COP532 – Internet Protocol Design Report

B611605 Colin Yuan
B632386 Yiwen Huang
B725022 Ting Ni
B730374 Matthew Beechey

Table of Contents

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

The diagram below (Figure 2-1) demonstrates the whole structure of the packet header and separate layers in the protocol. The complete header takes 6 Bytes in the packet. Though not everything included in the header have been implemented, there are sections in Chapter Design and reflection to discuss them.

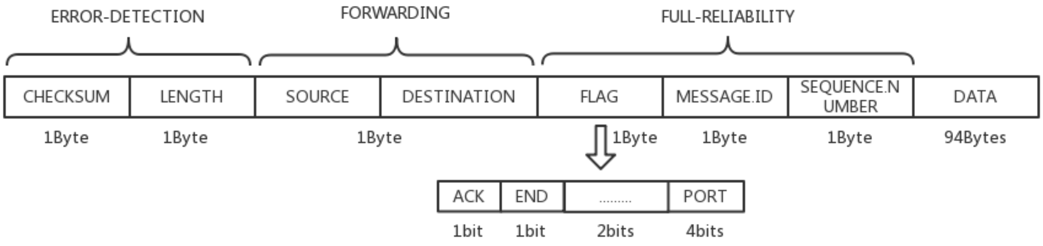


Figure 2-1

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 3 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 500ms, then the sender will resend the packet, up to five attempts. If no acknowledgement is received within 500ms 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 500ms 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 in flag area being changed to '1' (first bit of flag), 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 3 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' (the second bit of flag zone) 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 500ms.

2c – Routing (static) and Forwarding

In the Routing and Forwarding layer, there are two headers segments, simply 'source' (the source address) and 'destination' (the destination address) and each of them is 4-bit long which only allows 15 hosts (one is kept for broadcasting) in total within a tiny network.

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 includes a one-byte long checksum value which is calculated from the last 8 bits of the total sum up of the exact data part and anther byte for storing the length of real message in current packet.

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 500ms 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.

2e – Broadcast and Dynamic Routing

Broadcast and dynamic routing use the same header format as the static forwarding. But host number 15 (all ‘1’s in binary) remains for broadcasting and ‘0010’ which is last the 4 bits of the flag represents a routing message in this protocol.

Broadcast means that one host sends a message to everyone, which needs forwarding to access the hosts that are not the neighbors of the sender. Broadcast routing relies on link-state which uses n-way-unicast in this case, and in fact, it is a way to control flooding that each node maintains a list of source addresses, and the use of sequence numbers also helps to control flooding. To avoid flooding, there is a mechanism to check the source ID for not sending message back to source should be set, and to hold a record after receiving messages for only forward once is also important. Each multicast packet and sequence number of each broadcast packet will carry the IP address of all of the multiple recipients.

In dynamic routing, every router knows information from immediate neighbours. Thus, if there is a new node joining the routing, it should be added in the routing table. Otherwise, every router knows the cost between immediate neighbor and itself, so every router will choose the minimum cost way to transfer the messages. Thus, when routing table is changed every router should tell neighbors to update their routing table.

2f – File Transfer

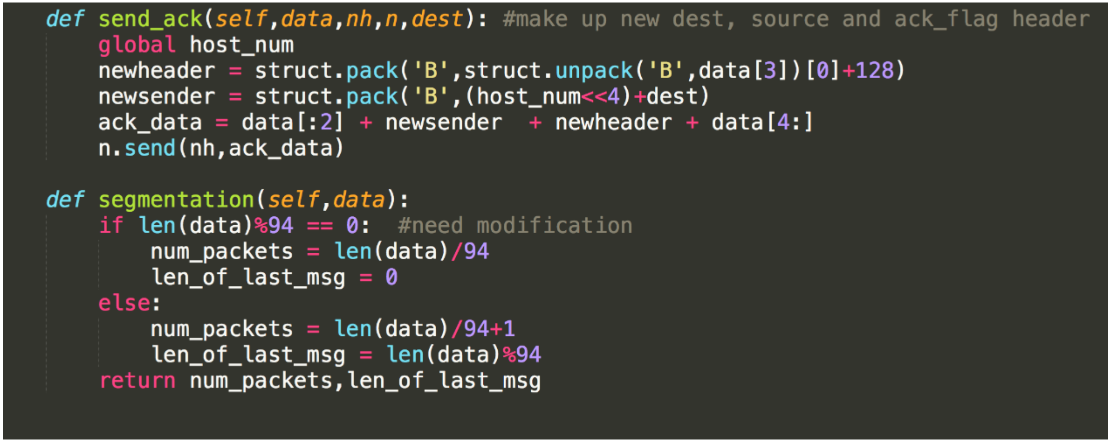
The protocol points out that flag ‘0001’ stands for file transfer and sequence number in reliability layer ‘0000000’ is also remained for filename. Besides, because we only have one sequence number for the filename, the result will be only one packet carries the filename which limits the length of the filename to up to 94 Bytes.

