Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33031 Computer Networks

Assignment #1: Publish/Subscribe Protocol

Alice Doherty, Student ID 19333356

November 2, 2021

Contents

1 Introduction 2
2 Theory of Topic 2
2.1 Internet Protocol (IP), Ports and Sockets 2
2.2 User Datagram Protocol (UDP) and Datagrams 3
3 Implementation 3
3.1 Header Design 3
3.2 Node 5
3.3 Publisher 5
3.4 Subscriber 8
3.4.1 Subtopics and Wildcards 10
3.5 Broker 10
3.5.1 PUBLISH Packets 11
3.5.2 SUBSCRIBE Packets 12
3.5.3 UNSUBSCRIBE Packets 13
3.6 Additional Features 14
3.6.1 Acknowledgements & Quality of Service 1 14
3.6.2 Retained Messages 15
3.7 Docker 15
4 Discussion 16
4.1 Header Design 16
4.2 Lua Wireshark Extension 17
5 Summary 18
6 Reflection 19
7 References 20

1 Introduction

The idea behind a publish-subscribe protocol is that there are publishers who continuously send out messages. Rather than these messages being sent directly to the desired recipient, they are sent to a broker who then forwards the messages onto interested receivers, or subscribers. These messages are categorized using topics. Each message a publisher sends out is associated with a topic, the broker then forwards on the message to any device subscribed to that topic.

At a high level my approach was to design three main components, a publisher, a broker, and a subscriber. Each of these can be run inside their own container in Docker to simulate the distinct devices they represent. My own custom header is implemented to carry the management data necessary to support the functionality between the components.

2 Theory of Topic

A large part of the time spent on this assignment involved researching the theory necessary to realise my solution. To avoid any assumptions being made about the concepts and protocols I have used, I will give an outline of the theory that underpins my solution. Specifically, in this section I will discuss the Internet Protocol, ports, sockets, the User Datagram Protocol, and datagrams.

2.1 Internet Protocol (IP), Ports and Sockets

When we talk about the Internet Protocol (IP), we are referring to the network layer in the Open Systems Interconnection (OSI) model. IP is used to deliver packets from a source host to a destination host which are identified using IP addresses. IP addresses are a unique and universal sequence of 32 or 128 bits depending on whether IPv4 or IPv6 are used. For this assignment, each container that is used to run each publisher, subscriber, or broker has its own IP address which is used to identify where to send packets to.

A port represents a communication endpoint. Whereas an IP address is used to identify a host in a network, a port number is used to identify a process or service. In my solution, the port number is always 50000 and is used to identify my protocol.

A socket is used as an endpoint for sending and receiving packets across a network. In my implementation, each broker, subscriber, and publisher instance has its own socket. This is defined using the DatagramSocket class, from the java.net library [1], which listens for incoming packets and sits on port 50000. My implementation also uses the InetSocketAddress class to represent the destination addresses of packets. An InetSocketAddress represents an IP socket address and takes in an IP address and a port number as parameters.

2.2 User Datagram Protocol (UDP) and Datagrams

The User Datagram Protocol (UDP) is a protocol that runs on top of IP and is part of the transport layer in the OSI model. It is connectionless and is mainly used for low-latency and loss-tolerating connections, due to its unreliability. No flow or error control is built-in so it is better suited for applications that can provide their own. Packets sent over UDP are referred to as datagrams. A UDP datagram consists of a header and the data section; see figure 1**.** The header is relatively small at 8 bytes and contains the source and destination port number, the length of the datagram, and a checksum field.

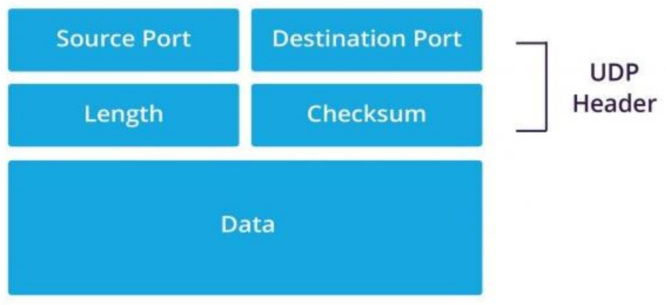


Figure 1: The basic structure of a UDP packet, or datagram.

For my protocol implementation, only the data section is relevant and my custom header and payload will be sent in this section. I have used the DatagramPacket class in the java.net library to represent these packets.

3 Implementation

3.1 Header Design

When I refer to the header in this section it is important to clarify that this is separate from the UDP header, as it is not possible to edit the UDP header in this scenario. My header is transported as part of the data section of a datagram and my protocol interprets my custom header and payload separately in its handling.

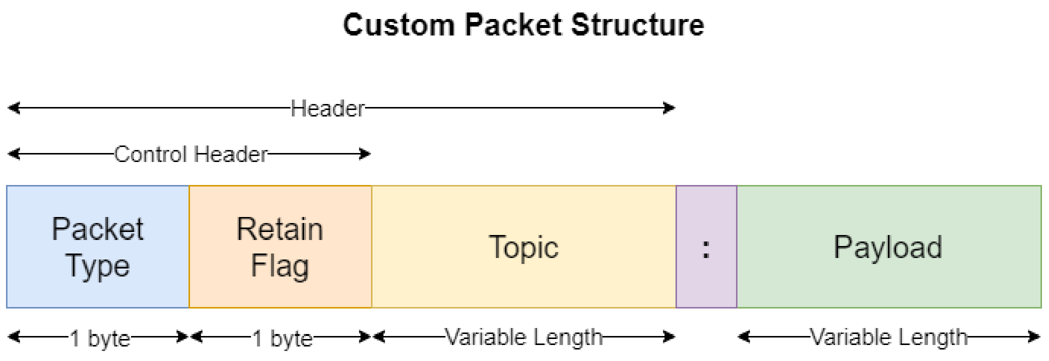


Figure 2: The custom packet structure of my protocol.

Figure 2 shows the custom packet structure for my protocol. The packet can be split into two main sections, the header and the payload. The payload is the actual data being sent without any management information. For example, if a sensor published the reading “12°C” to topic “temperature”, the payload is the sensor’s reading of 12°C.

Within the header there are two sections, the control header and the topic. The control header has a fixed length of two bytes. The first byte represents the packet type; see table 1.

Packet Type	Byte Value
PUBLISH	1
PUBACK	2
SUBSCRIBE	3
SUBACK	4
UNSUBSCRIBE	5
UNSUBACK	6

Table 1: Each of the packet types and their corresponding byte value. For example, if the first byte has a value of 1 the packet is a publish packet.

