CS2031 Telecommunications II
Assignment 1

Name: *Owen Burke*
ID: *15316452*

This report describes the design and implementation of the protocol summarised in the assignment specification provided. This implementation allows a client node to send a packet to a gateway, which is then forwarded to the server. The server then sends the gateway an acknowledgement which is also forwarded to the client. The classification of this acknowledgement is decided upon by the server, depending on whether the sequence number of the packet was what it expected. Also, if the client does not receive one of these acknowledgements within a specified time period, it resends the packet to the server. The server has also been designed to handle incoming data from multiple clients (described later).

Simplistic Implementation:

The basic implementation of this protocol began with the creation of the gateway. The gateway is essentially like an altered client, with an onReceipt method and a way of sending the packets it receives to some other node. Also the way in which this gateway has been implemented means that it is specific to a certain server, as seen by the ***DEFAULT_SERVER_DST_PORT*** field. So while the gateway can work on behalf of one or more clients, it can only co-ordinate with a single server.

Each client is also tied to the gateway with ***DEFAULT_GATEWAY_DST_PORT***. So the clients don't know of the existence of the server, only the gateway and it is up to the gateway to be the intermediary.

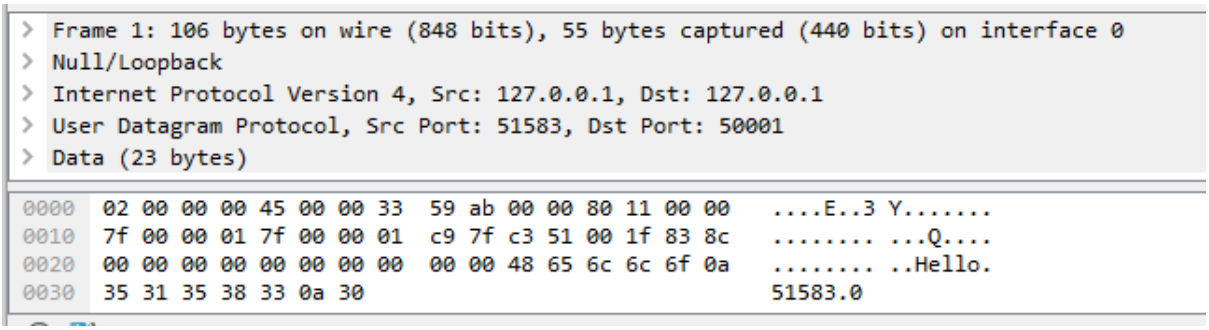
To implement this gateway, the client and server classes also had to be modified to include some necessary data in the packet, such as its source port (in order for the gateway to know where to send it on its return). This can be seen in the start method of the client class where the source port is added into the payload followed by the sequence number (this is simply a field which is incremented by one upon the reception of a positive acknowledgement).

```
byte[] payload= null;
byte[] header= null;
byte[] buffer= null;
```

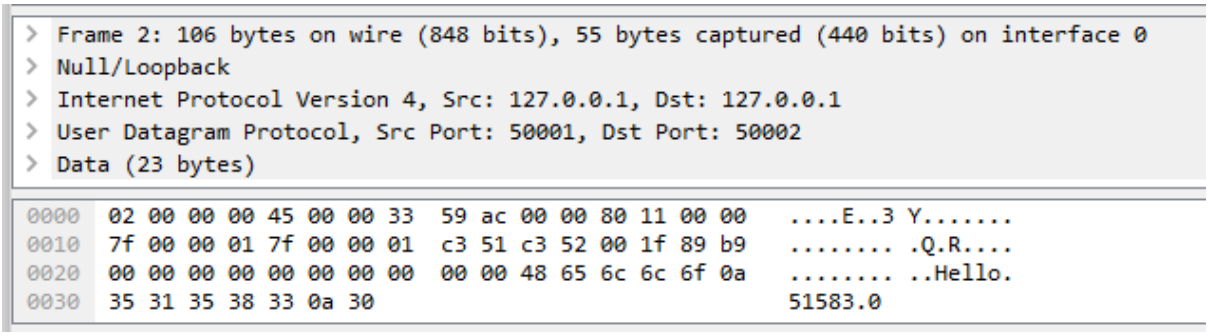
```
payload= (terminal.readString("String to send: ") + "\n" + DEFAULT_SRC_PORT
+ "\n" + sequenceNumber).getBytes();

header= new byte[PacketContent.HEADERLENGTH];

buffer= new byte[header.length + payload.length];
System.arraycopy(header, 0, buffer, 0, header.length);
System.arraycopy(payload, 0, buffer, header.length, payload.length);
```