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.



```
    def send_ack(self,data,nh,n,dest): #make up new dest, source and ack_flag header
        global host_num
        newheader = struct.pack('B',struct.unpack('B',data[3])[0]+128)
        newsender = struct.pack('B',(host_num<<4)+dest)
        ack_data = data[:2] + newsender  + newheader + data[4:]
        n.send(nh,ack_data)

    def segmentation(self,data):
        if len(data)%94 == 0:  #need modification
            num_packets = len(data)/94
            len_of_last_msg = 0
        else:
            num_packets = len(data)/94+1
            len_of_last_msg = len(data)%94
        return num_packets,len_of_last_msg
```

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 (Figure 3-1) is shown below and the three dictionaries of host 2 are also demonstrated for example (Figure 3-2).

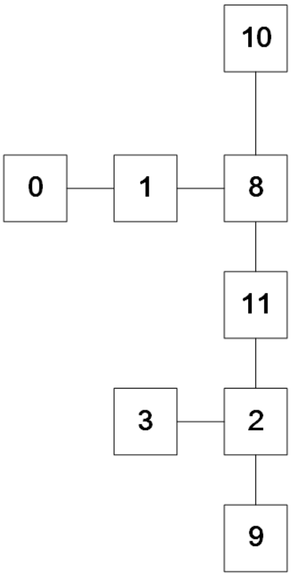
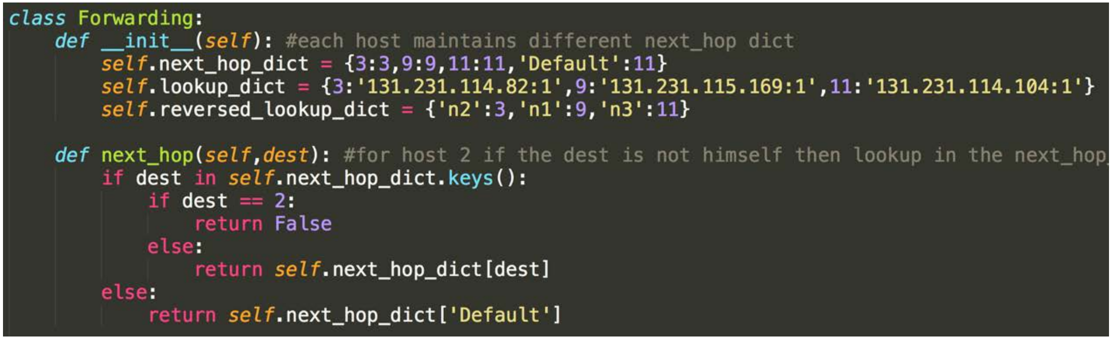


Figure 3-1



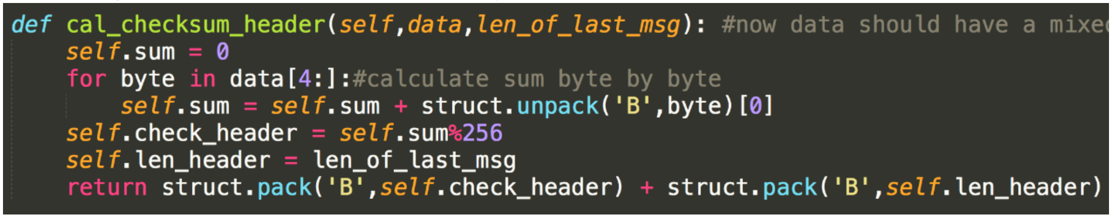
```
class Forwarding:
    def __init__(self): #each host maintains different next_hop dict
        self.next_hop_dict = {3:3,9:9,11:11,'Default':11}
        self.lookup_dict = {3:'131.231.114.82:1',9:'131.231.115.169:1',11:'131.231.114.104:1'}
        self.reversed_lookup_dict = {'n2':3,'n1':9,'n3':11}

    def next_hop(self,dest): #for host 2 if the dest is not himself then lookup in the next_hop
        if dest in self.next_hop_dict.keys():
            if dest == 2:
                return False
            else:
                return self.next_hop_dict[dest]
        else:
            return self.next_hop_dict['Default']
```

