Name: Sayan Acharya

Roll: 002010501009

Group: A1

Assignment: 2

Deadline: 22-26 August 2022

Submission date: 6th September

Assignment 2: Implement three data link layer protocols, Stop and Wait, Go Back N Sliding
Window and Selective Repeat Sliding Window for flow control.
Submission due: 22-26 August 2022 (in your respective lab classes)

Report submission due on: 28 August 2022
Sender, Receiver and Channel all are independent processes. There may be multiple Transmitter and
Receiver processes, but only one Channel process. The channel process introduces random delay and/or
bit error while transferring frames. Define your own frame format or you may use IEEE 802.3 Ethernet
frame format.

Hints: Some points you may consider in your design.
Following functions may be required in Sender.
Send: This function, invoked every time slot at the sender, decides if the sender should (1) do nothing,
(2) retransmit the previous data frame due to a timeout, or (3) send a new data frame. Also, you have to
consider current network time measure in time slots.
Recv_Ack: This function is invoked whenever an ACK packet is received. Need to consider network time
when the ACK was received, ack_num and timestamp are the sender's sequence number and timestamp
that were echoed in the ACK. This function must call the timeout function.
Timeout: This function should be called by ACK method to compute the most recent data packet's
round-trip time and then re-compute the value of timeout.
Following functions may be required in Receiver.
Recv: This function at the receiver is invoked upon receiving a data frame from the sender.
Send_Ack: This function is required to build the ACK and transmit.
Sliding window:
The sliding window protocols (Go-Back-N and Selective Repeat) extend the stop-and-wait protocol by

allowing the sender to have multiple frames outstanding (i.e., unacknowledged) at any given time. The
maximum number of unacknowledged frames at the sender cannot exceed its "window size". Upon
receiving a frame, the receiver sends an ACK for the frame's sequence number. The receiver then
buffers the received frames and delivers them in sequence number order to the application.
Performance metrics: Receiver Throughput (packets per time slot), RTT, bandwidth-delay product,
utilization percentage.