The packet created from this data is then sent to the gateway in the handleSend method.
On the reception of a packet, the gateway decides whether it has come from the server or from a client by `if(packet.getSocketAddress().equals(dstServerAddress))`.
If the packet comes from the client, the socket address is simply set to the address of the server with the setSocketAddress method. If it comes from the server, the payload is split into a string array using the new line statement as the separator so that the information can be taken out, such as the client port. This is needed to set the correct socket address. A new packet is then created which contains the same message as the original, just without the port number. This packet is then sent onto the client.
When the server receives a packet, the payload is split into an array in a similar way as the gateway. The message, the client port number and the sequence number are then printed to the terminal and a new packet is created with an acknowledgement message in the payload as well as the client port number and the sequence number it wants next. This is then sent back to the gateway to be forwarded to the client.
When the client receives this packet, it splits the payload into an array, sets the sequence number to the one just specified by the server and prints the acknowledgement message along with the new sequence number.
The resending of packets is implemented in the client class. I achieved this by writing a new method which deals, solely, with the sending of a packet called handleSend. In this method, the packet is sent and then the program waits for a given amount of time, in this case 2 seconds. The boolean hasReceived is set to true whenever a packet is received in the onReceipt method, so if it has not received a packet after those 2 seconds a message is printed saying time has run out and it is resending the packet. A recursive call to the handleSend method is then made. This recursion is what enables packets to continuously be resent after the timeout, until it has received an acknowledgement.

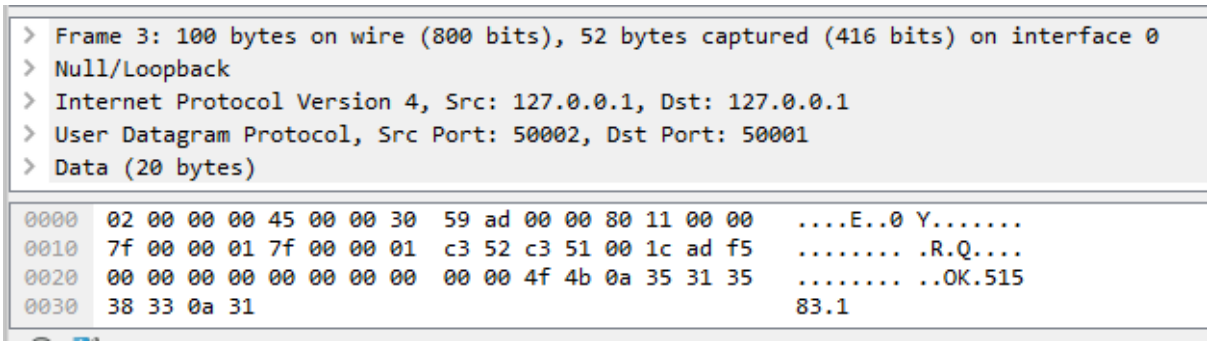
Simplistic Implementation Packet Description:



