



Paul Devaney | 15316723

CS2031 Telecommunications II - Assignment 1 Report

Introduction

The first assignment in CS2031 Telecommunications II was based on control flow and error handling. The task was to design and implement a gateway which would act as a middleman between a client and a server, to control the flow of packets between the two nodes. The Gateway would receive packets from a client and forward them onto the server. Once the server receive a packet an acknowledgement would be sent back to the Gateway and lastly this acknowledgement would be sent back to the client. The Gateway was to be designed to be able to receive packets from multiple clients and the client and server was to be able to handle errors in the sequence of packets being sent.

The aim of this assignment was to form a comprehensive understanding of sockets, datagram packets and threads and also to design our own protocol for each node in the communication system to understand and implement.

1-Client, Gateway, Server

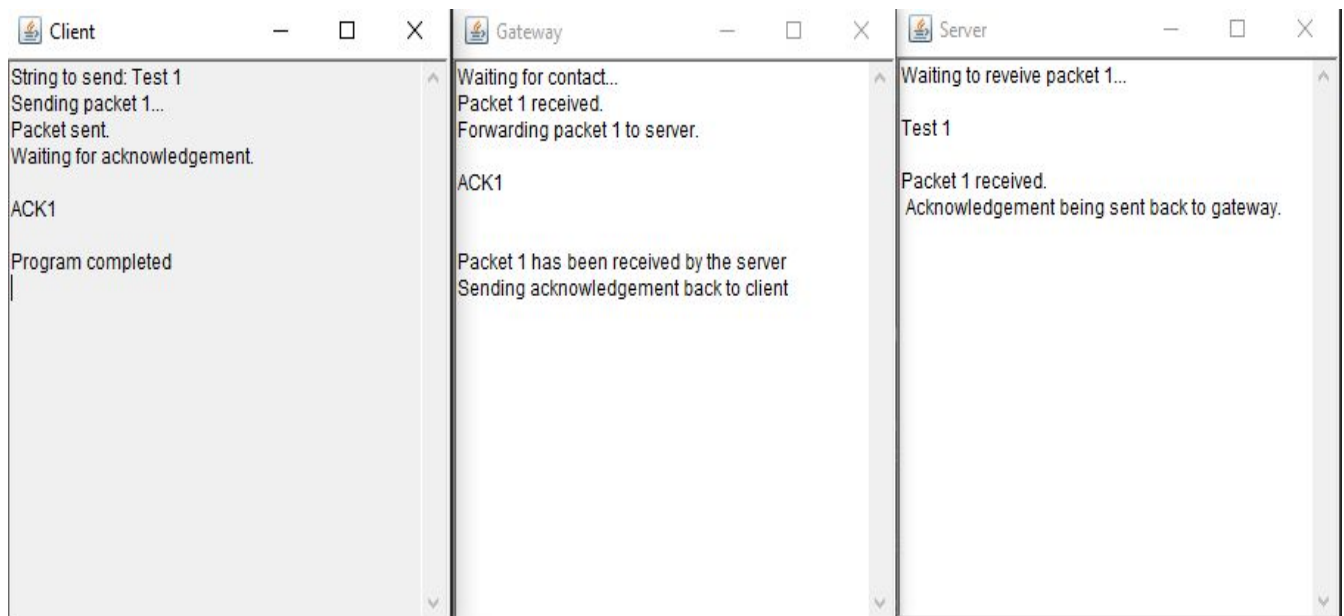
The first task at hand was the simplistic implementation of a client communicating through a gateway with a server. Basic classes for a client node and server node were given to us as a starting point. These classes simply sent a packet from the client to the server and sent an acknowledgment back. The Gateway class had to be implemented and the Client and Server classes were altered to implement the fully functioning communication system. In order for the three nodes to be able to communicate with each other, each of them first needed to use the same host. The local host acted as the host, as all nodes were running on the same machine. A datagram socket was created on each node to allow the sending and receiving of packets. The DatagramSocket constructor creates a datagram socket and binds it to the given port on the local machine. Each node in the program has its own port. In programming a port is defined as a "logical connection place" and specifically, using the Internet's protocol, [TCP/IP](#), the way a client program specifies a particular server program on a computer in a network.

