**Documentation MQTT**

The language of choice for our team to use as a base to code the MQTT protocol in Julia was the implementation in embedded C. The biggest initial issue the team faced was figuring out exactly what the protocol was and where to begin.

The way the program is intended to work is based on modules. By using modules we allow the program to have new elements to be easily added and altered rather then risking the need to change the entire program to implement or change a single feature of the protocol.

The most useful thing we used to test the code was the inclusion of the load.jl file. This allowed us to run the Julia program in command line which we found to be much more helpful. It also allowed us to put in print statements both before and in some cases after the methods that were called for certain things. Adding these print statements allowed us to more accurately find where the program fell down. if the program failed to enter a particular method then the print statements allowed us to more find that the method was the reason the program crashed as we could see it ran the previous method and then crashed when the new method was called.

The basis of the connect packet is to establish a connection to the server with a socket assigned to the connection to be able to keep the connection open and to be able to use the connection. If there is already a connection to the server then the new attempt at a connection will simply be rejected. In the event that the client is connected to the server the program must have a way in which to keep the connection open(alive). If the connection is left idle for too long then it will terminate. Normally the keep alive call just send a ping to the server at certain increments which are controlled by a timer.

The connect function must also serialize the connect in order to ensure both the client and the server are able to read the connect. Once the connection to the server is established and has been successfully serialized the client can then send the connect packet. If any part of the program does not behave as expected(i.e a flag not set to the right value, flags in the wrong order etc) then the connection will be terminated.

The send packet needs the client, length and timer all to be passed into the function when it is called. This is done in the connect function. As part of the connect function the program must send a connect packet to the server in order to successfully connect. Once the server receives the connect packet it deserializes the packet, reads the data, checks that all the data is correct, and will perform one of 2 actions. If all of the information is valid and in the right order then the server will accept the connection and send the client a CONACK packet in return, but, if the information is incorrect, in the wrong order or there is any other issue with the connect packet, then the server will terminate the connection.

When the connect packet is accepted the server must then construct and send back a CONNACK packet. If the session flag is 1 when the connect packet is sent then the server must set the session present flag to 0 in the CONNACK packet. Must also set a zero return code. If the session flag is set to 0 then the session present depends on if the server has already stored a session state for the ClientID in question. The session present must be 1 if the server has stored a session state. The server must return a session state 0 if there is no stored session state. This also results in setting a zero return code in the CONNACK packet that is received by the client. As long as the CONNACK packet is received and read properly then the connection should be fully completed as expected.

Once the connection is fully established we then moved onto the subscribe packet. We initially ran into an issue with the subscribe packet that we didn’t understand. After trying to run the code in a number of different ways it was discovered that there was no function created for the handler to pass into the subscribe packet. This was found to be the issue as when an empty function was written and then called into passed into the subscribe function then their appeared to be no issues(apart from the fact that we didn’t know what we needed to make the function do), but at least it meant we then knew what the issue was. Once all the variable are successfully passed into the Subscribe function the first thing the function does is check to see if the client is still connected. If the client is not connected then the will return an MQTTCLIENT\_FAILURE. As mentioned before if any part of the protocol does not happen in the intended order or with the corrected expected values then the request will fail next the client must serialize the subscribe and get the length of the subscribe so that it can inform the broker when it sends the subscribe packet how large the packet is. This is important as the broker must allocate resources in order to be able to handle the subscribe packet correctly and efficiently. Along with length the client must also send the broker the connection itself and a timer. The reason for the timer is to ensure the broker knows how long the client is going to wait before the connection will time out. The client will now wait patiently for the broker to respond to the subscribe packet with a SUBACK packet. The client also expects to receive back the client and the timer that it sent. Using waitfor allows the client to wait for the response before allowing the program to proceed as it cannot successfully carry out the next step without successfully completing the previous step in the chain. When the client receives the response from the broker the client must check that the response is what was expected. To do this the client must deserialize the response and check 2 things. First it will check the QoS of the response it receives. If the QoS = 0x80 then rc is set to MQTTCLIENT\_FAILURE, this represent a failed topic subscription. If QoS is set to anything that is not 0x80 then this means the subscription was a success and the rc is then set to MQTTCLIENT\_SUCCESS. In order to display any issues that might occur in this part of the function we have surrounded it with a try catch statement. This way any problem that may be unexpectedly encountered we can use a print statement in the catch to say exactly where the error occurred and print what the error was.

To date we have successfully managed to get the program to send the connect packet and receive and verify the CONNACK packet.