COP532 – Internet Protocol Design Report

Table of Contents

[1 – Abstract 3](#_Toc503351725)

[2 – Design 4](#_Toc503351726)

[2a – Reliability 4](#_Toc503351727)

[2b – Segmentation 5](#_Toc503351728)

[2c – Routing and Forwarding 5](#_Toc503351729)

[2d – Error Detection 6](#_Toc503351730)

[3 – Implementation 7](#_Toc503351731)

[3a – Reliability and Segmentation 7](#_Toc503351732)

[3b – Routing and Forwarding 7](#_Toc503351733)

[3c – Error Detection 8](#_Toc503351734)

[4 – Testing 9](#_Toc503351735)

[4a – Full reliability layer 9](#_Toc503351736)

[4b – Forwarding layer 9](#_Toc503351737)

[4c – Packet order 10](#_Toc503351738)

[4d – Checksum layer 10](#_Toc503351739)

[5 – Reflection on Coursework 11](#_Toc503351740)

[5a – Core design 11](#_Toc503351741)

[5b – Reliability and Segmentation 11](#_Toc503351742)

[5c – Routing and Forwarding 12](#_Toc503351743)

[5d – Error Detection 13](#_Toc503351744)

[6 – Appendices 14](#_Toc503351745)

# 1 – Abstract

The project was to design a protocol and develop chat programs to implement this protocol. This report will go into detail about the design decisions made for our project, and reflect on what the group might have done better if given the opportunity to redo the coursework. The report will also look into the design of the system, and the protocols themselves, why decisions on the protocols were taken and how that affected the system that the group produced.

# 2 – Design

## 2a – Reliability

The reliability design was focused around the protocol design agreed on by both groups. In this design, it was decided that it and segmentation would effectively work as one layer as their jobs were not vastly different enough to warrant their own protocols. In this, the protocol design agreed (along with segmentation) used 4 bytes of the 100 total bytes per packet. Within reliability specifically, there is the ‘acknowledgement’ bit (which would notify the receiver whether the incoming packet was simply an acknowledgement packet or a normal data packet), the ‘type’ nibble (which specified the real type of packet incoming, whether it be a file, a normal message, or routing information), and ‘end’ bit (which defines whether the current packet is the last packet of the current message ID).

When a host receives a packet, it will unpack the headers and read the message ID, linking it to any other packets with the same ID. Secondly, it will take the sequence number, which will define at what position the current packet needs to be positioned to reassemble the message in full. It will also check the ‘Flag’ byte to see what type of packet/message this is, and whether this is the final packet in the message.

When a packet is viewed to be intact and legitimate, an acknowledgement will be sent to the sender to inform them that the packet has been received and not to worry about resending any packets, both hosts can then move on to other communications. If an acknowledge is not received within 200ms, then the sender will resend the packet, up to five attempts. If no acknowledgement is received within 200ms of the fifth packet, then the packet is dropped and the sender will cease to resend it. This may result in the receiver getting an incomplete message, but it is necessary as it was deemed that five attempts at 200ms each was clearly enough for a host to respond to a packet.

The length of the data is necessary and included as it shows the receiver (if the full 94 bytes is not utilised by the sender) where the sender’s message ends, and the padding from ICNS begins. The receiver can then remove the padding where the sender states the message ends, and the message can be viewed in full with no padding.

An acknowledgement packet will have the relevant message ID and sequence number included, along with the ‘acknowledgement’ bit being changed to ‘1’, so the receiver knows that the packet is an acknowledgement packet and not a regular packet. With this system, the packet will have all of the important information from the receiver to the sender, for the sender to know that their packet has been taken and received correctly.

## 2b – Segmentation

The segmentation layer (which is grouped with reliability, as stated previously) utilises the remaining bits of the 4 bytes of the reliability and segmentation layer, this includes the ‘end’ bit (which signifies the end of the message, if there has been more than one packet), the message ID’ byte (which allows for the sender to have more than one message in flight at once, meaning if a host sends more than one packet for a message, both can be received and reassembled by the receiver without mixing up the packets from the different messages), and the ‘sequence number’ byte (which signifies the place a packet must be positioned when unpacking to form the message back in order).

When a host sends a message that will take more than the maximum 94 bytes, it will be segmented (all packets of the message will have the same message ID, which will be taken from a list when not in use), and the sequence number for each packet will notify the receiver in which order to place the packets to reassemble the message correctly and in order.

