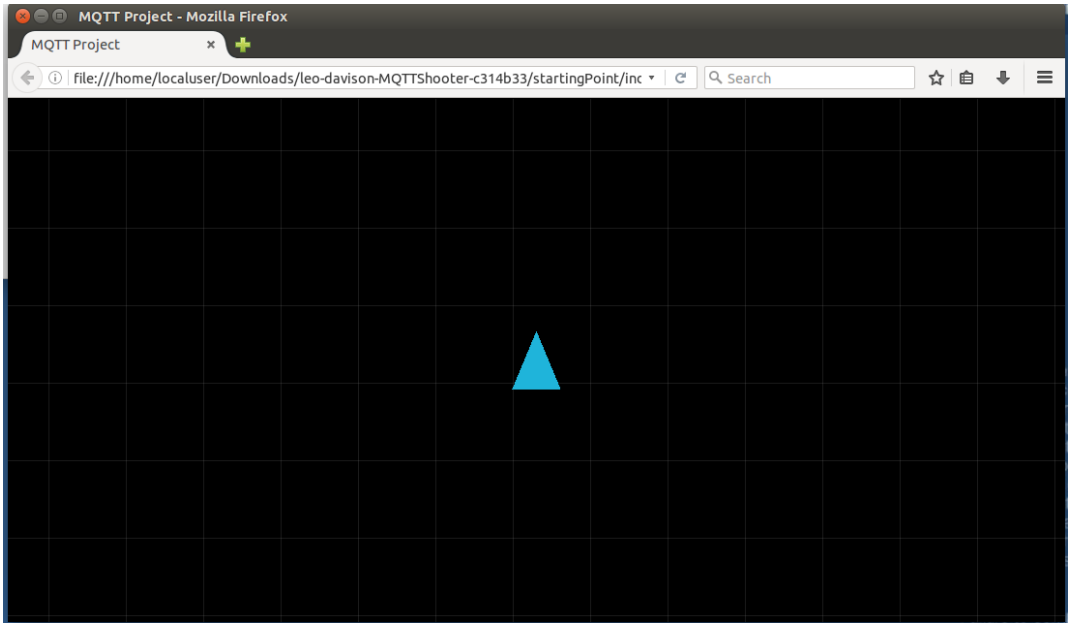


- The *instructions* directory contains a copy of these instructions.
- The *referenceSolution* directory contains a finished copy of the lab code
- The *startingPoint* directory contains the initial project code that you will work with
- The *subscriberUtility* directory contains a simple web page that allows you to connect to an MQTT server and create subscriptions to topics in order to see any messages that are published

i. Navigate into the ***startingPoint*** directory.
ii. Open the ***index.html*** file with Firefox (left double-click).



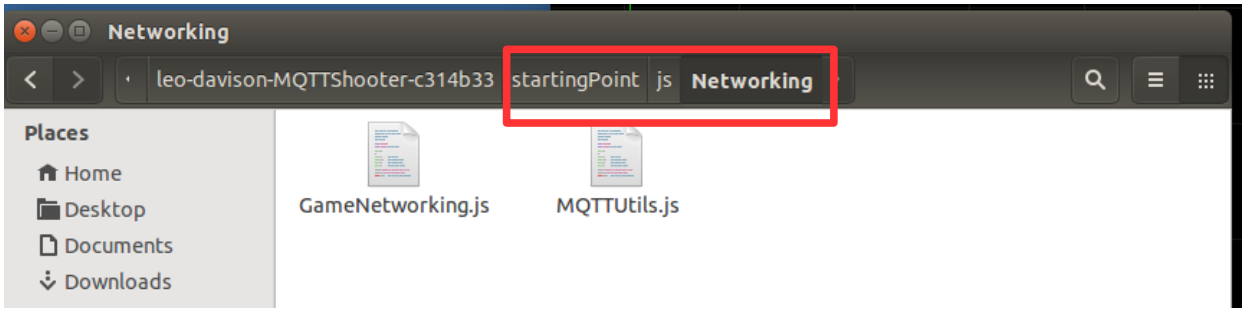
iii. When Firefox opens you should see a triangle representing the player space ship, atop a black background and a light grey grid. Verify that you are able to turn the ship using the left and right arrow keys, apply forward thrust using the up arrow key, and fire a projectile using the space bar key on the keyboard. (to reset the scene, just refresh the web page).



