Lab 4 report

Completed by Dominic Choi

UCID 30109955

Sender.py

```python
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:

```
PS D:\Schoolwork\2023-4 Fall\ENSF 462\ENSF-462-Labs\Lab 04> python Receiver.py 5678
Network: role is receiver
Receive message 1. Send ACK 1

Send message 1
Corruption detected! Send ACK 2
Corruption detected! Sending ACK 1

Send message 2
Receive message 2. Send ACK 2

Send message 2
Receive message 2. Send ACK 3

Send message 3
Receive message 3. Send ACK 3

Send message 3
Receive message 4. Send ACK 4

Send message 4
Receive message 5. Send ACK 5

Send message 5
Receive message 6. Send ACK 6

Send message 6
Receive message 6. Send ACK 7

Send message 7
Receive message 6. Send ACK 7

Send message 7
Receive message 7. Send ACK 7

Send message 7
Receive message 8. Send ACK 8

Send message 8
Receive message 9. Send ACK 9

Send message 9
Corruption detected! Send ACK 10
Corruption detected! Sending ACK 9

Send message 10
Receive message 10. Send ACK 10
```

```
PS D:\Schoolwork\2023-4 Fall\ENSF 462\ENSF-462-Labs\Lab 04> python Sender.py localhost 5678
Network: role is sender
Send message 1
Corruption detected! Send ACK 1
Timeout! Resend message 1
Receive ACK 2. Message successfully sent!

Send message 2
Corruption detected! Send ACK 2
Timeout! Resend message 2
Receive ACK 3. Message successfully sent!

Send message 3
Timeout! Resend message 3
Receive ACK 3. Message successfully sent!

Send message 4
Timeout! Resend message 4
Receive ACK 4. Message successfully sent!

Send message 5
Timeout! Resend message 5
Receive ACK 5. Message successfully sent!

Send message 6
Timeout! Resend message 6
Corruption detected! Send ACK 6
Timeout! Resend message 6
Receive ACK 7. Message successfully sent!

Send message 7
Timeout! Resend message 7
Receive ACK 7. Message successfully sent!

Send message 8
Receive ACK 8. Message successfully sent!

Send message 9
Corruption detected! Send ACK 9
Timeout! Resend message 9
Receive ACK 10. Message successfully sent!

Send message 10
Receive ACK 10. Message successfully sent!

PS D:\Schoolwork\2023-4 Fall\ENSF 462\ENSF-462-Labs\Lab 04>
```