Transport Layer Protocols

Project Report
11/25/2013
Instructor: Dr. Yao Liang
Student: Rubin Xiao, Zhihao Cao, Cong Qi
IUPUI CSCI 436
Principles of Computer Networking Fall 2013
Course Project
Cong Qi: 0002924538 congqi@iupui.edu
Zhihao Cao: 0002921224 caozh@iupui.edu
Rubin Xiao: 0002706451 rxiao@iupui.edu

ABSTRACTION

UDP sliding window protocol is an application layer protocol applied to cross-board communication in modern communication system.It uses the sliding window technology to ensure data packets transmitted in right sequence,avoid packet duplication and loss.This paper discusses the UDP sliding window protocol and its implementation.

Introduction

Project Environment

Seattle is a free platform which can support computer networking research. By acquiring computer networking nodes on Seattle Clearinghouse, users have node specification choices on WAN, LAN, and NAT. All Seattle resources have little impact on End-System's security because all the Seattle programs are running inside the Sandboxes. In addition, Seattle codes are written in REPY programming language, which is a subset of PYTHON programming language.

Project Purpose

Be familiar with the programming under UDP protocol, Stop-and-Wait protocol, and Go-Back-N protocol
To practice technical report skills
To develop a Go-Back-N protocol based on Sliding Window algorithm on Seattle platform

Description

Through the study of the networking principle, we have learned that the network is made up of several layers, and control by hundreds of protocols. In Application layer, we have DHCP, DNS, FTP, HTTP, IMAP, POP and so on. In Transport layer, we have TCP, UDP, DCCP, SCTP, RSVP and so on. In Internet layer, we have IP, OSPF, ICMP, ECN and so on. In Link layer, we have ARP/inARP, NDP, L2TP, PPP, MAC and so on.

In this project, we are going to write a reliable UDP protocol (Go-Back-N) on the Transport layer that can send multiple packets in a pipelined manner and keep track of each packet until it has been correctly acknowledged(Sliding Window). To accomplish this goal, we firstly need to establish a reliable data transfer protocol on top of the given UDP protocol, which is known as Stop-and-Wait protocol. Then, we need to extend the Stop-and-Wait protocol with the sliding window algorithm. The purpose of this algorithm is to enable the sender to use the limited network bandwidth more efficiently. This is achieved by sending more than one packets in a reasonable RTT before waiting for an expected acknowledgement.

The objects in our protocol are two end-systems: Sender and Receiver. The protocol requires the Receiver node be ready to accept incoming packets firstly and then allow the Sender node to send a file which can be segmented by user provided arbitrary packet size. In the Application layer of the running code, it will send each segment to the function slidingWindowSendPacket(). Then, the segment goes into the Transport layer and be controlled by our Sliding Window protocol. This protocol will first check if the current forwarding window is full.

If the window is not full:

The protocol will make the segment into a packet with header in the front and then send it to the Receiver node. Meanwhile, the protocol will trigger a timer to wait the acknowledgement of the packet. If it receives the acknowledgement and checks it is correct acknowledgement, which should be as current base index, the window base will be increased by 1. That means the current forwarding window has slide to the right and new packet will be accepted and the old base packet will be dropped. If it does not receives any acknowledgement or receives an incorrect acknowledgement, the protocol will act normal until timeout. The timeout function will sent all segments from the buffer (sent segments with no acknowledgement) and reset the timer. If the protocol still does not receive any acknowledgement, it will redo this procedure. Until it receives an acknowledgement, the window base will be increased by 1 and repeats above correct acknowledgment process.

If the window is full:

The protocol will not send any new packets. It only waits the incoming acknowledgments. If the acknowledgement index is equal to the current base index, sliding window action will perform.

In the Receiver node, it will receive the packet from the Sender node. The received packet will be put into a handler function. The handler function will firstly extract the header part (which is recognized as Packet Sequence Number) from the packet and compare it with Receiver's Expected Sequence Number.

If the Packet Sequence Number is Equal to the Expected Sequence Number, the protocol will write down the data field of the packet into its local file, and then send this Packet Sequence Number back to the Sender node as an acknowledgement.