Figure 3-2

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. Figure 3-3 shows how this is implemented in details.



```
def cal_checksum_header(self,data,len_of_last_msg): #now data should have a mixe
    self.sum = 0
    for byte in data[4:]:#calculate sum byte by byte
        self.sum = self.sum + struct.unpack('B',byte)[0]
    self.check_header = self.sum%256
    self.len_header = len_of_last_msg
    return struct.pack('B',self.check_header) + struct.pack('B',self.len_header)
```

Figure 3-3

4 – Testing

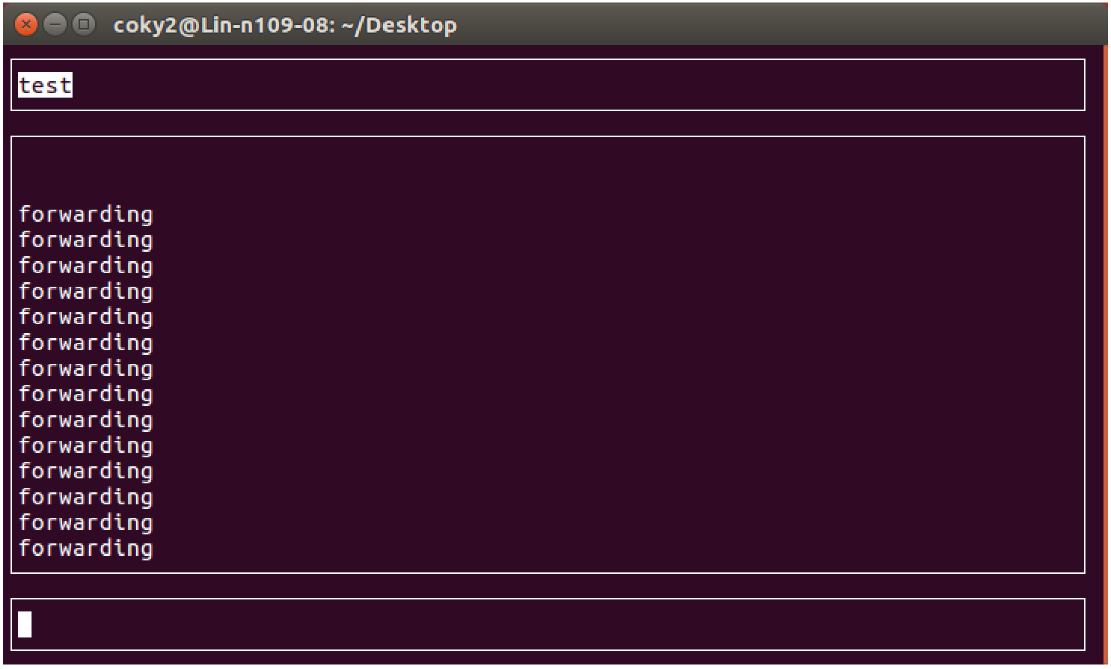
In this phase, we tested our program layer by layer and then combine them together and finally achieved the implemented part working across two groups.

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. The followed picture (Figure 4-1) indicate what happens to forwarding-only host while running.



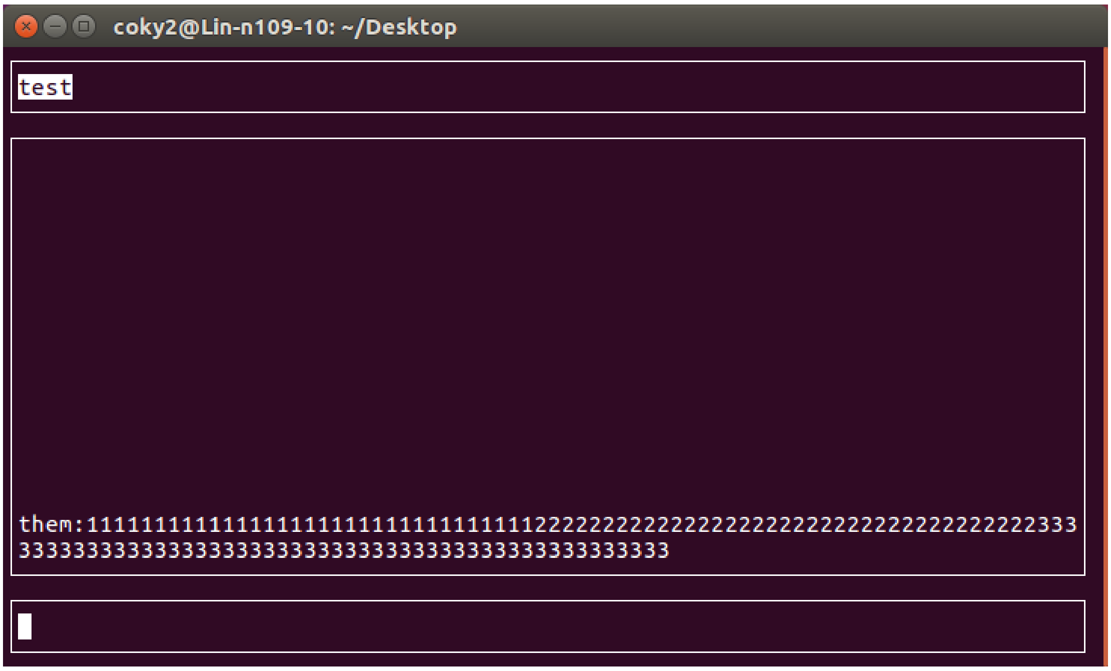
```
test
```

```
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
forwarding
```