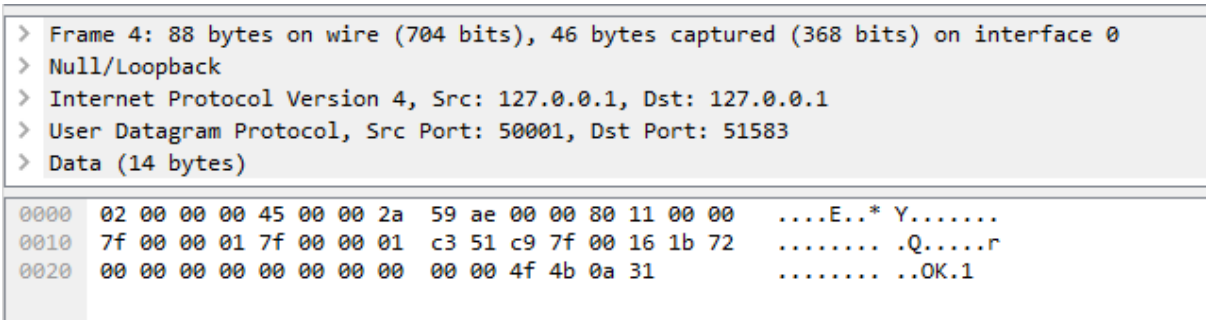
This is the packet that is sent from the client (51583) to the gateway (50001). The payload can be seen as the string “Hello” followed by the client source port, separated by a new line and then the sequence number “0”, also separated by a new line.



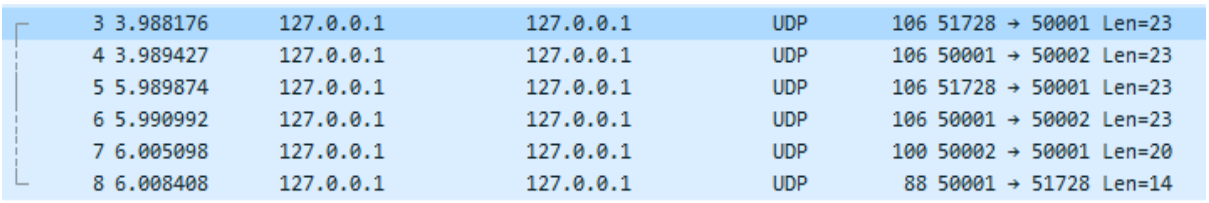
The same packet is then sent to the server (50002) from the gateway (50001).



This packet is then sent from the server onto the gateway upon the receipt of the above packet. The payload content is similar to those above, except for the message content which is now “OK” and the sequence number has been changed to the one the server expects next.



This new packet is then created and then sent onto the client. The payload of this packet is the same as the one above apart from the client port number, which has been removed and was used to set the socket address.



This is the order of packets that are sent when the timeout functionality is implemented. This shows that the client sends the packet to the gateway and the gateway sends it to the server, but the server doesn't respond. The client then resends the same packet to the gateway and the process is executed as usual.

Extended Implementation:

