TCP-like Reliable UDP Data Transfer Protocol Using Python

Amani Alinazi
CEN 536 – Final Project, 2nd Semester 1439-1440
Supervisor: Mohammed Alenazi, Ph.D
College of Computer and Information Sciences
Department of Computer Engineering
King Saud University, Riyadh, Saudi Arabia
Amanialinazi@gmail.com
438203191@student.ksu.edu.sa

***Abstract*—This report shows the design, implementation and evaluation of a protocol designed using python to enhance the reliability of UDP channel. After the design and implementation, the protocol will be evaluated using five different scenarios and three file types. The first file is PDF with size 10mb, the second one is compressed file that contains multiple types of files including video, images and pdfs; and it's of size 60 mb. The last file type is text and it'll be used to examine the security implemented in the protocol. The scenarios are set using tc in Linux environment and network emulated using host-only adapter. The five scenarios are: ideal, dropping, delaying, out of order and noisy channel which is mixture of the previous four scenarios. Finally, the results will be discussed in the evaluation section.**

Keywords—: UDP, Sockets, Congestion Control, Go Back N, Reliable Data Transfer, Python, Security.

I. INTRODUCTION

In this project, we need to implement a simple TCP-like reliable data transfer protocol using Python in the application layer using UDP as the unreliable channel. The new protocol must have several services. The first one is reliable data transfer; all segments must be delivered in order. Then a pipelined reliable data transfer which requires that multiple segments are sent at once such as Go Back-N protocol.

After that, congestion control will be implemented so sender do not overwhelm network buffers. And finally, transfer data securely by encrypting segments at the sender and decrypting them at the receiver. In this report I am going to show the design, implementation and evaluation of this protocol. In the evaluation section the protocol will be tested against multiple scenarios with

dropped, out-of-order, and delayed packets. These scenarios will be emulated using Linux Traffic Control application a.k.a tc and we will see how the protocol would react to them.

II. DESIGN

Starting to implement the enhancements of the protocol with the ordering service. This service can be implemented at the server side. By using sequence numbers, if it receives an unordered packet it will drop it immediately and send an ack of the last received packet to inform the sender that there is a packet loss. Then for a pipelined data transfer, the client set a window for sending multiple packets once.

And for each packet it will set a timer to indicate packet loss if the ack doesn't arrive. For the congestion control I implemented an algorithm similar to TCP-Tahoe. The sending window is controlled by the congestion window; cwnd, initially it will be set to on packet = MSS.

Client will send multiple segments and stop the ability of sending until it receives an ack, then it will slide-continue sending. The cwnd will be doubled every round until it reaches the threshold size which is 16*MSS (slow start) then it will be incremented by one to avoid the congestion.

The cwnd is controlled by the recipient of acks, if acks arrived in order it'll be doubled for every ack until threshold. If the ack received indicates loss of packets (doesn't equal the expected one), there will be a counter to count the duplicate ack. The window will be sent again if the counter reaches three without waiting for a time out and the cwnd is set to 1, otherwise the timer will be triggered, and window will be sent again.

To add secure data transmission several python libraries can be used such as Cryptography, simple-crypt and crypto. In this protocol I will use the simple crypt to encrypt the msg from the (file.read) and then decrypt it in the server side before writing it to the file. However, the protocol at server side is very simple and the code will be shown in the implementation part of this report, for the client side the sequence of instructions I wrote to design the code are shown below.

Algorithm 1: TCP-like Reliable UDP Data Transfer Protocol Using Python

Result: Files : pdf,compressed, txt

```
1: create UDP socket: SOCK DGRAM;
2: get ip,port and file to send from command line;
3: send through socket: open File, read msg in binary , attach header, start timer ;
4: get ack;
5: if ack==seq then
   | first packet recived , connection established, cancel timer, cwnd = cwnd*2;
else
   | send packet again, cwnd = 1;
end
cansend = True;
6: while msg and cansend: do
   | sender window size = cwnd;
   | if cwnd¿=threshld then
   |    | cwnd = cwnd+1;
   | else
   |    | cwnd = cwnd*2;
   | end
   | For( packetno in senderwindow) ;
   | while  msg and packetno¡cwnd and cansend: do
   |    | seq+1;
   |    | msg read ¿ pack ¿ encrypt ¿ send ¿ apped to window¿start timer ¿ cansend=False;
   |    | recive acks: ;
   |    | if ack==seq then
   |    |    | cansend=True, cancel timer;
   |    |    | if cwnd¿=threshld then
   |    |    |    | cwnd = cwnd+1;
   |    |    | else
   |    |    |    | cwnd = cwnd*2;
   |    |    | end
   |    | else
   |    |    | countdubacks+=1;
   |    |    | if countdubacks==3 then
   |    |    |    | cwnd=1 , retrans+=1, send window again;
   |    |    |    | recive ack;
   |    |    |    | if ack=seq then
   |    |    |    |    | break;
   |    |    |    |    | cancel timer, cwnd = cwnd+1;
   |    |    |    | else
   |    |    |    |    | continue
   |    |    |    | end
   |    |    | else
   |    |    |    | wait for timout to be triggered, cwnd=1
   |    |    | end
   |    | end
   | end
end
```

