COP532 – Internet Protocol Design Report

B424047 Simon Yan Lung Yip

B611638 Yang Zhou

B633595 Jie Su

Contents

1. Introduction

The Internet Protocol (IP) is a set of rules that are responsible for the format of data sent across the Internet or any other computer network. Internet protocols such as Transmission Control Protocol (TCP) and the Internet Protocol (IP) are open-source (non-proprietary) allowing communication between any set of interconnected networks.

The purpose of this project is to deepen the understanding, describe the design and the implementation of an Internet protocol. The Internet protocol is designed to allow two or more hosts to communicate in a specific manner via a chat program using Python programming language. The project will use the ICNS library's built-in features and functions for the UI of the chat system and underline network of sending the packet between hardware. The ICNS library was especially created and provided by the module leader Dr Iain Phillips to help students for this project. Three or four students joined together in a group to contribute, discuss and implement ideas to successfully achieve the project requirements.

2. Internet Protocol Design

Standardisation is a vital aspect that must be considered prior the implementation stage. Standards are documents that establish specifications and procedures designed. In this case, both group 1 and 2 must discuss each step during the design of the protocol to achieve a standard protocol that works for both groups. Before writing the codes, the protocol layers and fields must be discussed and agreed on. It is a very challenging project as changes happen after each meeting.

2.1 Stacking

The protocol will be implemented using the modular layering structure; this structure will allow the ability to deal with complex systems. Modularisation allow provide high and easy maintenance, updating of system by adding new modular and the most important feature that any change of implementation will be transparent to the rest of the program. Like the TCP/IP stacking, both sender and receiver have the same number of layers and it will be dealing with the packet in the same manner. After a long discussion between both group members, it was agreed to implement four layers to handle packets. The layers are Application, Reliability and Segmentation, Forwarding and Error Detection as shown in the figure below.

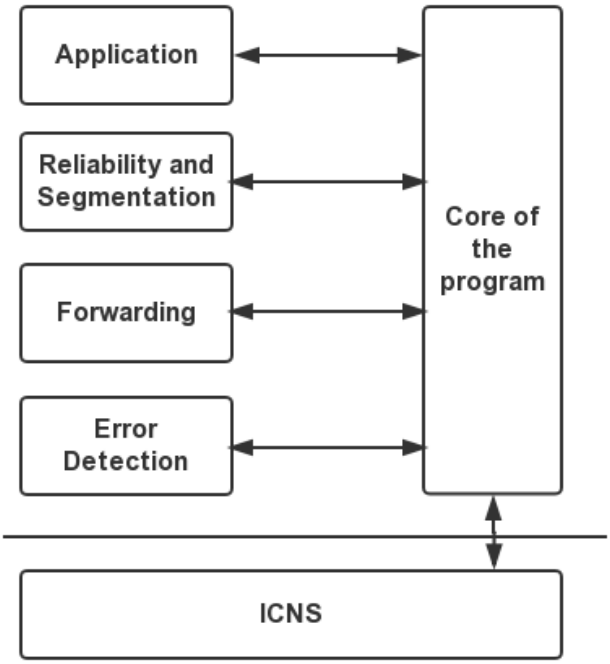

Figure 1 – Flow Chart

Reflection

By separating the information needed into different layers this allows ease of documentation, future expansion and modification to the protocol and a clear diversion of task in the groups. For each packet, a dedicated number of bytes which will contain information for each layer will be refer to as headers.

2.2 Reliability and Segmentation layer

The purpose of this layer is to ensure packet sent from sender to the receiver is authenticated, to ensure the packet sent is not lost and possible actions to take in the case of failure.

2.2.1 Reliability functionality

For this layer, the standard is to have a unique identification in the header for each packet we sent out. The group presented two possible ideas for this. One idea was to have a packet ID which will be unique for every packet generated and the segmentation will be dealt separately with a different ID and its corresponding segmentation number. Each ID will use one byte and segmentation will use one byte using a total of three bytes in the header of the packet. This idea ensures a clear distinction between the two layers.

The other idea was to have a message ID and a segmentation number in the packet where both headers will be used as a unique identification for this packet. The message ID and segmentation number will use one byte using a total of two bytes. This idea will save up to one byte compare to the other idea but combine the two layers together.

Reflection

Initially the first idea was favoured due to the ease of programming and a clear distinction between the different functionalities (reliability and segmentation). After various discussion and had successfully programmed and tested the use of packet ID, both group have decided that the second idea is better. The reason is because it uses less byte for our headers and both group have gain confidence in bitwise manipulation programming. Therefore, the message ID and a segmentation number are used as a unique identification for each packet, thus we have the reliability and segmentation layer joined as a whole.

2.2.2 Acknowledgment and segmentation functionality

For the acknowledgment, it was initially designed to use one bit as a flag taken from the message ID's byte which will indicate where a message is a normal message or an acknowledgment message. In other words, we have 1 bit for acknowledgment flag and 7 bits for message ID.

For the segmentation, similarly for acknowledgment where a flag will indicate which packet is the last packet for a message ID. This is necessary because if a message need to be separated into more than one packet, an indication is needed for the program to know when it has received the complete message. Initially, 1 bit for end packet flag and 7 bits for segmentation was used.

Later on, with the implementation, both group have decided to use 1 byte which contains all flags and type of packet in the header instead separating into 1 bit and 7 bits for flag and field corresponding to the flag.

Reflection

The reason for this decision was because it was inefficient and difficult to manage since there were flags in different area of the header and it is not possible to extract information from the flag without manipulating the byte. By combining all flags together, it is easier to manage and to program.

2.2.3 Process of authentication (Part of application layer)

The process of authentication is decided as followed, the sender send a packet with a unique message id and segmentation number. Once the receiver received the packet, it should send the same message to the send with the acknowledgment flag (bit) set to 1.

Once the sender received the acknowledgment packet and the message ID can be reused again once all packets related to the message ID have been authenticated. In the case where the receiver doesn't receive the message and where the sender doesn't receive the acknowledgment message, the sender will resend the packet to the receiver for five times with interval greater than 500ms.

In the case where the sender had fail to receive the acknowledgment packet after resending the packet for five time, the program will determine that the message has failed to send and inform the sender.

In the case of receiving duplicate message packet, it should be treated as a normal packet however it should not replace the original content of the packet. In the case of receiving duplicate acknowledgment packet, it should ignore the duplicate acknowledgment packet.

Reflection

While the agreed authentication process covers most of the basic fail case scenario, but it cannot guarantee all packet can be authenticated by the sender. This is due to the fact that the protocol is only limited to resend the packet for five times and after all resend, it is not possible to find out whether the message have not been received by the receiver nor whether the acknowledgment message have been lost during the packet transfer or whether the host has stopped working. The reason the process is limited of resend of five time is to avoid infinite looping of resending in the case when the destination of the packet can never be reached.

Also, for the ease of programming, the group decided that acknowledgment packet contain all information as the original packet but with an acknowledgment flag set to 1. This is very wasteful in term of memory and it is more vulnerable for third party intervention compare to normal packet transfer since it is pass the same packet contain the same information again. Hopefully, for future improvement it is possible to include and address these points.

In conclusion, the reliability and segmentation layer contain 1 byte where 2 bits contain our flags and the 6 bits is unused at the moment, 1 byte message ID and a 1-byte segmentation number.

2.4 Forwarding and Routing layer

In the project, the groups are under the assumption of working in a small network where each host (node) will be assigned with a number for identification. The objective is to be able to forward a message to a specified destination. To achieve this, the standard includes a source and destination field in our packet headers and to use static routing to forward the message.