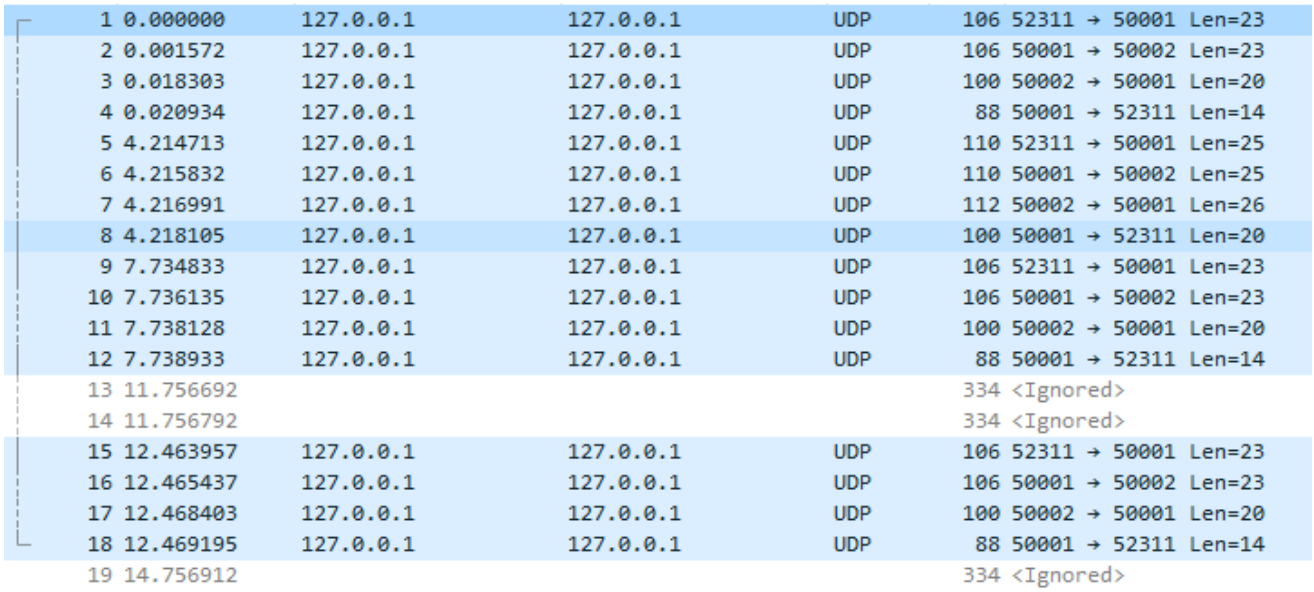
The extension to the simplistic implementation allows for the handling of multiple clients and the sending of negative acknowledgements from the server if it receives a packet with a sequence number it did not expect.
Both of these tasks are done using a hashmap, with the key being the client's port number and the value being the packet that came from that client. In onReceipt in the server class, the incoming packet is added into the hashmap with the client port number as its key if the sequence number is 0 for the first packet to come from that client (i.e. if the value for that port number is null) or if the sequence number is one greater than the number for the packet that is already in the hashmap for that client (i.e. the previous packet). If these criteria are met, the new packet is added into the hashmap for that client, replacing the old value for that key and a method is called to send a positive acknowledgement. Otherwise, the packet is not added and a method to send a negative acknowledgement is called.

In the sendPosAck method, a new packet is created with the message “OK”, followed by the client port number and the sequence number that the server expects next (the sequence number of the incoming packet plus one). This packet is then sent to the client.
The sendNegAck method is similar, only the number sent with the packet is the number that the incoming packet should have been (i.e. 0 for the first packet from a client or the number of the previous packet plus one).
In the client class, the onReceipt method checks if the acknowledgement is negative or positive. If it is positive the sequence number field is updated to the number expected by the server on the next send. If it is negative the sequence number is updated to what it should have been last time and asks for a new message to send. I chose to do this rather than cause a failure error as I thought that it would be best to avoid such things and not cause the program to crash. I also ask for a new message as there may have been an error in the message just as there was an error with the sequence number.

Extended Implementation Packet Description:

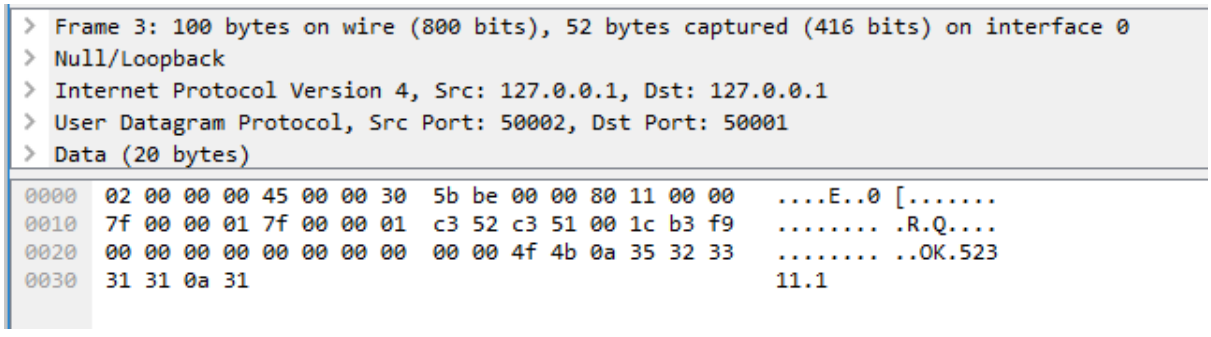
1	0.000000				660	<Ignored>
2	0.000055				660	<Ignored>
3	0.774472				68	<Ignored>
4	4.773176	127.0.0.1	127.0.0.1	UDP	122	52232 → 50001 Len=31
5	4.774726	127.0.0.1	127.0.0.1	UDP	122	50001 → 50002 Len=31
6	4.790086	127.0.0.1	127.0.0.1	UDP	100	50002 → 50001 Len=20
7	4.792968	127.0.0.1	127.0.0.1	UDP	88	50001 → 52232 Len=14
8	7.774457				68	<Ignored>
9	7.774505				68	<Ignored>
10	11.868881	127.0.0.1	127.0.0.1	UDP	122	52231 → 50001 Len=31
11	11.870085	127.0.0.1	127.0.0.1	UDP	122	50001 → 50002 Len=31
12	11.872497	127.0.0.1	127.0.0.1	UDP	100	50002 → 50001 Len=20
13	11.873393	127.0.0.1	127.0.0.1	UDP	88	50001 → 52231 Len=14
14	17.237084	127.0.0.1	127.0.0.1	UDP	122	52230 → 50001 Len=31
15	17.238401	127.0.0.1	127.0.0.1	UDP	122	50001 → 50002 Len=31
16	17.240444	127.0.0.1	127.0.0.1	UDP	100	50002 → 50001 Len=20
17	17.241228	127.0.0.1	127.0.0.1	UDP	88	50001 → 52230 Len=14
18	24.204050	127.0.0.1	127.0.0.1	UDP	134	52230 → 50001 Len=37
19	24.205380	127.0.0.1	127.0.0.1	UDP	134	50001 → 50002 Len=37
20	24.207719	127.0.0.1	127.0.0.1	UDP	100	50002 → 50001 Len=20
21	24.208625	127.0.0.1	127.0.0.1	UDP	88	50001 → 52230 Len=14
22	31.296528				334	<Ignored>
23	31.296579				334	<Ignored>
24	33.596142	127.0.0.1	127.0.0.1	UDP	134	52232 → 50001 Len=37
25	33.597442	127.0.0.1	127.0.0.1	UDP	134	50001 → 50002 Len=37
26	33.599897	127.0.0.1	127.0.0.1	UDP	100	50002 → 50001 Len=20
27	33.600710	127.0.0.1	127.0.0.1	UDP	88	50001 → 52232 Len=14
28	34.296134				334	<Ignored>
29	34.296172				334	<Ignored>
30	37.296177				334	<Ignored>
31	37.296221				334	<Ignored>
32	40.297152				334	<Ignored>
33	40.297189				334	<Ignored>
34	41.428691	127.0.0.1	127.0.0.1	UDP	134	52231 → 50001 Len=37
35	41.430079	127.0.0.1	127.0.0.1	UDP	134	50001 → 50002 Len=37
36	41.433016	127.0.0.1	127.0.0.1	UDP	100	50002 → 50001 Len=20
37	41.434248	127.0.0.1	127.0.0.1	UDP	88	50001 → 52231 Len=14
38	43.298262				334	<Ignored>

This image shows the implementation of multiple clients. The packets that are sent from clients are marked in black and it can be seen that some come from different clients (there are 3 clients in this case). The gateway responses and sending of acknowledgements immediately follow each client send. This image shows the 3 clients sending two packets each, although the packets for each client are interspaced between those of other clients. The responses from the server and gateway are similar to those of the simplistic implementation, seen above (all positive acknowledgements, etc.).