If the Packet Sequence Number is Less than the Expected Sequence Number, the protocol will record the highest expected Sequence Number, send this Packet Sequence Number back to the Sender node as an acknowledgement and let the current expected Sequence Number to be the Packet Sequence Number.

If the Packet Sequence Number is Bigger than the Expected Sequence Number, the protocol will only send its previous Expected Sequence Number back to the Sender node.

Project Difficulties

Programming Language

In this project, we write the protocol based on the repy programming language. We need to master the basic programming skills of Python. Beside, we should notice the syntax differences between the python and repy, since in the real code, the repy language has different uses of properties and syntax.

Concept

In addition to what we have known about how the UDP protocol works, we have to extend the functionality of it to accomplish the reliable sending. Besides, we need to add the sliding window algorithm to achieve our goal which is to implement the Go-Back-N protocol. In addition, by understanding the FSM of Go-Back-N protocol on the Ross's textbook, we have not realized a practical solution when an Acknowledgement is lost.

Algorithm

Finite State Machine

Sender FSM

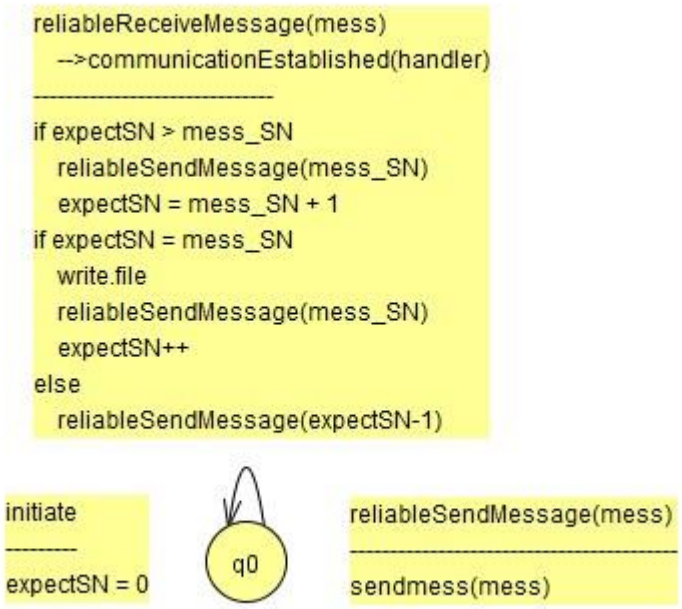
```
slidingWindowSendPacket(payload)
-------------------------
if  windowAvailable is True:
   if( nextSequenceNumber < windowBase + userWindowSize)
      packetStructure = nextSequenceNumber, payload
      reliableSendPacket(packetStructure)
      currentWindow.append(packetStructure)
      nextSequenceNumber++
      settimer
      return True
   else
      return False
else
   return False
```

```
Initiate
---------
windowBase = 0
nextSequenceNumber = 0
windowAvailable = True
```