The second byte represents the retain flag. If its value is 1, the retain flag is set to true and if the second byte is 0, the retain flag is set to false. The use of this retain flag will be outlined in section 3.6.2.

The second section that makes up the header is the topic and varies in length. Assuming that the topic only includes characters in the first 128 characters of UTF-8, then each byte will map to one character of the topic. The topic is followed by a colon, ":", character. This is to signify the end of the header. Any bytes following the colon are interpreted as the payload.

The advantages and disadvantages of my header design will be discussed in section 4.1.

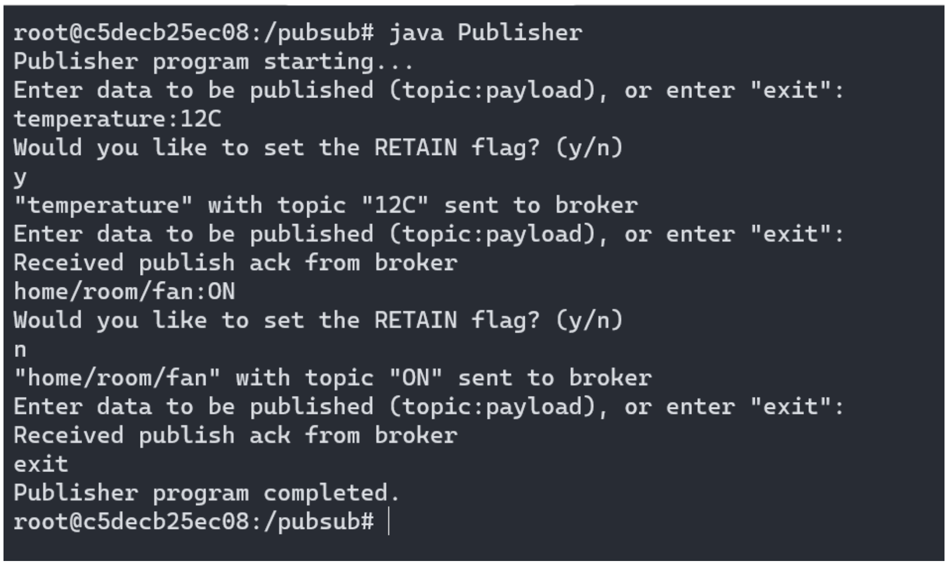
3.2 Node

My implementation consists of four classes, Node, Publisher, Subscriber, and Broker. The abstract Node class was provided on Blackboard and the Publisher, Subscriber, and Broker all extend this class. The main feature of the Node class is the Listener thread which continuously listens for incoming packets on a datagram socket. When a packet is received, it calls onReceipt. This function has a separate implementation in each of the Publisher, Subscriber, and Broker classes which determines what will happen when they receive a packet. The Node class also contains some helper functions and defines any constants used by the three subclasses. Some important constants that are defined include the different packet types and the protocol's port number. In my implementation, all components sit on port 50000 and are distinguished by their unique IP addresses. This means that any traffic in my protocol can be easily identified as anything on port 50000. Each Docker container will have a unique IP address, allowing for multiple publisher and subscriber containers to be created and run at any given time.

3.3 Publisher

The Publisher class is responsible for sending out data to the broker. A real-life example would be a temperature sensor. This sensor would continuously publish temperature readings to the broker with topics such as "temperature" or "home/bedroom/temperature". The publisher, however, does not need to be a sensor or to generate its own data. In my implementation, the user inputs the data and the topic to be published.

When Publisher.java is run, a new Publisher object is instantiated. The construction of a publisher includes setting the destination address to the broker's IP address, creating a new DatagramSocket, and telling the listener that the socket has been initialised and to start listening for incoming packets.



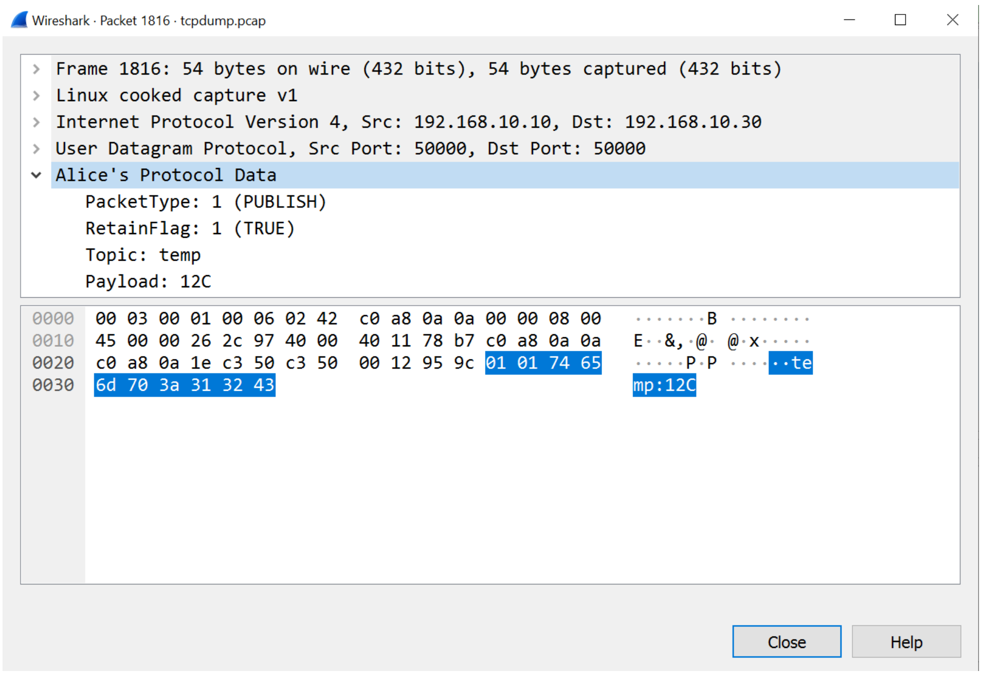
```
root@c5decb25ec08:/pubsub# java Publisher
Publisher program starting...
Enter data to be published (topic:payload), or enter "exit":
temperature:12C
Would you like to set the RETAIN flag? (y/n)
y
"temperature" with topic "12C" sent to broker
Enter data to be published (topic:payload), or enter "exit":
Received publish ack from broker
home/room/fan:ON
Would you like to set the RETAIN flag? (y/n)
n
"home/room/fan" with topic "ON" sent to broker
Enter data to be published (topic:payload), or enter "exit":
Received publish ack from broker
exit
Publisher program completed.
root@c5decb25ec08:/pubsub# |
```

Figure 3: Sample interaction between the user and the publisher.

