Transport Layer (Implementing Reliable Transport Protocol) — lab4 Part2

Contents

Introduction

The goal of this lab is to learn how to implement and analyze a reliable transport protocol. The lab has several **milestones**. Make sure you reach each one before advancing to the next.

For delivery, submit a PDF report where you answer **only** those steps that are marked with **REPORT(%)**. Additionally, you should submit the codes or capture files, if they were **explicitly** asked for. The percent point gives you an indication of the score of that question. Lab4 (both parts) counts for **4.5 points** of your final score in this course.

System Setup

In this lab, you will implement GO-BACK-N (**GBN**) as a reliable transport protocol on top of UDP. This will ensure a reliable transmission of packets in the presence of packet loss, packet corruption and Network delay (read section 3.4.3 of the book). You will use the same setup as in part1, see figure 1.

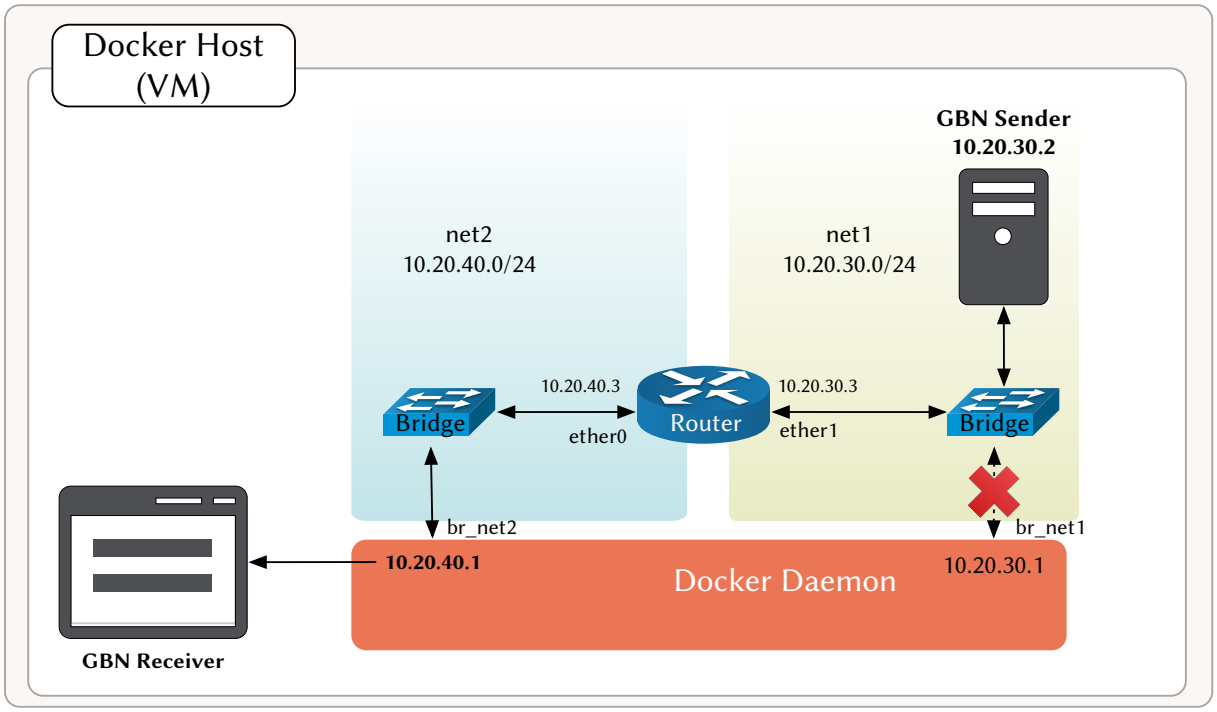

Figure 1: System Setup: Implementing Reliable Transport Protocol

- Start the containers with docker-compose, then delete the direct connection from "10.20.30.0/29" to the docker host and add a new route to ensure traffic is forwarded through the router.

  > *HINT: Refer back to part1*

1 Milestone 1 – Implementing GBN Sender

The skeleton code for the sender is provided below. You are to complete the skeleton code. You can also find the python skeleton file code at "server/gbnsender_skeleton.py". The places where you need to fill in code are marked with "====fill in here====".

You will use threads for sending packets and receiving acknowledgments simultaneously. Additionally, we will use **Lock** for thread synchronization. You can look at this article to get an idea of thread synchronization.

The provided implementation follows the description of the extended FSM presented in the book (section 3.4.3).