Reflection

The reason for using static routing is due to the simplicity of the design and easy to program since the network is already predetermined between the groups so there is no problem hardcoding the forwarding table into the program. The assumption of having a small network because of the limited time the group have given for this project. It would be difficult to create and test a large network given the limited time and participants.

Each host is represented with a unique number and each host will have a unique forwarding table which contain information about the closest next hop for every destination in our agreed network diagram. However, to send a packet through ICNS require a IP address and port number of the next host. Therefore, a lookup table is necessary to transform each unique number into the corresponding IP address and port number.

Under the assuming of having a small network, the group decided have use 4 bits for source number and 4 bits for destination number giving us a total of 1 byte for forwarding and routing.

Reflection

Once both group have successfully implemented and tested the static routing, the plan was to proceed to implement dynamic routing if there was time. This decision was made based on the fact that dynamic routing need a more complicated design and both group felt that at this stage it is more important to implement a simple working forwarding and routing process instead a complicated process.

2.5 Error Detection Layer

The error detection layer exists to prevent from receiving corrupted data. The checksum has a very simple structure. The process is to convert every byte (excluding the checksum byte and the padding) into integer number and sum all number together. Then mask the total sum with 255 to obtain a number between 0-255 which will be the final checksum for the packet.

For each hop, the host will check recalculate the checksum of the packet and compare the exist checksum in the packet. If the checksum is not equal, this implies the packet has been corrupted along the way. If the packet is corrupted, it will drop the packet and wait for the next packet.

In this layer, it is necessary to store the length of the data in the packet (excluding the headers). The reason is because ICNS automatically pad the packet into 100 bytes during transmission, since the checksum is sum of the data in the packet, we need to the length of the data to exclude sum generated by the padding. Therefore, it is necessary to have the information about the length of the data in this layer.

Reflection

The checksum algorithm used for this layer will provide the minimum error detection for each packet. However, this is only under the assumption that the corruption will cause change to the first 8 binary number from the right of the total sum of the packet. Otherwise it will not detect the corruption. Therefore, the current agreed algorithm for error detection is not perfect. For future improvement, if given a larger time scale for the project, the group can implement a better error detection algorithm and about to correct the corrupted packet instead of dropping the packet. If the algorithm can correct the corrupted packet, it would also be able to provide a higher level of integrity and authentication.

2.6 Other design and discussion

Other functionality that have been discussed are file transfer, dynamic routing, flooding.

2.6.1 File transfer

For file transfer, both group have decided to use the right 4 bits from the byte used to store reliability and segmentation flags. When 4 bits converted into integer, the integer will represent what type of packet it is being sent. For example, 0000 = regular message, 0001 = file transfer, 0010 = routing information. For file transfer, both group agreed that the first segment of a file transfer packet will only contain the name of the file and will only focus on text file for file transfer in the project.

2.6.2 Flooding message

There are several designs that has been discussed for flooding. The first method is to represent the flooding message with 1111 in the destination field of the forwarding layer. The flooding message will be sent to all neighbour of the host. Once the host receive a flooding message, it will be flooding to all neighbour of the received host except from the source. For any duplicate message, it will be dropped.

The other method is to use broadcast routing is used to send a message to every host in the network (flooding message). Broadcast routing is where a packet is sent from the source host to all other host in the network.

Reflection

Unfortunately, due to the lack of time both group were unable to reach a conclusion for the protocol of the flooding message and unable to implement this. For future improvement, it is possible to design an efficient algorithm for method one for flooding and avoid sending duplicate messages and eliminate the probability of broadcast storm.

2.6.3 Dynamic routing

For dynamic routing, both group decide with the distance vector approach instead of a link state approach because since distance vector is better suited due to better speed and memory than link state for a small network and the groups believe it is not necessary for every host to have complete topology of the network.

The agreed format for the routing message is <destination : weight ; destination : weight; ...>. The routing message format is a string where each character is 1 byte, so 4 byte per neighbour connection.

This calculate to a maximum of 23 neighbour connection messages in one packet which is enough to cover the possible 16 destinations for the network (since destination and source has a limit of 4 bits)

The method to update the routing information:

1. If new node, store in table immediately

2. If not new node: if cost greater than currently stored, then replace with the new connection.

3. If table changed, send new routing information to neighbours

Other conditions the groups have agreed on are:

- 20 seconds interval before resending routing information

- Message sending is asynchronous (nodes will not be synced, send routing message and start 20 second timer)

- If the neighbour does not reply in 2 routing message rounds, assume it is dead and remove from tables

- If new neighbour joins the network, and is connected to you, send routing message out immediately regarding new neighbour

Reflection

Due to time constraint, the group have not decided whether the existing design of the dynamic routing protocol is considered to be the final design. Therefore, it is not written in the protocol documentation.

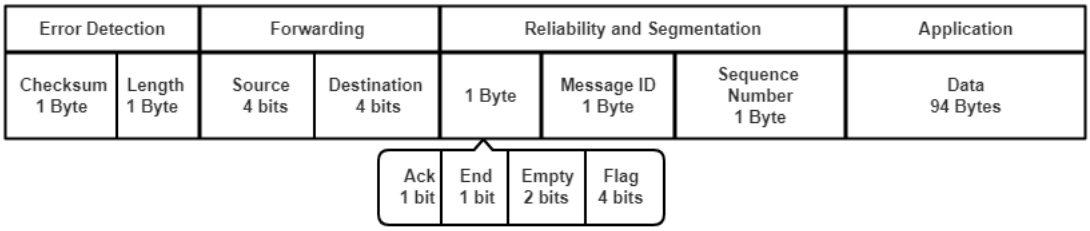

Figure 2 - protocol headers

3. Protocols Documentation

Considering the limitations of ICNS packet, several protocols of upper layers are required to be agreed with and implemented to provide more functionalities over communications between nodes. Based on the simple communication provided by the ICNS packet, the protocols should guarantee the segmentation for long message. Moreover, reliability in message transfer is necessary over network communication in reality. As for other further functions, the protocols provide the mechanisms for forwarding message and adding checksum to make sure the message integrity. In general, the protocol consists of three layers described in this section.

3.1 Reliability and Segmentation Layer Protocol

The Reliability and Segmentation Layer Protocol is designed for the reliable communications between node and node as well as divided long message into several segments. This layer is the first top layer in the stack of this task, which means it takes the raw data as input and encapsulate it.

3.1.1 Intentions

As the communications provided by ICNS packet is unreliable, the data loss rate arises when the connection state between hosts is busy or congested. This phenomenon influences the communications between hosts. Hence, a protocol that guarantee the mechanics for reliable communication is necessary.

Moreover, the maximum length of a packet, including headers, that can be sent through ICNS is 100 bytes which are not enough for long message, so mechanisms for segmentation is required in this protocol.

Segmentation needs to manipulate the original data directly and the reliability functionalities are implemented based on each single packet of a message. That is the reason that the Reliability and Segmentation Layer is the top upper layer.

Figure 3 shows the protocol layering in the perception of Reliability and Segmentation Layer Protocol.

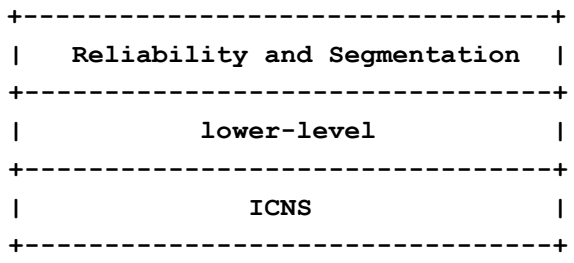

Figure 3: Protocol Layering

As showed in figure 3, the reliability and segmentation layer encapsulates the raw input message. This layer guarantees to segment the message into appropriate length, and resend the message based on the acknowledgment message of a packet being receiving or not. After that, the packet processed by this layer will be passed to lower-level layers.