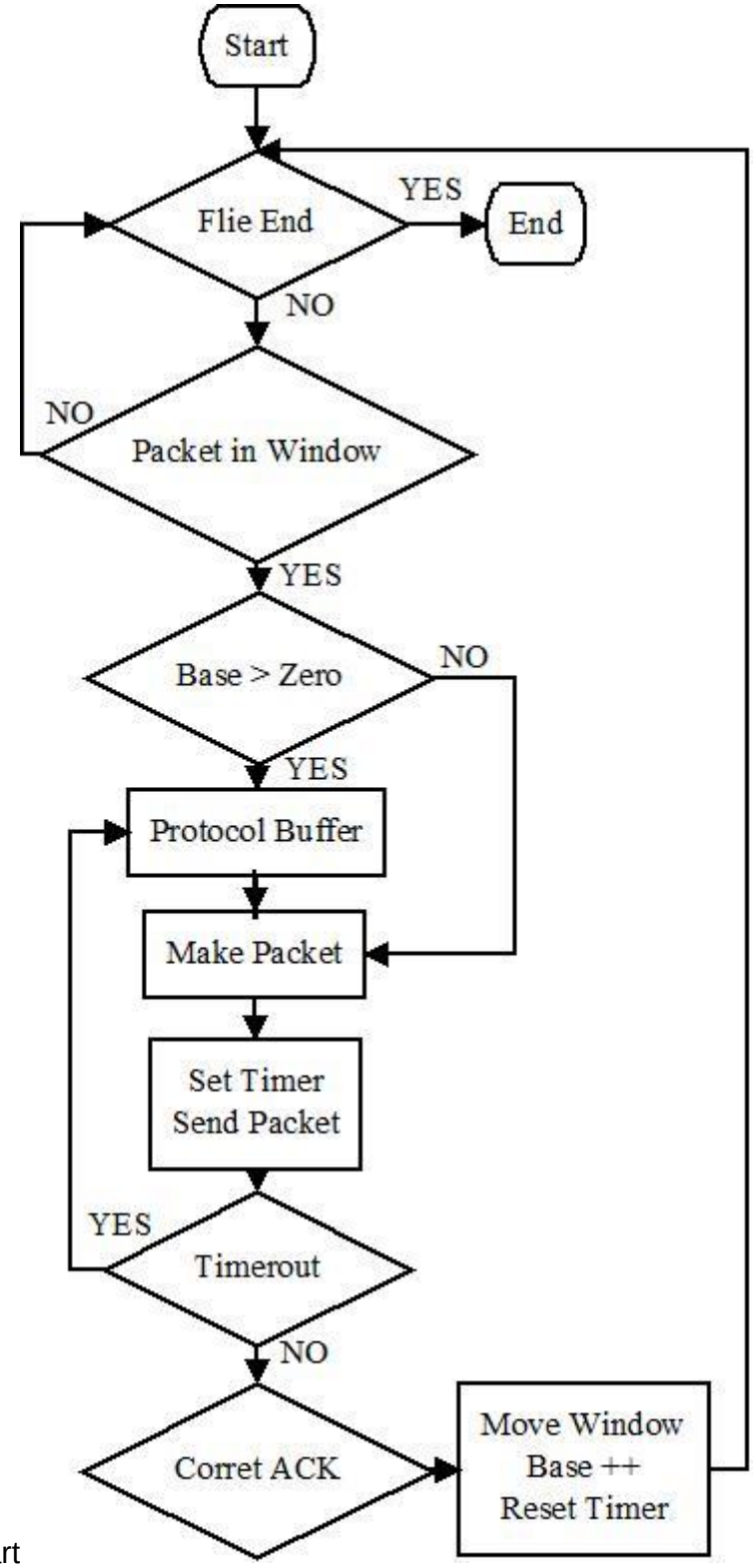

```
timeout
-------------------
start_timer
for item in currentWindow
   reliableSendPacket(item)
```

```
recvmess(mess)
  -->communicationEstablished(handler)
-------------------
if (ack == windowBase)
   mycontext['currentWindow'].remove(mycontext['currentWindow'][0])
   windowBase++
   if(windowBase == nextSequenceNumber)
      canceltimer()
   else
      resetTimer()
```