Code:(Stop and wait)

```
import socket
import threading
from convention import *

# socket.setdefaulttimeout(DEFAULT_TIMEOUT_MSG)


class Server:

  def __init__(self,host_name,port,file_to_send,extra_delay_error_func,server_id,window:int=1):
    self.sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    self.sock.bind((host_name,port))
    self.file_name=file_to_send
    self.error_maker=extra_delay_error_func
    self.server_id=server_id
    self.window_len=window

    self.sock.settimeout(DEFAULT_TIMEOUT_MSG)

  def start_server(self):
    self.sock.listen()
    while True:
      try:
        client,addr=self.sock.accept()
      except socket.timeout as tt:
        continue
```

```
      temp_thread=threading.Thread(target=self.handle_client,args=[client,addr])
      temp_thread.start()


  @staticmethod
  def make_16_bit_number(n):
    return bin(n)[2::].rjust(16,'0')

  @staticmethod
  def calculate_parity(s):
    val=0
    for i in s:
      val^=(ord(i)-ord('0'))
    return str(val)

  @staticmethod
  def encode(file_name):
    with open(file_name,"r") as f:
      message=''.join(f.readlines())
    encoded_message=""
    for i in message:
      encoded_message+=bin(ord(i))[2::].rjust(8,'0')
    listy=[]
    index=0
    serial_number=1
    while index<len(encoded_message):
      upto=index+MAX_LEN_CONTENT
      if upto>len(encoded_message):
        upto=len(encoded_message)


main_message=Server.make_16_bit_number(serial_number)+encoded_message[index:upto]

      listy.append(main_message+Server.calculate_parity(main_message))
      index=upto
      serial_number+=1
    listy.append(EXIT_MESSAGE)
    return (listy,serial_number-1)

  def handle_client(self,client:socket.socket,addr):
    client.settimeout(DEFAULT_TIMEOUT_MSG)
    all_messages,total_serial=Server.encode(self.file_name)
    print([len(i) for i in all_messages])
    index=0
```

```
    while index!=len(all_messages):
      message_to_send=self.error_maker(all_messages[index])
      ret=client.send(message_to_send.encode(FORMAT_MSG))

      # print(f"msg sent: {message_to_send}, sender side return code: {ret}")

      try:
        msg=client.recv(MAX_LEN_MSG).decode(FORMAT_MSG)
      except socket.timeout as tt:
        print(f"timeout before receving ack from client by {self.server_id}")
        continue
      except ConnectionAbortedError as cae:
        print(f"connection to client closed by {self.server_id}.")
        break
      index+=1
    print(f"message sending done by {self.server_id}.")
    client.close()

if __name__=="__main__":
  # sender=Server("localhost",5000,'file1.txt',ZERO_ERROR_NO_DELAY_FUNC)
  sender=Server("localhost",5000,'file1.txt',SOME_ERROR_SOME_DELAY_FUNC)
  sender.start_server()

import socket
import threading
from convention import *

class Listener:
  def __init__(self,error_delay_func,client_id):
    self.sock=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    self.error_maker=error_delay_func
    self.client_id=client_id

  def start_listener(self,host_name,port):
    addr=(host_name,port)
    temp_thread=threading.Thread(target=self.establish_connection,args=[addr])
    temp_thread.start()

  @staticmethod
  def error_detect_parity(s:str)->bool:
    val=0
    for i in s:
      val^=(ord(i)-ord('0'))
    return val!=0
```

```
  @staticmethod
  def extract_message_part(s:str)->str:
    return s[16:-1]

  @staticmethod
  def decode(s:str)->str:
    serial_number=int(s[0:16],2)
    string=Listener.extract_message_part(s)

    assert(len(string)%8==0)

    packs=len(string)//8
    message=""
    for i in range(packs):
      message+=chr(int(string[i*8:(i+1)*8],2))
    return (message,serial_number)

  def establish_connection(self,addr):
    self.sock.connect(addr)
    while True:
      msg=self.sock.recv(MAX_LEN_MSG).decode(FORMAT_MSG)
      msg=self.error_maker(msg)

      # print(f"msg len is: {len(msg)}")

      if msg==EXIT_MESSAGE:
        self.sock.close()
        return
      if Listener.error_detect_parity(msg):
        print(f"Faulty Message Received by {self.client_id}!")
        continue

      message,serial_number=Listener.decode(msg)
      print(f"INCOMING MESSAGE FOR SERIAL:{serial_number} by {self.client_id}")
      print(f"{message}")

      send_msg="ACK"
      self.sock.send(send_msg.encode(FORMAT_MSG))



if __name__=="__main__":
  listener=Listener(ZERO_ERROR_NO_DELAY_FUNC)
```

```
# listener=Listener(SOME_ERROR_SOME_DELAY_FUNC)
listener.start_listener('localhost',5000,)
```

SendingFile:

This is a message.
Hope this gets sent.

Output:

```
Faulty Message Received by 1!
timeout before receving ack from client by 1
Faulty Message Received by 1!
timeout before receving ack from client by 1
Faulty Message Received by 1!
timeout before receving ack from client by 1
Faulty Message Received by 1!
timeout before receving ack from client by 1
Faulty Message Received by 1!
timeout before receving ack from client by 1
INCOMING MESSAGE FOR SERIAL:1 by 1
This is a message.
Hope t
Faulty Message Received by 1!
timeout before receving ack from client by 1
Faulty Message Received by 1!
timeout before receving ack from client by 1
INCOMING MESSAGE FOR SERIAL:2 by 1
his gets sent.
Faulty Message Received by 1!
timeout before receving ack from client by 1
message sending done by 1.
```

