Lab 4 report

Completed by Dominic Choi

UCID 30109955

|  |
| --- |
| Sender.py |
| import argparse  import RDT  import time  if \_\_name\_\_ == "\_\_main\_\_":      parser = argparse.ArgumentParser(          description="Quotation sender talking to a receiver."      )      parser.add\_argument("receiver", help="receiver.")      parser.add\_argument("port", help="Port.", type=int)      args = parser.parse\_args()      msg\_L = [          "sending message - 1",          "sending message - 2",          "sending message - 3",          "sending message - 4",          "sending message - 5",          "sending message - 6",          "sending message - 7",          "sending message - 8",          "sending message - 9",          "sending message - 10",      ]      timeout = 2  # send the next message if no response      time\_of\_last\_data = time.time()      rdt = RDT.RDT("sender", args.receiver, args.port)      for msg\_S in msg\_L:          resendMessage = msg\_S          rdt.rdt\_3\_0\_send(msg\_S)          # try to receive message before timeout          rcvPkt = None          while rcvPkt == None:              rcvPkt = rdt.rdt\_3\_0\_receive()              if rcvPkt is None:                  # If timeout occurs resend message                  if time\_of\_last\_data + timeout < time.time():                      rdt.rdt\_3\_0\_send(resendMessage, True)                      time\_of\_last\_data = time.time()                  else:                      continue              elif rcvPkt == True:                  rcvPkt = None              elif rcvPkt:                  if int(rcvPkt.msg\_S) != rdt.seq\_num:                      print(                          f"Receive ACK {rcvPkt.msg\_S}. Resend message {rdt.seq\_num}"                      )                      rdt.rdt\_3\_0\_send(resendMessage)                      rcvPkt = None          time\_of\_last\_data = time.time()          # print the result          if rcvPkt:              print(                  f"Receive ACK {rcvPkt.seq\_num}. Message successfully sent!\n"              )              rdt.seq\_num += 1      rdt.disconnect() |
| Reciever.py |
| import argparse  import RDT  import time  if \_\_name\_\_ == "\_\_main\_\_":      parser = argparse.ArgumentParser(description="Uppercase conversion receiver.")      parser.add\_argument("port", help="Port.", type=int)      args = parser.parse\_args()      timeout = 10  # close connection if no new data within 5 seconds      time\_of\_last\_data = time.time()      rdt = RDT.RDT("receiver", None, args.port)      while True:          # try to receive message before timeout          rcvPkt = rdt.rdt\_3\_0\_receive()          if rcvPkt is None:              if time\_of\_last\_data + timeout < time.time():                  print("Timeout: No more data. Closing connection.")                  break              else:                  continue          time\_of\_last\_data = time.time()          if rcvPkt == True:              print(                  f"Corruption detected! Sending ACK {rdt.seq\_num - 1}\n"              )              rdt.rdt\_3\_0\_send(str(rdt.seq\_num - 1))          else:              print(                  f"Receive message {rcvPkt.seq\_num}. Send ACK {rdt.seq\_num}\n"              )              rdt.rdt\_3\_0\_send(str(rcvPkt.seq\_num))              if rcvPkt.seq\_num == rdt.seq\_num:                  rdt.seq\_num += 1      rdt.disconnect() |
| RDT.py |
| import Network  import argparse  from time import sleep  import hashlib  class Packet:      ## the number of bytes used to store packet length      seq\_num\_S\_length = 10      length\_S\_length = 10      ## length of md5 checksum in hex      checksum\_length = 32      def \_\_init\_\_(self, seq\_num, msg\_S):          self.seq\_num = seq\_num          self.msg\_S = msg\_S      @classmethod      def from\_byte\_S(self, byte\_S):          # If packet is corrupt, resend the message          if Packet.corrupt(byte\_S):              return True          # extract the fields          seq\_num = int(              byte\_S[                  Packet.length\_S\_length : Packet.length\_S\_length                  + Packet.seq\_num\_S\_length              ]          )          msg\_S = byte\_S[              Packet.length\_S\_length + Packet.seq\_num\_S\_length + Packet.checksum\_length :          ]          return self(seq\_num, msg\_S)      def get\_byte\_S(self):          # convert sequence number of a byte field of seq\_num\_S\_length bytes          seq\_num\_S = str(self.seq\_num).zfill(self.seq\_num\_S\_length)          # convert length to a byte field of length\_S\_length bytes          length\_S = str(              self.length\_S\_length              + len(seq\_num\_S)              + self.checksum\_length              + len(self.msg\_S)          ).zfill(self.length\_S\_length)          # compute the checksum          checksum = hashlib.md5((length\_S + seq\_num\_S + self.msg\_S).encode("utf-8"))          checksum\_S = checksum.hexdigest()          # compile into a string          return length\_S + seq\_num\_S + checksum\_S + self.msg\_S      @staticmethod      def corrupt(byte\_S):          # extract the fields          length\_S = byte\_S[0 : Packet.length\_S\_length]          seq\_num\_S = byte\_S[              Packet.length\_S\_length : Packet.seq\_num\_S\_length + Packet.seq\_num\_S\_length          ]          checksum\_S = byte\_S[              Packet.seq\_num\_S\_length              + Packet.seq\_num\_S\_length : Packet.seq\_num\_S\_length              + Packet.length\_S\_length              + Packet.checksum\_length          ]          msg\_S = byte\_S[              Packet.seq\_num\_S\_length + Packet.seq\_num\_S\_length + Packet.checksum\_length :          ]          # compute the checksum locally          checksum = hashlib.md5(str(length\_S + seq\_num\_S + msg\_S).encode("utf-8"))          computed\_checksum\_S = checksum.hexdigest()          # and check if the same          return checksum\_S != computed\_checksum\_S  class RDT:      ## latest sequence number used in a packet      # seq\_num needs to alternate between 0 and 1??      seq\_num = 1      ## buffer of bytes read from network      byte\_buffer = ""      def \_\_init\_\_(self, role\_S, receiver\_S, port):          self.network = Network.NetworkLayer(role\_S, receiver\_S, port)      def disconnect(self):          self.network.disconnect()      def rdt\_3\_0\_send(self, msg\_S, timeout=False):          p = Packet(self.seq\_num, msg\_S)          if timeout:              print(f"Timeout! Resend message {self.seq\_num}")          else:              print(f"Send message {p.seq\_num}")          self.network.udt\_send(p.get\_byte\_S())      def rdt\_3\_0\_receive(self):          ret\_S = None          byte\_S = self.network.udt\_receive()          self.byte\_buffer += byte\_S          p = ""          while True:              # check if we have received enough bytes              if len(self.byte\_buffer) < Packet.length\_S\_length:                  if ret\_S:                      return p                  return ret\_S  # not enough bytes to read packet length              # length of packet              length = int(self.byte\_buffer[: Packet.length\_S\_length])              if len(self.byte\_buffer) < length:                  if ret\_S:                      return p                  return ret\_S  # not enough bytes to read the whole packet              # create packet from buffer content and add to return string              p = Packet.from\_byte\_S(self.byte\_buffer[0:length])              if p == True:                  print(                      f"Corruption detected! Send ACK {self.seq\_num}"                  )                  self.byte\_buffer = ""                  return True              ret\_S = p.msg\_S if (ret\_S is None) else ret\_S + p.msg\_S              # clear the buffer              self.byte\_buffer = self.byte\_buffer[length:]              # if this was the last packet, will return on the next iteration  if \_\_name\_\_ == "\_\_main\_\_":      parser = argparse.ArgumentParser(description="RDT implementation.")      parser.add\_argument(          "role",          help="Role is either sender or receiver.",          choices=["sender", "receiver"],      )      parser.add\_argument("receiver", help="receiver.")      parser.add\_argument("port", help="Port.", type=int)      args = parser.parse\_args()      rdt = RDT(args.role, args.receiver, args.port)      if args.role == "sender":          rdt.rdt\_1\_0\_send("MSG\_FROM\_SENDER")          sleep(2)          print(rdt.rdt\_1\_0\_receive())          rdt.disconnect()      else:          sleep(1)          print(rdt.rdt\_1\_0\_receive())          rdt.rdt\_1\_0\_send("MSG\_FROM\_RECEIVER")          rdt.disconnect() |
