CS2031 Telecommunications II

Assignment 1 - Gateway

Seán Hassett

1. Theory

1.1 Objective

The objective of this assignment was to come up with a gateway implementation that could receive traffic from multiple clients and forward the information to one of multiple servers while accounting for the possible loss of packets during transfer.

1.2 Stop & Wait

My intended solution to the assignment was to implement a Stop and Wait ARQ solution.

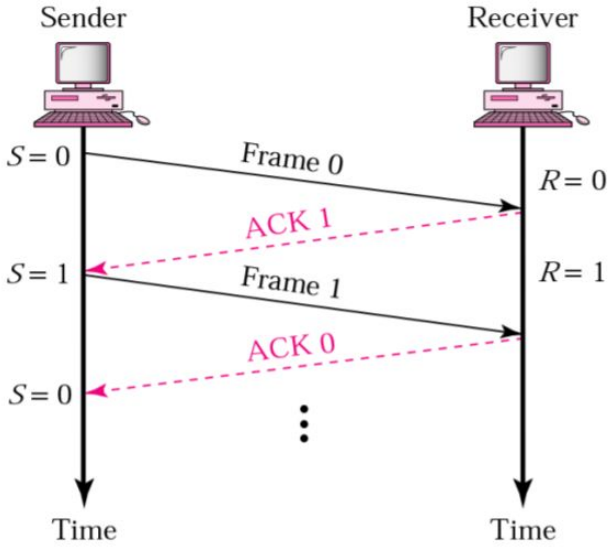

Fig 1.1. Stop and Wait ARQ, courtesy of B. Forouzan

As seen in the diagram in Fig 1.1, the principal behind stop and wait is for the sender to send numbered packets to the receiver. It then waits to receive a response from the receiver.

If the receiver receives a good sequence number, it requests the next packet, if it receives a sequence number other than the one it was expecting, it requests the one it was expecting. If a packet gets lost in between the sender and receiver, no response is sent from the receiver. In this case, the sender waits for a prescribed amount of time before resending the current packet.

If the ACK is lost then the sender behaves the same as if no response was received and sends the same packet again which is not the packet the receiver was expecting. In this case the receiver will ignore the duplicate packet and again request the next one.

While not implemented in my project, error detection bits can also be included in a packet. The receiver will also check these error detection bits according to whatever error detection protocol is in use and will not send the response unless the error detection is passed.

1.3 Advantages and Disadvantages

Some of the advantages of Stop and Wait are that it is straightforward to implement since packets are all dealt with one at a time. Dropped packets are also easy to account for for the same reason. The big disadvantage is that it is slow and inefficient. Senders must always wait for a response from the receiver before they can send the next packet.

1.4 Improvements

1.4.1 Sliding Window

A different approach to Stop and Wait is to use a Sliding Window approach. This allows the client to send not just one packet at a time but a defined number of packets.

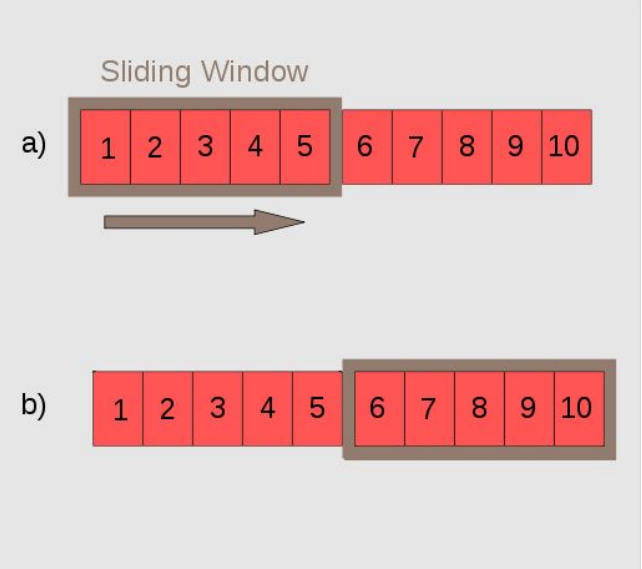

Fig 1.2 Sliding Window.
https://tariqaustralia.files.wordpress.com/2013/07/555px-sliding_window_svg.png

As seen in Fig 1.2, the sender here is sending a window of five packets at a time. This means it can send five packets before it has to wait for a response from the receiver, which is obviously much more efficient. Each packet still has its own sequence number and

the receiver expects to receive each one. If a packet is dropped, say number 3 in the diagram, the sender sends a new window starting from sequence number 3.

The window also means that the receiver can receive packets in the window out of order. As long as all the packets in the window are received in some order, an ACK will be sent for the first packet after the original window, number 6 in our diagram.