Receiver FSM

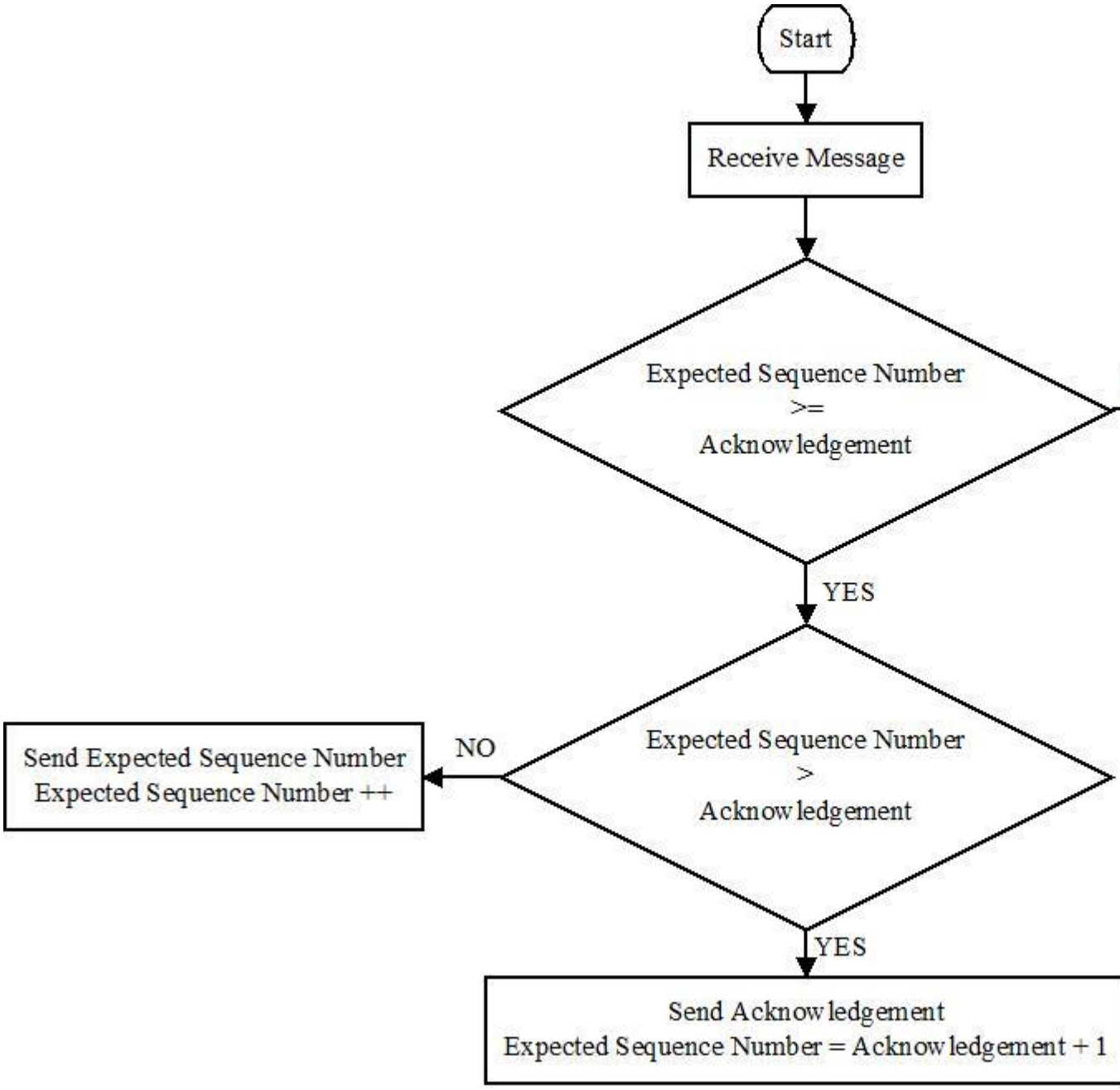


Flow Chart



Sender Flow Chart

Receiver Flow Chart



Work Distribution

All team members have participated all four individual long-period coding process in a group working environment. The group working environment was set up as following.

Date and Time	Location

11/12/2013 14:45 to 11/12/2013 18:00	IUPUI Library Room What?
11/19/2013 14:45 to 11/19/2013 18:00	IUPUI Library Room What?
11/24/2013 14:00 to 11/25/2013 1:00	Cong Qi's Home
11/25/2013 16:00 to 11/25/2013 22:00	Cong Qi's Home

Individual workload can be found in the following table.