3.1.2 Operation description

The reliability and segmentation layer is responsible for providing reliable communications and long message segmentation. The main idea of this layer is providing reliability and segmentation services.

Reliability

The protocol requires that the sender can resend the message if the sender does not receive an acknowledgement(ACK) within a timeout interval.

Accordingly, the sender needs to maintain a timer for recording the timeout interval for every packet. Moreover, combination of message ID and sequence number is unique identifier for a packet. Based on this identifier, the receiver can send an acknowledgment for a specific packet back, so that the sender can remove a specific message from waiting queue.

Once all the packets of a message are acknowledged by the receiver, the sender reuses the message ID. Furthermore, even if the receiver receives a duplicate packet, it should send an acknowledge back. However, if the sender receives a duplicate acknowledgment packet, it should ignore the second acknowledgement packet. The acknowledge packet contains the same thing as the packet it received, but the ACK field is required to change.

Segmentation

The sender is responsible for dividing the massage into several packets, so that the ICNS can transfer the long message. The segments from the same message share the same message ID but different sequence number. For sender side, it can recover the whole message based on sequence number when it obtains all the packets. Moreover, the protocol maintains the END field to notify the final packet of a message.

3.1.3 Function specification

Figure 4 shows the header format of Reliability and Segmentation Layer Protocol. The header includes 3 bytes where the first byte is for flags like ACK, END and TYPE which is to notify the message type. The second and third bytes are message ID and sequence number respectively.

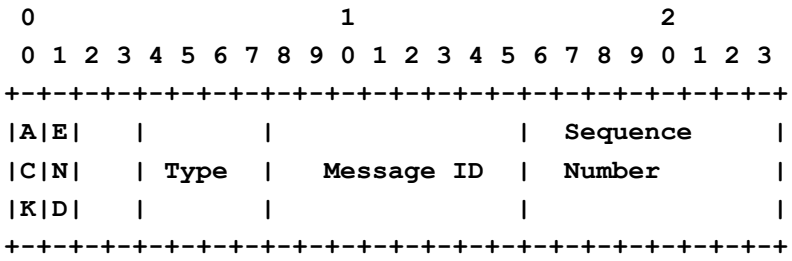

Figure 4

- Acknowledge Number: 1 bit

The ACK field takes 1 bit where 0 is represented as normal packet which is required a ACK as reply, while 1 is represented as ACK packet that acknowledge a specific packet identified by message ID and Sequence Number.

- END Number: 1 bit

The END field also takes 1 bit for notifying the final packet of a long message. 1 is represented as the final message. If a message is short and only consists of 1 packet, the field in that single packet is 1.

- Type Number: 4 bits

Type number declares the types of the message. The host may send the normal message from UI or send a file. Moreover, for dynamic routing, the nodes are required to send the routing table to neighbours. Hence, the Type number helps distinguish these different types of message.

The types are showed as follow:

0000: normal message

0001: file

0010: routing information

- Message Number: 8 bits

This field specifies the amount of message that the sender can maintenance and help receiver get the segments of the same message together. For example, if a sender sends two different messages to receiver with the same sequence number, the receiver can integrate these two messages respectively rather than drop one of them.

- Sequence Number: 8 bits

Sequence number is generated by the sender when segmenting the message into different segments. In addition, the Sequence numbers of a long message are required continuous, so the sender cannot randomly generate them.

The sequence number begin with 0 (00000000) and end up with 255 (11111111).

3.1.4 Other parameters

- TIMEOUT: 500ms

The sender would wait for a TIMEOUT interval after sending a packet out. If the sender does not receive an acknowledgement after TIMEOUT interval, it will resend the message.

- RETRY: 5 times

The maximum amount of sending the same message after not receiving acknowledgment. After that, the sender will not resend the message although it does not receive an acknowledgment.

3.1.5 Message Format

As TYPE field notifies the message type that the data field contains, an agreement on data format of each type of message is essential.

For normal message, the data field just simply includes the input message. The system can extract the message from data field and recover it based on sequence number, message number as well as END field.

As for the file massage, it is a little bit different. The first packet with sequence number as 0 contains only the file name, while the rest of sequence number starting from 1 contains the contents. The system can obtain the file name from the first packet with TYPE as 0001 as well as the sequence number as 0. Then, the file contents can be extracted from the following packets with TYPE as 0001 as well as sequence number starting from 1.

Finally, the routing massage is with TYPE as 0010. The format of data is illustrated as follow:

Destination1 : weight1; Destination2 : weight2 …

For example, if node 1 receives a message with Type setting as 0010 and the data field being ‘3:1;4:2;5:3’ from node 2, it means node 2 sends a routing message to node 1 and informs node 1 that it can get to node 3, 4 and 5 with costs as 1,2,3 respectively.

The strategy of sending routing information is basically sending the message when the table is changed immediately. If the table is not changed, one node is required to sending the table every 20 seconds to keep alive. If one node is not being heard within two routing message rounds, it is dead and removed.

When a node is received a routing message, it will update the table when a new node is added, or the cost of an existing node is smaller. The next hop is updated as the source destination in the routing information packet.

3.2 Forwarding Layer Protocol

Forwarding Layer Protocol is designed for nodes forwarding the packets if a network topology is given so that the nodes can send the message to other nodes through intermediate neighbours. This layer is built under the Reliability and Segmentation Layer, so it takes the encapsulated packet as input.

3.2.1 Intensions

ICNS packet provides the methods for sending message to a specific neighbour as well as sending to all neighbours. However, when the node intends to send a message to the nodes which do not have direct connections, ICNS cannot work in this situation. In real world, the nodes can send the message to each other if there are connections between them, so achieving the forwarding part is necessary to expand the network topology structure.

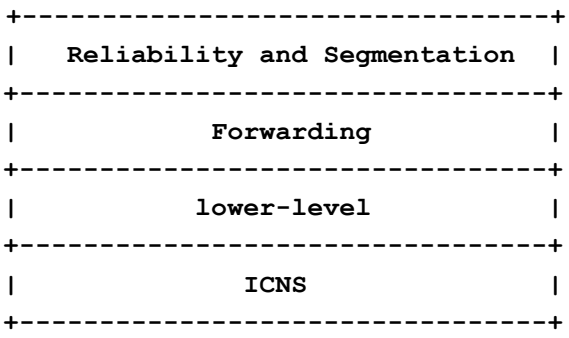

Figure 5: Protocol Layering

Figure 5 shows the protocol layering from Forwarding Layer perception. The input of Forwarding layer is the encapsulated packet from Reliability and Segmentation Layer, so the forwarding layer do not know the structure of the encapsulated packet. After adding the Forwarding header to the packet, it passes the packet to the lower-level layers

3.2.2 Operation description

The operations of the Forwarding layer are simple. This layer needs to decapsulate the packet, and extract the source and destination. If the destination is this node, the node will generate an acknowledgment packet where the source and destination are flipped. The acknowledgment packet will send back to the source address which is destination address in acknowledgment packet. If the destination is not the node, the node will forward the message to the next hop based on the Forwarding Table and Look Up Table.

The Forwarding Table matches the destination nodes with the next hops, while the Look Up Table matches the nodes to IP addresses and network address. The header of Forwarding layer consists of source and destination. They are both represented as node numbers rather than IP addresses.

As the ICNS uses the IP address and network address to identify a host, Look Up Table helps transform the node number into IP address and network number, so that the node can send the packet through ICNS. Forwarding table helps determine the next hop.