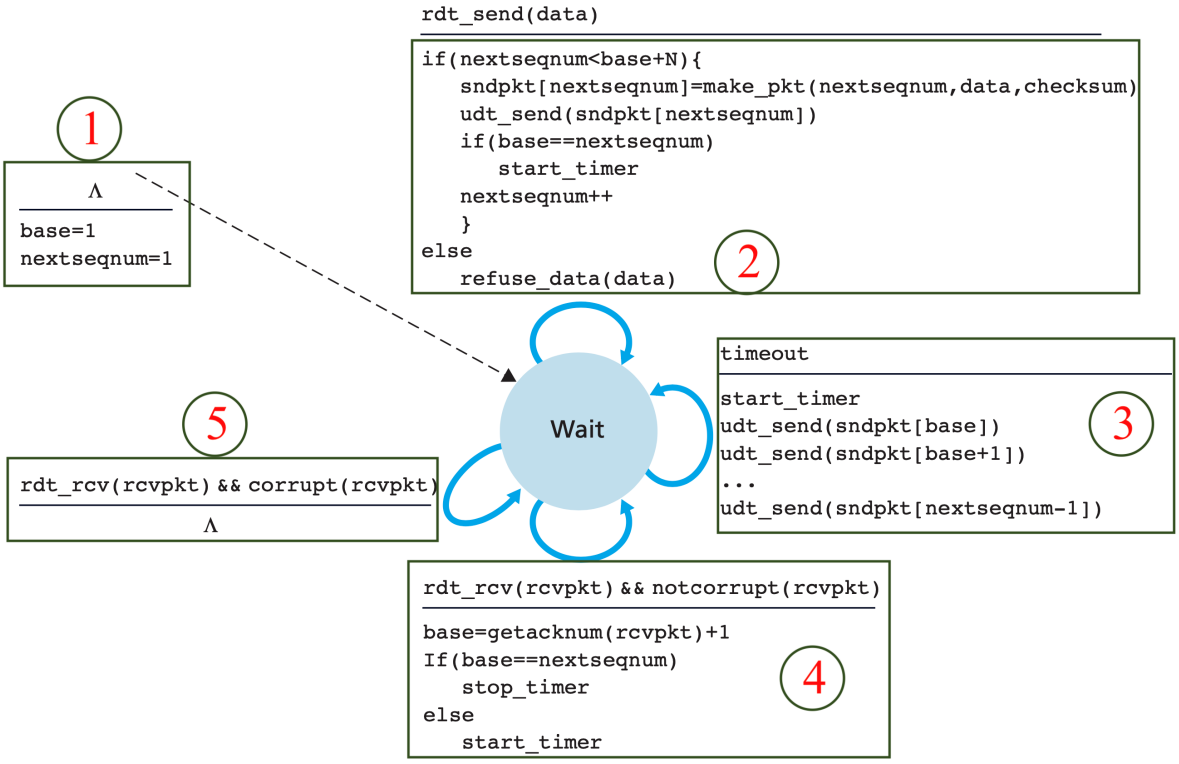

Figure 2: GBN sender (Kurose, J. and Ross, K)

For simplicity, the sender reads a text file and creates a packet from every line of text (every line will be the payload of a single packet). The receiver will extract the lines from the received packets and write them to a text file. In the end, the implementation will reliably copy the text file from the sender to the receiver, even in the presence of packet loss, packet corruption or network delay.