Code:(Go back N)

```
import threading
from server import Server
import socket
from convention import *
import time


class go_back_n_server(Server):

  def __init__(self, host_name, port, file_to_send, extra_delay_error_func,
server_id,window_len=4):
    super().__init__(host_name, port, file_to_send, extra_delay_error_func, server_id)
    self.recv_set=set()
    self.window_len=4
    self.all_messages=[]
    self.time_sent_mappa={}

  def get_ack(self,client:socket.socket):
```

```
  #add serial numbers to the dict
  try:
    while True:
      full_ack=client.recv(MAX_LEN_MSG).decode(FORMAT_MSG)
      serial_no=int(full_ack.split()[-1])
      self.recv_set.add(serial_no)
      print(f"received ack for {serial_no}")
  except:
    print("connection stopped from client side")

def send_window(self,client:socket.socket,index):
  print(f"sending window: {index} to {index+self.window_len}")
  for i in range(index,min(index+self.window_len,len(self.all_messages))):
    client.send(self.all_messages[i].encode(FORMAT_MSG))
    # print(f"len of msg is: {len(self.all_messages[i].encode(FORMAT_MSG))}")
    time_now=time.time()
    self.time_sent_mappa[i+1]=time_now
    sleep(1)


def handle_client(self, client: socket.socket, addr):

  self.all_messages,total_serial=Server.encode(self.file_name)
  index=0
  self.all_messages=self.all_messages[:-1] # excluding the disconnect message


  #start receiver thread
  receiver_thread=threading.Thread(target=self.get_ack,args=[client])
  receiver_thread.start()

  while index!=len(self.all_messages)+self.window_len:
    prev_index=index-self.window_len
    serial_no_prev=prev_index+1

    while(serial_no_prev>=1 and serial_no_prev not in self.recv_set):
      #wait for the timeout amount to pass
      cur_time=time.time()
      # print(f"gonna check serial no: {serial_no_prev}")
      if cur_time-self.time_sent_mappa[serial_no_prev]<DEFAULT_TIMEOUT_MSG:
        sleep(1)
        continue

      self.send_window(client,prev_index)
```

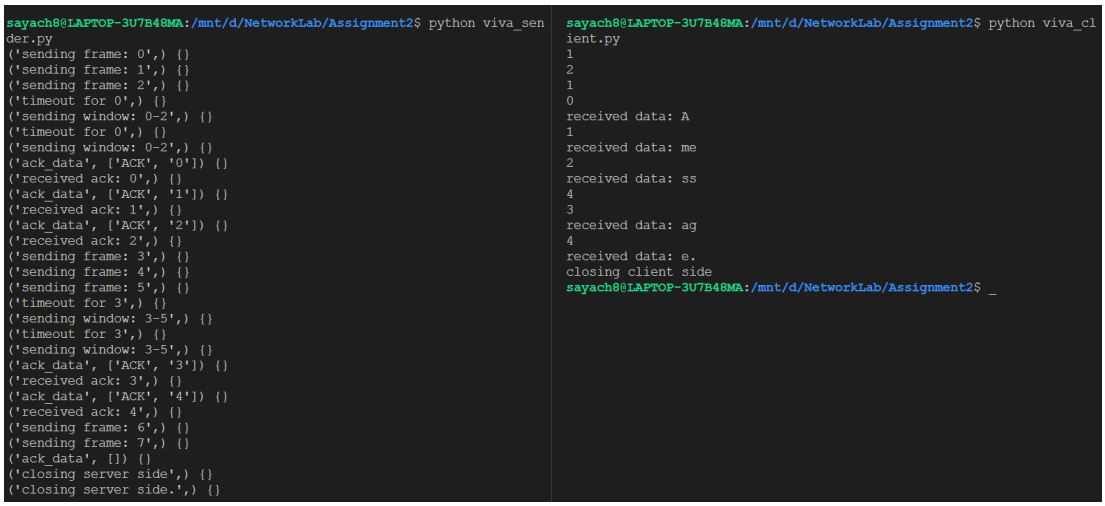
```
        sleep(1) # give 1 second for buffering

      if(index<len(self.all_messages)):

        message_to_send=self.error_maker(self.all_messages[index])
        client.send(message_to_send.encode(FORMAT_MSG))
        # print(f"len of msg is: {len(self.all_messages[index].encode(FORMAT_MSG))}")

        self.time_sent_mappa[index+1]=time.time()
        sleep(1)
      index+=1

    try:
      while True:
        client.send(EXIT_MESSAGE.encode(FORMAT_MSG))
    except:
      print("connection closed from client side")

  def start_server(self):

    self.sock.listen()
    while True:
      try:
        client,addr=self.sock.accept()
      except socket.timeout as tt:
        continue
      temp_thread=threading.Thread(target=self.handle_client,args=[client,addr])
      temp_thread.start()

from client import Listener
from convention import *
import threading,time

class go_back_n_listener(Listener):


  def start_listener(self,host_name,port):
    addr=(host_name,port)
    temp_thread=threading.Thread(target=self.establish_connection,args=[addr])
    temp_thread.start()

  def establish_connection(self,addr):
    self.sock.connect(addr)
    while True:
```

```
    msg=self.sock.recv(MAX_LEN_MSG).decode(FORMAT_MSG)
    msg=self.error_maker(msg)

    if msg==EXIT_MESSAGE:
      print("closing client node")
      self.sock.close()
      return
    if Listener.error_detect_parity(msg):
      print(f"Faulty Message Received by {self.client_id}!")
      continue

    message,serial_number=Listener.decode(msg)
    print(f"INCOMING MESSAGE FOR SERIAL:{serial_number} by {self.client_id}")
    print(f"{message}")

    send_msg=f"ACK {serial_number}"
    self.sock.send(send_msg.encode(FORMAT_MSG))
```