RESULT

By considering the conceptual implementation of Go-Back-N protocol, we have realized that there are basically three scenarios should be set up in order to testing our algorithm. In the following, we will try to summarize those three events. First of all, we recognize one possible scenarios is Packet Lost through the data transfer period. Secondly, the scenario could be summarized as sender resends un-received packets in an arbitrary period. Since the first two scenarios could be tested on the same approach,we create a testing code named as ATestSenderCode.repy. In this testing code, sender would not send packet 3 as the usual approach as others. Be specific, packet 3 would not be sent to receiver at the first time, but would be sent to receiver when it's time out. Snippet source code would be the following.

```
if int(packetStructure[0]) == 3:
        mycontext['currentWindow'].append(packetStructure)
    else:
        reliableSendPacket(ServerIP, ServerPort, packetStructure, ClientIP,
ClientPort)
        mycontext['currentWindow'].append(packetStructure)
```

From sender's perspective（Figure: Sender Testing Code - Sender Side - Screen Shot）, we can realize that after sending out packet 2, sender sends packet 4. When time out reaches, sender resend packet 2 through packet 6. From receiver's perspective （Figure: Sender Testing Code - Receiver Side - Screen Shot）, receiver is expecting packet 3 when it's receiving packet 2 through packet 6. The third scenario could be recognized as Acknowledgement Lost. We have created a testing code by using the following code snippet(please refer souce code named as ATestReceiverCode.repy).

```
#testing purpose || do not send ACK3 || means ACK losts
    if (mess_seq == 3) and (int(mycontext['expectSN']) == 3) and
(mycontext['flagNoACK3'] == True):
    mycontext['expectSN'] = mycontext['expectSN'] + 1
    mycontext['flagNoACK3'] = False
```

```
    else:
    if str(mycontext['expectSN']) >= str(mess_seq):
        #Write the received file to "file1"
        if "server_time_out" in mycontext:
        canceltimer(mycontext["server_time_out"])
        mycontext["outfile"].write(mess)
        if str(mycontext['expectSN']) > str(mess_seq):
        reliableSendMessage(remoteIP, remoteport, str(mess_seq), getmyip(),
int(callargs[0]))
        mycontext['expectSN'] = mess_seq + 1
        else:
        reliableSendMessage(remoteIP, remoteport, str(mycontext['expectSN']),
getmyip(), int(callargs[0]))
        mycontext['expectSN'] = mycontext['expectSN'] + 1
        print "    expected SN", str(mycontext['expectSN'])
    else:
        print "    else: expected SN", str(mycontext['expectSN'])
        reliableSendMessage(remoteIP, remoteport, str(mycontext['expectSN'] -
1), getmyip(), int(callargs[0]))
```

By observation, from the perspective of receiver, when it is receiving packet 3, it does not perform an acknowledge action (please refer Figure: Receiver Testing Code - Receiver Side - Screen Shot). From sender side, sender does not receive ACK 3 until timeout event occur.

Figure: Sender Testing Code - Sender Side - Screen Shot

```
Sender make packet as:  1,DEF
sender try to send 1,DEF
sender received ACK: '1' from 200.0.206.169:63117
sender expected ACK: 1
move window 2
Sender make packet as:  2,GHI
sender try to send 2,GHI
sender received ACK: '2' from 200.0.206.169:63117
sender expected ACK: 2
move window 3
Sender make packet as:  3,JKL
Sender make packet as:  4,MNO
sender try to send 4,MNO
sender received ACK: '2' from 200.0.206.169:63117
sender expected ACK: 3
Sender make packet as:  5,PQR
sender try to send 5,PQR
sender received ACK: '2' from 200.0.206.169:63117
sender expected ACK: 3
Sender make packet as:  6,STU
sender try to send 6,STU
sender received ACK: '2' from 200.0.206.169:63117
sender expected ACK: 3
time out event
sender try to send 3,JKL
sender try to send 4,MNO
sender try to send 5,PQR
sender try to send 6,STU
```

Figure: Sender Testing Code - Receiver Side - Screen Shot

```
Received message: '1,DEF' from 200.0.206.203:63117
mess_seq:1, mess_body:DEF
     expected SN 2
Received message: '2,GHI' from 200.0.206.203:63117
mess_seq:2, mess_body:GHI
     expected SN 3
Received message: '4,MNO' from 200.0.206.203:63117
mess_seq:4, mess_body:MNO
     else: expected SN 3
Received message: '5,PQR' from 200.0.206.203:63117
mess_seq:5, mess_body:PQR
     else: expected SN 3
Received message: '6,STU' from 200.0.206.203:63117
mess_seq:6, mess_body:STU
     else: expected SN 3
Received message: '3,JKL' from 200.0.206.203:63117
mess_seq:3, mess_body:JKL
     expected SN 4
Received message: '4,MNO' from 200.0.206.203:63117
mess_seq:4, mess_body:MNO
     expected SN 5
Received message: '5,PQR' from 200.0.206.203:63117
mess_seq:5, mess_body:PQR
     expected SN 6
Received message: '6,STU' from 200.0.206.203:63117
mess_seq:6, mess_body:STU
     expected SN 7
```

