CS2031 ASSIGNMENT 1 - GATEWAY

OVERVIEW

The current implementation of this program consists of three main classes; the Client, Server and Gateway. Each of these classes are sub-classes of Node.java superclass, and all function similarly. In their current functionality, the client (and each of its instances) solely communicates with the gateway which acts between the server and clients, as per the project specifications. A protocol for the information stored within the packets was created to handle the communication.

1.PACKET ENCAPSULATION AND PROTOCOL

1.1 'Encapsulation'

As part of this implementation a protocol was created to simplify the communication between the server, gateway and multiple clients. Generally, throughout these programs the buffer of the packets will have the same layout in this order:

1. Payload
2. Port number
3. Sequence number

Each of these elements of the buffer will be separated by a string added to the Node.java class known as the 'SPLITTER', which is just 3 unseparated semi-colons ';;;'. As a result, most packets sent between the client and server will be constructed and sent as such:

```
DatagramPacket response;
response = (new StringContent(responseMessage + SPLITTER + clientPort + SPLITTER +
sequenceNumber)).toDatagramPacket();
packet.setSocketAddress(packet.getSocketAddress());
socket.send(response);
```

Each variable has a previously set value.

This is a very simple method of ensuring that the gateway knows which message was sent by which client, and to which client to send a response from the server.

1.2 Separating elements of the buffer

Since the payload, the port, and the sequence number are all sent as part of a string contained in the buffer, there needs to be a way of separating these so the program can obtain the relevant data. To do this, three methods were created and added to the Node.java superclass, they are;

1. extractMessage(DatagramPacket packet),
2. extractPort(DatagramPacket packet)
3. extractSequenceNumber(DatagramPacket packet)

Here is a snippet of code for the extractMessage() function:

```
/*
 *            A method which can extract the payload (or message) which is
 *            encapsulated inside a packet assuming the correct layout is
 *            met. This is done by splitting the buffer of an incoming
 *            packet into an array of strings and returning the relevant
 *            data.
 *
 *            @param DatagramPacket packet -  the packet from which you want
 *            to extract the encapsulated message
 *
 *            @return - the encapsulated message string
 */
      public static String extractMessage(DatagramPacket packet) {
            StringContent content = new StringContent(packet);
            String packetContent = content.toString();
            String[] packetContentArray = packetContent.split(SPLITTER);
            return packetContentArray[0];
      }
```

Each function will split the incoming buffer into an array of strings, this is based on the previously mentioned SPLITTER string. The relevant data is returned, which is determined by its index in the array. All the functions act identically, the former will return a string, while the latter two will parse the string to an integer then return that integer. The port number usually refers to the port of the client that sent the packet.

A weakness of design is that the protocol must be strictly followed i.e. each string being sent must be in the form of STRING + PORT + SEQUENCE. As a result, throughout the programs you will often see 0 to fill these gaps if the port isn't totally relevant to that specific data transmission.

2.CLIENT

2.1 Determining Port numbers

Upon launching the Client program, the user will be presented with a terminal which reads at the top the current port number of this client (see **Figure 1** below). Each port number is randomly generated using the rand.nextInt() function from the java.util.Random library. A constant called 'MAX_CLIENT_PORT' is used to ensure that a client could not have the same port as the gateway or server. Using a random number generator to create the port was found to be the most feasible way of ensuring that no two instances of the client would have the same port number. In the case that they did have the same port, an exception would be caught and resolved by just rolling for another random number.

Earlier iterations of this program attempted to use a static variable 'offset' which would be added to the port that increased by 1 each time the program was launched, however this proved to not work as intended and a choice was made to use random numbers.

This system ignores any sort of protocols for address allocation in real world applications, however is sufficient for the purposes of this implementation.

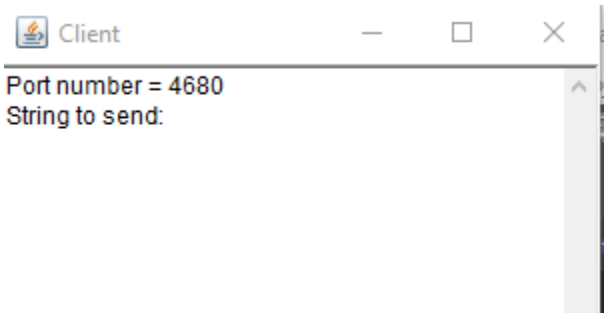

Figure 1: What you should see upon launching the client

2.2 Handling Sequence Numbers

Each instance of the client has its own sequence number, which is initially set to 0. Upon receiving the correct acknowledgement from the server this sequence number will iterate by 1. If the incorrect acknowledgement is received from the server then the sequence number will decrement by 1 and resend the previous packet. This will continue to happen until the correct acknowledgment is received.