The ‘end’ bit will be ‘0’ up until the last packet is created and sent, at which point it will be altered to ‘1’, which tells the receiver that the packet is the final packet of a message, meaning that any other packets after this will almost certainly be duplicates. If the bit is corrupted and therefore flipped to its opposite, this will be spotted in the checksum calculation and the packet dropped with no acknowledgement being sent by the receiver, so the sender will be forced to automatically resend the packet after 200ms.

## 2c – Routing and Forwarding

In the Routing and Forwarding layer, there are two headers segments, simply ‘source’ (the source address) and ‘destination’ (the destination address).

The routing is performed on a hop-by-hop basis, meaning that the packet will be sent simply by a single jump all the way to the destination. This should make the transmission more optimal as each host will know where to send the packet given the destination address. With the ‘hop-by-hop’ system, the checksum is calculated and checked at every host, so if the packet is corrupted, it will be dropped before it gets to the destination, meaning the sender will resend the packet when no acknowledgement is received.

For example, host 1 (H1) may be connected to H2, and H2 to H3, and H2 also to H4. If H1 wants to send a packet to H4, H2 will know that given H4 is one of its neighbours, it can route it directly there. However, if H4 was connected to H3 instead, H2 would not know about H4, and so would send it to H3, who would then be able to forward the packet to its correct destination. This system relies on the entire network working properly together, and each host having up-to-date Forwarding and Lookup tables.

In the program, there are two tables dedicated for routing and forwarding. The ‘Lookup Table’ (which contains the neighbour name and the corresponding IP address), and the ‘Forwarding Table’ (which tells the host where to send a packet to get to another host). With this system, it makes the design of the protocol flexible and easily expandable as more hosts can be added at will and simply included in the Forwarding and Lookup tables, simply doing this will then fully include the new host in the network and it will be able to partake in discussions on the program.

## 2d – Error Detection

Error detection is performed on a hop-by-hop basis, meaning that any host that a packet passes through will check the checksum of the packet which was generated by the sender against its own generated checksum (given the packet). If the checksums match, then the packet it forwarded along to its next destination given its Forwarding and Routing header information, otherwise, if the checksum is different it can be assumed the packet data is corrupted, and the packet dropped.

No acknowledgement will be sent and no notification to any other host will be sent, this is because the sender will then wait out the remaining time from the 200ms allotted for that packet and resend it. This system ensures that there should not be any corrupted packets being delivered to the receiver at any point.