File:

A message.

Output:



Code:(Selective Repeat)

```
    from server import Server
from client import Listener
import threading,time,socket
from convention import *

from collections import deque

class selective_repeat_server(Server):
```

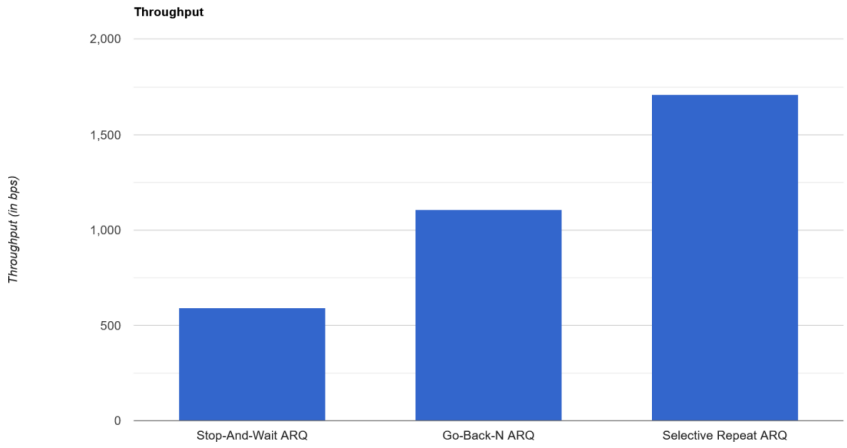
```
  def __init__(self, host_name, port, file_to_send, extra_delay_error_func, server_id, window: int
= 1):
    super().__init__(host_name, port, file_to_send, extra_delay_error_func, server_id, window)
    self.all_messages=[]
    self.recv_dict=dict()
    self.time_sent_mappa=dict()
    self.queue=deque()


  def get_ack(self,client:socket.socket):
    #add serial numbers to the dict
    try:
      while True:
        full_ack=client.recv(MAX_LEN_MSG).decode(FORMAT_MSG)
        ack_type,serial_no=full_ack.split()
        serial_no=int(serial_no)

        if(ack_type=='NAK'):
          self.recv_dict[serial_no]=-1
        else:
          self.recv_dict[serial_no]=1

        print(f"received {ack_type} for {serial_no}")
    except:
      print("connection stopped from client side\n")

  def send_one_frame(self,serial_no,frame_content:str,client:socket.socket):
    try:
      client.send(frame_content.encode(FORMAT_MSG))
      timing=time.time()
      self.time_sent_mappa[serial_no]=timing
      self.queue.append((serial_no,timing))
      sleep(1)

    except Exception as e:
      print("Connection to server closed.\n")
      print(f"{e}")

  def handle_client(self, client: socket.socket, addr):
    self.all_messages,total_serial=self.encode(self.file_name)
    self.all_messages=self.all_messages[:-1:1]
    index=0

    #start receiver thread
```