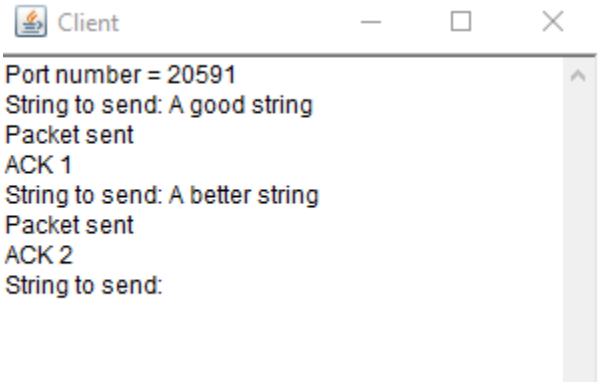

Figure 2: An example of the back and forth communication from the client.

2.3 Timer Mechanism

This implementation also supports a simple timeout mechanism using the setSoTimeout() method. Upon sending a packet the timer will begin to count down from 5 until it receives a response from the server. If the timer manages to reach 0, the *SocketTimeoutException* exception will be thrown. This is all inside a try-catch block which will then resend the packet. If the client receives a response, the setSoTimeout() method will be called and the timer set to 0 (which is effectively infinity).

N.B To have multiple clients active at once just run the client program multiple times.

3.GATEWAY

3.1 DESIGN

The Gateway program serves as the middle program between the clients and the server. As such, it contains much of the same functionality of both. The gateway has its own port at port number 40789 (known as the constant *GATEWAY_PORT*), chosen just because it was the same port as in the assignment outline. Upon receiving a packet, the gateway will print to the terminal that there is an incoming packet from one of the clients (referred to by port number), or from the server (Server). The gateway will then confirm that the packet has been sent on to the intended recipient (see **Figure 3)**. The choice was made to not print the message to the screen of the terminal.

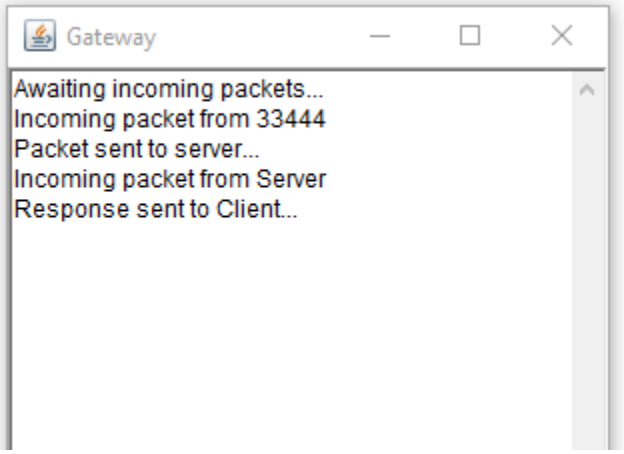

Figure 3: The Gateway in action

3.2 Client or Server?

Upon receiving a packet, the Gateway must be able to determine if that packet is coming from one of the clients, or the server. This is done by simply comparing the port number of the incoming packet to the *SERVER_PORT* integer (50001). The gateway will then reconstruct the packet with relevant variables such as the sequence number and the port from which the packet was received (in the case it came from one of the clients), before sending it on to its intended recipient.

4.SERVER

4.1 Design and Operation

Upon receiving a packet, the Server will split the buffer up into its elements using the aforementioned extraction methods. The server will then print this message to the screen with the port number of the sender, before sending a response to the client. Assuming the correct sequence number has been received the server will construct and send its reply consisting of the correct ACK and the client to which the response is being sent. In this case the response will always have a sequence number of 0 as the message string already includes an ACK with the next expected sequence.

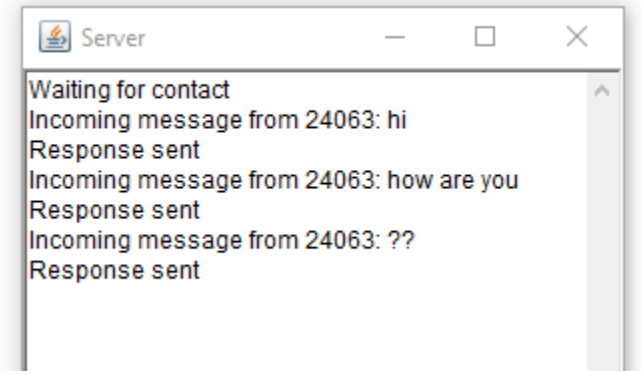

Figure 4: Example of incoming packets

4.2 Multiple Clients, Multiple Sequence Numbers

Since this implementation can support multiple active clients, all of which could be assumed to have different sequence numbers, a system had to be put in place to account for this. This was solved by creating a Hashmap known as "*sequenceList*" that would contain this information. The key would be the port number of the client, while the corresponding value would be the sequence number. If the server received a packet from a specific client for the first time, the expected sequence number to which the packet would be compared to would be 0, then the relevant data would be added to the Hashmap. If the incorrect sequence number was in the packet, the response would be set to "NAK" and the user would get an error on the terminal. On the contrary, if it was the correct number it would then be incremented by one and stored in the Hashmap. The acknowledgment would then

be sent to the client. It should be noted this Hashmap stores the expected sequence number that should be received on the next transmission.