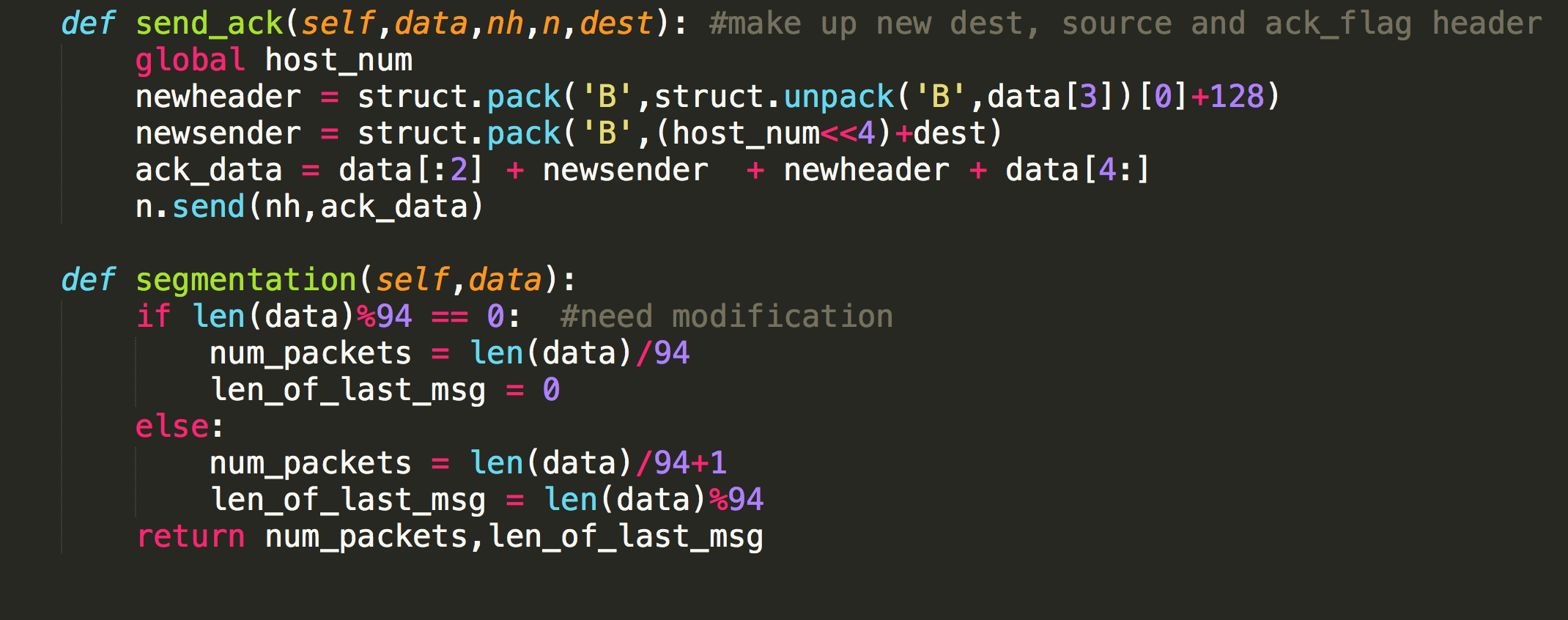
# 3 – Implementation

From the global view of this program, mainly we have 3 classes in total which represent the three specific layers in our protocol. Moreover, these 3 layers are under control of a steering method which includes a non-stop ‘while’ loop. For every loop, the program listens to either the sending part or the receiving part and do different things when entering one of them. For example, when FD (file descriptor) is ready, the procedure sends all the packets in packet\_list and keeps the packet state unless it is well delivered while when NET is ready the procedure checks the destination to do whether forwarding or receiving.

## 3a – Reliability and Segmentation

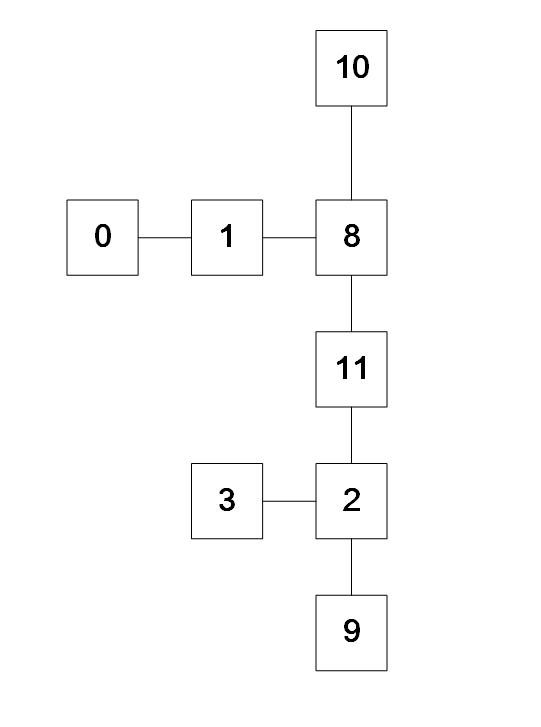
This project starts with a very simple reliability layer with only a one-byte header which contains one bit for acknowledgement and 7 bits for the randomly generated message\_id. The simple\_reliability layer which in the code is wrapped as a class holds several functions together such as initiation, encapsulation, decapsulation and send\_ack. The idea of classification is the fundamental idea for designing rest layers in the protocol.