Now that sockets had been opened on each node with their own ports, packets could be sent between the two. A DatagramPacket is created, DatagramPackets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another based solely on information contained within

that packet. Initially, I designed the gateway to simply send one packet from the Client to the Gateway, the Gateway would then forward this packet onto the Server. Once the server had received the packet, it would send an acknowledgment back to the Gateway, and the Gateway would forward the acknowledgment back to the Client. For a packet to be sent, the socket address had to be created from the port number of the receiving node and the host. An InetAddress object creates the address and the setSocketAddress method is used to set the address of the packet to be sent. The send() method then send the packet. Here is a visualisation of the basic initial implementation:

```
Client(Port 50000)----- [Packet] ----->Gateway(Port 50001)
Gateway(Port 50001)----- [Packet] ----->Server(Port 50002)
Server(Port 50002)----- [Acknowledgement] ----->Gateway(Port 50001)
Gateway(Port 50001)----- [Acknowledgement] ----->Client(Port 50000)
```

This is how the program looked in the terminals:

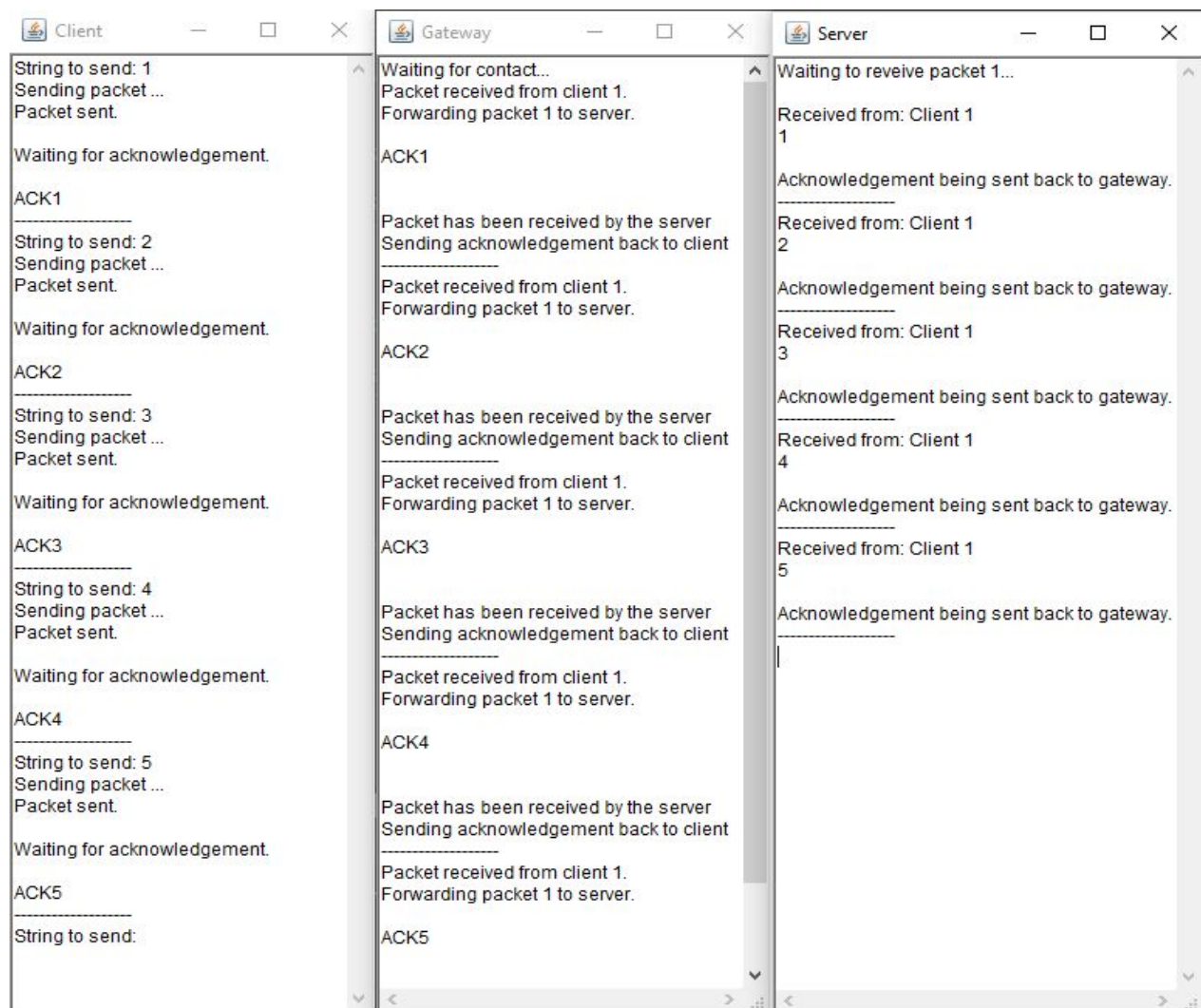



As shown above, the packet had successfully been sent from the Client to the Server through the Gateway, and an acknowledgment had been sent back through the Gateway to the Client. The main difficulty faced here was making the gateway recognise from which node a packet was coming from. To solve this issue, I began creating the protocol. The first byte in the buffer would be 1 or 0, if it was 1 this told the gateway that the packet was coming from the Client, and if it was 0 this told the

Gateway that the packet was coming from the server. To implement this, there was a setClientFlag method created in the Client class and a setServerFlag setup in the Server class. These methods first made use of the getData() method of a DatagramPacket to extract the byte array. They then changed the first byte accordingly, and lastly they used the setData() method which updated the buffer with the flag. Once the communication was working, a loop was set up so the the client could continuously send packets. This introduced the need for flow control and error handling.