3: Setup the MQTT server connection details

Before you implement the messaging capabilities used by the game networking logic, you will need to update some configuration variables which hold the host information of the MQTT server that the client will connect to, and the name of the game session used to differentiate between multiple games using the same MQTT server.

i. Returning to the file explorer window where you opened the ***index.html*** file, navigate into the ***js/Networking*** directory.



ii. The ***GameNetworking.js*** file contains all of the code for implementing the multiplayer logic for the game. It does not have any messaging capabilities itself. Instead, it has a function which expects to be called with data from a remote client, and when it wishes to send it's own data it calls out to another object – the ***MQTTUtils*** object.

iii. The ***MQTTUtils*** object is defined in the ***MQTTUtils.js*** file. Unfortunately, the object does not currently contain any implementation for the messaging functions it is supposed to provide for use by the game networking code. You will implement these functions in steps 4 through 7.

iv. Open the ***GameNetworking.js*** file with ***gedit*** (left double click). You need to update the following variables: ***Networking.gameName*** and ***Networking.server***.

```
Networking.gameName = "game_mygamename";
Networking.server = "bandicoot0.hursley.ibm.com";
Networking.port = 20004;
```

v. ***Networking.server*** defines the host for the MQTT server. An appropriate hostname or IP address string will be provided to you by the lab host.

vi. ***Networking.gameName*** is used to identify an instance of the game. All clients which share the same server and game name will be part of the same game. The value is inserted into the topic strings (which define the logical channels) that the games messages are published to (and subscribed to). While you are developing your solution, you should choose a value that will be unique to you so that you don't overlap with other lap participants, perhaps including your name. For example: **`"game_LeoDavison".`**

vii. Save the ***GameNetworking.js*** file and then close it. You should not need to modify it further until you have completed the remaining steps.

4: Implement the function to connect to the MQTT server

From within the ***js/Networking*** directory of the starting solution, open the ***MQTTUtils.js*** file with ***gedit*** (left double-click). Notice that it contains the definition of a JavaScript object which has a number of stub implementations of functions. The functions that have names beginning with a capital letter are those which are being used by the game networking code – for example **`ConnectToServer`** and **`SubscribeToTopic`**.

During initialisation of the game, the networking code will call the **`ConnectToServer`** function of the **`MQTTUtils`** object, so you should implement this first. The Paho JavaScript MQTT client library has been included in the project and is available for you to use.

The primary object that you will create and interact with in order to make use of the Paho library is the **`Paho.MQTT.Client`** object. This object contains a number of functions which enable you to connect to an MQTT server, publish messages and subscribe to topics.

The documentation for this object can be found here (a link can also be found on the homepage configured in Internet Explorer, under the **Useful Information** section):
http://www.eclipse.org/paho/files/jsdoc/symbols/Paho.MQTT.Client.html

The homepage for the Paho JavaScript MQTT client also contains a small, but complete example of how the library can be used. You may find this useful as a reference (again, a link can also be found in the **Useful Information** section of the homepage):
https://www.eclipse.org/paho/clients/js/

i. In ***gedit*** review the **`ConnectToServer`** function stub and delete the log statement:

```
ConnectToServer : function(clientID, server, port, onConnectCallback, onConnectionLostCallback, willTopic, willData) {
    console.log("TODO: connect to server");
},
```

ii. We need to perform a number of actions over the next few steps in order to properly implement this function. These can be summarised as:
   1. create a new **`Paho.MQTT.Client`** object
   2. setup callback functions to handle events such as a new message arriving or a dropped connection
   3. create a new **Paho.MQTT.Message** object which will be used as a ***will message*** – this is a message that will be automatically published in the event that our client is abnormally disconnected.
   4. Use the **`connect`** function of the **`Paho.MQTT.Client`** object we created

iii. Create the new **`Paho.MQTT.Client`** object with the following code:

```
this.client = new Paho.MQTT.Client(server, port, clientID);
```

Note: the **clientID** variable is a string name which is used to uniquely identify this client to the MQTT server.