| Network.py |
| import argparse  import socket  import threading  from time import sleep  import random  import RDT  class NetworkLayer:      prob\_pkt\_loss = .2      prob\_byte\_corr = .1      prob\_pkt\_reorder = 0      sock = None      conn = None      buffer\_S = ""      lock = threading.Lock()      collect\_thread = None      stop = None      socket\_timeout = 0.1      reorder\_msg\_S = None      def \_\_init\_\_(self, role\_S, receiver\_S, port):          if role\_S == "sender":              print("Network: role is sender")              self.conn = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)              self.conn.connect((receiver\_S, port))              self.conn.settimeout(self.socket\_timeout)          elif role\_S == "receiver":              print("Network: role is receiver")              self.sock = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)              self.sock.bind(("localhost", port))              self.sock.listen(1)              self.conn, addr = self.sock.accept()              self.conn.settimeout(self.socket\_timeout)          self.collect\_thread = threading.Thread(name="Collector", target=self.collect)          self.stop = False          self.collect\_thread.start()      def disconnect(self):          if self.collect\_thread:              self.stop = True              self.collect\_thread.join()      def \_\_del\_\_(self):          if self.sock is not None:              self.sock.close()          if self.conn is not None:              self.conn.close()      def udt\_send(self, msg\_S):          if random.random() < self.prob\_pkt\_loss:              return          if random.random() < self.prob\_byte\_corr:              start = random.randint(RDT.Packet.length\_S\_length, len(msg\_S) - 5)              num = random.randint(1, 5)              repl\_S = "".join(random.sample("XXXXX", num))              msg\_S = msg\_S[:start] + repl\_S + msg\_S[start + num :]          if random.random() < self.prob\_pkt\_reorder or self.reorder\_msg\_S:              if self.reorder\_msg\_S is None:                  self.reorder\_msg\_S = msg\_S                  return None              else:                  msg\_S += self.reorder\_msg\_S                  self.reorder\_msg\_S = None          totalsent = 0          while totalsent < len(msg\_S):              sent = self.conn.send(msg\_S[totalsent:].encode("utf-8"))              if sent == 0:                  raise RuntimeError("socket connection broken")              totalsent = totalsent + sent      def collect(self):          while True:              try:                  recv\_bytes = self.conn.recv(4096)                  with self.lock:                      self.buffer\_S += recv\_bytes.decode("utf-8")              except socket.timeout as err:                  pass              if self.stop:                  return      def udt\_receive(self):          with self.lock:              ret\_S = self.buffer\_S              self.buffer\_S = ""          return ret\_S  if \_\_name\_\_ == "\_\_main\_\_":      parser = argparse.ArgumentParser(description="Network layer implementation.")      parser.add\_argument("role", choices=["sender", "receiver"])      parser.add\_argument("receiver")      parser.add\_argument("port", type=int)      args = parser.parse\_args()      network = NetworkLayer(args.role, args.receiver, args.port)      if args.role == "sender":          network.udt\_send("MSG\_FROM\_SENDER")          sleep(2)          print(network.udt\_receive())          network.disconnect()      else:          sleep(1)          print(network.udt\_receive())          network.udt\_send("MSG\_FROM\_RECEIVER")          network.disconnect() |

Outputs:A screen shot of a computer

Description automatically generated