3.2.3 Function specification

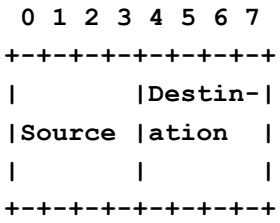

Figure 6

Figure 6 shows the header of the Forwarding Layer. The header consists of 2 bytes. The first byte is source address, while the second byte is destination address.

- Source: 4 bits

The source address is the node number which is needed to transform into IP address and network number while sending the packet through ICNS. The range of source is from 1 (0001) to 14 (1110).

- Destination: 4 bits

The destination address is also represented as node number, and the range is also from 1 (0001) to 14 (1110). The 15 (1111) in destination field is designated for flooding.

Figure 7 shows a simple network topology. Based on the topology, the Forwarding Table and Look Up Table from node 1 are structured as figure 8:

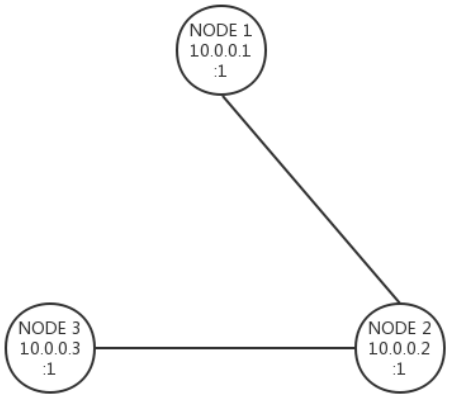

Figure 7: Network topology

Forwarding Table

Destination	Next hop
1	1
2	2
3	2
Default	2

Look Up Table

Node	IP:NET
1	10.0.0.1:1
2	10.0.0.2:1

Figure 8: Forwarding Table and Look Up Table

For static routing, the Forwarding Table of node 1 contains all the nodes in the network topology, while the Look Up Table includes only itself and the nodes which are directly connected to. Moreover, Forwarding Table requires a default gateway which can be set manually.

If Node 1 intends to send a message to Node 3, the packets contain the source as 1 and destination as 3. The Forwarding Layer firstly looks up the Forwarding Table to search the Next hop for destination 3. Then, based on Look Up Table, the layer

transforms the Next hop into IP:NET format which can be recognised by ICNS, so that the message can be transmitted.

If Node 1 receive a packet whose destination is another node, it will repeat the process like above. It searches the Forwarding Table and Look Up Table, and then, forwards it out. As for flooding, basically, the system sends the message to all the neighbours that connect to it, but the neighbours are not required to forward the message.

3.3 Checksum Layer Protocol

As network provided by ICNS is not reliable, except for drop rate, corruption rate is also needed to be taken into account. Another problem is that the ICNS add the padding automatically as the tail of packet if the packet is less than 100 bytes. The checksum layer is responsible for checking the data integrity and removing the padding of the data.

3.3.1 Intentions

The corruption rate of the real network will influence the data integrity. When the data is corrupted, the message received has been changed. In this scenario, the corrupted data is required to drop. Hence, Checksum layer is designed for checking whether the data is changed. Moreover, the Checksum layer records the length of data field, which help to remove the padding. Figure 9 shows the Checksum layer where the packet from Forwarding layer is input and required to encapsulated. The Checksum layer is the final layer designed in this task, and directly connects to the ICNS. Accordingly, the output of the Checksum layer a string that the ICNS can encapsulate.

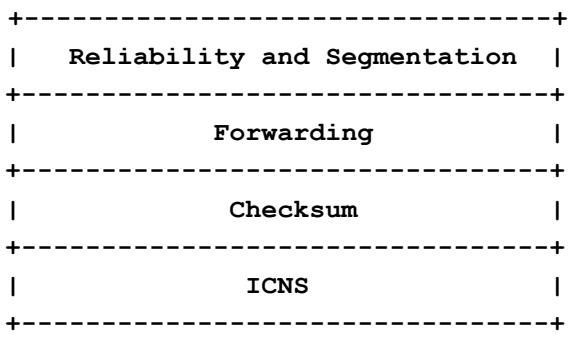

Figure 9: Protocol Layering

3.3.2 Operation description

- Checksum

Checksum calculation is simple. Each byte of raw data in binary is added up and finally the last 8 bits is used for checksum. When a sender intends to send a message, the layer will calculate the checksum of the data in packet and add into the header field. For every hop, the host will recalculate the checksum of the packet and compare this value with the existing checksum in the packet to ensure that the packet is not corrupted. If the packet is corrupted it will be dropped. Once the receiver obtains the data, it will recalculate the checksum in terms of Data Length

filed. If the result matches the value in checksum field, the data is accepted. Otherwise, the data will be dropped without acknowledgment.

- Data Length

The Data Length is the length of raw data in the packet without any headers from upper layers. This field helps the receiver calculate the checksum of packet.

3.3.3 Function specification

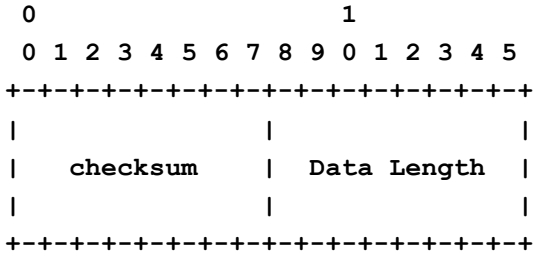

Figure 10

The Checksum Layer header is showed as figure 10. Checksum is 8 bits and the Data Length is also 8 bits.

- Checksum: 8 bits

The 8 bits of checksum is generated by the last 8 bits of the total sum of data in a packet. The checksum is simple so some drawbacks will occur. For example, if two corresponding bits flipped, the checksum is not changed, but the data has been changed.

- Data Length: 8 bits

From the discussion above, all the headers from all the designed layer is 6 bytes. Hence, 94 bytes in a single packet can be used for storing the message data, which requires at least 7 bits. As byte stream is easier to deal with, the data length is allocated 8 bits as a byte.

3.4 Other Protocols

3.4.1 File Transfer

For file transfer, each packet will have type 0001 in the reliability and segmentation layer. The type represents the type of packet it is being sent (See section 3.1.3 for details) In file transfer, first segment 0 have been decided to only contain the name of the file and every other segment will contain the content of the file.
This file transfer protocol is designed to only sent text file across the network and not account for other file types.

4. Implementation

4.1 Layering

In this section, the details of implementing each layer are discussed. In general, functions of Encapsulate and Decapsulate are two important methods in each layer, which are responsible for adding header and getting rid of header respectively.

Moreover, as the ICNS takes string data as input, the data flow in this layering structure is string type. The package named struct in Python help encode and decode the bytes. Hence, layering implementation is about generating the headers by manipulating the bits, and transforming each 8 bits into a character of 1 byte by using struct package. Then the layer attaches the header in front of the packet received from upper layer. Finally, it passes the packet to the lower layer.

4.1.1 Reliability and Segmentation layer

The methods in this layer is showed as table 1.

Table 1: methods in Reliability and Segmentation Layer

Methods	Functions
Segmentation()	Responsible for segmenting the raw data from user input.
Encapsulate()	Adding the header to the data from upper layer.
Decapsulate()	Getting rid of header in a packet from lower layer.
Generatedac()	Responsible for generating an acknowledgment when receiving a package.