Figure: Receiver Testing Code - Sender Side - Screen Shot

```
Sender make packet as:  0,ABC
sender try to send 0,ABC
sender received ACK: '0' from 200.0.206.169:63117
sender expected ACK: 0
move window 1
Sender make packet as:  1,DEF
sender try to send 1,DEF
sender received ACK: '1' from 200.0.206.169:63117
sender expected ACK: 1
move window 2
Sender make packet as:  2,GHI
sender try to send 2,GHI
Sender make packet as:  3,JKL
sender try to send 3,JKL
sender received ACK: '2' from 200.0.206.169:63117
sender expected ACK: 2
move window 3
Sender make packet as:  4,MNO
sender try to send 4,MNO
sender received ACK: '4' from 200.0.206.169:63117
sender expected ACK: 3
time out event
sender try to send 3,JKL
sender try to send 4,MNO
sender received ACK: '3' from 200.0.206.169:63117
sender expected ACK: 3
move window 4
sender received ACK: '4' from 200.0.206.169:63117
sender expected ACK: 4
move window 5
```

Figure: Receiver Testing Code - Receiver Side - Screen Shot

```
mess_seq:0, mess_body:ABC
     expected SN 1
Received message: '1,DEF' from 200.0.206.203:63117
mess_seq:1, mess_body:DEF
     expected SN 2
Received message: '2,GHI' from 200.0.206.203:63117
mess_seq:2, mess_body:GHI
     expected SN 3
Received message: '3,JKL' from 200.0.206.203:63117
mess_seq:3, mess_body:JKL
Received message: '4,MNO' from 200.0.206.203:63117
mess_seq:4, mess_body:MNO
     expected SN 5
Received message: '3,JKL' from 200.0.206.203:63117
mess_seq:3, mess_body:JKL
     expected SN 4
Received message: '4,MNO' from 200.0.206.203:63117
mess_seq:4, mess_body:MNO
     expected SN 5
```

CONCLUSION

In this paper, we present a reliable and efficient way to transmit data over unreliable channels that can lose, reorder and duplicate messages by using Sliding Window technique, multiple packets can be transmit at a pipeline manner and each packet will be correctly acknowledged, which is quite efficient in terms of response time compare to stop-and-wait. It also allows the user to control the packet size and windows size to suit the application.

Theoretical Performance Compare

Problems Regarding to Our Implementation

no LOCK. maybe this is reason why command console print statement is mess up. what we are thinking is concurrency issue, which SEATTLE did a bad job. Besides, all running threads are interchanged on charging IO channel.

Testing code and protocol code are not separated. Hard to do a regular source code maintenance.

In the receiver end, there is a tradeback regarding to Acknowledgement Lost. When the receiver receive the retransmit packets, it will receive the duplicated packets. We recognize this as a trade back not a problem since receiver knows the ordered packets by introducing another variable to keep counting the packet with the largest Sequence Number at any instance time. Then, we write the packet with the largest Sequence Number out to file structure.

Future Work and Expectation

A writefile function is needed to handle in-order packets received from sender and write data into an output file after server receive all packets.

Extend current protocol to a more efficient algorithm such as “Selective Repeat”, which will only resend unacked packet.

REFERENCE

Stop-and-Wait Protocol

https://seattle.poly.edu/wiki/EducationalAssignments/StopAndWait

Sliding Window Protocol

https://seattle.poly.edu/wiki/EducationalAssignments/SlidingWindow

Kurose | Ross “Computer Networking: A Top-Down Approach (6th Edition)