Figure 4-1

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 (see figure 4-2) by reassembling the arrived messages and displaying them in the correct order.



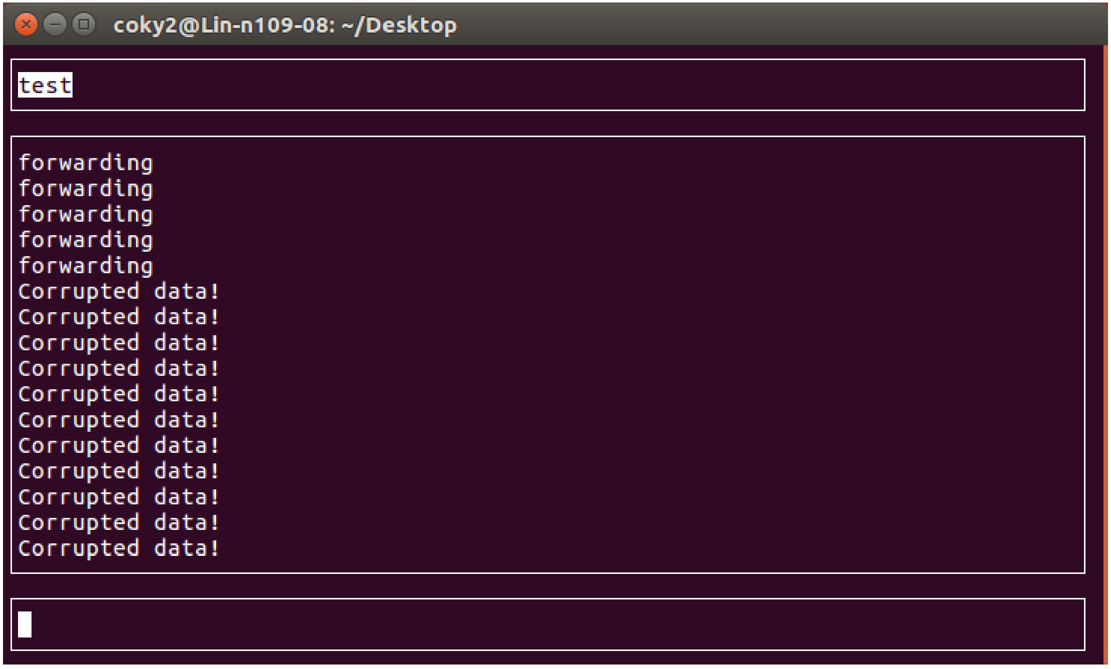
```
test

them:1111111111111111111111111111111112222222222222222222222222222222222222333
333333333333333333333333333333333333333333333333
```

Figure 4-2

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 (see figure 4-3).



```
test
```

```
forwarding
forwarding
forwarding
forwarding
forwarding
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
Corrupted data!
```

Figure 4-3

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 500ms, 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.

5e – Broadcast and Dynamic Routing

There is a risk for using broadcast when a host uses broadcast to send message bring acknowledgement back to source, then it may be form a loop between several hosts. Though keeping the source host info and avoid sending back to it somehow addresses this issue, it works well only in small networks and it would not help when network routing changes after sending out the broadcast. An alternative way is to use shortest path to determine which host to send the broadcast.

Dynamic routing and Broadcast are two possible enhancements, however, there are some problems in carrying out. Low feasibility of broadcast and time limit also obstructs achievement of them.

6 – Conclusions

During the process of negotiation with another group, the importance of well-documented protocols is very important, or separate groups could not get the different versions of the protocol implementation working. In spite of this, the global design and management of the code is as essential as the protocol itself. For instance, the layers are seen as classes in our code and they all interact with the control strategy within one method. The positive point of doing so it is easy implementation at the very beginning when only one or two layers are well-discussed and are going to be implemented.