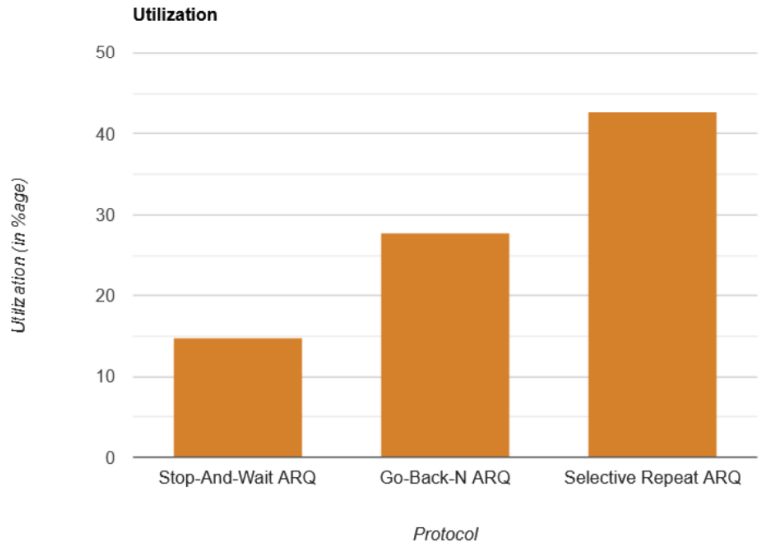
```
    receiver_thread=threading.Thread(target=self.get_ack,args=[client])
    receiver_thread.start()

    while index!=len(self.all_messages)+self.window_len:
      prev_index=index-self.window_len
      serial_no_prev=prev_index+1

      if(serial_no_prev>=1 and (serial_no_prev not in self.recv_dict or
self.recv_dict[serial_no_prev]==-1)):
        #wait for the timeout amount to pass
        cur_time=time.time()
        # print(f"gonna check serial no: {serial_no_prev}")
        if cur_time-self.time_sent_mappa[serial_no_prev]<DEFAULT_TIMEOUT_MSG:
          sleep(1)
          continue

        self.send_one_frame(serial_no_prev,self.all_messages[serial_no_prev-1],client)

        sleep(1) # give 1 second for buffering

      while(len(self.queue)>0 and self.queue[0][1]<time.time()):
        serial_no_now=self.queue[0][0]
        self.queue.popleft()
        if serial_no_now in self.recv_dict and self.recv_dict[serial_no_now]==1:
          continue
        # print("printing from here")
        self.send_one_frame(serial_no_now,self.all_messages[serial_no_now-1],client)


      if(index<len(self.all_messages)):

        message_to_send=self.error_maker(self.all_messages[index])
        self.send_one_frame(index+1,message_to_send,client)


        # client.send(message_to_send.encode(FORMAT_MSG))
        # print(f"len of msg is: {len(self.all_messages[index].encode(FORMAT_MSG))}")
        # sleep(1)

      index+=1

    try:
      while True:
        client.send(EXIT_MESSAGE.encode(FORMAT_MSG))
```