- Segmentation methods takes the data as input and simply divided into several packets with maximum length of 94 bytes.

```
def segmentation(self,data):
    segnum = math.ceil(len(data)/94.0)
    listdata = []
    if segnum > 1:
        for i in range(int(segnum)):
            newdata = data[0:94]
            listdata.append(newdata)
            data = data[94:]
    else:
        listdata.append(data)
    return listdata
```

Figure 11: Segmentation method

- Encapsulate takes message type and a list of packets segmented by Segmentation method as inputs. In this method, the message ID is randomly chosen from the message ID set from 1 to 254. The chosen ID will be removed from the message ID set. When the message ID has been fully acknowledged by every packet, the message ID will be put into message ID for reusing. Then, the Encapsulate will manipulate the bits to set the ACK, END and FLAG, and using struct package in python to encode them.

```
def Encapsulate(self,format1,listdata): #
    messageID = random.choice(list(set(self.packetIDlist)-set(self.existList)))
    listlen = len(listdata)
    self.existList.append(messageID)
    self.messagesq[int(messageID)] = range(0,listlen)
    output = []
    for i in range(listlen):
        ack = 0
        if i == listlen-1:
            end = 1 << 6
        else:
            end = 0

        if format1 == "f":
            port = 1
        elif format1 == "r":
            port = 2
        else:
            port = 0  # normal
        field = ack|end|port
        header = struct.pack('!BBB',field, messageID, i)
        newData = header + listdata[i]
        output.append(newData)
    return messageID,output
```

Figure 12: Encapsulate method

- Decapsulate method takes the packet from lower layer as input and return the Reliability and Segmentation Layer hearders and data through unpack method provided by struct package.
- Generatedac method will keep the message ID and sequence number of the received packet. It simply changes the ACK flag into 1 and then return the new packet. In figure 13, '|128' means setting the ACK as 1.

```
def generatedac(self,f, messid,seq, data):
    flag = f | 128
    header = struct.pack('!BBB',flag, messid ,seq)
    newData = header + data
    return newData
```

Figure 13: Genneratedac method

4.1.2 Forwarding layer

The methods in Forwarding layer are showed as Table 2:

Table 2: methods in Forwarding layer

Methods	Functions
Encapsulatelist()	Responsible for adding the Forwarding layer header to the packets of the senders.
Encapsulate()	Encapsulating the acknowledgment packet.
Generatedest()	Generate the destination address from user input.
Getdest()	Unpack the destination address from the received packet.
Checkdest()	Check the destination address of received packet.
Decapsulate()	Extract the forwarding layer header fields.
Gennextdigest()	Responsible for generating the next hop IP and Network when sending or forwarding.

- Encapsulatelist and Encapsulate methods are basically the same. The former one is for adding header to every packet from a message, so it takes a list of packets as input. The second method is responsible for encapsulate the acknowledgment packet, so its input is only a single packet which is needed to be acknowledged.

```
def Encapsulatelist(self,dest,data):
    output = []
    for item in data:
        sour = self.source << 4
        header = sour | dest
        newdata = struct.pack('!B', header) + item
        output.append(newdata)
    return output
```

Figure 14: Encapsulatelist method

- Genneratedest method takes the user input from UI as input. If the user types '/destination address ', for example '/3 ' ,as beginning, the method will extract the destination address, like '3', as the destination. Otherwise, the Generatedest method will use the default destination address of the UI.
- Getdest method is for extracting the destination address from a received packet.
- Decapsulate method is used for extracting all the header fields from a received packet, so that the program can check each field.
- Checkdest is responsible for checking the destination address extracted by Getdest or Decapsulate method. If the destination is the receiver, the packet will be sent to the upper layer. Otherwise, the packet will be forwarded.
- Gennextdigest method helps to return the digest value corresponding to a specific host with IP. The digest is generated. The send function in ICNS can send the packet out with the digest and network number being input.

```
def getnextdigest(self,dest):
    nexthop = self.ForwardingTable[dest]
    tup = self.LookupTable[nexthop]
    hexdegest = self.iptohex[tup]
    return hexdegest
```

Figure 15: Getnextdigest method

4.1.3 Checksum layer

The methods in Checksum layer are showed as follow:

Methods	Functions
Add_checksumlist()	Responsible for calculating the checksum of a list of packets generated from a massage.
Add_checksum()	Calculating the checksum for an acknowledgment packet being sent.
Getlength()	Get the data length from the header field.
Check_checksum()	Designed for checking the checksum of a new packet.

- Add_checksumlist method helps calculate the checksum for each packet segmented from a message and add the Checksum layer header to the front.

While the add_checksum method is for calculating the checksum and adding it to the acknowledgment packet.

```
def add_checksumlist(self, packets):
    output = []
    for packet in packets:
        checksum = 0
        toalsum = 0
        for x in packet[4:]:
            tup = struct.unpack('!B', x)
            checksum += int(tup[0])
        totalsum = checksum
        checksum = checksum & 255
        leng = len(packet)-4
        header = struct.pack('!BB', checksum,leng)
        new_packet = header + packet
        output.append(new_packet)
    return output, totalsum
```

Figure 16: Add_checksumlist method

- Check_checksum method is basically a reverse calculation of add_checksum. Based on the length in the length field of checksum layer, the check_checksum method can calculate the checksum, and then compare it with the checksum in the header. If they are the same, it passes the packet to upper layer, otherwise, the packet is dropped.

Reflection

Some methods in the class can be merged together, so that the structure of the coding is clearer.

4.1.4 Application layer

In this program there are 5 threads including the main threads which divide the task of the host. The main thread is used for the ui of the chat system, the first thread is used for preparing headers of the packet and sending packets, the second thread is used for processing the received packets, the third thread is used for checking authentication, timeout and resending packets when needed and the fourth thread is used for outputting messages into the ui chat windows. Details explanation of the threads will be in the following sections.

4.1.4.1 Main Thread

The main thread is the central thread which manages the chat system, storing the necessary information needed for the program and putting data into queue for other threads to process. Firstly, the main thread stores information of the forwarding tables of all host available in the group, lookup table and the hexdigest of all neighbours (which is needed for sending packets in the icns layer).

Next, the main thread has four queues, a send queue, a receive queue, a time queue and an output queue. When the user enters the chat system and enter an message, the message is grabbed and put into the send queue as string in order to prepare a packet for that message to be sent through.

To start the program, it is necessary to specify which host to start. The specified host is grabbed by the program and it will choose the right forwarding table for the host. The main thread has a menu with three options,

1. Enter the Chat room,
2. Show neighbours in networks
3. End Program

The first option enables the chat system, which allows the terminal to send message to different host. The second option shows the user the details about the neighbours of the host in the predefined network. The third option is to end the program.

When the user enables the chat system, it will provide a list of neighbours available and to the ask the user to set a default neighbour to communicate to. Once the user has selected, the terminal to change into the chat window format allowing the user to enter message and read message sent by other users.

Once the chat window is enabled, the program will continue in a while loop. Within the while loop, the program will continue to wait for a keyboard input from the chat window and incoming packets from ICNS layer. When the program receives a keyboard input, it will convert the input as a string and it will put into the send queue for the first thread to be processed. When the program receives a incoming packet, it will put into a receive queue for the second thread to be processed. Using different threads allows us to divide each task and process them concurrently. It has become possible to send packets, receive packets, counting timeout, output to screen and receive keyboard concurrently and eliminate the issue of waiting for one process (e.g sending) to finish before moving on to the next task.

4.1.4.2 First Thread

The first thread solely deals with creating new packets and sending them to its destination. It is designed to continuously take an item (in order word string or keyboard message) from the send queue. For each item, first it will check for its destination by using the function generatedest() from the forwarding class (details about the function can be find in above section). The function will return the destination, type of message and the data (message content) which will be used later for encapsulation. Next, using the segmentation() in the reliability class, the data will be separated in a list where each item is set to have less than 94 bytes.