III. IMPLEMENTATION

1. Server.py:

```
import socket
import time
from struct import *
from simplecrypt import encrypt, decrypt
from cryptography import *
from Crypto.Cipher import AES
from Crypto.Cipher import *
import base64

import sys #Import sys module to handle command line arguments
PORT = (int(sys.argv[1]))  #Get port number from command line and
convert it to integer
s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM) #Create
UDP socket
s.bind(('',PORT)) #Bind the port number to the socket
filerecieved=open("output/RecievedUDP.txt","wb") #In output
folder, Open file to write recived data
packed_data, addr = s.recvfrom(calcsize('ii4096s')) #Recieve the first
data chunk from address
password='this is password'
data=unpack('=ii4096s',packed_data) #unpack the recived data
print data[0] , data[1]
ack1 = str(data[0]) #get the packet seq from header
seq=0
se=str(seq)
ack = pack('i',seq) #send ack to client
msg=decrypt(password,(data[2])) ####decrypt the msg using key
starting_time=time.time()
########check if the first packet in sequnce and write to the file####
if ack1 == se:
    print "packet in order ack will be sent",len(data[2]), "ack=" ,
ack1
    filerecieved.write(msg) #Write the recived data
    s.sendto(ack,(addr))
    seq+=1
else :
  print "packet dropped , out of order" , ack1 , se , seq , ack
  s.sendto(ack,(addr))
canwrite=True
#recwin=[recelement]
while True and canwrite: #Write the remainnig recieved chunks to
the output file
      packed_data, addr = s.recvfrom(calcsize('ii4096s')) #Recieve
the first data packet from address
      #packed_data= password.decrypt(ciphertext)
############extract seq from header######
      data=unpack('=ii4096s',packed_data)
      ack1 = data[0]
      #msg=data[2]
      msg=decrypt(password,(data[2])) ####decrypt the msg using
key
      ending_time=time.time()
      print "recived seq is:", ack1 ,"my seq is:", seq
      ########check if the first packet in sequnce and write to the
file####
      if ack1 == seq:
        print "packet in order ack will be sent is nu" , "ack=" ,
ack1 , len(data[2])
        print "seq nu is",seq
        ###to count the time needed to recive the file###
        print " time started reciveing:",starting_time , "time last
packet recived:",ending_time
        filerecieved.write(msg) #Write the recived data
        ack = pack('i',ack1)
        s.sendto(ack,(addr))
        seq+=1
      else :
          print "packet dropped , out of order"
          print "recived seq is:", ack1 ,"my seq is:", seq
          s.sendto(ack,(addr))
          print " Last one I recived is " , data[0]
          print " time started reciveing:",starting_time , "time last
packet recived:",ending_time

print "finish"
```

```
filerecieved.close()
s.close() #close socket
```