The user is prompted to enter the data that they want to publish in the form topic:payload. One advantage of using a colon as a delimiter is that topics and data can contain whitespace. Subtopics can also be used, with each level being separated by a forward slash, "/". For example, if a sensor in room 1 of the Hamilton in Trinity College was publishing data, an example input could be "Trinity/Hamilton/room1/temperature:18 °C". Topics are case sensitive, i.e roomone is not the same as roomOne, and whitespace can also be used in topics, i.e "room one" is valid. The use of wildcards will be described in more detail in section 3.4.1 but note that wildcards cannot be used when publishing data, only when subscribing to topics. The user is next asked if they want to set the RETAIN flag. The data entered by the user is then converted to a byte array and the first byte is set to PUBLISH, to indicate the packet type, and the second byte is set to TRUE or FALSE depending on whether the user wishes to set the RETAIN flag or not. This packet is then sent to the broker. The user will again be prompted to enter data they want to send, and can send as many packets as they want. The program will end if the user types "exit".

```
protected byte[] makeDataByteArray(String message) {
        byte[] buffer = message.getBytes();
        byte[] data = new byte[CONTROL_HEADER_LENGTH + buffer.length];
        System.arraycopy(buffer, 0, data, CONTROL_HEADER_LENGTH,
buffer.length);
        return data;
    }


public synchronized void sendMessage(String message, String retainChoice)
throws Exception {
        byte[] data = makeDataByteArray(message);
        data[TYPE_POS] = PUBLISH;

        // Set the retain flag to TRUE or FALSE depending on the user's choice
        if(retainChoice.equalsIgnoreCase("y")) {
            data[RETAIN_FLAG] = TRUE;
        } else if(retainChoice.equalsIgnoreCase("n")) {
            data[RETAIN_FLAG] = FALSE;
        }

        DatagramPacket packet = new DatagramPacket(data, data.length,
dstAddress);
        socket.send(packet);
}
```

Listing 1: The functions makeDataByteArray (from Node.java) and sendMessage (from Publisher.java). This is the main code that sets the bytes and sends a publish packet to the broker.



```
> Frame 1816: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 192.168.10.10, Dst: 192.168.10.30
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Alice's Protocol Data
    PacketType: 1 (PUBLISH)
    RetainFlag: 1 (TRUE)
    Topic: temp
    Payload: 12C
```

```
0000  00 03 00 01 00 06 02 42  c0 a8 0a 0a 00 00 08 00   ·······B ········
0010  45 00 00 26 2c 97 40 00  40 11 78 b7 c0 a8 0a 0a   E··&,·@· @·x·····
0020  c0 a8 0a 1e c3 50 c3 50  00 12 95 9c 01 01 74 65   ·····P·P ······te
0030  6d 70 3a 31 32 43                                  mp:12C
```

Figure 4: A sample PUBLISH packet captured in Wireshark. Note that the first byte is 1 which indicates that this a PUBLISH packet.

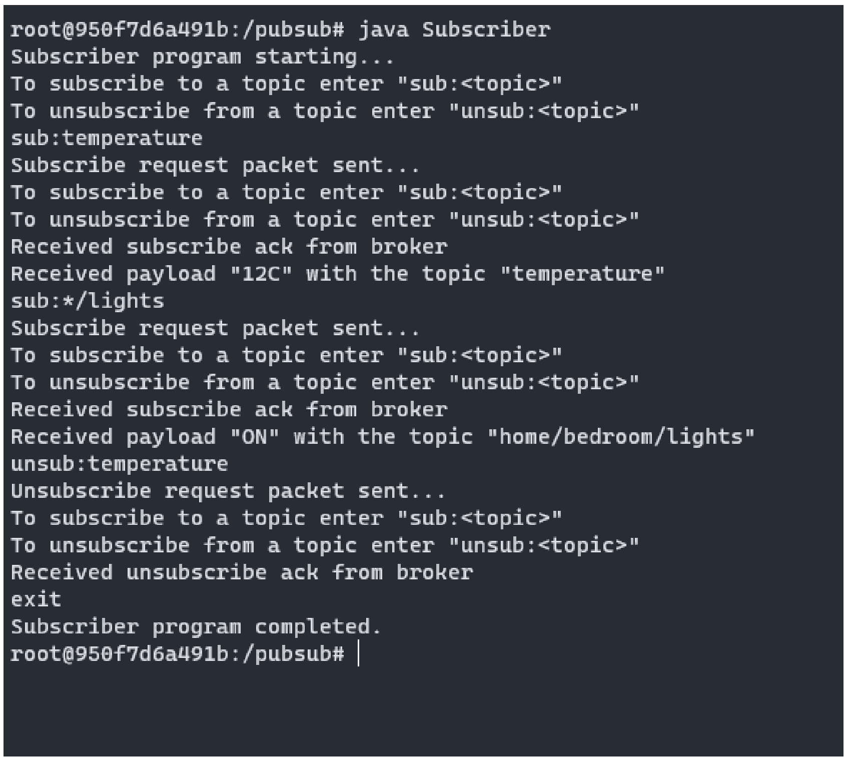
The publisher contains additional functionality to support acknowledgements and retained messages and these will be described in section 3.6.1 and 3.6.2 respectively.

3.4 Subscriber

A subscriber is a device that wishes to receive all data published to a topic that is specified by subscribing to that topic. For example, a dashboard that wishes to receive all readings about the location of a sensor could subscribe to the topic “location”. Then, any data published to that topic will get forwarded to the dashboard. An important note, the subscriber cannot request data at a specific time. It will only receive the data when a publisher publishes it and cannot request a reading from the publisher.

There is no direct communication between the publishers and subscribers, and like the publisher, the subscriber communicates directly with the broker. The subscriber has two main functions, to subscribe or unsubscribe to a topic, and to receive packets for the topics they are subscribed to. Similar to the Publisher class, when Subscriber.java is run, a new Subscriber object is instantiated. The construction of a subscriber includes setting the destination address to the broker’s IP address, creating a new DatagramSocket, and telling the listener that the socket has been initialised and to start listening for incoming packets.