2. Implementation

2.1 Introduction

I decided early on that I would use Python as my language of choice for this assignment. Having some prior experience in Python, I thought it would an interesting project to get some more practice with the language. The key library used in the project is the socket library. This library contains everything needed for creating sockets between client and server scripts and transferring data. The main drawback I found with the library was that it already handled much of what needed to be implemented in the assignment and it could be restrictive over how much control you had over what was sent. Thus, the protocol I implemented for the project is of course unnecessary in reality and only implemented for the purposes of the assignment.

2.1.1 Quick Note

While it probably doesn't need to be said, the classes in my project need to be run in the same directory as the packet_utils.py class so that it can be imported.

2.2 Main Classes

I will now give a rundown on the primary classes utilised in my implementation.

2.2.1 packet_utils.py

This class contains methods for conveniently dealing with packets according to the protocol I have defined. It is implemented by the client, gateway and server.

The packet structure I have designed is as follows:

As seen, the packet consists of a 14 byte header with the payload starting from byte 14. The packet_utils class has a create_packet() method which takes in these six pieces of information as parameters and constructs a hex representation of them, using leading zeroes where necessary to preserve the indexing of the bytes.

The sequence number uses 2 bytes, as such the maximum value for the sequence number is 2^16. After this point the sequence numbers wrap around to 0 again. For Stop &

Wait the required maximum sequence number is only 2. As such, the two bytes aren't strictly necessary for our implementation as the project actually enforces a maximum sequence number of 2 and runs perfectly..

The IP's are 4 bytes each, one byte for each element of the IP address. The port numbers are 2 bytes each since the maximum value for ports is 2^16.

The sequence number and ports are treated as integers and formatted as hex strings with leading zeroes. The IP values are treated as four separate integers, each integer is formatted as a hex string and the dots are ignored (but replaced by the unpack method). The data is passed in as a stream of bytes. Each byte is converted to a hex string and added to the end of the overall hex string. The hex representation is really stored as a string and so is just for visualising the created packet. This hex string is encoded as bytes and the encoded version is what is returned as the packet.

packet_utils also has an unpack() method which takes in packet as a parameter and returns a namedtuple containing the six pieces of information stored as plaintext strings. The namedtuple allows for direct referencing of the packet segments by name instead of by abstract numbering. For example, I can access packet.destination_ip instead of accessing packet[3]. This improves the readability of the code.

Finally, packet_utils has a to_string() method. This prints a hex representation of the packet and prints each of the packet segments as plaintext. Realistically there is no need for this method but I used it in the gateway to verify that the packets received were correctly structured.

2.2.1 serverN.py

There are three server scripts in the project files which are identical except that they have their own unique port numbers. The purpose of having three servers was to showcase the gateway's ability to connect to multiple servers at once. This is achieved by having the server handle traffic to and from the gateway on separate threads for each packet.

Once a server instance is initialised, it creates a socket at an address which is hardcoded into the class. It then listens for an incoming connection request.

Upon receiving a connection request, the server calls the socket.accept() method which creates a new socket. This new socket is what is used to transfer data between the server and whatever made the connection request, in our case the gateway. The port number for the new socket is decided by the socket library and is assigned randomly. The documentation for the library doesn't detail how the port number is assigned. It is one of the

limiting factors of the library that, to the best of my knowledge, the user can't assign the port number.

Once this new socket is created, the server creates a thread and listens for incoming packets. Upon receipt of a packet, the server calls the packet_utils.unpack() method to access the information in the packet.

To keep track of sequence numbers, each server has a dictionary which it uses to map client_ids to sequence numbers. The server uses the source port contained in the packet to identify clients and derives its expected sequence number from the dictionary based on how many packets it has already received from that client.

There is a chance of error here if the server is running for a long time that a new client appears on the same port as an older client. In this case the server would not know the difference between the new client and the old client and would expect the wrong sequence number. Since the ports appear to allocated by socket.accept() randomly but still sequentially, the chances of the server running long enough for the port numbers to wrap around are probably minimal.

The server sets its destination IP and port to the same values as the source IP and port received in the packet and then it checks to see if the received sequence number matches its expected sequence number. If they match, the server prints out the data from the packet and sends a response asking for the next packet. If they don't, the server sends a response asking for the packet with the expected sequence number.

2.2.2 gateway.py

The gateway initialises in a similar way to the server. It creates a socket at its predefined address and then listens for incoming connection requests. On receipt of a connection request from a client, it creates a new thread for the client and uses the socket created by socket.accept() to send and receive information. Each client is set on a timeout, so that if the gateway doesn't receive anything from the client in an amount of time, it closes that connection and the client must send a new connection request.