However, as the layers scale up, it is quite hard to consider the individual steering strategy for newcomers without affecting the existing layers. After the whole protocol has been well documented, the concept of a traditional 5-layer network model breaks up and our protocol takes its place. This is crucial in fully comprehending the traditional network model and what does each layer deal with. The Full_Reliability layer in our protocol does similar thing that TCP does for example. The difference is that we only acknowledge the current packet while TCP also expect next message in the queue (see details in previous section).

In conclusion, though our protocol has quite a lot of shortcomings compared to TCP, the achievement is still very clear given the short design and development time. More importantly, we receive not only the programming know-how and protocol designing strategy but also the global view of the whole network system and its fundamentals.

7 – Appendices

7a – Matt's Reflection

My contribution to the project of designing and development of a protocol focused mainly around the design itself, documentation and liaising with the other group to make sure the protocol was properly understood. In hindsight, I would have preferred to take a more active role on the development of the protocol, besides my bug-fixing and other development input, but unfortunately the system utilised (being a single file program) made it incredibly difficult to have more than a single primary developer, and I think this was a similar situation with the second group.

Having said that, I was very pleased with the outcome of the project and think that the development went a smoothly as could have been expected, although there was a bit of breakdown of communication between the groups at times, leading to some compatibility issues towards the start of the assignment. These were remedied, and inter-group development and compatibility continued throughout the project thereafter.

I thought my role in the design of the protocol was important, particularly in the class meetings, as I took a lead on the design, trying to get the groups to decide on a final design so development could properly continue. As previously stated, my role in the physical development was limited, as having more than a single primary developer on a single computer would have dramatically slowed down the development, and given that the entire group (including myself) had direct input on every process in the design and development of the protocol throughout the entire assignment, I think the decision to limit the group to one main developer was the correct one, in order to get the most out of the progress we were making in the in-class meetings.

It could be argued that a system whereby multiple files were utilised, therefore modulating development and allowing more than a single programmer to work on the project at a time could be utilised, however the downside to this is the means of communication between the members of the group, which was at times lacking, although never was critical. Because of this, I did what I could to ensure that everyone had the same idea of the protocol, making sure that everyone understood the structure of the packet headers and trying to force communication during the in-class meetings, and this worked, as we ended up with a functional header structure.

I very much enjoyed the module, mainly because of the freedom allowed to the groups to decide their own structures, and because of the freedom allowed to define the packet structure, whether it was the most optimal design or not. This led to a lot of trial and error design and development, which I believe made everyone learn why certain aspects of a header needed to be modulated and others didn't, for example.

7b – Colin’s Reflection

Basically, I have done most of the code part and testing assignment. I am also involved in protocol design with classmates whether in-class meeting or group discussion in lab session. Due to the lack of group work, our project progress once stuck. Anyway, I spend extra hours during the Christmas time to catch up with the other group’s progress. Finally, our project meets the all the basic requirements in our protocol.

Though only one programmer in a group can manage his code well and works out uniform type of code, it still could be better if we have more group work. Sharing the load and gaining knowledge of team spirit is equally important as the coursework itself I suppose.

I sincerely enjoyed this module very much since Ian act as not only a teacher but also a guide to us which gives us more space to make the decision on both protocols and implementations. I have ever been taught one similar module in China, but teacher mixed the layers up and pushed all of them to us in a rush so that we didn’t fully understand how each layer really works in the protocol. However, after this module, I can be proud of saying I am a network staff now. So, thanks to Iain’s quick idea of setting up this module too.

7c – Ni Ting’s Reflection

My contribution to this coursework is testing and debugging, to ensure the accuracy of the protocol. After we finished our code design and basic communication test, we tried to connect to another group, and then we tested whether we implemented all the basic functionality required by the protocol. From the designer’s point of view, our protocol basically implements all the basic connection-oriented functions of transmission and communication. Nobody can be deny the fact that Internet network layer services are not reliable, IP does not guarantee the delivery of datagrams, does not guarantee the orderly delivery of datagrams, nor does it guarantee the integrity of the data in the datagram. Therefore, I spent so much time to discuss the reliability of the protocol with my group members and then we need to send reliable messages according to our protocol design, and we complete reliable communication with other nodes.

Reliability is ensured by applying sequence numbers and checksums in the reliability layer. By specifying sequence number for each message and requiring the receiver to remember the segments to detect repetitions and to prevent unnecessary retransmission. The checksum ensures that the packet will never be modified. The sequence number is used to give each packet's order. This layer of the overtime mechanism can be effectively avoiding the loss of data packets. In other words, the sender did not receive the corresponding packet acknowledgment in time and the resend operation will be triggered.

During the process of the protocol design and implementation, in my opinion, the most two important fields in a segment are the sequence number and acknowledge number. These two fields are the key part of the protocol for reliable transport services. At the same time, we can correctly identify the message type.