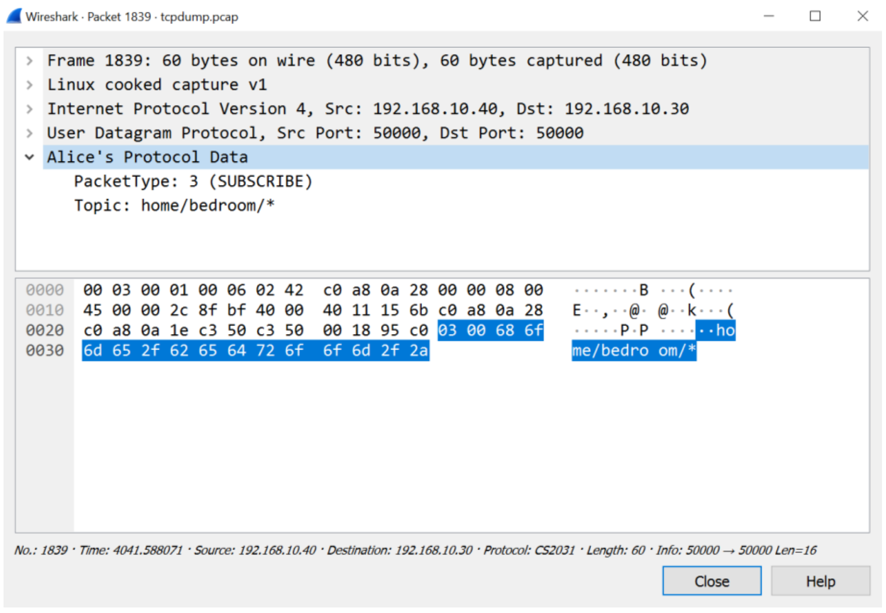
The user is prompted to either subscribe or unsubscribe to a topic by entering “sub:<topic>” or “unsub:<topic>”. By default, the subscriber is not subscribed to any topics. In my implementation, the subscriber creates the topic, i.e if the subscriber sends a request for a topic that hasn’t been subscribed to before, it is created. When the user enters a subscribe request, the topic is turned into a byte array, and the first byte is set to SUBSCRIBE. This means that the packet sent to the broker consists only of the header, i.e there is no payload, as only the topic is needed for a subscribe request. If the user makes a unsubscribe request the process is the same but instead the first byte is set to UNSUBSCRIBE. The code for this is very similar to the publisher code shown in listing 1.



```
root@950f7d6a491b:/pubsub# java Subscriber
Subscriber program starting...
To subscribe to a topic enter "sub:<topic>"
To unsubscribe from a topic enter "unsub:<topic>"
sub:temperature
Subscribe request packet sent...
To subscribe to a topic enter "sub:<topic>"
To unsubscribe from a topic enter "unsub:<topic>"
Received subscribe ack from broker
Received payload "12C" with the topic "temperature"
sub:*/lights
Subscribe request packet sent...
To subscribe to a topic enter "sub:<topic>"
To unsubscribe from a topic enter "unsub:<topic>"
Received subscribe ack from broker
Received payload "ON" with the topic "home/bedroom/lights"
unsub:temperature
Unsubscribe request packet sent...
To subscribe to a topic enter "sub:<topic>"
To unsubscribe from a topic enter "unsub:<topic>"
Received unsubscribe ack from broker
exit
Subscriber program completed.
root@950f7d6a491b:/pubsub# 
```

Figure 5: Sample interaction between the user and the subscriber.

Once subscribed to a topic, that subscriber will then receive any packets published to that topic. If the user requests to unsubscribe from a topic, from that point on they will no longer receive packets with that topic. Each subscriber can subscribe to as many topics as they want and each topic can have multiple subscribers.



```
> Frame 1839: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 192.168.10.40, Dst: 192.168.10.30
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Alice's Protocol Data
    PacketType: 3 (SUBSCRIBE)
    Topic: home/bedroom/*

0000  00 03 00 01 00 06 02 42  c0 a8 0a 28 00 00 08 00   ·······B ···(····
0010  45 00 00 2c 8f bf 40 00  40 11 15 6b c0 a8 0a 28   E··,··@· @··k···(
0020  c0 a8 0a 1e c3 50 c3 50  00 18 95 c0 03 00 68 6f   ·····P·P ······ho
0030  6d 65 2f 62 65 64 72 6f  6f 6d 2f 2a               me/bedro om/*

No.: 1839 · Time: 4041.588071 · Source: 192.168.10.40 · Destination: 192.168.10.30 · Protocol: CS2031 · Length: 60 · Info: 50000 → 50000 Len=16
```

Figure 6: A sample SUBSCRIBE packet captured in Wireshark. Note that the first byte is 3 which indicates that this a SUBSCRIBE packet and that there is no payload, only the header.

3.4.1 Subtopics and Wildcards

Subtopics and wildcards are supported in my implementation. As mentioned previously, the publisher can specify topics with multiple levels. On the subscriber side, the subscriber can either subscribe to the exact topic, or to a topic containing a wildcard, denoted with an asterisk, "*". For example, if you want all temperature readings from any room in the Hamilton you could subscribe to the topic "Hamilton/*/temperature". This means that no matter the room, the subscriber will receive the packet. The wildcard also covers multiple levels. For example, this subscriber will receive packets such as "Hamilton/roomX/temperature" and "Hamilton/floorX/roomX/temperature". The wildcard can be placed at any level in the topic. See table 2 for more examples.

Subscribed to Topic	**Will Receive**	**Will Not Receive**
home/bedroom/*	home/bedroom/fan	home/bedroom
	home/bedroom/window/blind	home/floor1/bedroom
*/temperature	home/room1/temperature	home/temperature/sensor
	garage/temperature	temperature/sensor
home/*/temperature	home/anyroom/temperature	home/room/temperature/sensor
	home/floor1/room1/temperature	houses/home/room/temperature

Table 2: Examples of topics a device can be subscribed to and the topics of the packets it will and will not receive. For example, if subscribed to "home/bedroom/*", packets with the topic "home/bedroom/fan" will be received.

The subscriber also contains additional functionality to support acknowledgements and this will be discussed in section 3.6.1.

3.5 Broker

The broker is essentially the middle man of the protocol and determines who to publish the data to. Publishers continuously send packets to the broker who looks up what devices are subscribed to the topics of those packets, and forwards them onto the relevant subscribers. The broker also receives requests to subscribe or unsubscribe to topics and updates accordingly. Although my implementation can support multiple publishers and subscribers, I have chosen only to support one broker at any given time.

When Broker.java is run, a new Broker object is instantiated. The construction of a Broker includes creating a new DatagramSocket, and telling the listener that the socket has been initialised and to start listening for incoming packets. The broker has a subscriberMap of type HashMap<String, HashSet<InetSocketAddress>>. This maps each topic to a set of addresses of all the subscribers to that topic. The choice of using a HashSet over an ArrayList was because HashSets do not allow duplicate values, therefore if a subscriber tries to subscribe to a topic they are already subscribed to the subscriber's address won't be added twice, without additional checks needing to be made.

No input is taken in from the user. The broker continuously listens for a packet and when one is received, it checks the first byte of the packet to determine the packet type using a switch statement. The packet can either have the type PUBLISH, SUBSCRIBE, or UNSUBSCRIBE, or if it fits none of these cases it prints an error message. I will next discuss what happens in each of these cases.