Each item will be sent as a different packet but with the same message ID to the destination. Using the reliability class, forwarding class and checksum class, the list of separated data will be encapsulated into the agreed protocol format. (Details of the fields added during each encapsulation can be found in the above sections)

The encapsulation function of the reliability class takes two inputs. The first input is the type of message which was obtained from the generatedest() before and the second input is the list of separated data. This encapsulation will return a message ID used for the encapsulation and list of packets encapsulated with the reliability layer.

The encapsulation function of the forwarding takes two inputs. First input is the destination of the packets obtained which was again obtained from generatedest() and the send input is list of packets encapsulated with the reliability layer. This encapsulation will return a list of packets encapsulated with reliability and forwarding layer. Lastly, the list of packets will be encapsulated with checksum class using the function add_checksumlist() and return with the value of the checksum calculated and list of complete packets ready for sending.

In order to send the packets to their destination, it is necessary to work out the next hop to our destination using the forwarding table and transfer the destination into a suitable format which can be read by the ICNS layer. The getnextdigest() function in the forwarding class will return next hop into a format readable by the icns layer.

Finally, using send function from the ICNS library, all packets are sent by looping through the list of packets.

Lastly, looping through the list packets, the thread will convert each packet into object using the packettime class. This class obtain message ID, segment number, data of packet, destination of the packet, time send and number of time the packet has been resent. Each object is than put into the timeout queue for the third thread to be processed.

4.1.4.3 Second Thread

The second thread deals with all incoming packets by continuously taking packets from the received queue through the main thread. Firstly, once the thread received a packet, it will first run check_checksum() function from the checksum class to check is the packet is corrupted. (Detail of check_checksum() function can be found in section above). If the packet is corrupted, it will discard the packet and will put an error message into the output queue where the fourth thread will output the message onto the chat window. If the packet is not corrupted, the thread will proceed to check decapsulate a part of forwarding layer of the packet to obtain the destination of the packet.

Using checkdest() function from the forwarding layer, it will check if the destination of the packet is reached (equal to the host number). If the destination is not reached, the thread use getnextdigest() function to obtain the next hop to the destination and send the original packet to its next hop through ICNS layer.

If the destination of the packet is reached, the thread will proceed to complete decapsulation of the packet obtaining the following information: length, source, destination, flags, message ID, sequent number and data.

Next, the thread checks if the packet is an authentication packet, the thread will put an message into the output queue telling the user that it has received an authentication packet and remove the an entry from the messagesq dictionary in the reliability class using the messageid and sequence number.

When a packet is encapsulated with the reliability class, it will create a new entry in the messagesq dictionary where the message ID is the key and the value will be a

list of sequence number used for this message ID. When the program received an authentication packet, it will remove the sequence number of the message ID stored in the dictionary.

If the packet is not an authentication packet, it will then generate authentication packet to be sent back to the source. If the packet is of type 0 (in order word a normal message), the thread will store the data into dictionary named buffer with the key as the message ID. The buffer dictionary has key as the message ID and each key will have a list of list where each item in the list will have the following format: [[sequence number, data, length]], except the first item. The first item will have the following format: [-1, flag for end packet obtained, flag for output to screen]. The first item is used as an indicator the two flags.

If the end packet is obtained flag = 1, flag for output to screen = 0 and if the item in the list matches the sequence number of the end packet. The thread will concatenate all data from the item and put the complete message into the output queue for it to be outputted onto the screen.

4.1.4.4 Third Thread

This thread strictly deals with the time out of every packet sent in this program. The thread will continuously get item from the timeout queue and check if the packet needed to be sent. First, the thread will check if the item (packet) has exceeded the time specified in the protocol for resending. If not, the item will be put back into the queue. Otherwise, the thread continues to check for conditions for the packet to be resend. The thread has two boolean value, resend and ackbool initially set as true.

Next, the thread will check if the packet has received an authentication from the receiver by checking if its entry in the messagesq dictionary in the reliability class exists. If the entry has existed, the ackbool value will be set as false. This implies the program has not received the authentication for this packet therefore the packet will be resent. If the ackbool value is set as true, this implies the packet has received the authentication and set the resend value to false, so the packet will not be resent, and it is dropped from the memory.

If the resend value is true and the number of time the packet has been resent is less than 5, the packet will be resent, and the count value of that item will increment by 1. If the count value greater than 5, it will output to the chat window it has encounter an error and it is unsure whether the message has been sent.

Lastly, if the timeout queue is empty, the thread will proceed to clean out excess key (memory) from the messagesq dictionary. The thread will check if there are any keys in the messagesq dictionary that has no entries. If there exist keys that has no entries, this implies that all the packets under that message id has been authenticated and it will remove the key from the dictionary.

4.1.4.5 Fourth thread

The final thread is used purely for outputting to the chat window. The process is to take item from the output queue and output this item as a string to the chat window. The reason for having a separate thread for output is because if different threads output to the screen at the same time, it will cause an error to the program and the content will not display properly. However, by using a thread and a queue, it has become possible to output to the screen without collision due the to the queue and without effecting other process due to the process being independent by this thread.

4.1.5 Summary of the flow of the program.

In summary, the main thread takes keyboard input and receive incoming packets and put them into the send queue and receive queue. The first thread will take every item in the send queue, encapsulate the item into packets and send it to the next hop nearest to the destination. The second thread will take every item in the receive queue and perform various tasks such as calculating the correct checksum, checking authentication, sending authentication packet, storing and putting complete message into output queue and forwarding packets to their destinations. The third threads will take every item in the timeout queue where it will check various condition for resending. Lastly, the fourth thread will strictly take item from output queue and output them to the chat windows.

Reflection

The use of threads and queue allows handling of multiple sending, receiving, outputting and resending concurrently which makes the program fast and efficient. However, this in return made the program very complicated and very hard to debug and test. It is possible to create this program without the use of threads and it might be suitable to do so since the project is working with a small network. Nonetheless, the use of threads allows the possibility of expanding the network while keeping the program fast and efficient.

4.1.6 List of functionalities of the program

In summary, the list of functionalities of the program implemented are:

- Able to show all immediate neighbours of the host
- Open chat window and select default neighbour to send message to.
- Sending messages which includes
    - Getting keyboard input from chat window
    - Segment message into sizeable packets
    - Encapsulates packet with reliability and segmentation, forwarding layer and error detection layer.
    - Send packets through icns layer
    - Forwarding packets which involve
        - Using static routing which lead to creating forwarding table for each host and lookup table
- Receive incoming packets which involves
    - Checking whether the packet is corrupted
    - Checking if it is an acknowledgment packet
    - Create and send acknowledgement packets

    - Decapsulate packet to extract information
    - Storing incomplete message and printing complete messages
- Timeout algorithm which involves (authentication and timeout)
    - Checking if the sent packet exceeds wait time for resend
    - Checking if the program has received an acknowledgment for the sent packet
    - Check whether the sent packet have been resent five times
    - Resend the packet

Reflection

The functionalities which haven't been implemented are file transfer, dynamic routing and flooding message due to insufficient time. However, the class had discussed and made decisions on the design of these functionalities which can be found in the design section.

5. Testing

In this section, the general functionalities of the programme are tested to illustrate the results achieved in this module. Two main functionalities are sending and receiving correctly both for short message (single packet) and long message (multiple packets). The section is organised with showing the initialisation of chatting room first, then the message transmission.

5.1 Initialisation

Once the programme is activated on command line, the first option is entering the chat system, which enables the terminal to send message to different hosts. The main menu that will be showed on the terminal is showed as follow.