Through continuous testing and modification, this protocol can provide two or more users with a reliable way to transfer information and files. It implements a reliable flow control message protocol. The packet can be forwarded correctly. It can implement the datagram forwarding routing protocol. Providing lookup table and forwarding table to deal with. It can be choosing to provide multicast forwarding and routing services.

According to the Internet protocol design, in order to reduce the complexity of the protocol design and debugging process, we have adopted the principle of protocol layering. Each layer to achieve correct functions, each layer is built on its lower layer. At the same time, this principle also allows different team members to focus on one level over a period of time without having to worry about the lower level of implementation. This method not only improves group work efficiency and saves time, but also makes the refinement of functions easier to achieve. Therefore, it is a good experience for me to understand the structure of an Internet protocol, how to design a reliable protocol and implement it.

7d – Yiwen’s Reflection

In this module, I have learnt a lot of new knowledges about Internet protocol, I have done experiments about chat room by using UI and I have drawn the clear detailed header picture for our group. We discuss which method should we use or what essential information should header contains in the class. It is a quite flexible module for me, because most things about protocol design are discussed between two groups rather than being told in advance.

After doing experiments in the lab, I recognize that the timeout rule is quite significant and should be set to carry out retransmission after a period time without acknowledgement, which is to cure the losing messages in underlying service. The duration of timeout is quite an important point in this design, because if duration of timeout is too short that will lead to duplication of messages, however, if duration of timeout is too long that will bring other problems.

I also have several questions about acknowledgement, after enquiring Ian I have learnt that if messages have been received, but acknowledgement has been lost when transmission, sender also will retransmission. In this situation, receiver have received two same packets, so receiver will discard the second one immediately according to its own record, then send another acknowledgement to sender.

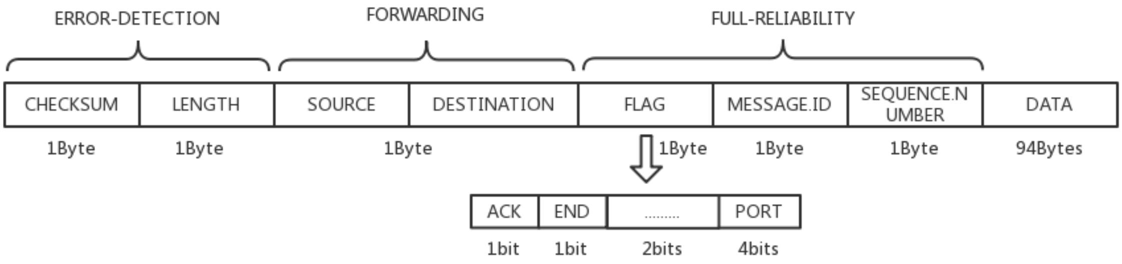
The other important problem should be solved in this design is corruption control, the traditional mechanism to control the corrupt error is to use checksum or error-detecting code. The general principle of corruption control is that correct blocks satisfy a certain standard, however, the corrupt blocks do not satisfy this. It can help protocol to distinguish correct blocks from corrupt blocks, but the accuracy of filter still relies on the complexity of the standard. And I consider that checksum is a good way to filter the corrupt messages and it is also an important part for error-detection.

Broadcast routing relies on link-state and it equals to n-way-unicast, and in fact, it is a way to control flooding that each node maintains a list of source address. To avoid flooding, the mechanism to check source id for not sending message back to source should be set, and to hold a record after receive messages for only forward once is also significant.

When we discussed about dynamic routing and file transfer which are two possible enhancements, we tried to achieve broadcast and dynamic routing, however, there are some problems in carrying out these two objects. And low feasibility of broadcast also obstructs achievement of dynamic routing and file transfer.

In the summary, I certainly learn many knowledges not only from class but also from discussions we made, and this method can inspire our inspiration to solve the problems we faced at that time.

7e – Packet Structure



7f – Routing and Forwarding Documentation

1 INTRODUCTION

The Forwarding and Routing Protocol is designed for providing a simple, but effective forwarding mechanism for the protocol. It will provide a ‘hop-by-hop’ service, meaning that the service provides a mechanism of routing the packet by the next hop address. For example, Host 1 (H1) may be connected to H2, and H2 to H3, but not H1 to H3, so for a packet going from H1 to H3, the next hop from H1 would be H2.

This document declares the functions required to be implemented by the Forwarding and Routing Protocol, and agreement on initial state as well as final state.

1.1 Intention

Forwarding and routing in a very simple, one-to-one communication is easy as there is only the single connection, however, when the network of hosts gets more complex, there is a need for a network map from each host, which decides where to send a packet if that packet is not destined for that host itself. To achieve this, forwarding and routing is used, which should allow for hosts to effectively act the role of a router and decide where in the network the packet should be sent to get to its ultimate destination.