3.5.1 PUBLISH Packets

If the broker receives a packet of type PUBLISH it is a publish request from one of the publishers. The function sendMessage is called. This function pulls everything after the first two bytes, i.e the topic and the payload, out of the packet. It then iterates through the subscriberMap and the topic of the packet is compared to each of the topics in the subscriberMap. If there is a match, it then iterates through the HashSet of InetSocketAddresses for that topic and sends the packet to each of those addresses. In order to handle subtopics and wildcards, each of the topics in the subscriberMap are converted to the appropriate regex form by replacing each "*" with ".*?". This means, if the packet's topic is "home/temperature", regex can be used to match it with a topic like "home/.*?" in the subscriberMap, therefore sending the packet to all subscribers subscribed to the topic "home/*".

```
public synchronized void sendMessage(DatagramPacket receivedPacket) throws Exception {
        byte[] receivedData = receivedPacket.getData();
        byte[] buffer = new byte[receivedPacket.getLength()-CONTROL_HEADER_LENGTH];
        System.arraycopy(receivedData, CONTROL_HEADER_LENGTH, buffer, 0, buffer.length);
        String content = new String(buffer);

        String[] splitContent = content.split(":");
        String topic = splitContent[0];

        for (Map.Entry<String, HashSet<InetSocketAddress>> entry :
subscriberMap.entrySet()) {
            String subscriberTopic = entry.getKey();
            // Convert topic to regex to ensure accurate matching for wildcards
            String regexSubscriberTopic = subscriberTopic.replace("*", ".*?");

            if(topic.matches(regexSubscriberTopic)) {
                HashSet<InetSocketAddress> subscribers = entry.getValue();
                Iterator<InetSocketAddress> i = subscribers.iterator();
                while(i.hasNext()) {
                    InetSocketAddress addr = i.next();
                    DatagramPacket packet= new DatagramPacket(receivedPacket.getData(),
receivedPacket.getLength(), addr);
                    socket.send(packet);
                    System.out.println("Packet \"" + content + "\" sent to " + addr);
                }
            }
        }
        sendAck(PUBACK, receivedPacket);
    }
```

Listing 2: The sendMessage function from Broker.java. This is called when a publish packet is received. It looks up the topic within subscriberMap and, if it finds a match, sends the packet to all the addresses in the set mapped to that topic.

3.5.2 SUBSCRIBE Packets

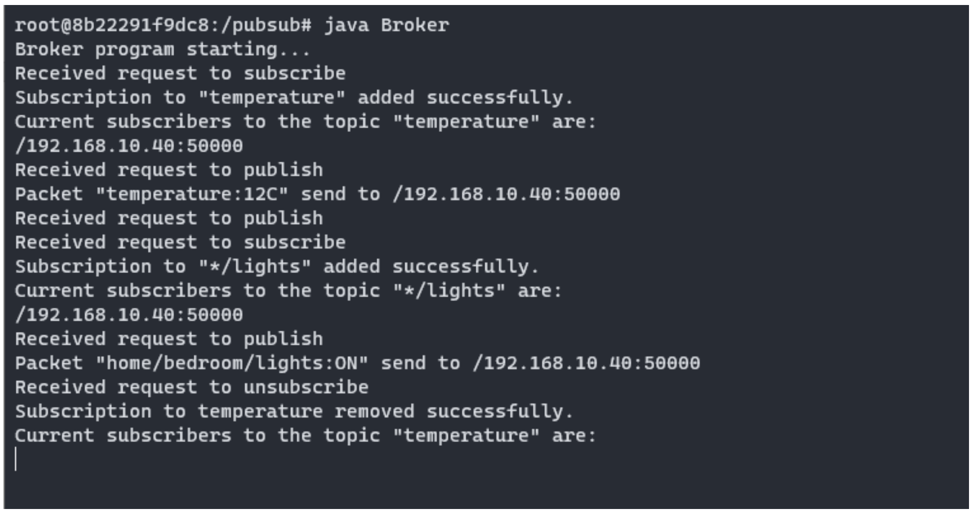
If the broker receives a packet of type SUBSCRIBE, it is a subscribe request from one of the subscribers. The function subscribe is called in this case. This function pulls out everything after the first two bytes from the packet, which in this case will just be a topic. It can then extract the address of the subscriber by using the function getSocketAddress on the received packet. If the topic is not already in the subscriberMap then a new entry is created where the topic is the key, and a new HashSet with the subscriber’s address in it is the value. If the subscriberMap already contains the topic then the subscriber’s address is simply added to the existing HashSet of addresses.

```
    private void subscribe(DatagramPacket packet) throws IOException {
        byte[] data = packet.getData();
        InetSocketAddress subscriberAddr = (InetSocketAddress) packet.getSocketAddress();
        String topic = getStringData(data, packet);

        if (!subscriberMap.containsKey(topic)) {
            // If the subscriberMap doesn't already contain the topic, create a new HashSet
for it.
            HashSet<InetSocketAddress> subscribers = new HashSet<InetSocketAddress>();
            subscriberMap.put(topic, subscribers);
        }

        if (subscriberMap.get(topic).add(subscriberAddr))
            System.out.println("Subscription to \"" + topic + "\" added successfully.");

        System.out.println("Current subscribers to the topic \"" + topic + "\" are:");
        HashSet<InetSocketAddress> subscribersCheck = subscriberMap.get(topic);
        Iterator<InetSocketAddress> i = subscribersCheck.iterator();
        while(i.hasNext()) {
            System.out.println(i.next());
        }
        sendOutRetainedMessage(topic, packet);
        sendAck(SUBACK, packet);
    }
```

Listing 3: The subscribe function from Subscriber.java. This extracts the topic from the received packet and updates subscriberMap so that each topic correcting maps to its subscribers' addresses.

3.5.3 UNSUBSCRIBE Packets

If the broker receives a packet of type UNSUBSCRIBE, it is a unsubscribe request from one of the subscribers. The function unsubscribe is called in this case. This function pulls out the topic and the subscriber's address the same was as subscribe. It then removes that address from the HashSet that corresponds to the given topic.



```
root@8b22291f9dc8:/pubsub# java Broker
Broker program starting...
Received request to subscribe
Subscription to "temperature" added successfully.
Current subscribers to the topic "temperature" are:
/192.168.10.40:50000
Received request to publish
Packet "temperature:12C" send to /192.168.10.40:50000
Received request to publish
Received request to subscribe
Subscription to "*/lights" added successfully.
Current subscribers to the topic "*/lights" are:
/192.168.10.40:50000
Received request to publish
Packet "home/bedroom/lights:ON" send to /192.168.10.40:50000
Received request to unsubscribe
Subscription to temperature removed successfully.
Current subscribers to the topic "temperature" are:
```

Figure 7: Sample output from the broker. Note that it does not take in any user input.