iv. The **Paho.MQTT.Client** object provides a number of callback facilities that enable you to respond to particular events. Two that we are interested in inform us when messages are received, and if our connection to the server is lost. Notice that there are two function stubs in this file which can be assigned to these callback properties in the **Paho.MQTT.Client** object, **onMessage** and **onConnectionLost**. Assign these callbacks using the following code:

```
this.client.onConnectionLost = this.onConnectionLost;
this.client.onMessageArrived = this.onMessage;
```

v. Two of the arguments passed to the **ConnectToServer** function are called **onConnectCallback** and **onConnectionLostCallback.** These are references to functions that wish to be called when a server connection is established or lost, respectively. This code should keep a reference to these functions so that we can ensure they are called at the appropriate times. Do this using the following code:

```
this.userOnConnect = onConnectCallback;
this.userOnConnectionLost = onConnectionLostCallback;
```

vi. If you refer to the documentation for the Paho client, you will see that the **Paho.MQTT.Client.connect** function takes, as an argument, a JavaScript object which can specify a number of properties (which affect how the client connects to the server). For our purposes, we are only interested in two of the properties. The first is the **onSuccess** property, which allows us to specify a callback function which will be called once a connection request has been successfully actioned. We create the connection options object and assign the user provided callback to the **onSuccess** property using the following code:

```
var connectOpts = {onSuccess: onConnectCallback};
```

vii. The remaining arguments that were passed to the **ConnectToServer** function are **willTopic** and **willData**. These specify the topic a message should be sent to in the event that this client is abnormally disconnected from the server, and the data that the message should contain. The will message property is the second of the connection options that we are interested in, and it expects to be given a **Paho.MQTT.Message** object. Create this object and set the appropriate connect options property using the following code:

```
this.willMessage = new Paho.MQTT.Message(willData);
this.willMessage.destinationName = willTopic;
connectOpts.willMessage = this.willMessage;
```

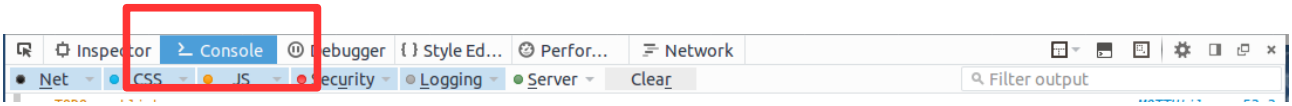
Note: We are storing a reference to the will message in our **MQTTUtils.Client** object. This is so that we can reuse the message if we need to reconnect to the server at any point

viii. The final step in our **`ConnectToServer`** function is to call the **`connect`** function on our **`Paho.MQTT.Client`** object. This (if successful) will result in the function assigned the **onSuccess** connection property being called. Add this call using the following code:

```
this.client.connect(connectOpts);
```

ix. Finally, save the ***MQTTUtils.js*** file and switch to the Internet Explorer window in which you previously opened the ***index.html*** file. If you closed it, reopen it now (from the ***startingPoint*** directory).

x. Inside Firefox, press **F12** to open the developer tools, and select the **Console** tab:



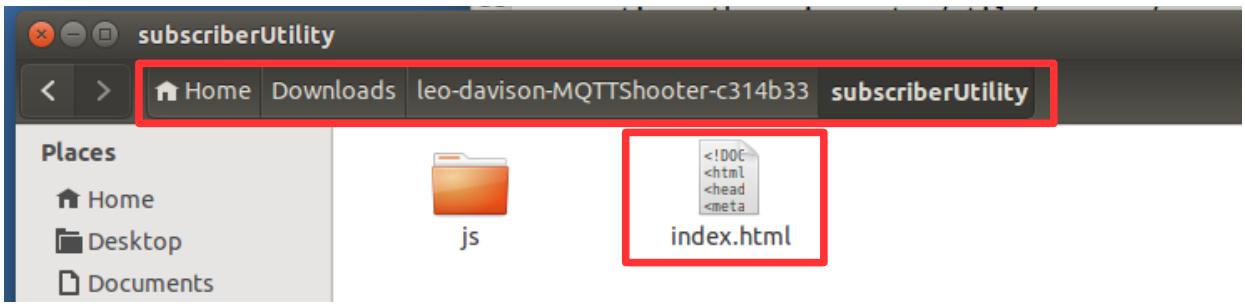
xi. Refresh the page in order to reload the code and check that you see a log message in the developer tools console with the text "**on connect**". There may be warnings in the console from third-party libraries that deliberate use deprecated functions to be compatible with a wide range of browsers - these can be ignored. This log message is issued by the networking code in the **`onConnect`** callback, and indicates that you have successfully established an MQTT connection to the server. You will also notice additional log messages in the console containing text such as "**TODO: publish a message**". This indicates that the networking code, having been connected, is now attempting to send data using the **`MQTTUtils`** functions which we have not yet implemented:

5: Implement the function to publish messages

The next task will be to implement the **`MQTTUtils`** function (**`PublishMessage`**) which enables the networking code to send data out to the network by publishing messages.