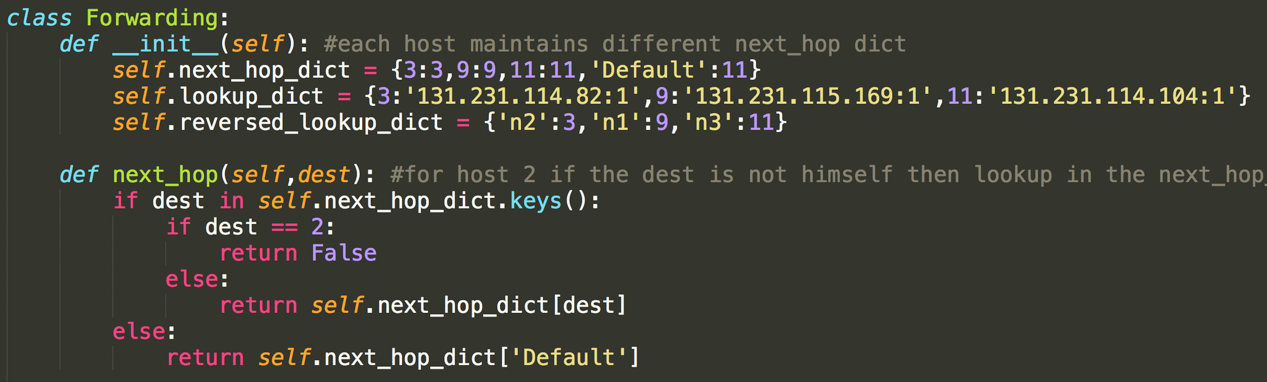
The Full\_reliability layer focuses on the global unique packet number, segmentation and acknowledgement which is a more advanced and smarter version of simple\_reliabilty. In regarding to global packet number, a randomly generated 8-bit long message id and a continuous sequence number from 0 to 255 make up the global unique number. Segmentation deals with the total input and split them into one single packet every 94 bytes (the other 6 bytes are left for the complete header). It is in this layer that decides how many packets are the chatroom going to send. Function send\_ack swaps the destination and source host number in the packet and changes the acknowledge flag to 1 and repack the packet again.



## 3b – Routing and Forwarding

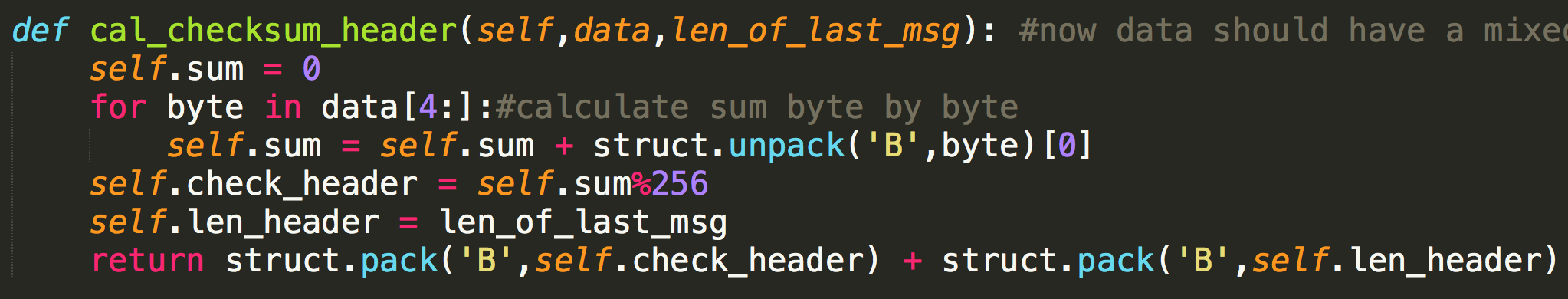
In order to do forwarding, three hard-coded dictionaries are initialized in class Forwarding which are next\_hop\_dict, lookup\_dict, reversed\_lookup\_dict respectively. Every individual host has its own version of the three dictionaries because in this static routing network each host has different neighbour hosts. Basically, the forwarding header consists of a source host number and a destination host number. The encapsulation function in this layer shift the source number to left 4 times in binary first to create the higher 4 bits in the header, then the result will concatenate with the destination host number in order to make up the 8 bits long header in this layer. Function next\_hop figures out the next neighbour this host is going to send to to reach one step closer to the destination by referring to the next\_hop\_dict. The diagram of our tiny static network is shown below and the three dictionaries of host 2 are also demonstrated for example.





## 3c – Error Detection

The very beginning of the packet header goes the checksum header. In fact, checksum header has two parts (checksum value and length of message in the packet) and each has a length of one byte. The checksum starts from summing up the binary number of the message byte by byte and then do modulo of 256 to fetch the last 8 bits as the checksum value.



# 4 – Testing

In this phase, we tested our program layer by layer and then combine them together and finally achieved the implemented part working across two groups.