You can write the code in your computer using your favorite python IDE, then copy the code files to the container either using `scp` or the mounted volumes.

```
import socket
from threading import Thread
from threading import Lock
import sys
import time
import binascii


PORT = 5000
SENDER_IP = #====fill in here====
RECEIVER_IP = #====fill in here====
RECV_BUFFER = 1024
# Create a UDP socket
UDP_SOCKET = #====fill in here====
UDP_SOCKET.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
# Bind the UDP socket to SENDER_IP and PORT
#====fill in here====


# Send a packet over unreliable data transprot(UDP)
def udt_send(pkt):
    print(f"send pkt{int.from_bytes(pkt[0:5], byteorder = 'little', signed = True)}")
    #Send "pkt" to the receiver
    #====fill in here====


# Creates a packet in bytes from seqnum, data and checksum
def make_pkt(seqnum, data, checksum):
    seqnum_bytes = seqnum.to_bytes(5, byteorder = 'little', signed = True)
    data_bytes = data.encode('utf-8')
    checksum_bytes = checksum.to_bytes(5, byteorder = 'little', signed = True)
    return seqnum_bytes+checksum_bytes+data_bytes


# Computes cyclic redundancy check (CRC), the 32-bit checksum of (seqnum + data)
def compute_checksum(seqnum, data):
    seqnum_bytes = seqnum.to_bytes(5, byteorder = 'little', signed = True)
    data_bytes = data.encode('utf-8')
    checksum = binascii.crc32(seqnum_bytes+data_bytes)
    return checksum

# Get acknowledgement number from a received packet
def getacknum(rcvpkt):
    acknum = int.from_bytes(rcvpkt[0:5], byteorder = 'little', signed = True)
    return acknum

#check if a received packet is not corrupted
def notcorrupt(rcvpkt):
    acknum = int.from_bytes(rcvpkt[0:5], byteorder = 'little', signed = True)
    checksum = int.from_bytes(rcvpkt[5:10], byteorder = 'little', signed = True)
    ACK = rcvpkt[10:].decode('utf-8')
    computed_checksum = compute_checksum(acknum, ACK)
    return (checksum == computed_checksum)
```

```
#receive packets from UDP socket
def rdt_rcv():
    # Receive a rcvpkt (acknowedgement) from the receiver
    rcvpkt, address = #====fill in here====
    print(f"rcv ACK{int.from_bytes(rcvpkt[0:5], byteorder = 'little', signed = True)}")
    return rcvpkt


#End transmission and exit the main thread
def end_transmission():
    global thread_lock
    empty_data = ''
    checksum = compute_checksum(nextseqnum, empty_data)
    empty_pkt = make_pkt(nextseqnum, empty_data, checksum)
    UDP_SOCKET.sendto(empty_pkt, (RECEIVER_IP  ,PORT))
    print('end transmission')
    thread_lock.acquire()
    sys.exit()


# class that provides: starting, stopping, and checking a timer
class Timer(object):
    def __init__(self, timeout_interval):
        self._start_time = 0.0
        self._timeout_interval = timeout_interval
    def start(self):
        self._start_time = time.time()
    def stop(self):
        self._start_time = time.time()
    def running(self):
        return self._start_time > 0.0
    def timeout(self):
        return ((time.time() - self._start_time) >= self._timeout_interval)


#Figure1, circle 1: Initial state of GBN sender
#We start with 0 to be compatible with python (0 based indexing)
base = 0
nextseqnum = 0

#GBN parameters
N = 4
#====================
# CHANGE TIMEOUT_INTERVAL TO SEE EFFECTS OF POOR NETWORK CONDITIONS
#====================
TIMEOUT_INTERVAL = 3

timer = Timer(TIMEOUT_INTERVAL)
thread_lock = Lock()
```

```
    #Main thread sending packets,
    def send():
        global thread_lock
        global base
        global timer
        global N
        global nextseqnum

        #read text file (send_data.txt) and initialize empty sndpkt list
        with open('/home/ttm4200/work_dir/senddata.txt') as f:
            data_lines = f.readlines()
        sndpkt = [None]*len(data_lines)

        #start a thread to receive ACK
        Thread(target=receive, daemon=True).start()

        #loop until you send all data
        while base < len(data_lines):
            thread_lock.acquire()
            N = min(N, len(data_lines) - base)

            #Figure1, circle 2:
            if (nextseqnum < base + N):
                data = data_lines[nextseqnum]
                checksum = compute_checksum(nextseqnum, data)

                # make packet using make_pkt function
                #====fill in here====

                # send the packet using udt_send function
                #====fill in here====

                # start the timer if the base equal nextseqnum
                #====fill in here====
                #====fill in here====

                # increment nextseqnum
                #====fill in here====

            else:
                #refuse_data
                pass

            # Figure1, circle 3: if timeout, resend all packets (from the base)
            if timer.timeout():
                print(f"pkt{base} timeout")
                # Hint: reset the nextseqnum to the base, so it will go through
                # the loop again and send packets from base
                #====fill in here====

            thread_lock.release()
        end_transmission()
```

```
    # Daemon thread  for  receiving acknowledgment
    def receive():
        global thread_lock
        global base
        global timer
        global nextseqnum
        while True:
            rcvpkt = rdt_rcv()
            thread_lock.acquire()
            #Figure1, circle 4:
            if rcvpkt  and notcorrupt(rcvpkt):
                #get the acknum (getacknum) and set the base to it plus one
                #====fill in here====
                base=getacknum(rcvpkt)+1
                #stop the time if base equeal nextseqnum, otherwise start it
                #====fill in here====
                #====fill in here====
                #====fill in here====
                #====fill in here====

            ##Figure1, circle 4:
            else:
                #if the are no received packet or corruped packets, stay in the loop
                pass

            thread_lock.release()


    if __name__ == '__main__':
        send()
        UDP_SOCKET.close()
```

2 Milestone 2 – Implementing GBN Receiver

The skeleton code for the receiver is provided below. You are to complete the skeleton code. You can also find the python skeleton file code “gbnreceiver_skeleton.py” in lab4 directory. The places where you need to fill in code are marked with “====fill in here====”.