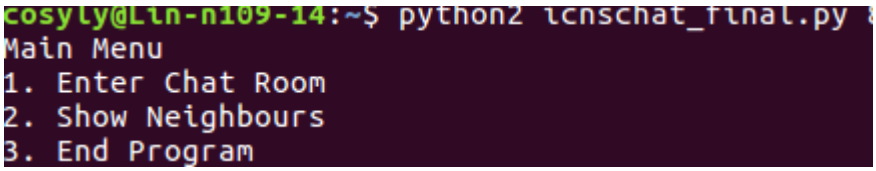

Figure 17: Main menu

The second option shows the details of neighbours in the network. From the figure below, the IP address, network number and hexadecimal digest of IP address are showed.

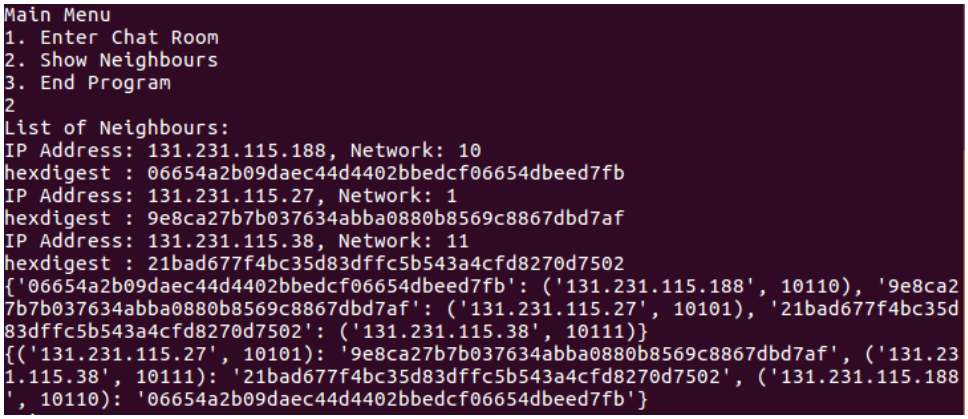

Figure 18: Show neighbours

The third option is to exit the program.

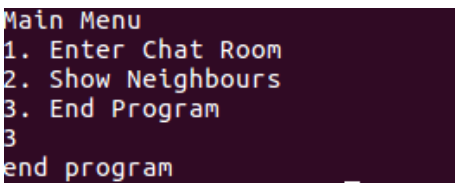

Figure 19: End program

5.2 Testing the application layer

Once the 2nd option is selected and a default host that is communicating to is chosen, the user interface of ICNS is open. Then, user can write messages in the bottom box then it will be sent across the network

```
cosyly@Lin-n109-14:~$ python2 icnschat_final.py 8
Main Menu
1. Enter Chat Room
2. Show Neighbours
3. End Program
1
Avaliable hosts :  [1, 10, 11]
Select default host: 11
```

Figure 20: Choose default neighbour

Figure 21: Chat room

5.3 Testing protocol stack

5.3.1 Sending short message

After opening the chat room GUI, the communication between two alive nodes can be performed. if node 8 sends a message to node 10, it needs to type a ‘/10’ in the front of main content, as it represents the destination (receiver). The following example is used '/10 hello 10 I am 8'. After sending it, it will show the details in the middle box. The results are showed as follow. They show the process of encapsulating a message into a packet that is received by ICNS.

If the sender received an acknowledgment packet and the packet is not corrupted, sender will be removed the packet from the waiting list and will not resend the packet.

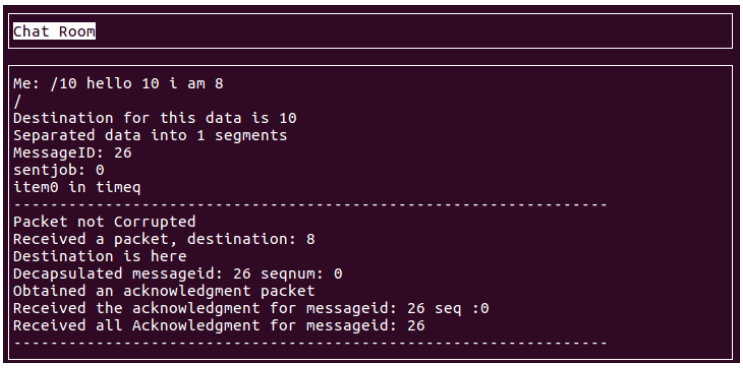

Figure 22: Sender side

The following figure shows the results of receiver which will check the checksum first. If it is not corrupted, it checks the destination whether the packet is for itself. If the destination is here, the receiver checks the message ID and sequence number, then print it out on the screen.

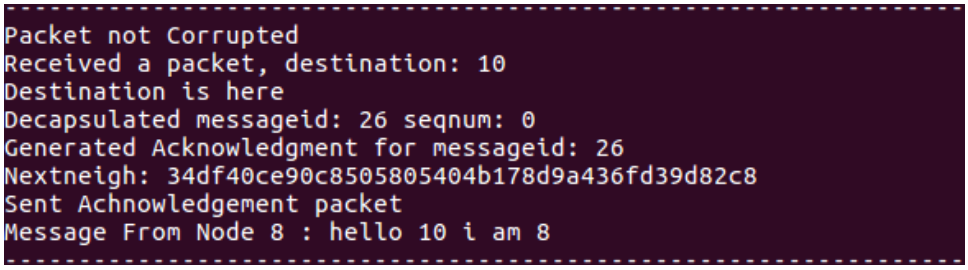

Figure 23: Receiver side

If node 8 is the intermediate node between node 10 and node 11, when node 10 sends message to node 11, node 8 will forward it. From the figure below, it shows the packet destination of the message and passing packet to next hop.

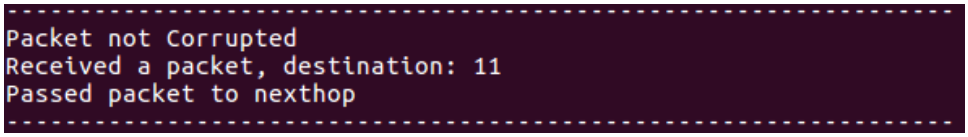

Figure 24

5.3.2 Sending long massage

When sending a long message, the Reliability and Segmentation layer will segment the message into segments with 94 bytes for data and 6 bytes for header. The figures below show the result of sending and receiving a long message.

After sending it, when the sender receive all the acknowledgments for every packet generated from a long message, it will print ‘Received all acknowledgments for messageid: ’. Moreover, the figure bellow shows within the time interval, the sender did not receive the acknowledgements and resend the data.