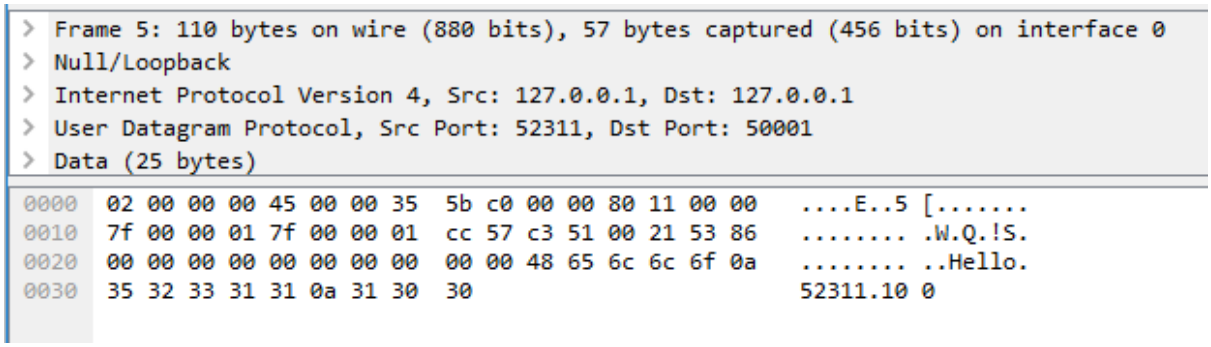
1	0.000000	127.0.0.1	127.0.0.1	UDP	106 52311 → 50001 Len=23
2	0.001572	127.0.0.1	127.0.0.1	UDP	106 50001 → 50002 Len=23
3	0.018303	127.0.0.1	127.0.0.1	UDP	100 50002 → 50001 Len=20
4	0.020934	127.0.0.1	127.0.0.1	UDP	88 50001 → 52311 Len=14
5	4.214713	127.0.0.1	127.0.0.1	UDP	110 52311 → 50001 Len=25
6	4.215832	127.0.0.1	127.0.0.1	UDP	110 50001 → 50002 Len=25
7	4.216991	127.0.0.1	127.0.0.1	UDP	112 50002 → 50001 Len=26
8	4.218105	127.0.0.1	127.0.0.1	UDP	100 50001 → 52311 Len=20
9	7.734833	127.0.0.1	127.0.0.1	UDP	106 52311 → 50001 Len=23
10	7.736135	127.0.0.1	127.0.0.1	UDP	106 50001 → 50002 Len=23
11	7.738128	127.0.0.1	127.0.0.1	UDP	100 50002 → 50001 Len=20
12	7.738933	127.0.0.1	127.0.0.1	UDP	88 50001 → 52311 Len=14
13	11.756692				334 <Ignored>
14	11.756792				334 <Ignored>
15	12.463957	127.0.0.1	127.0.0.1	UDP	106 52311 → 50001 Len=23
16	12.465437	127.0.0.1	127.0.0.1	UDP	106 50001 → 50002 Len=23
17	12.468403	127.0.0.1	127.0.0.1	UDP	100 50002 → 50001 Len=20
18	12.469195	127.0.0.1	127.0.0.1	UDP	88 50001 → 52311 Len=14
19	14.756912				334 <Ignored>

This image relates to the sending of negative acknowledgements from the server. The test that has been implemented triggers a negative acknowledgment on the second packet sent from the client (in client, the sequence number is set to 100).