The provided implementation follows the description of the extended FSM presented in the book (section 3.4.3).

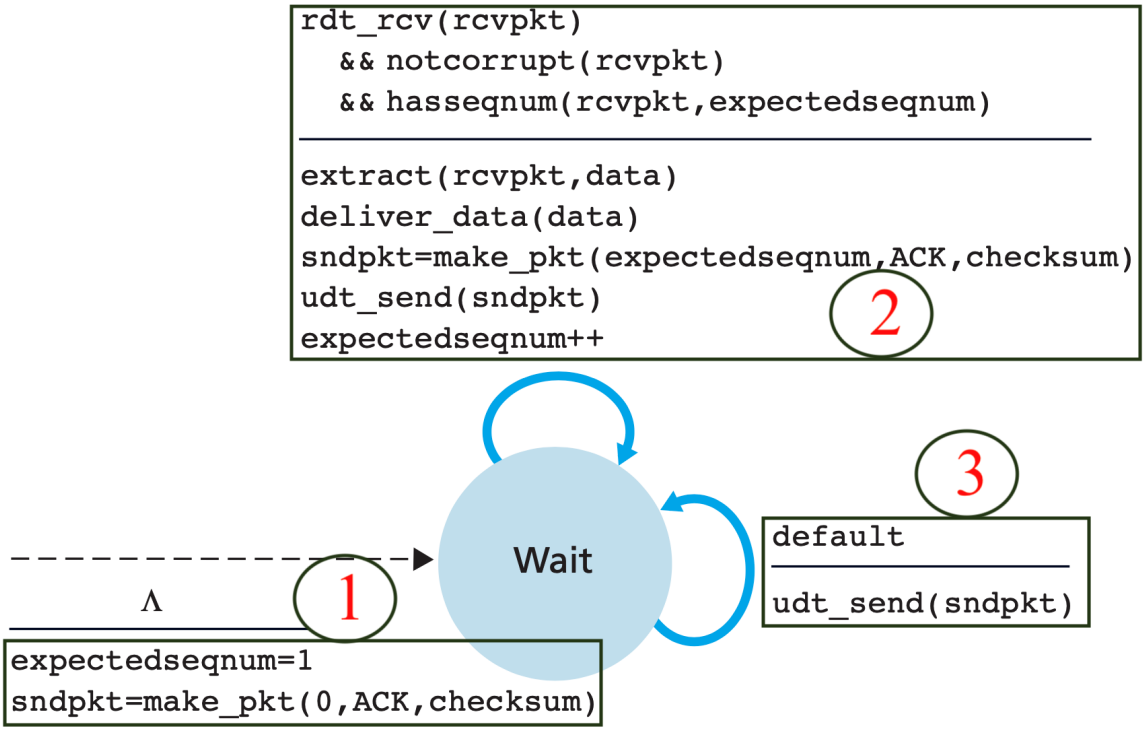

Figure 3: GBN receiver (Kurose, J. and Ross, K)

```
import socket
import sys
import binascii

RECEIVER_IP = #====fill in here====
SENDER_IP = #====fill in here====
PORT = 5000

#====================
#ACKNOWLEDGEMENT MESSAGE
#====================
ACK = ""

RECV_BUFFER = 1024
# Create a UDP socket
UDP_SOCKET = #====fill in here====
UDP_SOCKET.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind the UDP socket to RECEIVER_IP and PORT
#====fill in here====


# Receive packets from UDP socket
```

```
    def rdt_rcv():
        # Receive a rcvpkt  from the sender
        rcvpkt, address = #====fill in here====
        return rcvpkt


    # Send packet over unreliable data transprot(UDP)
    def udt_send(pkt):
        print(f"send ACK{int.from_bytes(pkt[0:5], byteorder = 'little', signed = True)}")
        #Send "pkt" to the sender
        #====fill in here=====


    # Creates a packet in bytes from seqnum, data and checksum
    def make_pkt(seqnum, ACK, checksum):
        seqnum_bytes = seqnum.to_bytes(5, byteorder = 'little', signed = True)
        ACK_bytes = ACK.encode('utf-8')
        checksum_bytes = checksum.to_bytes(5, byteorder = 'little', signed = True)
        return seqnum_bytes+checksum_bytes+ACK_bytes


    # Check if a received packet is not corrupted
    def notcorrupt(rcvpkt):
        acknum = int.from_bytes(rcvpkt[0:5], byteorder = 'little', signed = True)
        checksum = int.from_bytes(rcvpkt[5:10], byteorder = 'little', signed = True)
        ACK = rcvpkt[10:].decode('utf-8')
        computed_checksum = compute_checksum(acknum, ACK)
        return (checksum == computed_checksum)


    # Extract seqnum,checksum and  payload from a received packet
    def extract(rcvpkt):
        data = rcvpkt[10:].decode('utf-8')
        if data:
            print(f"rcv pkt{int.from_bytes(rcvpkt[0:5],byteorder='little',signed=True)},deliver")
        else:
            sys.exit()
        return data


    # Write received data to a file
    def deliver_data(data):
        with open('./rcvdata.txt', 'a') as f:
            f.write(data)


    # Check if a received packet has a seqnum as the expectedseqnum
    def hasseqnum(rcvpkt, expectedseqnum):
        seqnum = int.from_bytes(rcvpkt[0:5], byteorder = 'little', signed = True)
        return (seqnum == expectedseqnum)


    # Computes cyclic redundancy check (CRC), the 32-bit checksum of (seqnum + payload)
    def compute_checksum(seqnum, payload):
```

```
        seqnum_bytes = seqnum.to_bytes(5, byteorder = 'little', signed = True)
        payload_bytes = payload.encode('utf-8')
        checksum = binascii.crc32(seqnum_bytes+payload_bytes)
        return checksum


    # Start receiving packets
    def receive():
        # Figure2, circle 1: Initial state of GBN receiver
        expectedseqnum = 0
        checksum = compute_checksum(-1, ACK)
        sndpkt=make_pkt(-1, ACK, checksum)

        while True:
            rcvpkt = rdt_rcv()

            # Figure2, circle 2:
            if rcvpkt and notcorrupt(rcvpkt) and hasseqnum(rcvpkt, expectedseqnum):
                data = extract(rcvpkt)
                deliver_data(data)
                checksum = compute_checksum(expectedseqnum, ACK)

                # make sndpkt using make_pkt function
                #====fill in here====

                # send the packet using udt_send function
                #====fill in here====

                # increment expectedseqnum
                #====fill in here====

            # Figure2, circle 3: send ack of the last received pakt
            else:
                # Hint: use udt_send function
                #====fill in here====
                print(f"rcv pkt{int.from_bytes(rcvpkt[0:5],byteorder='little',signed=True)},discard")


    if __name__ == '__main__':
        receive()
        UDP_SOCKET.close()
```

3 Milestone 3 – Running GBN Implementation

- After completing the skeleton code, run the code of the receiver in the in your **VM**. Then run the code of the sender in the "server" container (use **Python3**). Debug your code and make sure that the process of sending packets and receiving acknowledgements corresponds to the GBN operation.

Q1. **REPORT(23%):** In your report, submit the completed code in python files and screenshots of the sender and receiver output.

4 Milestone 4 – Analyzing GBN Operation

Now you will test your GBN implementation in the presence of packet loss, packet corruption and network delay.

- Introduce a 50% packet loss in the router, on the interface "ether0". Then run your GBN implementation.

Q2. **REPORT(4%):** Provide a screenshot of your implementation output in the presence of packet loss and briefly explain what happened.

> *HINT: refer to page 254, figure 3.22 in the book. Remember that the* `tc` *traffic shaping commands apply only to egress (outgoing traffic).*

- Introduce a 50% packet corruption in the router, on the interface "ether0". Then run your GBN implementation.

Q3. **REPORT(4%):** Provide a screenshot of your implementation output in the presence of packet corruption and briefly explain what happened.

- Introduce a network delay in the router, on the interface "ether0". Then run your GBN implementation. Change the amount of network delay and observe what happens when the delay is bigger or lower than the timeout interval in the sender.

Q4. **REPORT(4%):** Provide a screenshot of your implementation output in the presence of network delay. Explain the significance of the timeout interval with regards to network delay.

5 Optional Exercise

We implemented an "end of transmission signal" by sending an empty packet. The receiver checks every packet (in the "rcvpkt" method) and if it finds an empty packet, then it will close the socket. However if the "end of transmission signal" got lost, the receiver will keep the socket open forever.

Q5. **Extra Credit:** Implement a better mechanism for "end of transmission signal". This mechanism must address the fact that "end of transmission signal" can be lost itself.

Contributors

Abdulmajid Murad, David Palma, Stanislav Lange, Mathias Pettersen, Sebastian Fuglesang, Christian Lewin.