3.6 Additional Features

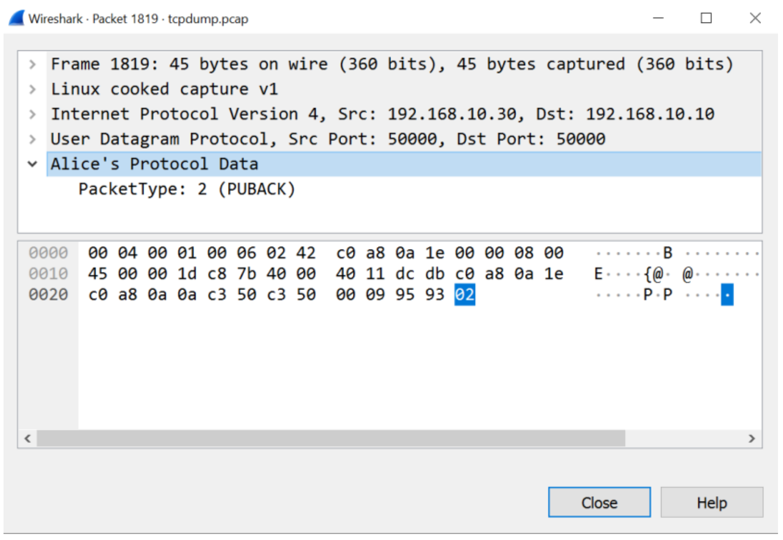
On top of the basic functionality of my protocol, I have implemented some additional features. As already outlined, my implementation can support subtopics and wildcards but there are two additional features I would like to discuss. These are acknowledgements and retained messages.

3.6.1 Acknowledgements & Quality of Service 1

Quality of Service (QoS) is a way of measuring the performance of a network service. In MQTT [2] [3] there are three QoS levels which define a guarantee for the delivery of a message. I have decided to implement something akin to MQTT's QoS level 1. This guarantees that the message is delivered at least once to the receiver. When the publisher sends a packet to the broker that packet is stored in a temporary variable and a timer is started. On the broker's side, after it has forwarded the packet onto any subscribers, it sends a PUBACK, or publish acknowledgement, back to the publisher of the original packet. If the publisher does not receive a PUBACK before the timer runs out it will resend the saved packet to the broker.

On the subscriber side, these acknowledgements have also been implemented. The subscriber should receive a SUBACK after a successful subscription request, and an UNSUBACK should be received after a successful unsubscribe request. Again if these acknowledgements aren't received in the time frame, they will be resent.

These ACK packets are lightweight and consist of only one byte, the value of which indicates which type of ACK it is; see table 1 from above.


Wireshark · Packet 1819 · tcpdump.pcap

```
> Frame 1819: 45 bytes on wire (360 bits), 45 bytes captured (360 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 192.168.10.30, Dst: 192.168.10.10
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Alice's Protocol Data
    PacketType: 2 (PUBACK)

0000  00 04 00 01 00 06 02 42  c0 a8 0a 1e 00 00 08 00   ·······B ········
0010  45 00 00 1d c8 7b 40 00  40 11 dc db c0 a8 0a 1e   E····{@· @·······
0020  c0 a8 0a 0a c3 50 c3 50  00 09 95 93 02            ·····P·P ·····
```

Figure 8: Sample PUBACK packet captured in Wireshark. Note that it consists of 1 byte which determines its type.

3.6.2 Retained Messages

As previously mentioned, the second byte in the header represents the RETAIN flag. Note, the retain flag is only used in PUBLISH packets. Normally, if a message is published to a topic and no one is subscribed to it the message will get discarded. However, if the RETAIN flag is set to true the broker will save the last message published to that topic and forward it onto a subscriber if it subscribes to that topic at a later time. For example, if you have a sensor that publishes whether a door is open or closed to the topic “door”, if a device subscribes to “door” it would have to wait until the status changes to get a message stating whether the door is open or closed. However, in this case it would be useful to get the current status, i.e the last message sent to the topic, when the device subscribes to the topic and this is possible with retained messages.

In terms of the implementation, the broker has a retainedMessageMap of type HashMap<String, byte[]>. When the broker receives a PUBLISH packet, it will check the second byte to see if the RETAIN flag is set. If it is set to TRUE it calls retainMessage. This function will extract out the topic and the data byte array from the received packet and insert it into retainedMessageMap, where the topic is the key and the byte array is the value. This means that at any given time there will only be one message retained per topic. This retained message will be the last message published to that topic with the RETAIN flag set. When a subscription request is made, the function sendOutRetainedMessage is called. The broker looks up the topic that the subscriber is looking to subscribe to in the retainedMessageMap and if there is a retained message for that topic is it forwarded to the new subscriber.

3.7 Docker

Docker [4] is used to run my solution. Each publisher, subscriber, and broker instance is run inside its own Docker container. This means that each container has a unique IP address and multiple publishers and subscribers can be run at once. To create my network and containers I used Docker Compose and the YAML file which defines my containers can be found in the code submitted to Blackboard. This creates two publishers, two subscribers, and one broker, as well as a container called tcpdump [5] which is used to capture the network traffic in a PCAP file.

4 Discussion

4.1 Header Design

A major part of this assignment, relates to the design of the header and the balancing act between adding more management information to increase efficiency and increasing the overhead. I chose to keep my header relatively small, and looked at the MQTT header design [6] to see which features I would like include in my protocol. The first byte representing the packet type is key to the protocol and I believe the benefits of being able to determine the packet type from 1 byte has very few disadvantages. Of course, by limiting the packet type to 1 byte you are limiting the number of packet types you can have to 258, however, I cannot imagine any publish-subscribe protocol needing that many packet types.

The second byte of my protocol represents the RETAIN flag. The inclusion of this adds, in my opinion, a useful feature for many real-world application scenarios. However, one down side is that the flag is only needed for PUBLISH packets being sent from the publisher to the broker. If I had more time, I would consider removing this byte from SUBSCRIBE packets and maybe also from PUBLISH packets that are being sent from the broker to the subscriber as it is not necessary.

Another design decision I made, was to hardcode the QoS to 1 and not give the user the option to select different values. There are two main reasons for this. First, I did not have time, and did not think it was necessary to add support for QoS 2 in this assignment. Secondly, as the majority of lightweight protocols like the one I have designed usually stick to QoS 1, I thought it made sense to make that the default.