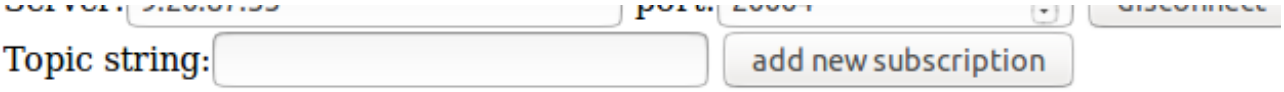
In order to aid the development of this feature, the code that was downloaded contains a simple web page which allows you to connect to a server, create subscriptions, and then view any messages that are published to those subscriptions. This can be used to verify that messages are in fact being successfully published once you have implemented the **`PublishMessage`** function. Let's open this now.

i. Return to the top level directory of the downloaded project, and navigate into the ***subscriberUtility*** directory and open the ***index.html*** file in Firefox:



ii. Using the same server details you were previously given, enter the **IP** and **port** in the form on the page and click on the **connect** button:

iii. If a connection is successfully established, a new section of the form will appear that will allow you to enter topic strings to subscribe to:



iv. Each time you add a subscription, you will see a new text area appear on the page, which shows any messages received on the specified topic, for example:

Note on topic strings: A topic string is made of one or more topic levels which are separated by a forward slash. Typically the levels are used to progressively group a topic space into more specific categories. For example: *sports/tennis/players/roger_federer/stats*. Topic strings can also contain certain wildcard characters, '+' and '#'. The '+' is a ***single level wildcard*** and can be used to match any topic at a single level in a topic tree. For example:
sports/tennis/players/+/stats which would match both
sports/tennis/players/roger_federer/stats and
sports/tennis/players/andy_murray/stats

The '#' is a ***multi-level wildcard*** and will match any number of levels within a topic. The '#' wildcard must be specified on its own or following a '/'. In all cases, it must be the last character of the topic string. For example:

sports/# would match
sports/tennis and
sports/football and
sports/tennis/players

Whereas *sports/+* would match
sports/tennis and
sports/football but **not**
sports/tennis/players

There are additional rules and restrictions pertaining to wildcards. Such information, along with a detailed specification of the MQTT protocol can be found here:
http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html#_Toc398718106

Adding the publish functionality

We will use the subscriber utility page to subscribe to the topics used in the multiplayer logic, but first let us implement the message publishing functionality.

i. Return to editing the ***MQTTUtils.js*** file from the ***startingPoint/js/Networking*** directory.

ii. Review the **`PublishMessage`** function in the **`MQTTUtils`** object. Notice that it takes two arguments, a **`topic`** to publish the message to, and **`messageData`** which should be contained in the payload of the message that is published:

```
PublishMessage : function(topic, messageData) {
    console.log("TODO: publish a message");
},
```

iii. The **`Paho.MQTT.Client`** object provides a **`send`** function which will publish a message to the connected server. So, all we need to do is create a new **`Paho.MQTT.Message`** object and pass this to the **`send`** function. Remove the log statement in the body of the **`PublishMessage`** function and replace it with the following code:

```
var msg = new Paho.MQTT.Message(messageData);
msg.destinationName = topic;
this.client.send(msg);
```

iv. Save the ***MQTTUtils.js*** file and switch back to the browser window that contains the game screen.