5. Examples and Captured Packets

Figure 5 shows an example of multiple clients active at once:

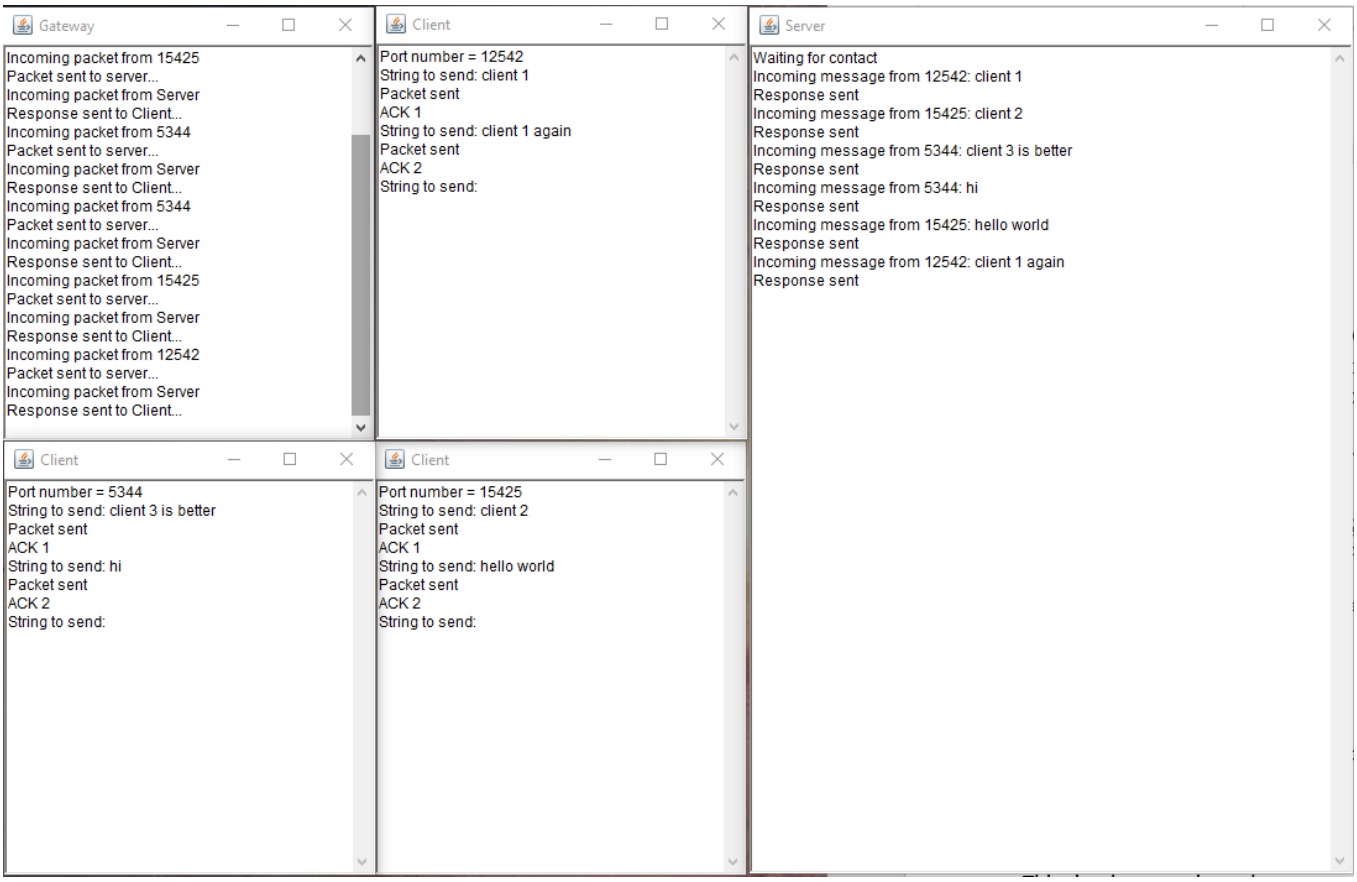

Figure 5

The following images show the packets captured on wireshark:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	127.0.0.1	127.0.0.1	UDP	73	12852 → 40789 Len=29
2	0.004273119	127.0.0.1	127.0.0.1	UDP	77	40789 → 50001 Len=33
3	0.007873717	127.0.0.1	127.0.0.1	UDP	71	50001 → 40789 Len=27
4	0.018015736	127.0.0.1	127.0.0.1	UDP	71	40789 → 12852 Len=27