## 4a – Full reliability layer

Firstly, a droprate of 0.2 is brought into our program to test if the Full\_reliability does its work and successfully resends the dropped packets between two adjacent hosts (host2 & host3) in our network diagram. At first, the multiple packets message could not be delivered properly. Gradually, we found the issue is that **os.read(fd,100)** restrains the length of the message. After changing it to **os.read(fd,300),** the multi-packet message flies to its destination correctly. Besides, a bug that the host will resend all the previous unsent packets is addressed by initialising the packet list when a new message comes.

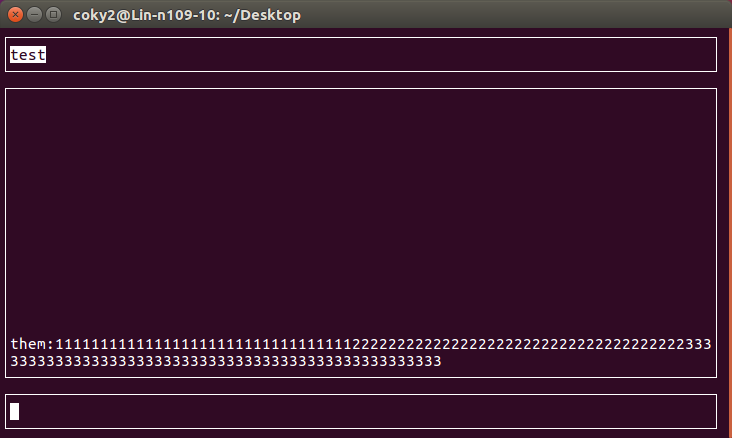
## 4b – Forwarding layer

Secondly, a third host (host9) is created to gain the result of forwarding layer. We have quite a lot attempts to achieve forwarding but fails in different situations. One major problem is that the so-called ‘router’ does not forward our message to its destination and sends destination acknowledgement back to destination host. Later, we found that we mess up the destination number and source number in the header and meanwhile the unreasonable control strategy also counts to this bug. The right of control strategy is supposed to check first if the message needs to be forwarded and then either enter next loop (means it is a ‘forwarding’ message and do nothing but jump to next loop) or do receiving part and check if it need to send back acknowledgement. In a word, ‘routers’ shall only do forwarding and do no more things on swapping the source and destination host number.



## 4c – Packet order

Thirdly, during the test, sometimes multi-packet message arrives in disorder because that the first packet has been dropped and resent while the second packet has not. This issue is done by reassembling the arrived messages and displaying them in the correct order.



## 4d – Checksum layer

Last but not least, sometimes (especially when packets drop happens) the checksum value does not match the correct value. The inconsistent checksum value issue is handled by initializing the packet list (discussed in previous section). Moreover, when testing between two groups, the different length of message encapsulated in the header eventually leads to message unaccepted.



Finally, we agreed on using the exact length of the message as the number in header and this bug is fixed as well.

# 5 – Reflection on Coursework

## 5a – Core design

The core design focused around having a spine for the main program, and then creating instances of each method (layer of the protocol) in the spine to utilise, meaning on a programming standpoint, it was clear and concise, but it was also usable in the sense that it would not take much effort to alter the program. This was useful as it allowed for easy documentation and commenting of the program. With this, it meant that other developers could edit the code without being confused by a program that had no defined borders for each layer.

The alternative was to not have any clear and defined separation between the different protocol layers, which would have been easier to implement, but significantly harder to document and edit, when that became necessary. The decision to use a ‘spine’ approach was made very early on, to give as much time as possible to set out a good idea of the shape the program would take.

## 5b – Reliability and Segmentation

The Reliability and Segmentation layer was originally to be totally separated, with reliability being performed at a different stage to segmentation. Unfortunately, this added a lot of complexity to the protocol header, since it would be very difficult to unpack the reliability header to retrieve the sequence number and message ID in order to perform the necessary reliability checks, but then also repack them or send them individually to the segmentation layer (particularly the sequence number) to make sure that the packets were organised correctly.

In the end, it was decided at a higher level that both reliability and segmentation would occur at the same state, making it much easier to synchronise the groups and streamlining the protocol design and creation. With this system, the program could unpack the sequence number and message ID in one go and utilise them as needed. This meant that we had to combine two layers that to the mind would be thought of as totally separated, but time constraints and simplicity forced the decision. Given the opportunity and more time, it might have been decided that separation was the best idea for modularity, and a more complex, but better-defined system might have been preferred.