2. Client.py:

```
import socket
from struct import *
import threading
import time
from simplecrypt import encrypt,decrypt
from cryptography import *
import sys #import sys module to handle command line arguments
HOST = sys.argv[1] #Get the host adress from command line
PORT = (int(sys.argv[2])) #get the port number from command line
and to convert it to integer
Filetosend = (sys.argv[3]) #Getthe name of the file to be sent to the
server
addr=HOST,PORT
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#Create UDP client socket
password='this is password'

#timer=threading.Timer(1.0,mytimer)
mss=1 #maximum segment size (1024*4, one packet )######
cwnd=1*mss ####intial congestion window size#####
packetno=1 #to count packets in sender window
winelement=0 # window element to be appended
seq=0 #sequence numbers for ordering
seqwin=[seq] #sequence numbers window
rtt=0.401085138321 ###sample round trip time
retransm=0 #initial no of retransmissions
touts=0 #initial no of timeouts

##############timer##############
def mytimer():
  #sretransm=retransm
  cwnd=1 ### set the cwnd to 1 for timeout
  print ("timout , packet resending seq=", seq )
  print "cwnd is:", cwnd
  addr=HOST,PORT
  ########send the lost packet again##########
  #timeout4=threading.Timer(3,mytimer)
  #timeout3.join()
  #print " I Send you packet with seq=",seq
  print "No of Retransmissions:",retransm , "No of Timeouts:",
touts
  print "window resending",c
  for winelement in window:
      s.sendto(winelement,(addr))
      #ack, addr = s.recvfrom(calcsize('i'))
      #ack1=unpack('i',ack)
      #print "No of Retransmissions:",retransm , "No of Timeouts:",
touts
  cansend=True
  timeout1.cancel()
#touts+=1
#timeo=timeo+1
#######to print the cwnd ######
cwndwindow=[]
cwndwindow.append(cwnd)
###########Establish Connection############
originalfile=open("input/"+Filetosend,"rb") #open and read the file
msg= originalfile.read(1024*4) #read the first chunk of the file
window=[] #Window to buffer packed data
data=encrypt(password,msg) #Encrypting the msg
length=len(data)
#########Create Header and add it to the packet#######
packed_data= pack('ii4096s',seq , length ,data)
#ciphertext=encrypt('Amani', packed_data)
s.sendto(packed_data,(addr)) #send the packed data to address
print "packet 0 sent"
####Start the timer########
timeout1=threading.Timer(6,mytimer)
timeout1.start()
#######Recive ack from server########
ack, addr = s.recvfrom(calcsize('i'))
ack1=unpack('i',ack)
if ack1[0] == seq:
    print "ack recieved connection established"
    timeout1.cancel()
    cwnd=cwnd*2
    sucsess=1
else:
  s.sendto(packed_data,(addr))
  window.append(packed_data)
  print "packet resent"
  cwnd=1
  timeout1.cancel()
cwndwindow.append(cwnd)
########counters#######
packetcount=0 #to count the total no of packets sent
countdubacks=0 # to detect 3dub acks
cansend= True # to control sending proccess
threshd=16 #threshld value , 16 segments, 64 KB

###########start loop to send the file##############
while msg and cansend:
  ######## the sender window size maintained by the
cwnd#########
  senderwindow=range(0,cwnd+1)
  ####congestion contor####
  if cwnd>=threshd:
        cwnd=cwnd+1
  elif cwnd>=100:
        cwnd=100
  else:
    cwnd=cwnd*2
  cwndwindow.append(cwnd)
  packetno=0 ####### count no of packets sent every round
  print "########sender window size
is:",len(senderwindow),"and cwnd is:",cwnd+1,"#############"
  ###########start sending window#########
  #while msg and cansend:
  for packetno in senderwindow :

    while msg and packetno<cwnd and cansend:
     seq+=1
     sucsess+=1
     msg= originalfile.read(1024*4)
     #data=msg
     data=encrypt(password,msg)
     length=len(data)
     packed_data= pack('ii4096s',seq , length ,data)
     #ciphertext=encrypt(password, packed_data)
     s.sendto(packed_data,(addr))
     #timeout1=threading.Timer(5,mytimer)
     timeout1.join()
```

```
        startingtime=time.time()
        winelement=packed_data
        window.append(packed_data) ###buffer every packet sentin the
a window
        packetno+=1
        packetcount+=1
        seqwin.append(seq) ##seq number window
        cansend=False
        print "window length",len(packed_data) ,
"packetconut=",packetcount, "packet no in
window:",packetno,"sent" , "seq=",seq , "cwnd:",cwnd
        #########after sending the window wait for acks#########
        timeout1=threading.Timer(0.01,mytimer)
        timeout1.start()
        ack, addr = s.recvfrom(calcsize('i'))
        endingtime=time.time()
        ack1=unpack('i',ack)
        ###check if recived ack in sequence (Cumliative ack) or there is a
packet loss###
        print "No of Retransmissions:",retransm , "No of Timeouts:",
touts , "success", sucsess
        if ack1[0] >= seq :
            print "ack no" ,ack1[0] ,"is recieved , seq=" ,seq
            print "window length is" , len(window)
            print "cwnd=" , cwnd , "and mss=" , mss
            print "rtt is",rtt
            sucsess+=1
            #timeout1.cancel()
            if cwnd>=threshd:
               cwnd=cwnd+1
            else:
               cwnd=cwnd*2
            timeout1.cancel()
            cansend= True
            #cwndwindow.append(cwnd)
          ##############Check for three dublicate acks############
        else:
            countdubacks+=1
            #cwnd=1
            #if cwnd>=threshd:
               #cwnd=cwnd+1
            #cwndwindow.append(cwnd)
            timeout1.cancel()
            if countdubacks==3:
              cwnd=cwnd=1
              retransm+=1
              print "3 dub acks detected , packet:",ack1[0]+1, "lost,
Resend it again"
              for winelement in window:
                  print "window resending , lost packet:",ack1[0]
                  s.sendto((winelement),(addr))
              ack, addr = s.recvfrom(calcsize('i'))
              ack1=unpack('i',ack)
                  #print "No of Retransmissions:",retransm , "No of
Timeouts:", touts
              if ack1[0] == seq :
                    sucsess+=1
                    countdubacks=0
                    if cwnd>=threshd:
                      cwnd=cwnd+1
                    else:
                      cwnd=cwnd*2
                    #cwndwindow.append(cwnd)
                    timeout1.cancel()
                    cansend=True

                    break
                #timeout2=threading.Timer(3,mytimer)
            else:
                timeout1.cancel()
                #touts+=1
                #cansend=True
                cwnd=1
          else:
            #timeout1=threading.Timer(3,mytimer)
            #timeout1.start()
            touts+=1
            print "continue..."
            #cansend=True
            #continue
        #except:
          #print " what's happenn"

        #print "##### window recived , cum ack no" ,ack1[0] ,"arrived ,
my seq=" ,seq,"#####"

      cansend=True
print "CWND" , cwndwindow
print "No of Retransmissions:",retransm , "No of Timeouts:",
touts , "Success:",sucsess
print "window length",len(window) ,
"packetconut=",packetcount, "packet no in
window:",packetno,"sent" , "seq=",seq , "cwnd:",cwnd
#s.close() #close the socket
```