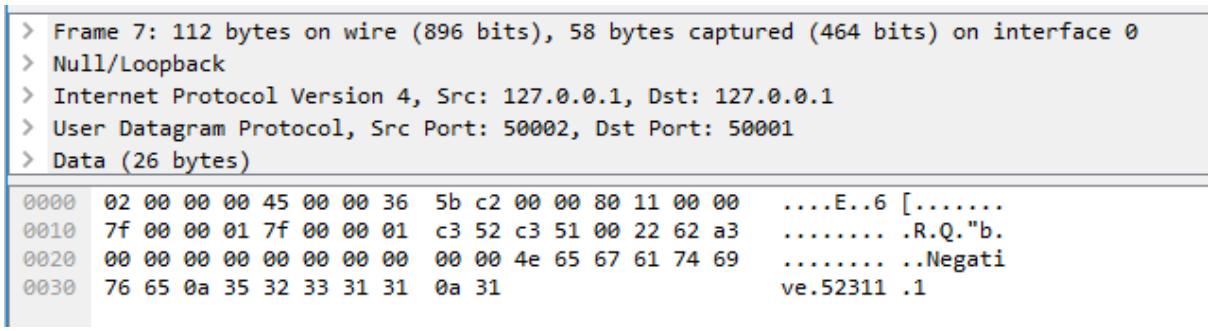


```
> Frame 3: 100 bytes on wire (800 bits), 52 bytes captured (416 bits) on interface 0
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 50002, Dst Port: 50001
> Data (20 bytes)

0000  02 00 00 00 45 00 00 30  5b be 00 00 80 11 00 00   ....E..0 [.......
0010  7f 00 00 01 7f 00 00 01  c3 52 c3 51 00 1c b3 f9   ........ .R.Q....
0020  00 00 00 00 00 00 00 00  00 00 4f 4b 0a 35 32 33   ........ ..OK.523
0030  31 31 0a 31                                       11.1
```

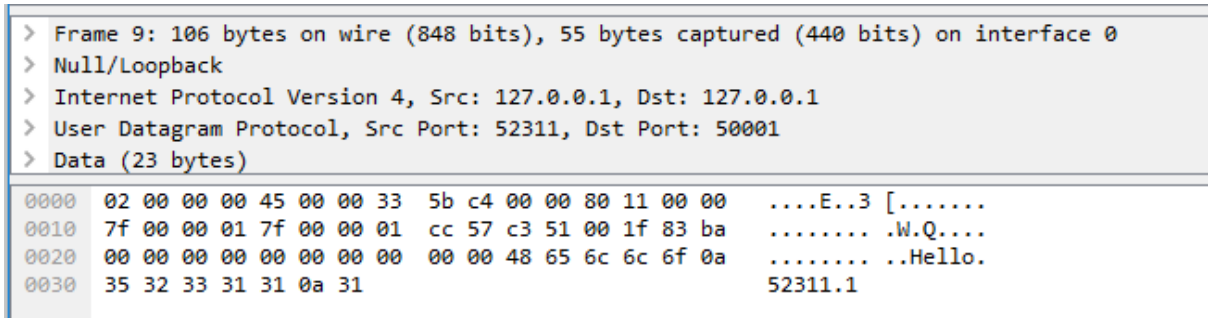
This packet is clearly a positive acknowledgment and contains the sequence number that the next packet it receives should be.



This packet sent from the client will trigger a negative acknowledgment as the sequence number is 100, when it should be 1.



This is the negative acknowledgment sent from the server. The packet contains the sequence number the incoming packet should have been and what the next packet will be, 1.



Here the sequence number has been updated to the number provided to us by the server in the negative acknowledgement.

Reflection:

-Advantages:

The advantages introduced by the design of the protocol include:

- If the sequence number is incorrect, the program will recover (asks for the message again) rather than ending the whole operation.

- The use of a hashmap allows for the handling of a high number of clients rather than limiting them to a specific number i.e. 10.

-Disadvantages

The disadvantages introduced by the design of the protocol include:

- The placing of the sequence number and client port in the payload may cause issues in some cases that I am not aware of (might have been better to put them in the header or something).
- The way that I have implemented the timeout/resending of packets may not be as efficient as using the setSoTimeout function.

Overall I found the assignment relatively manageable. I found the task of actually writing the classes and solving the programming problems fairly straight forward. Although, I am still unsure as to how one or two things work under the hood, such as threads. But, on the whole, I found it very doable. In terms of hours spent, I would guess in the mid 20 hours range, although I am not sure. That's just a rough guess.