Figure 6: All the packets sent in order of Client -> Gateway -> Server then Server -> Gateway -> Client

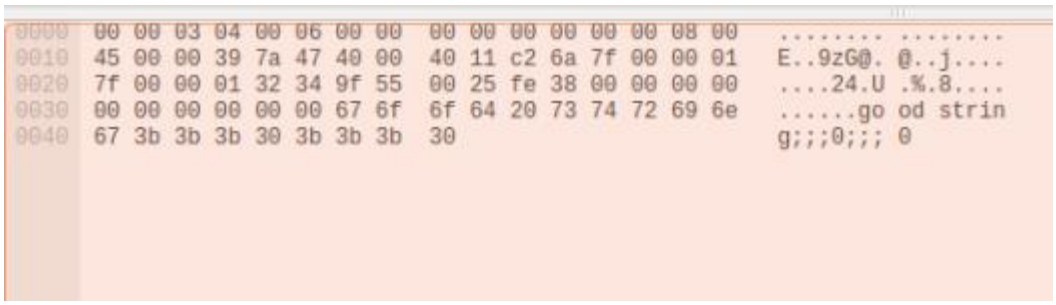

Figure 7: The packet sent from a client to the gateway, note that the sequence number and port are both 0, as the port hasn't been set by the gateway yet.

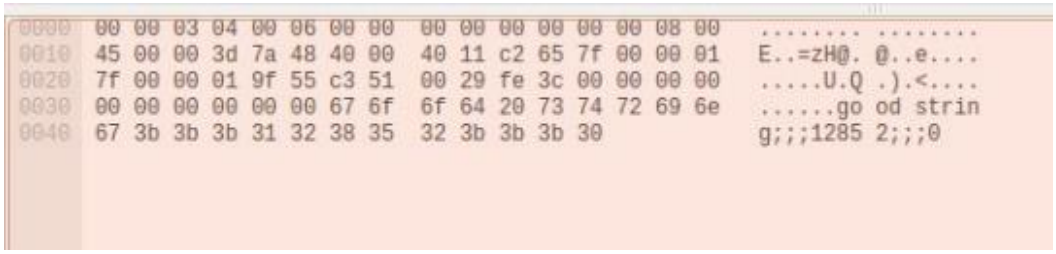

Figure 8: Here we can packet sent from the gateway to the server, this packet includes the original source port of the packet.

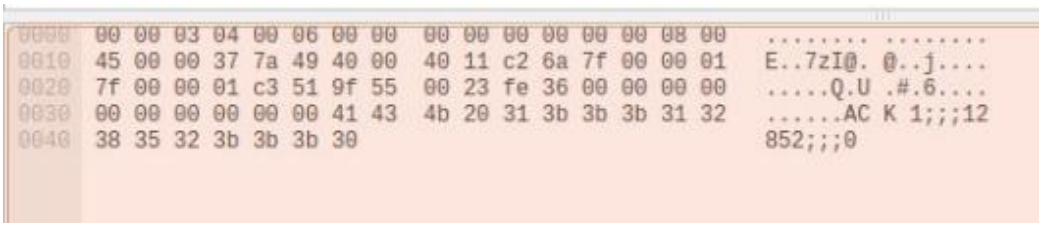

Figure 9: The response as it is being sent from the Server to the gateway.

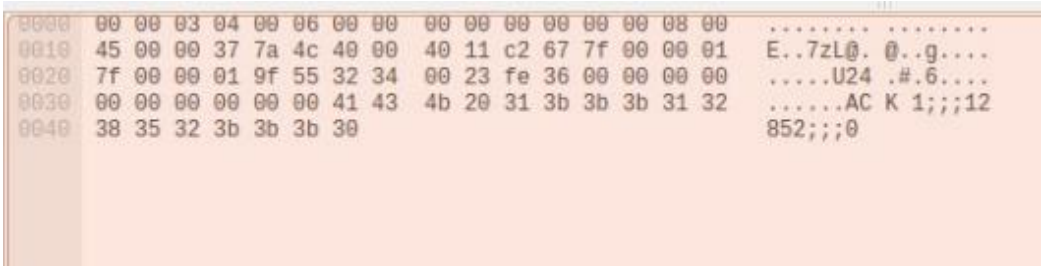

Figure 10: The response finally arriving at the client.