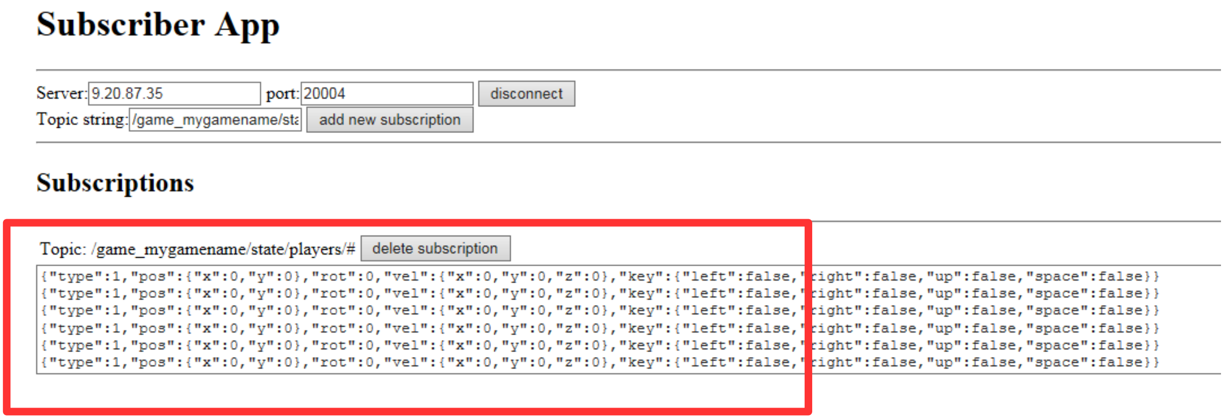
v. Refresh the game window in Internet Explorer to reload the code. In order to validate that the game is now able to publish messages, add a subscription to the subscriber utility page of the following form:

/<your_game_name>/state/players/#

where <your_game_name> is the value you assigned to **`Networking.gameName`** in step3. So, if you had set a game name of “game_LeoDavison”, then you should subscribe to:

/game_LeoDavison/state/players/#

If you have created the appropriate subscription in the subscriber utility, and you have implemented the **`PublishMessage`** function as above, and refreshed the game web page, you should see messages being received such as this:



6: Implement the function to subscribe to topics

Now that the game client is able to publish messages, we want to be able to subscribe to topics so that we can receive published messages from other players. Again, the **`Paho.MQTT.Client`** object provides us a useful function for achieving this – the **`subscribe`** function.

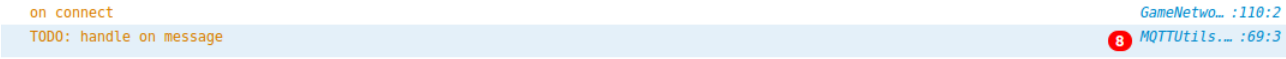
i. Return to editing the ***MQTTUtils.js*** file and review the **`SubscribeToTopic`** function. Notice that it takes a single argument, which is the topic string to which we wish to subscribe:

```
SubscribeToTopic : function(topic) {
    console.log("TODO: subscribe to a topic");
},
```

ii. All we need to do now, is replace the log statement with the following code:

```
this.client.subscribe(topic);
```

iii. If you save the file, and switch back to the game web page in Internet Explorer (and enable the developer tools using **F12** if they aren't still open) and refresh the page, you should observe that we now see new messages being logged to the console:



iv. These log messages show that the we are receiving messages that have been published to subscriptions to which the networking code has subscribed to, using our newly implemented function. These log messages are being output from the **`onMessage`** function of the **`MQTTUtils`** object. Currently, these messages are not being passed on to the network code.

Note: The reason you are seeing messages being delivered to the game are because the network code subscribes to a topic including a wildcard, and so we are receiving our own published messages.

7: Implement the function to handle the receipt of messages

In order for the networking code to be able to process the messages being received, the **`MQTTUtils`** object needs to pass them on. This is what the **`messageHandler`** is for. The networking code provides the **`MQTTUtils`** object with a function to call whenever messages are received. This is assigned to the **`messageHandler`** property.

i. Return to editing the ***MQTTUtils.js*** file and review the **`onMessage`** function. Notice that it takes a single argument, which is a message that has been received. Also recall that we previously configured the **`Paho.MQTT.Client`** object to call this function when messages are delivered to us:

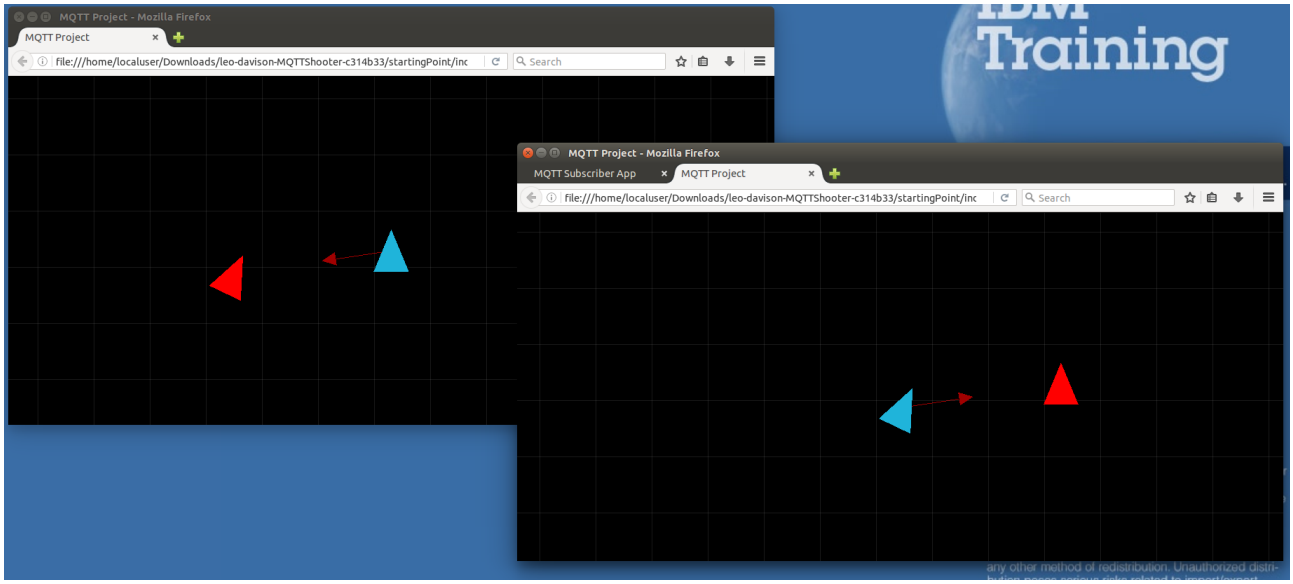
```
onMessage : function(message) {
    console.log("TODO: handle on message");
},
```

ii. All we need to do is check that we have been given a handler function to call, and then call it with the received message. Replace the log statement with the following code to achieve this:

```
if (MQTTUtils.Client.messageHandler != null) {
     MQTTUtils.Client.messageHandler(message);
}
```

iii. Save the file, and switch back to the game web page in the browser. If the developer tools are still open, you can close them now.

iv. Refresh the page to reload the code. Open a second instance of the game in a new browser window. You should now see two ships (although initially they will be on top of each other, so fly one of them out of the way):



v. You should now be able to fire projectiles at the other ship to temporarily remove it from play, and both game windows should show the two ships in the appropriate positions.

When you have reached this point, close one of your game instances and then ask the lab host for the shared game name which will enable you to join a game with the other lab participants. You will use this to update the **`Networking.gameName`** variable in the ***GameNetworking.js*** file.

Lab Summary

You have taken an application with no messaging capabilities and, through the use of the Eclipse Paho JavaScript MQTT client and the IBM MessageSight for Developers Virtual Appliance, you have added the ability for the application to communicate via Publish / Subscribe messaging.

This was achieved in what amounts to 3 basic steps:

1. Creating a connection to a server
2. publishing messages
3. subscribing to topics

This small amount of additional application code means that data can be exchanged asynchronously between zero to many other applications or devices.

Received data could have originated from another web client such as the application you worked with, or it could just have easily have been published from a native application on a mobile phone, or even from sensors attached to a person!

Bonus Challenge for the Reader

If you have been able to complete the core lab task and have some time remaining, then feel free to attempt the following extension. The lab host will be happy to provide additional guidance or ideas.

Create a score table web page for the game

No real game is complete without some sort of scoring mechanism. Using what you have learnt in this lab session, create a new web page which includes JavaScript code to subscribe to the same topics as the game client. Using the data you receive, assign points to a player each time they successfully hit another ship with a projectile. Display these scores, updating live, in the web page.

Can you make the scores available to new instances of the scores web page?

Hints/Tips:

- collision information is published to */<game_name>/state/collisions/<hit_player_name>*
- perhaps retained messages could be useful for keeping the scores around? Research retained messages, and use the Paho documentation to learn how to make a message retained
- The ***GameNetworking.js*** file constructs data that is sent, review it to determine message structure

References

IBM MessageSight: http://www-03.ibm.com/software/products/en/messagesight

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

MQTT: http://mqtt.org/