IV. EVALUATION

For Evaluation, I will test my script and generate scenarios on three file types with five suggested list of channels that are :Ideal Channel: no changes by the tc application, Dropping Channel: only drops packets, Delayed Channel: only delays packets, Out-of-order Delivery Channel: only delivers packets out-of-order and Noisy Channel: dropping, delayed, and out-of-order delivery channel.

I will test the protocol's reliability, react to packet loss, delay and out of ordering. Also, I will see the rate of transmission, goodput, cwnd, number of transmissions needed and timeouts. The tests will be implemented using two linux Vm's with IP's provided in figures below on interface enp0s8.

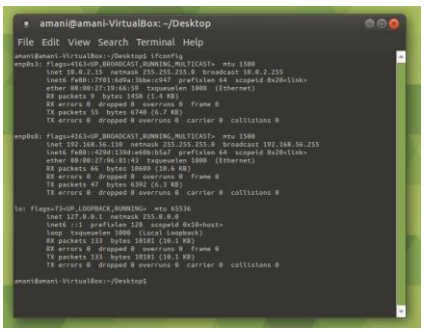


Fig1. Server's VM.

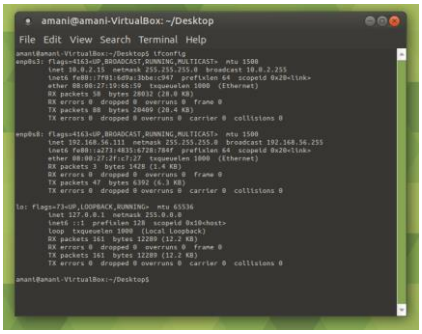


Fig2. Client's VM.

4.1 PDF File

4.1.1 Scenario I: Ideal Channel:

This scenario will be implemented to test the protocol reliability to transfer a pdf file of size around 10Mb. First, by running the server.py on receiver's VM. Then we run client.py at the sender's VM. We get the result shown in figure below. The file transfer success, but there was some noise, so it needed to retransmit tow times and one timeout triggered. We can see that the number of success is 15 which represents the number of rounds needed to transfer the file which proves that protocol send multiple segments once of sender window size = cwnd.

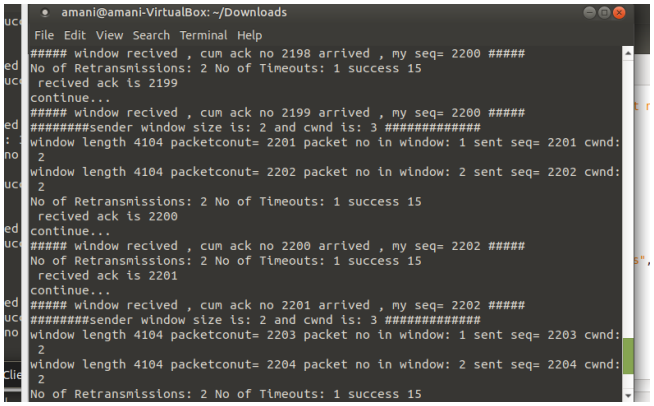


Fig3. PDF ideal channel.