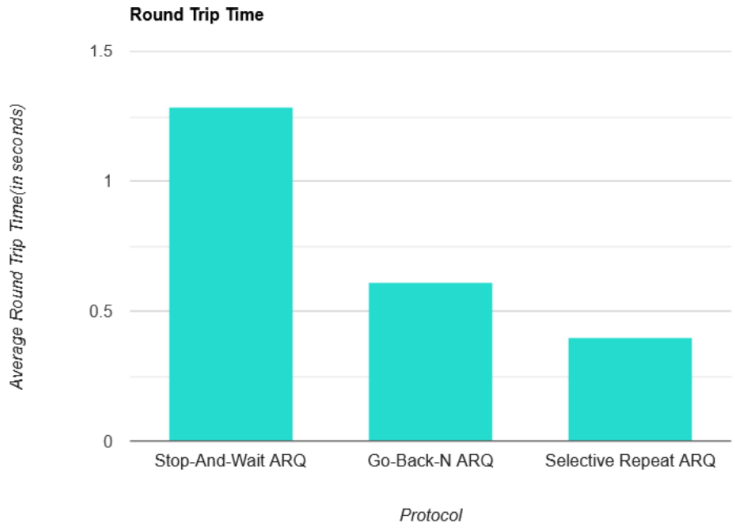
```
    except:
      print("connection closed from client side at end_msg\n")
import socket
import threading
from client import Listener
from convention import *

class selective_repeat_listener(Listener):
  def __init__(self, error_delay_func, client_id):
    super().__init__(error_delay_func, client_id)

  def start_listener(self,host_name,port):
    addr=(host_name,port)
    temp_thread=threading.Thread(target=self.establish_connection,args=[addr])
    temp_thread.start()

  def establish_connection(self,addr):
    self.sock.connect(addr)
    while True:
      msg=self.sock.recv(MAX_LEN_MSG).decode(FORMAT_MSG)
      msg=self.error_maker(msg)

      # print(f"msg len is: {len(msg)}")

      if msg==EXIT_MESSAGE:
        self.sock.close()
        return
      if Listener.error_detect_parity(msg):
        print(f"Faulty Message Received by {self.client_id}!")
        send_msg="NAK -1"
        self.sock.send(send_msg.encode(FORMAT_MSG))

        continue

      message,serial_number=Listener.decode(msg)
      print(f"INCOMING MESSAGE FOR SERIAL:{serial_number} by {self.client_id}")
      print(f"{message}")

      send_msg=f"ACK {serial_number}"
      self.sock.send(send_msg.encode(FORMAT_MSG))
```

Output:

```
INCOMING MESSAGE FOR SERIAL:1 by 1
A message.
received ACK for 1
INCOMING MESSAGE FOR SERIAL:0 by 1
```







Discussion:
Stop-And-Wait is memory efficient as the sequence numbers are only 0 and 1 and thus, keeps a copy of just 1 sent frame. If the channel is thick and long, the potential of the channel is wasted because we are just waiting for an ACK from the receiver, whereas we could have Page 69 of 136 sent a few packets lined up next too at the same time. This would boost the throughput to a great extent. The need to utilize more of the channel brings us to Go-Back-N ARQ, where we send many frames before waiting for ACK. This ensures that many frames are in transit at the same time, which is desired when the bandwidth-delay product is high. But here the receiver needs to accept the frames in order. So a timer is maintained on the sender side to resend the frames, in case the frame or ACK was lost during transit and thus the frame was either not acknowledged or the sender didn't receive the ACK. Whenever such happens, all the frames from the last acknowledged frames are resent by the sender. In Selective Repeat ARQ, multiple frames are in transit and the channel is also utilized well. The improvement here is that the receiver can accept the frames in any order. It just needs to make sure that the data is delivered to the file accurately. As a result, the frames within a window can be acknowledged in any order. 1 NAK can inform regarding the last missing packet and 1 ACK can serve as ACK for the previously received ACKs as well because an ACK is transferred only when the frames are converted in order to message and delivered to the file. This releases contention on the channel. But the out-of-order hack necessitates individual timers, so more memory overhead is present on the sender side and special care must be given to synchronization issues.

Comment:

I got to learn about the various data sending protocols used in network systems.