Another alteration made during the design phase of this section of the protocol was to move the ‘ACK’ bit to a dedicated byte in the total header, which we labelled ‘Flag’. This byte of the header consisted of the ‘ACK’ bit (to define whether the packet was an acknowledgement packet or a normal data packet), the ‘END’ bit (to define whether the current packet is the last packet for that specific message ID) and the ‘Type’ (which defines what type of data is contained in the packet, for example, ‘0000’ is a normal message, ‘0001’ is a file and ‘0010’ is routing information).

Naturally, this left 16 different options for the type of message, which left a lot of space for expansion. This was thought of as necessary as the additional bits are so minute in scale to the overall packet length that there was little to lose by leaving expansion room.

Reliability was also separated from error detection in the protocol, which again might seem an odd decision, but the fact that a hop-by-hop error detection had been agreed meant that specific section of the header had to come right at the start, meaning that it is before the source and destination sections of the Forwarding/Routing part of the protocol, the Reliability and Segmentation parts, and of course the data. This was a trade-off again for flexibility and simplicity. Originally it was attempted to include this in the Reliability header, but it was very difficult on paper to justify and in the end, it was decided that separation was the best policy, given the importance of error detection to the overall specification.

## 5c – Routing and Forwarding

The Routing and Forwarding headers were some of the more interesting to design, as there were a few different ways that they could be implemented. The system that ended up being agreed on was one that utilised a ‘Forwarding Table’ (which would have been used anyway – is in essence just pairing a neighbour ID to a unique host ID), but also a ‘Lookup Table’, which took the globally unique host ID’s, and then mapped out where the next hop will be in the network to reach any given destination.

This was particularly useful as static routing was utilised given the small nature of the project, but also because it allowed for rapid expansion of the network if needed. Given we limited our potential network size to a maximum of 15 hosts, static routing was more than acceptable, although all preparations were made for dynamic routing to be implemented if more time was available in the end. This however never came to pass as there was a significant amount of other work to complete.

A major positive of the table system we implemented was that we did not need to repeatedly include in the code the IP addresses of the host’s immediate neighbours. As we instead utilised a globally known ID, we could simply tie the IP address of a neighbour to their allocated ID, meaning it was possible to communicate with that host using the ID, rather than the IP thereafter.

However, a downside to implementation of this system was that it was more complicated to physically program in the project. This was because every host needed their own unique tables as they only know about their own immediate neighbours, so every host will have a totally different set of tables to every other host. Along with this downside, the IP addresses had to be hardcoded in the code itself, meaning that if any of the lab computers IP addresses changed, they had to be manually altered in the code itself.

A solution to this issue would have been a system of dynamic routing, which would have meant that each host individually propagated data around the network, expanding their own tables as and when changes were made to their immediate neighbours. As mentioned however, though the protocol was designed, it was never developed and implemented due to the pressing time restraints.

## 5d – Error Detection

The hop-by-hop nature of the error detection meant that no corrupted packets would be delivered to the destination host, but also that there was a large amount of overhead on the network. This is because of the ‘automatic resend’ nature of the protocol if no acknowledgement is received within 200ms, but also because it allows quite a few reattempted resends (being 5), meaning for one packet being sent, it could take up to a full second for that packet to be completely sent with no issues, but also that it might end up sending the same data 5 times.

An alternative system of error detection would have been and end-to-end implementation, which would have meant significantly less overhead on the network and host, but the potential for more corrupted packets getting through to the destination host. This would have been bad, as there was no system designed which accounted for a host receiving a sequence number that technically does not exist, so the destination host will have sent an acknowledgement back to the sender for a sequence number that it never despatched. This could have caused a number of issues given the numerous potential implementations that could have occurred between the groups. Thankfully, a hop-by-hop system is certain to prevent this outcome.

Alternatively, error detection could have been included in the reliability layer. A major issue with this was the would have been that it forced the packet to be checked only at the end nodes, rather than the hop-by-hop checks that we wanted. This is different to an implementation such as TCP, in this respect, but for a system of the size of this, it works very well. However, for an extremely large network, where a packet may hop through hundreds of nodes to get to a destination, hop-by-hop may mean that the packets simply take too long to get to the destination, and end up expiring through a time-out.

# 6 – Appendices