Forwarding and Routing Protocol is a hop-by-hop protocol as it allows for better routing, especially on a static network. The service assumes it will be deployed above an unreliable layer such as ICNS.

```
                 Protocol Layering

     +------------------------------------+
     |     Reliability and Segmentation    |
     +------------------------------------+
     |            Forwarding               |
     +------------------------------------+
     |            Lower Layer              |
     +------------------------------------+
     |               ICNS                  |
     +------------------------------------+

                     Figure 1
```

The Forwarding layer will receive data packets from the Reliability and Segmentation layer, which will have encapsulated its own header data, Forwarding will then encapsulate its own header into the packet. The interface between Forwarding and the lower layers, such as ICNS is important because it allows for the hop-by-hop mechanics of the layer.

1.2 Operation

The primary reason for the Forwarding layer is to provide a hop-by-hop service in routing and forwarding a data packet from source to destination, via the most optimal route. To achieve this, the position of the layer on the system is important, as well as multiple tables implemented on each host. If the protocol was higher in the table, then it would only secure forwarding and routing at either end of the packet route, as it stands, it secures the forwarding and routing at a hop-by-hop basis.

Given the system will initially be a static routing layout, the system needs to know the next hop for the route (just its neighbours). The Lookup Table will have the next hop for every address available depending on the destination. The Forwarding Table will have a host's IP Address and their respective Global ID.

2 FUNCTION SPECIFICATION

2.1 Header format

The packet will currently be simply 1B (1 byte) in total, with a source and destination header, each composed by 4 bits (totally 1 byte in total).

```
Forwarding Protocol

  1 2 3 4 5 6 7 8
  +-+-+-+-+-+-+-+-+
  |   S   |   D   |
  +-+-+-+-+-+-+-+-+

     Figure 2
```

Source (S): 4 bits

Refers to the source's Global ID (to be represented in binary, but in the Lookup Table, that will be converted to decimal).

Destination (D): 4 bits

Refers to the destination's Global ID (to be represented in binary, but in the Lookup Table, that will be converted to decimal).

Both source and destination being limited to 4 bits means that there can be a maximum of 15 hosts in total (2^4 = 16, minus one for the broadcast), this means that the network is limited in size, but this is solely because of the static routing protocol currently being implemented. In the future, when dynamic routing is implemented, there may need to be more hosts available.

7g – Reliability and Segmentation Documentation

1. INTRODUCTION

The Reliability and Segmentation Protocol is designed for guaranteeing hosts-to-hosts reliable communications and dividing messages into segments so that the length of each packet does not exceed 100 bytes.

This document declares the functions required to be implemented by the Reliability and Segmentation Protocol, and agreement on initial state as well as final state.

1.1 Intention

Communications based on ICNS package are unreliable, which may cause packages lost problem when sending messages between hosts. It is essential to agree on a protocol that provides several mechanisms for achieving reliable communications as upper-layer of ICNS. Moreover, as ICNS stipulates maximum length of 100 bytes for each packet, the protocol is also required to implement segmentation function for users who intend to send a long message.

Reliability and Segmentation Protocol is a reliable end-to-end protocol and provides segmentation service. The protocol assumes that it can be performed above a simply unreliable communication service as a lower level protocol, for example ICSN.

Protocol Layering

The interface between Reliability and Segmentation protocol and Application process is illustrated as a set of functions that process as well as encapsulate the messages obtained from user. For instance, the call related to encapsulation is responsible for adding important information as header to message, which makes the message recognisable within Reliability and Segmentation layer between two hosts.

The interface between Reliability and Segmentation Protocol and lower level protocol is important in terms of the data flow in the network stack. The output of this protocol depends on the documents of lower level protocol.

1.2 Operation

As demonstrated above, the purpose of Reliability and Segmentation Protocol is providing reliable communication and message segmentation services between hosts. To achieve these goals, the systems are required implement in the following areas:

Reliability
Segmentation

Reliability:

The protocol simply guarantee that sender can resend the packets when waiting acknowledgment (ACK) until a timeout interval. This requires each end system, the sender, maintenances a timer and records the time of each packet. Considering the sender is able to send different message, message ID is set to record different message. As to the receiver, it is required to send back a ACK when receiving a packet so that using both the sequence number from segmentation and message ID identifies a unique packet in ACK.

Segmentation:

The sender need to divide the message into small blocks so that each massage received by ICNS layers is no more than 100 bytes. Sequence numbers are set to identify the order of packets based on message ID. Moreover, the Length field notify the length of data in each segment so that the receiver can recover the message and get rid of paddings. As for receiver, the sequence numbers are used to specify the correct order of each message.