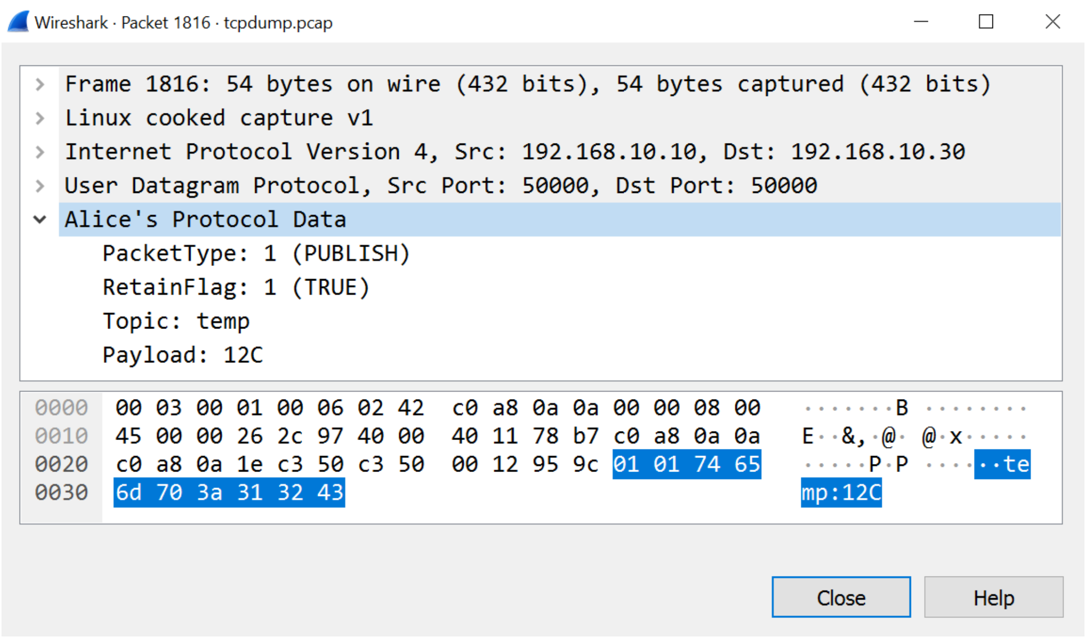
Looking at the MQTT control header you can also see it is 1 byte, whereas mine is 2 bytes. The first four bits encode the packet type, and the second two bits encode extra flags such as the QoS and the retain options. I considered designing mine like this to reduce my control header from 2 bytes to 1 byte whilst retaining the same functionality. However, I decided against this as having each field in its own byte made it very easy to access using array notation, and setting and getting specific bits would require more code and masks, which I did not think was worth the hassle.

One final design decision regarding the header, was the decision to convert the topic from a String to a byte array and send that as it is, rather than mapping each topic to a value and having a fixed length for the topic field. The main reason for this were the benefits when it came to creating and searching for topics. Especially since I did not create fixed topics for my protocol and allowed the user to subscribe to an endless amount. As well, taking subtopics into account, the number of possible topics a user may want also increased. If I were to have designed the protocol to only support specific topics, for example if it was designed specifically to handle weather data and the only topics needed were temperature, humidity, and location, then it would make sense to map each of these topics to a value and store them in a single byte in the header, but that is not the functionality I was looking to support.

Also note, that while many protocols store the destination and source addresses within their header I did not do this. The simple reason being that the source address is easily accessed from the DatagramPacket using getSocketAddress, and the destination address is simply the broker or is stored in the subscriberMap, as outlined previously.

4.2 Lua Wireshark Extension

To help tell Wireshark how to interpret my protocol, and to visualise it more clearly, I wrote a Lua file defining the elements of my protocol. This file can be found as part of my code submission on Blackboard.

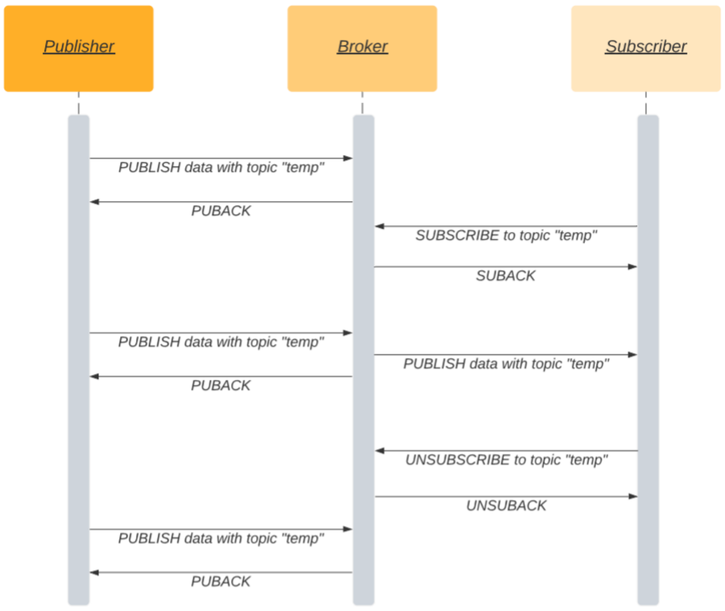


```
> Frame 1816: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
> Linux cooked capture v1
> Internet Protocol Version 4, Src: 192.168.10.10, Dst: 192.168.10.30
> User Datagram Protocol, Src Port: 50000, Dst Port: 50000
v Alice's Protocol Data
    PacketType: 1 (PUBLISH)
    RetainFlag: 1 (TRUE)
    Topic: temp
    Payload: 12C

0000  00 03 00 01 00 06 02 42  c0 a8 0a 0a 00 00 08 00   ·······B ········
0010  45 00 00 26 2c 97 40 00  40 11 78 b7 c0 a8 0a 0a   E··&,·@· @·x·····
0020  c0 a8 0a 1e c3 50 c3 50  00 12 95 9c 01 01 74 65   ·····P·P ······te
0030  6d 70 3a 31 32 43                                  mp:12C
```

Figure 9: This screenshot shows my custom protocol in Wireshark. Each section of my protocol design can be seen: PacketType, RetainFlag, Topic, and Payload.

5 Summary

In summary, my protocol features three main components, the broker, publishers, and subscribers. The publishers and subscribers can send packets to the broker who then determines what to do depending on the type of packet. Figure 10 and figure 11 summarise the communication and flow between each of these components. Note that in these figures only one publisher and subscriber are used for simplicity, but it is possible to have multiples of both.