I captured this figure while the file was transmitting, and we can see that multiple packets are sent once; the packet count is the sequence number of packet and packet no is the packet number in the sender window:

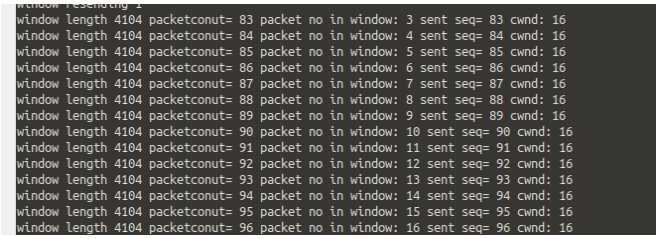
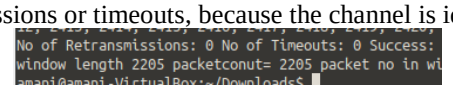
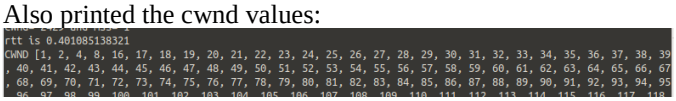


Fig4. Multiple data transfer.

I run the script again and the result was successful without any retransmissions or timeouts, because the channel is ideal:



Also printed the cwnd values:



We can see that it starts with slow start then when it reaches the threshold it will be additively increased, and no losses recorded since channel is ideal:

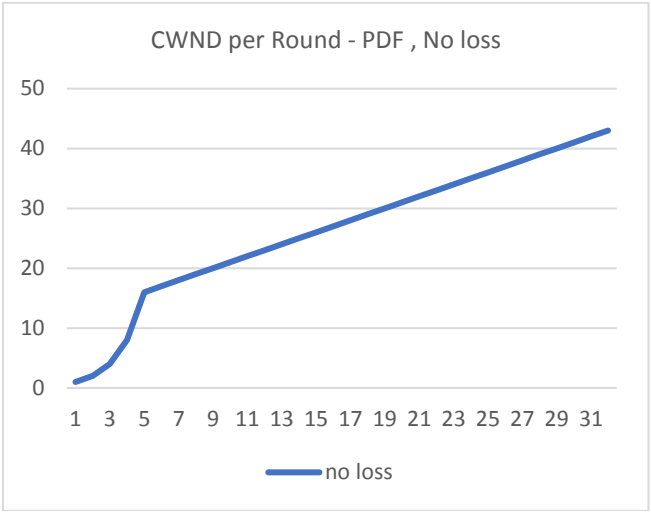
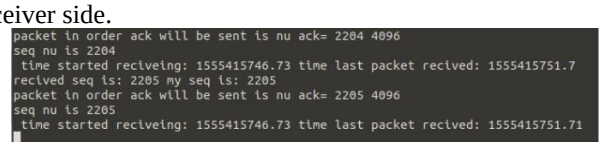


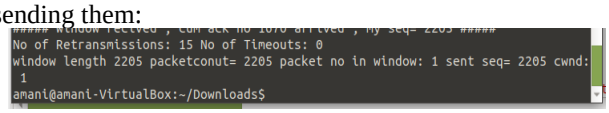
Fig5. Cwnd , ideal channel.

The time to receive the file at server is 4.98s, and the goodput here = 10/4.98= 2mb per second. We can see below the time at receiver side.

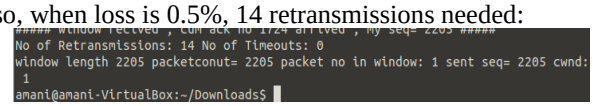


4.1.2 Scenario II: Packet loss Channel

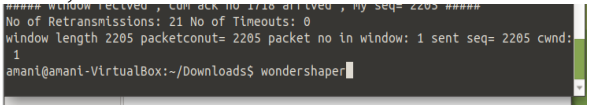
This scenario will be implemented to test the protocol reliability to transfer a pdf file of size around 10Mb with different loss rates. I tested it under 0.2, 0.3, 0.5, 1, 2, 3 loss percentages. when loss is 0.2%, 15 retransmissions needed because multiple packets will be lost, and they will be indicated by the three duplicate ack statement when receiver continues resending them:



Also, when loss is 0.5%, 14 retransmissions needed:



Here the figure below shows the needed retransmissions when loss is 1%, which are 21 retransmissions:



When loss is 2%, 24 retransmissions needed:

```
##### Window recived , cum ack no 1725 arrived , My seq= 2205 #####
No of Retransmissions: 24 No of Timeouts: 0
window length 2205 packetconut= 2205 packet no in window: 1 sent seq= 2205 cwnd:
 1
amani@amani-VirtualBox:~/Downloads$
```

When loss is 3%, 39 retransmissions needed:

```
##### Window recived , cum ack no 1692 arrived , My seq= 2205 #####
No of Retransmissions: 39 No of Timeouts: 0
window length 2205 packetconut= 2205 packet no in window: 1 sent seq= 2205 cwnd:
 1
amani@amani-VirtualBox:~/Downloads$
```

I captured time of receiving at server when loss 0.5%:

```
 time started recieving: 1555415632.76 time last packet recived: 1555415639.16
recived seq is: 2204 my seq is: 2204
packet in order ack will be sent is nu ack= 2204 4096
seq nu is 2204
 time started recieving: 1555415632.76 time last packet recived: 1555415639.16
recived seq is: 2205 my seq is: 2205
packet in order ack will be sent is nu ack= 2205 4096
seq nu is 2205
 time started recieving: 1555415632.76 time last packet recived: 1555415639.16
```

The goodput here = 10/7.16= 1.3966 mb per second, which is lower than the ideal channel because several retransmissions needed. Also, the congestion window was recorded for each scenario and as we can see its growth in the graph below. We can see that all of them started with slow start then additive increase, but the performance is minimized by the incrementing percentage of loss.