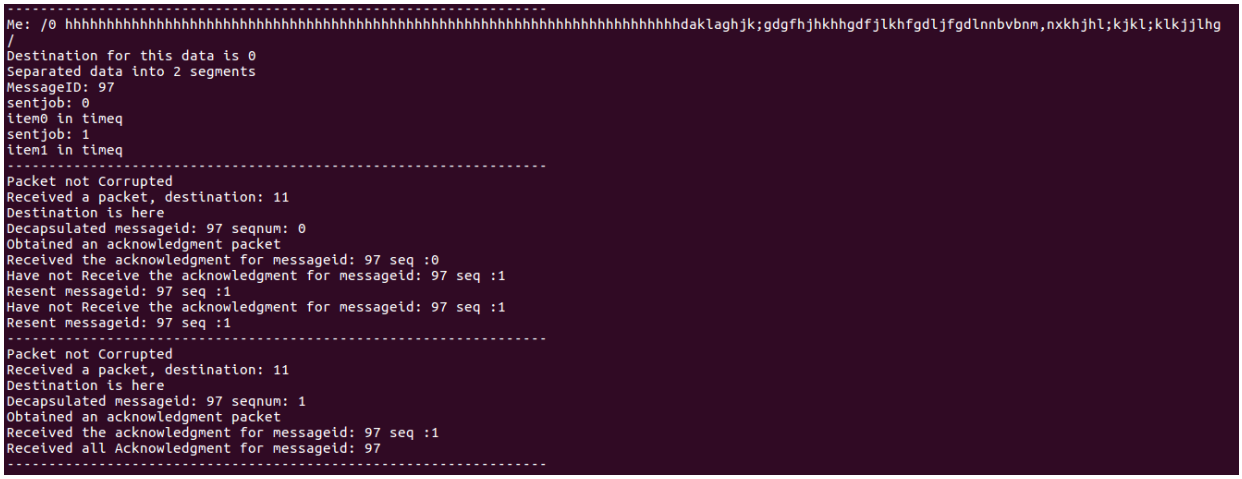

Figure 25: Sender side for long message.

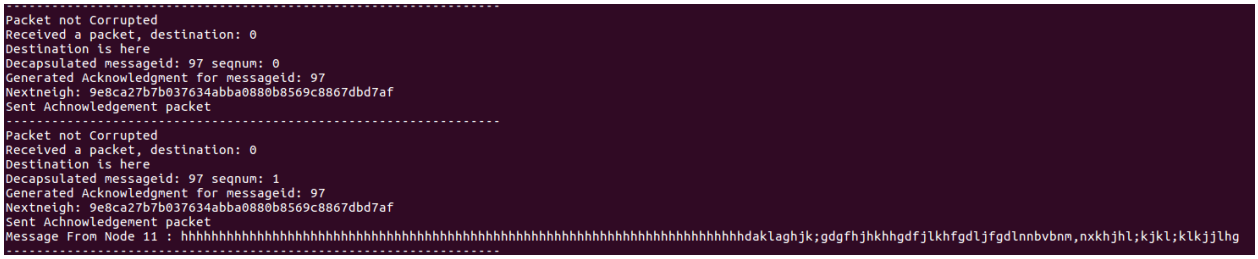

Figure 26: Receiver side for long message.

6. Conclusion

In this task, a simple communication system is achieved with reliable massage transmission, forwarding, checksum and timeout functionalities. The protocol stack agreed on the discussion sessions contains three layers: Reliability and Segmentation Layer, Forwarding layer and Error detection layer. With these three layers, the system guarantees the reliable communication point to point both for short and long message. Moreover, the system can forward the packets to next hop when the destination addresses are other nodes. Furthermore, the system uses simple error detection using checksum and some parameters to avoid the corruption and get rid of paddings.

From the test section, it shows it works well within a group which all nodes are implemented with the same code. It also works well with inter-group with different implementation. Hence, most of functionalities agreed in the lecture have been implemented.

As the time limitation, although the header considers the flags for file transfer and dynamic routing, the system is just prepared for normal message. Moreover, some details of the layering are still required to be discussed further, such as, the format of forwarding table with the costs of each link. Furthermore, some methods can be replaced by other advanced algorithms or strategies. For example, the checksum implemented for now is simple and it may not work for some circumstance, so it is required more time for discussing about the method and implementing more complicated one.

Future improvement for this project would be to consider better algorithm for checksum and to invest more time on the security of the protocol such as encryption of the message being sent.

Overall, the system is working now both within single group and inter-group. Most of functionalities agreed in the lectures have been achieved. Based on the topology in the laboratory and the node numbers allocated to each group in the lectures, two versions of the protocol stack can communicate with each other. While it is unfortunate that some of the designs has not been implemented but having fully functional chat system across groups with different implementation shows that this project has been a success.

7. Self-Reflection

7.1 Reflection Page – Simon Yan Lung Yip

In this project, I believe that I am one of the member that has always initiated and lead during our discussion of the protocol layers. In our individual group, my contribution in the coding are:

- The main menu interface
- Introducing multi-threading,
- The sending process using various function from various layer classes to complete the packet,
- The receiving process, using various function to decapsulate packet using the various layer classes, checking for corrupted data, checking and sending for acknowledgement packet, forwarding packets, storing and outputting completed message when all packets from message ID have been received,
- The design of time out and resend algorithm and its implementation,
- Various testing of the program to ensure all coding works within the group.

My contribution to the report is the design section and some of the implementation section. As a whole, I believe the project went very well.

Future improvement and reflection from this module would be to consider advantages and disadvantages properly and realistic for the protocol and for the project, to further research about existing protocol and implement useful element for the project, to make proper design plan for the implementation of the program, to improve leadership skills, to be able to make clear concise decision during the discussion. This is important point to mention because I personally believed that while we have a lot of idea, many discussions were repeated back and forth because of the lack of decision being made. However, I want to also take in the consideration that some students were being unresponsive and uncooperative which is one of the factor for stall discussion.

While there might be some people who did not participate in all the discussion of the design, I personally believe that the amount of the work and the quality exceed the time we have given and limitation of participants for this project (Both discussion and group work). I personally want to thank our lecturer Ian Napier for the support he had given us during this project. Without his support, I believe our discussion will not go (and end) as smooth and positive without him.

7.2 Reflection Page – Yang Zhou

In lectures and lab sessions we discussed and built a new protocol stack which includes Reliability and Segmentation Layer, Forwarding Layer and Checksum Layer.
In the lecture section, I participated the discussion and expressed my ideas:

- I wrote the Reliability and Segmentation Layer documentation and uploaded to the discussion forum.

In our group, I am responsible for：

- the code of building the protocol stack which are represented as classes in Python. The classes provide the interfaces for every thread to use, so that a message can be encapsulated and decapsulated.
- debugging for the programme with Simon. In this process, we looked into the details of the results from the protocol stack, and check the formats of input and output in every thread are correct.
- Testing our system with another group based on the topology in laboratory board with every group having 4 nodes, and it works with inter-group.

As for the repot：

- I wrote the section 3 Protocols Documentation and part of section 4 Implementation about layering.

After this module, I improved my programming skills with python. In addition, A new protocol stack designed by ourselves boots our confidence and deepen the understanding of the network communication.

7.3 Reflection Page – Jie Su

In this project, my focus is on learning the composition of a good network protocols and writing the report. My contribution to the report is the testing section and introduction section. I tested my team's project and gave feedback to the team members for discussion and correction to make sure our project works well. My team members are excellent and really help me a lot. And I really appreciate the guidance of lecturer Ian Napier on this module.

Although I was not able to participate in the discussion of design at first, it was because I didn't know enough about how to make a network protocol. However, in later lectures, I recorded the content and results of the group discussion, then went online to search some relevant information and understood the principles of hierarchical design of network protocols and the functions of the packet headers. So I can make some diagrams and apply them to the report.

To be honest, I am not good at programming, and I would rather play a more active role in the development of the protocol. In coding, I contributed a little bit of checksum part, which was applied to our program through the adjustment of my team members. So during the Christmas holidays, I worked on python. Actually, I've been learning python since the beginning of the semester. It allows me to follow my team's progress in the last week and complete the reports and project with my team members.

In general, our project went well, and the expected ideas were achieved well. Although file transfer and dynamic routing are not completed due to time constraints. Future improvements may require the addition of dynamic routing to make forwarding more efficient and intelligent, and work on security, such as encryption, to make our projects more formal.

It was a module that taught me a lot. Because we were completely free to design the structure and head of the package, we learned in practice and constantly correcting errors.