Publisher
Broker
Subscriber
PUBLISH data with topic "temp"
PUBACK
SUBSCRIBE to topic "temp"
SUBACK
PUBLISH data with topic "temp"
PUBACK
PUBLISH data with topic "temp"
UNSUBSCRIBE to topic "temp"
UNSUBACK
PUBLISH data with topic "temp"
PUBACK

Figure 10: A diagram showing the basic flow of network traffic between each of the components in my solution.

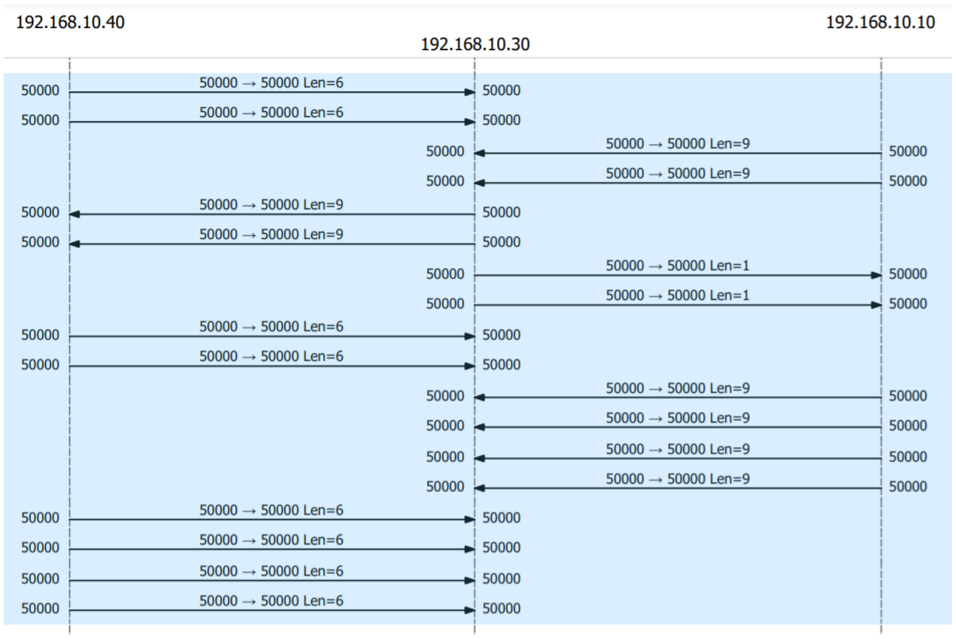


Figure 11: A capture of the flow of traffic between each of the components in my solution using Wireshark's Flow Graph feature. Note that 192.168.10.10 is the publisher, 192.168.10.30 is the broker, and 192.168.10.40 is the subscriber and all traffic sits on port 50000.

Six different packet types were mentioned within this report and a summary of the contents of these packets is below in figure 12.

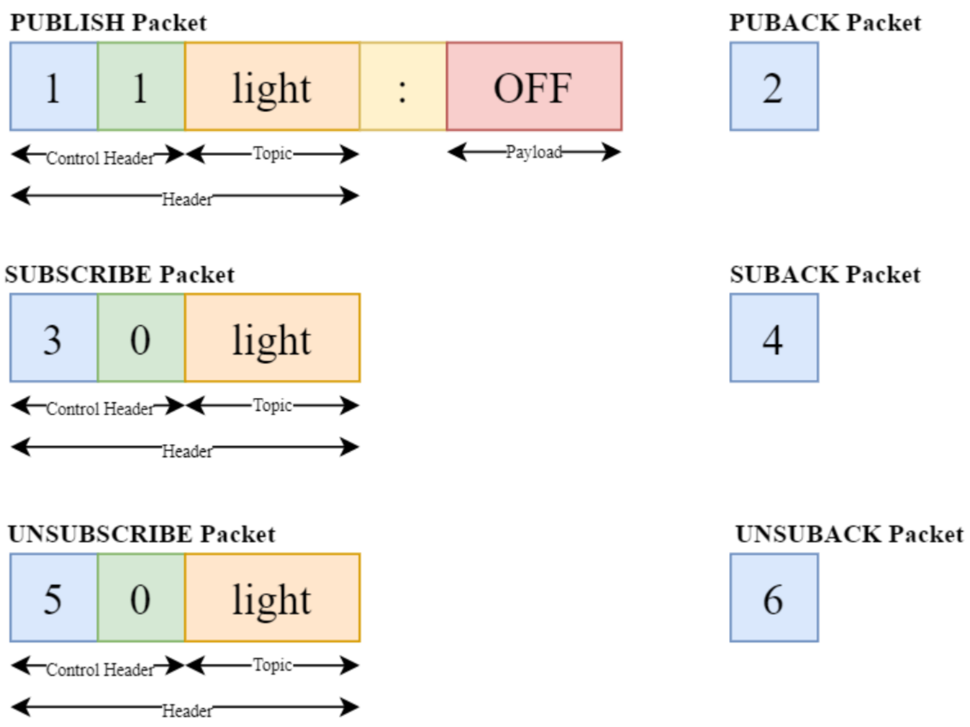


Figure 12: A summary of the contents of each of the different packet types.

Docker is used to run my solution, with the broker, subscribers and publishers each having their own container within my Docker network. On top of the basic publish and subscribe functionality, my protocol also supports additional features such as wildcards, QoS 1 guarantee, and retained messages.

6 Reflection

Overall, throughout the course of this assignment I have learned a huge amount of networking theory. Additionally, I also have acquired other practical skills such as how to use Docker, and a greater proficiency in Unix. If I were given more time to work on my solution there are a few small things I would spend more time on. The first is that, although it uses the same logic as the publisher side, the subscriber timer mechanism and the resending of a packet in the event that no SUBACK or UNSUBACK is received does not fully work, however, the logic still stands. Another thing I would consider if I were given more time is support for multiple brokers but once again this was not a feature I considered essential so focused my time elsewhere. A huge amount of time and effort was put into this assignment and I was constantly faced with issues that seemed trivial taking hours to sort out, for example getting Wireshark working in Docker. Given the time constraints, I am very pleased with the working solution I have come up with and the additional features I have implemented.

7 References

[1] Oracle. Package java.net. https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html, visited Oct 2021.

[2] MQTT. MQTT project page. https://mqtt.org/, visited Oct 2021.

[3] HiveMQ. MQTT Essentials. https://www.hivemq.com/tags/mqtt-essentials/, visited Oct 2021.

[4] Docker. Docker project page. https://www.docker.com, visited Oct 2021.

[5] Docker Hub. kaazing/tcpdump Docker Image. https://hub.docker.com/r/kaazing/tcpdump, visited Oct 2021.

[6] Open Lab Pro. MQTT Packet Format. https://openlabpro.com/guide/mqtt-packet-format/, visited Oct 2021.