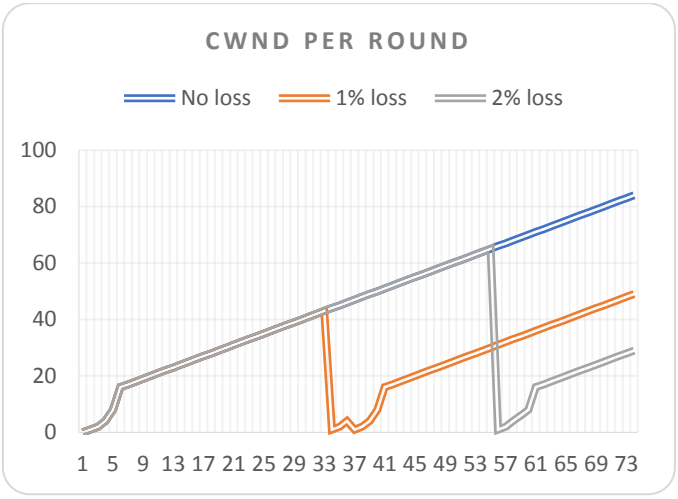


Fig6. Cwnd , dropping channelles.

4.1.3 Scenario III: delayed Channel

In this scenario I tested it for 100ms, 200ms delays. When delay is 100ms I get the result below. And we can see the number of timeouts and retransmissions is high, because I set a small timer to test how the protocol would behave.

```
##### Window recived , cum ack no 1433 arrived , My seq= 2205 #####
No of Retransmissions: 142 No of Timeouts: 961
window length 2205 packetconut= 2205 packet no in window: 1 sent seq= 2205 cwnd:
 1
amani@amani-VirtualBox:~/Downloads$
```

And the protocol also tested when delay is 200ms, getting result below and it shows same behavior as the first one. But the number of timeouts increased because the delay increased:

```
##### Window recived , cum ack no 1573 arrived , My seq= 2205 #####
No of Retransmissions: 62 No of Timeouts: 1041
window length 2205 packetconut= 2205 packet no in window: 1 sent seq= 2205 cwnd:
 1
amani@amani-VirtualBox:~/Downloads$
```

4.1.4 Scenario IIII: Out of Order Channel

Using: tc reorder we can set this channel, but the usage of it requires adding delay also:

```
[2]+  Stopped                 python UDP_Server.py 55881
amani@amani-VirtualBox:~/Downloads$ sudo tc qdisc add dev enp0s8 root netem delay 10ms
 reorder 1%
[sudo] password for amani:
amani@amani-VirtualBox:~/Downloads$
```

As we can see in the result below 2 retransmissions needed because of out of order packets at channel:

```
No of Retransmissions: 1 No of Timeouts: 1 Success
window length 2205 packetconut= 2205 packet no in
nd: 3941
```

But it took the file 340.05 s to arrive. Achieving goodput of 10/340.05 = 0.0294 mb which is low because of the delay added.

4.1.5 Scenario IIIII: Noisy Channel

I added 0.5% loss , 10ms delay and 1% reordering:

```
amani@amani-VirtualBox:~/Downloads$ sudo tc qdisc del dev enp0s8 root netem
amani@amani-VirtualBox:~/Downloads$ sudo tc qdisc add dev enp0s8 root netem delay 10ms
 reorder 1% loss 0.5%
amani@amani-VirtualBox:~/Downloads$
```

Got the following result, 3 retransmissions 1 timeout because some packets will be lost and will be retransmitted, and a timeout will be triggered because of the delay:

```
, 2119, 2134, 2141, 2157, 2172, 2181, 2197, 2209]
No of Retransmissions: 3 No of Timeouts: 1 Success: 5334
window length 2205 packetconut= 2205 packet no in window:
```

Time needed to receive file: 359.92s, goodput = 10/359.92=0.027mb. And it's low performance is caused by the noise in the channel.

```
Last one I recived is  2205
time started recieving: 1555533725.62 time last packet recived: 1555534084.92
```

4.2 Compressed File

The compressed file of size around 60mb contains different types of files: PDFs, Images and Video as shown in figure7.