2. FUNCTION SPECIFICATION

2.1 Header Format

The header of Reliability and Segmentation Protocol is simple as shown in figure 2. The header consists of 4 bytes where the first byte is Flag field that includes ACK, END and Port. The remaining three bytes are Message ID, Sequence Number and Length respectively.

Acknowledge Number: 1 bit

The ACK takes 1 bit to note whether it is a ACK packet. 0 represents a normal packet while 1 means an ACK packet.

End Number: 1 bit

End number notifies the final packet of a massage. 1 means the final packet otherwise it is not the final packet. If a message consists of only one packet, the End number is set as 1.

Type: 4 bits

The type declares the types of message. As the host may send normal message from user interface, file type and routing information from time to time. Receiver can have different functions to deal with different types of message.

The types are agreed as follow:
0000: normal message
0001: file
0010: routing information

Message ID: 8 bits

This field specifies the amount of message that the sender maintenances and declare the message when the receiver receive a message. For example, if the sender sends two messages to receiver at a time, the receiver receives two packets with the same sequence. In this situation, receiver differentiates these two messages based on the Message ID.

The ID begins with 0 (00000000) to 255 (11111111).

Sequence Number: 8 bits

Sequence Number is from 0 (00000000) to 255 (11111111). The sender is responsible for segment the message into several packets and assigns each a continuous sequence number. The receiver can recover the message based on the sequence numbers.

Length: 8 bits

This field shows the total length of the segment data from original message. Receiver can extract the data and integrate the original message from each packet with the same Message ID based on Sequence Number and Length.

Other parameters:

TIMEOUT: 500ms

If the sender cannot receive a ACK from receiver within a TIMEOUT period, the sender is required to retransmit the packet in case the packet has not reached the receiver.

RETRY: 5 times

The sender will resend 5 times if the sender cannot get a ACK, which means the sender will wait 5 TIMEOUT intervals. After that, the packet will not be retransmitted again.

7h – Error Detection Documentation

1 INTRODUCTION

The Error Detection header is designed to provide a simple, but effective mechanism for the detection of anomalies in messages. It will provide a 'hop-by-hop' service, meaning that the service provides a mechanism of checking the host-provided checksum at every single hop, in case of data corruption.

This document declares the functions required to be implemented by the Error Detection header, and agreement on initial state as well as final state.

1.2 Intention

Error detection in a reliable protocol is vital. In that vein of thought, there are two primary systems which can be implemented to allow for the detection of corrupted data. The first of which is hop-by-hop, which checks the calculated checksum at every host hop, meaning that no matter where in a network the data packet it, if corruption is detected, then the packet is dropped.

The alternative system an end-to-end system whereby the calculated checksum is only compared at the end of a packet's journey. This means that at every hop, the host will only forward the packet to the next location, and will not care about the integrity of the data. Only at the receiver host will a corrupted packet be dropped.

```
                  Protocol Layering

        +------------------------------------+
        |           Error Detection          |
        +------------------------------------+
        |     Reliability and Segmentation   |
        +------------------------------------+
        |             Forwarding             |
        +------------------------------------+
        |             Lower Layer            |
        +------------------------------------+
        |                ICNS                |
        +------------------------------------+

                      Figure 1
```

The Error Detection Layer will receive packets from the Reliability and Segmentation layer, and will then add its own header on, which is just two bytes in total. The checksum, and the total length of the message a user is sending. They have to be at the top of the protocol stack as every host needs immediate access to them.

1.2 Operation

The primary reason for the Error Detection layer is to provide a hop-by-hop service in detection of corrupted data in a data packet from source to destination. To achieve this, the position of the layer itself is vital as at every host hop, that specific host needs quick access to the checksum, to compare it to a checksum that itself calculates. If there are discrepancies in the checksum, then the packet is dropped, and it is then left to the Reliability Layer to send a new packet.

2 FUNCTION SPECIFICATION

2.1 Header format

The header itself will be 2 bytes in total. The first byte of which is the checksum itself, and the second byte, which is the length of the message from the user.

```
         Error Detection Protocol

 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |   Checksum    |   Length      |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2

Checksum: 1 byte

This refers to the calculation of the checksum itself. The 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 does a modulo of 256 to fetch the last 8 bits as the checksum value.

Length: 1 byte

This simply refers to the length of the data from the user (the message) that is stored in the available 94 bytes of the data section of a packet. This is necessary as every packet has to be a total of 100 bytes, so if a message is less than 94 bytes, the remainder is added through padding in the ICNS layer. The length byte simply tells the receiver where the message ends and the padding begins, so the receiver can read the message without any of the added padding.