2- Flow control and error handling

The implementation at this stage now allowed a single client to repeatedly send and receive packets. Below is an example of how each terminal looked at this point:





Now that the Client and Server nodes were sending and receiving multiple packets, it was necessary for each packet to have its own unique sequence number so that the server could check whether it was receiving the packets in the right order. The protocol I created made the second byte in the header act as the sequence number of a packet. I created a `setSeqNumber(DatagramPacket Packet)` method which would update the third byte in the header to the appropriate sequence number before the packet was sent from the client. The server was designed to know what sequence number it was expecting next. If the sequence number of the packet received by the server was the same as the sequence number it was expecting the server would print out the contents of the packet and send an acknowledgement back to the gateway to say that it had successfully received the correct packet. This acknowledgement was then sent back to the Client. However, if the sequence number of a packet received at the Server was not the same as the expected sequence number, the Server would send an error message back to the Gateway to be forwarded to the Client. In order to deal with the sending and receiving of errors, the protocol included an error flag. The `setErrorFlag` method set the third byte in the header to a 1 if there had been an error, and when the Gateway received a packet from the server it would check whether or not the error flag had been set. If it had been set, an error message is printed at the client end, if not the acknowledgement packet is printed.

The client node also made use of the `setSoTimeout()` method of a `DatagramSocket` to handle timeout errors. This method causes an exception to be thrown if a packet's receive method has not returned within a given time. This method was used as a timer mechanism, once a packet had been sent to the gateway from the Client the timer began, and if the socket did not receive a packet back within the given time the socket timed out. If the socket timed out, the `resend()` method was called which resent the packet to the gateway.

After implementing these two error handling measures, the implementation was much more complete and able to deal with all possible errors. Packets could now flow through the communication system without the possibility of losing sequence or timing out. This is how the system now looked as packets travelled through each node:

Client	Gateway	Server
String to send: 1 Sending packet ... Packet 1 sent.	Packet received from client. Forwarding packet 1 to server.	Received from: Client
Waiting for acknowledgement.	Packet has been received by the server Sending acknowledgement back to client	2 Acknowledgement being sent back to gateway.
ACK1	Packet received from client. Forwarding packet 1 to server.	Received from: Client
String to send: 2 Sending packet ... Packet 2 sent.	Packet has been received by the server Sending acknowledgement back to client	3 Acknowledgement being sent back to gateway.
Waiting for acknowledgement.	Packet received from client. Forwarding packet 1 to server.	Received from: Client
ACK2	Packet has been received by the server Sending acknowledgement back to client	4 Acknowledgement being sent back to gateway.
String to send: 3 Sending packet ... Packet 3 sent.	Packet received from client. Forwarding packet 1 to server.	Received from: Client
Waiting for acknowledgement.	Packet has been received by the server Sending acknowledgement back to client	5 Acknowledgement being sent back to gateway.
ACK3	Packet received from client. Forwarding packet 1 to server.	Received from: Client
String to send: 4 Sending packet ... Packet 4 sent.	Packet has been received by the server Sending acknowledgement back to client	6 Acknowledgement being sent back to gateway.
Waiting for acknowledgement.	Packet received from client. Forwarding packet 1 to server.	Received from: Client
ACK4	Packet has been received by the server Sending acknowledgement back to client	7 Acknowledgement being sent back to gateway.
String to send: 5 Sending packet ... Packet 5 sent.	Packet received from client. Forwarding packet 1 to server.	Received from: Client
Waiting for acknowledgement.	Packet has been received by the server Sending acknowledgement back to client	8 Acknowledgement being sent back to gateway.
ACK5	Packet received from client. Forwarding packet 1 to server.	Received from: Client
String to send: 6 Sending packet ... Packet 6 sent.	Packet has been received by the server Sending acknowledgement back to client	
Waiting for acknowledgement.		

Conclusion

In conclusion, the communication system I made allowed packets to travel seamlessly from the Client through the Gateway to the Server, and the server would then send acknowledgements back through the Gateway to the Client. A unique protocol was designed which was used by each node to carry out the necessary actions, and mechanisms were put in place so that common errors such as errors in the sequence of packets and socket timeouts were dealt with accordingly.

After completing this assignment I can safely say I have learned a great deal about sockets, packets, threads, control flow, error handling and many other aspects of computer



networks. Before starting this assignment I had very little knowledge of how information is sent and received and how protocols are designed and implemented. I spent approximately 10 hours on this assignment in total, including research, writing the program and writing the report.