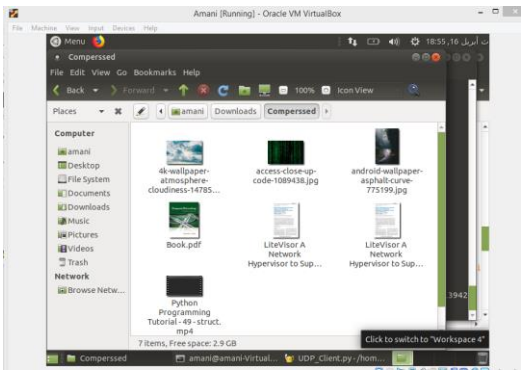
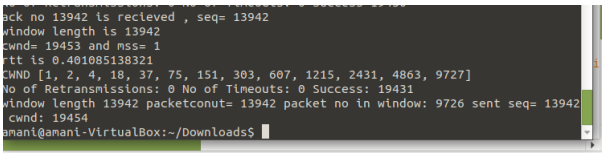


Fig7. Compressed file.

4.2.1 Scenario I: Ideal Channel:

Here in ideal channel, the file transferred with no problems. I set the congestion control to exponential increase without setting initial threshold, so the file will be transferred in small amount of time since no loss would occur. Since we can see that it's incremented by one each cumulative ack received and doubled each round.

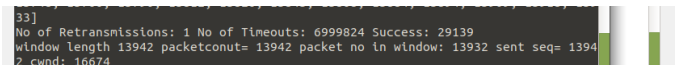


4.2.2 Scenario II: Packet loss Channel

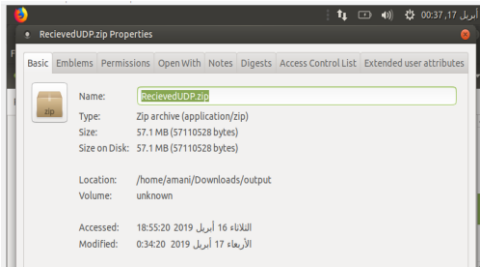
When loss is 1%, I get the result shown below. Which shows that multiple packets were lost and retransmitted and also timeouts will be triggered because of the loss percentage in channel:

4.2.3 Scenario III: delayed Channel

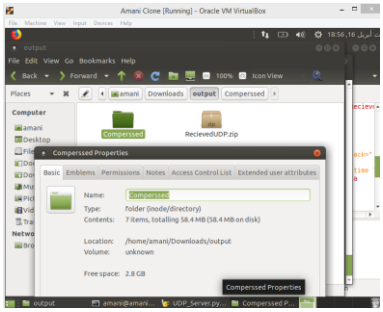
Here I tested the protocol when delay is 100ms and I set packet timeout to 0.1, which will lead for a lot of premature timeouts. The file received after 4.5 hours, since every packet will be timed out several times even when resending the timeout will be triggered because of delay & timer, so the counter for timeouts resulted a very large number! but it arrived correctly.

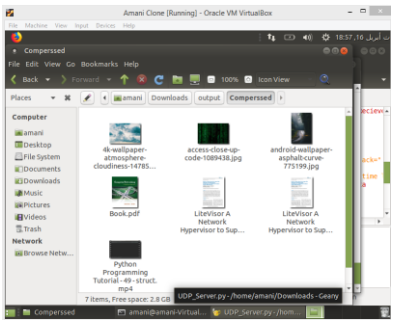


Here we can see the time file created until it last edited, which indicates the time of file creation until writing the last segment.



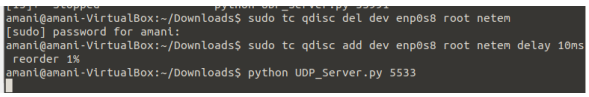
And here we can see that the file received correctly after extracting it and the files aren't corrupted:



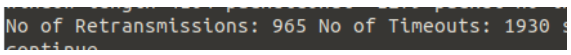


4.2.4 Scenario IIII: Out of Order Channel

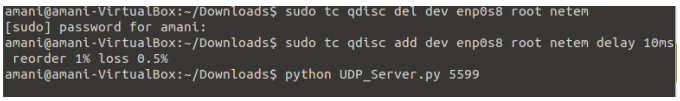
Using tc to add the channel characteristics:



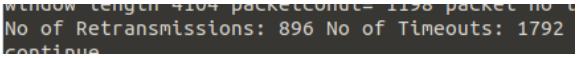
I got this result, which shows that the unordered packets needed to be retransmitted and some of them timed out because of delay and reordering channel.