When the gateway receives a packet, it expects the payload of the packet to be another packet. This nested packet is where the gateway derives its destination address.

Since the address of the new socket cannot be known until the connection is accepted by the gateway, it is seemingly impossible for the client to know the address of that socket prior to connecting to the gateway. As such, the client is forced to send blank values for the source IP and source destination components of the packet. To rectify this, the

gateway unpacks the received packet and repacks it with the correct source address information before sending it to the gateway.

Having taken those steps, the gateway creates a new socket and sends a connection request at the destination address defined in the client packet. Upon successful connection, it forwards on the client packet (with the corrected source address) and then waits for an ACK from the server. Upon receipt of an ACK, it simply forwards it straight on to the client.

Since I wanted the client instance to have the functionality of deciding which server each packet goes to without having to start a new instance for each server, the server and gateway are quite wasteful in terms of creating and destroying sockets. For each packet the gateway receives, it creates a new socket, as does the server. Since the gateway doesn't know where client packets are bound until it receives them, it can't have a predefined socket already connected to the destination. It might be more practical to have a client instance restricted to only one server and thereby enable the gateway to keep the same socket for packets from that client, but I decided that it seemed more realistic that the client may want to send packets to different destinations without restarting.

2.2.3 client.py

This class is used to interface with the gateway by sending packets with user-written messages as the payload. When a client instance is initialised, it creates a socket at the gateway address and sends a connection request to the gateway. As such, it is important that a gateway instance is already running before the client is initialised. It is also important that an instance of the destination server is running before the client attempts to send a packet.

When the client is successfully connected to the gateway, it prompts the user for some input. When it receives this input, it prompts the user to select a destination server, 1, 2 or 3. When it has the destination server, it then allocates a sequence number. Similar to the server class, each client instance maintains a dictionary so that it can know which sequence number is expected at which server.

The destination addresses of each server are predefined in the client class so once it knows which server the user wants, it knows the destination IP and port. As mentioned before, the client at this point doesn't know it's own address so it uses zeroes for the source IP and port, to be corrected by the gateway.

With this information and with the encoded user message as the payload, the client uses packet_utils to create a packet destined for the server. It then immediately creates a new packet, this time with the gateway address as the destination and the first packet as the

payload. Because the gateway has already accepted the connection to the client, this packet does not actually need the destination address but it is included anyway for the sake of consistency.

The client then sets an acknowledgment boolean to false and waits to receive an ACK from the server. Upon receipt of the ACK, it checks to see that the expected sequence number is not the sequence number of the packet just sent. If it is, then it resends the same packet, if not it asks the user for new input for a new packet. If the client does not receive an ACK within five seconds of sending its packet to the gateway, it resends the packet. This five second value is controlled by a predefined global variable and is easily changed.

3. Testing

3.1 Main Classes

To test some of the aspects of my implementation, it was necessary to create some additional classes in order to test certain aspects.

3.1.1 lossy_server.py

This is a near copy of the other server classes. The key difference with this class is that it will do a random number check on receipt of a packet with a 1 in 500 chance that it pretends to have not received the packet. In this case it sends back an ACK asking for the same packet again. This allowed me to verify that the client behaved correctly in the event of a dropped packet, by resending the “lost” packet.

3.1.2 bombard_client.py

This is a near copy of the client class. The key difference with this class is that instead of asking for a message to be sent, it continually sends a large amount of packets in an infinite loop as fast as possible. This client always sends packets to the lossy server. Running two bombard clients simultaneously allowed me to test how the gateway handled a large amount of traffic from multiple sources.

3.2 Other Testing

3.2.1 Timeout

To test the client resending a packet on a timeout, I simply commented out some lines in the gateway code so that the gateway didn’t send on the packet and thus no response is received from the server. As expected/desired, the client continually resends the same packet every 5 seconds as defined by our timer variable. A potential improvement that could be made here would be to have the client give up after some amount of failed attempts and display an error message instead.

3.2.2 Packet Capture

By using RawCap to capture packets on localhost and then using Wireshark to read the resulting data dump, I was able to inspect the packets that were being sent by my project. A screenshot of one such packet is below:

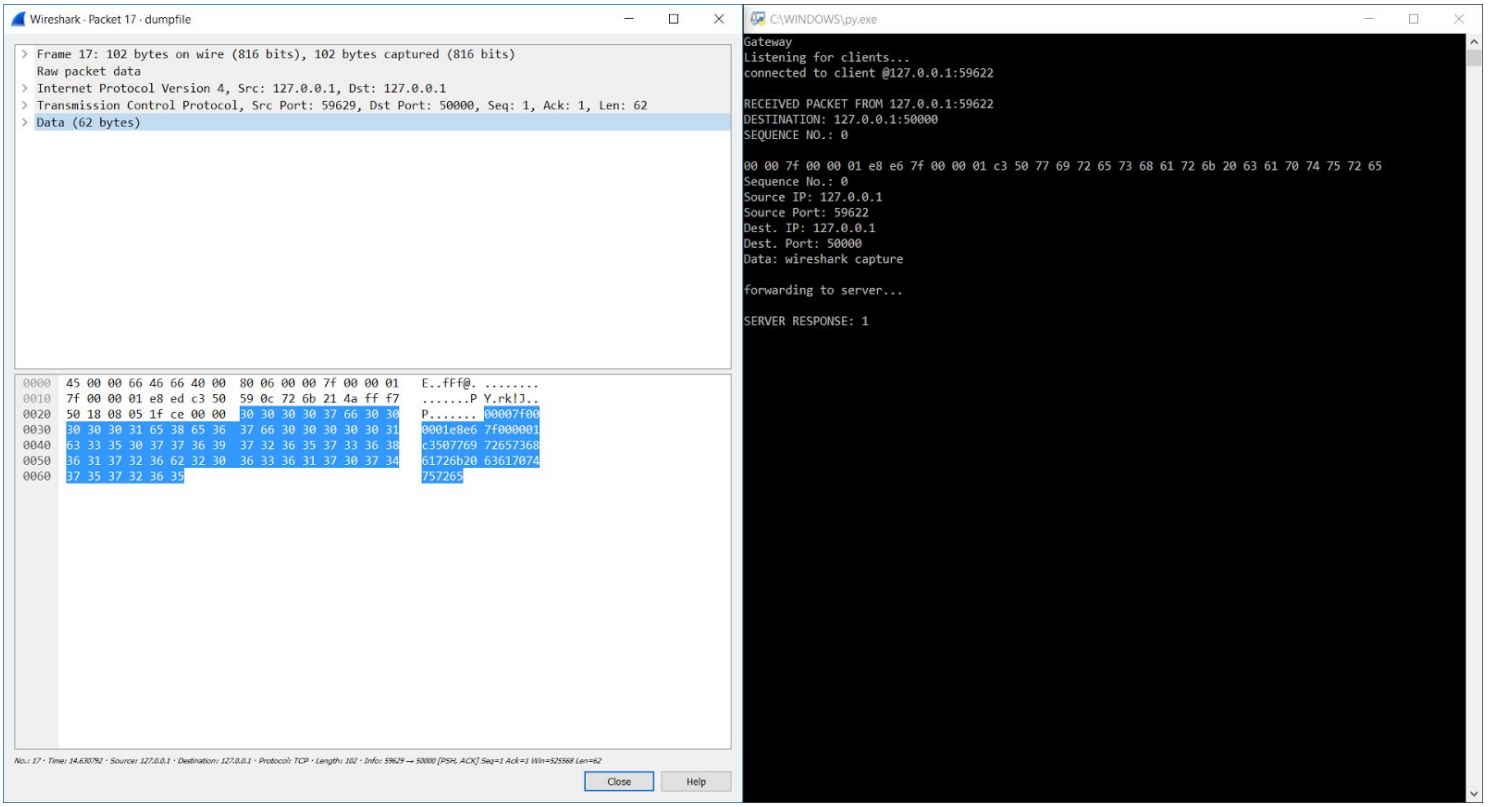

Fig. 3.1. Captured packet

As seen in the captured packet in Fig. 3.1, the socket library attaches its own header information to the packet and treats the packets created in our project as the payload for its packets. If we inspect the payload of the captured packet and the hex dump printed by the gateway, we can see that they match perfectly and so the information that we expect to see transferred is indeed seen in the captured packet.

We can also see that the server is receiving the packet from port 59629, the port assigned when the server accepted the connection to the gateway and created a new socket. Again, due to the nature of the socket library, there is no way for us to know what port the gateway will send packets to the server on without having the server print it to screen for us.

4. Reflection

Overall I enjoyed working on this assignment and found it to be a worthwhile endeavour. It was a good way of learning the material from the lectures and reinforcing some concepts. I feel like a better implementation than Stop and Wait could perhaps have been implemented but otherwise I am reasonably happy with my implementation.

Outside of the course material I enjoyed the opportunity to work in a language which we do not get to utilise very often. It was a good chance to work with a new library and try to understand how it was best implemented. It was also the first time I implemented threading in a project which was a good learning experience.

One of the drawbacks to using Python was not having access to the existing Client and Server classes. However, there was good documentation online which got me started in the right direction. Probably the biggest drawback to using Python was the lack of control over some aspects of the assignment, especially the port numbers assigned to sockets. Looking back, I am not fully convinced it is the best language to have done the project in but the learning experience was still worth it.