4.2.5 Scenario IIIII: Noisy Channel

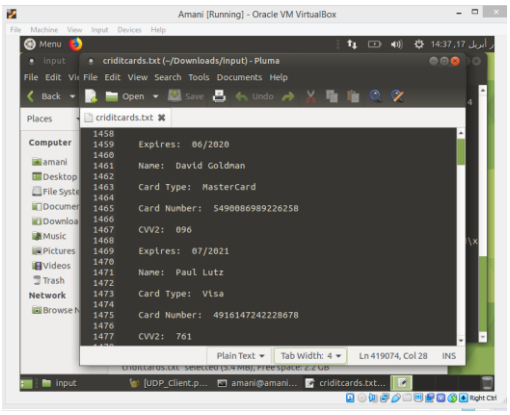


Result shows large number of retransmissions and timeouts because of noise in channel:

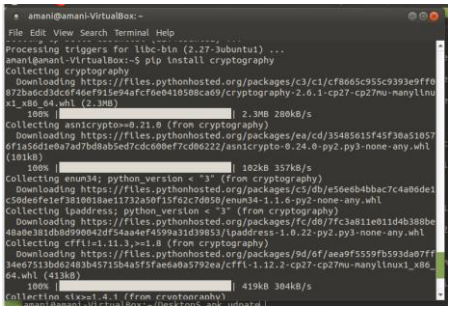


4.3 Text File

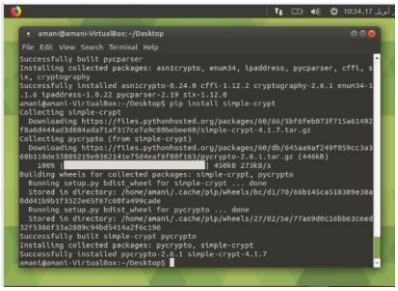
In this part I'm going to test a text file with contents below, before and after using the encryption.



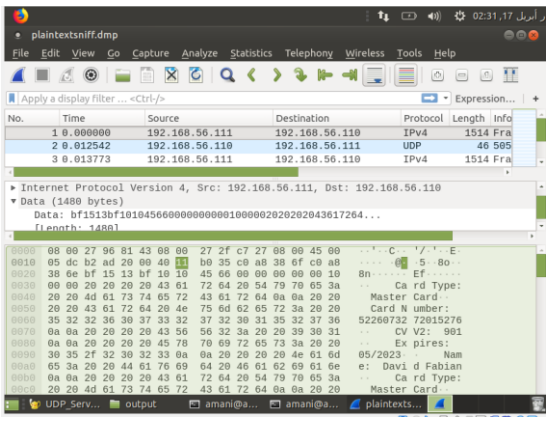
First, we need to install the libraries, installing cryptography:



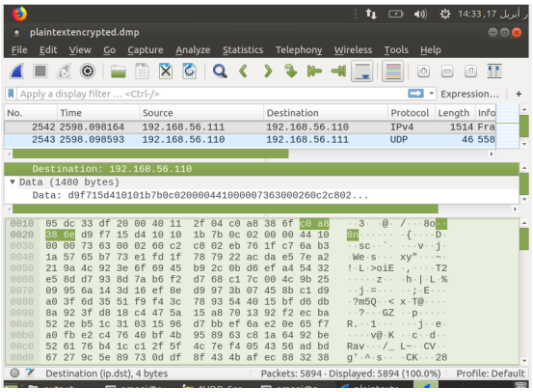
Installing simple crypt:



Capturing the data transfer before using encryption and using Wireshark we can see that the contents of plaintext can be readable, and we can clearly read the credit card numbers, names, dates etc. as shown in next figure.



After that I encrypted the data then I Captured the traffic and we can clearly see that the contents are encrypted and no longer readable which make the communication secure.



V. Conclusion and Future Work

In this report, I evaluated the performance of TCP-Like reliable UDP. As we saw in results above, that the designed protocol guarantees reliable data transfer under multiple scenarios, different data types and sizes. But it has a quite slow performance and when adding the encryption, it'll be very slow compared to TCP socket transfer. A lot of work can be implemented to it such as using the selective repeat protocol to transfer multiple data and buffer at receiver side. Also, flow control can be implemented so the sender won't overflow the receiver. In addition, another congestion control algorithm can be implemented such as Reno, New-Reno, Cubic. UDP is an unreliable channel, implementing such enhancements can provide reliability and it can be safely used if TCP isn't preferred. In Conclusion, there is no single algorithm that can completely overcome the problem to congestion and unreliable nature of the network. Every algorithm has his own advantages and disadvantages to solve the network problems of TCP, UDP protocols.

Acknowledgment

This project report has been written For CEN536, a graduate computer networks Course at King Saud University. I would like to acknowledge Dr. Mohammed Alenazi for his effort and support during this course.