Assignment 2

*Computer Networks (CS 456/656)*
Spring 2023
A congestion controlled pipelined RDT
*Due Date: **July 4th 2023, at midnight (11:59 PM)***
Work on this assignment is to be completed individually

Assignment Objective

The goal of this assignment is to implement a **congestion controlled pipelined Reliable Data Transfer (RDT) protocol over UDP**, which could be used to transfer a text file from one host to another across an unreliable network. The protocol should be able to handle packet loss, packet reordering, and duplicate packets. For simplicity, your protocol is unidirectional, i.e., data will flow in one direction (from the sender to the receiver) and the acknowledgements (ACKs, i.e., control messages) in the opposite direction. To implement this protocol, you will write two programs: `sender` and `receiver`, with the specifications given below. To test your implementation, we have provided a third program, the `network_emulator` that will emulate an unreliable communication link.
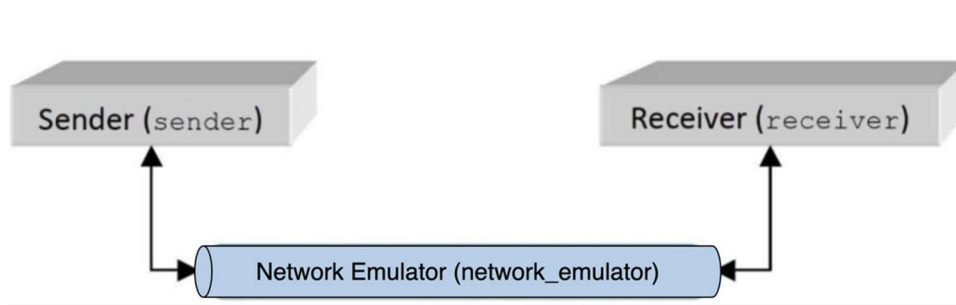
When the `sender` needs to send packets to the `receiver`, it will send them to the `network_emulator` instead of sending them directly to the `receiver`. The `network_emulator` will then forward the received packets to the `receiver`. However, it may randomly discard or reorder the received packets. The same scenario happens when the `receiver` sends ACKs to the `sender`.

Packet Format

All packets exchanged between the sender and the receiver should have the following structure:

```
integer type;          // 0: ACK, 1: Data, 2: EOT, 3: SYN
integer seqnum;        // Modulo 32
integer length;        // Length of the String variable 'data'
String data;           // String with Max Length 500
```

Each integer field is a 4-byte unsigned integer in **network byte order**. The `type` field indicates the type of the packet. It is set to 0 if it is an ACK, 1 if it is a data packet, 2 if it is an end-of-transmission (EOT) packet, and 3 if it is the synchronization (SYN) packet (see the definition and use of EOT and SYN packets below). For data packets, `seqnum` is the modulo 32 sequence number of the packet. The `seqnum` of the SYN packet and the first data packet should be zero. For ACK packets, `seqnum` is the sequence number of the packet being acknowledged. The `length` field specifies the number of characters carried in the data field. It should be in the range of 0 to 500. The `data` string should be exactly `length` bytes long. For EOT, SYN and ACK packets, `length` should be set to zero. A reference implementation of the packet format is provided to you as a Python 3 file named "packet.py". Please view the code at the end of the file for sample usage of the Packet class.

Sender Program (`sender`)

You should implement a sender program, named `sender`. Its command line inputs include the following in the given order:

- `<host address of the network emulator>`,
- `<UDP port number used by the emulator to receive data from the sender>`,
- `<UDP port number used by the sender to receive ACKs from the emulator>`,
- `<timeout interval in units of millisecond>`, and
- `<name of the file to be transferred>`

Upon execution, the sender program goes through 3 stages: (i) connection establishment, (ii) data transmission, and (iii) connection termination. In the connection establishment stage, the sender sends a SYN packet to the receiver (through the network emulator) every 3 seconds, until it receives a SYN packet from the receiver. The SYN packets have a `seqnum` of 0.

In the data transmission stage, the sender program reads data from the specified file and sends it using the congestion controlled RDT protocol to the receiver via the network emulator. The initial window size should be set to N=1 packet, and the seqnum for the initial data packet is set to 0. After all content of the file has been transmitted successfully to the receiver (and **all corresponding ACKs have been received**), the sender transitions to the connection termination stage.

In the connection termination stage, the sender sends an EOT packet to the receiver every 3 seconds, until it receives an EOT packet from the receiver. The sender can close its connection and exit only after it has received ACKs for all data packets it has sent and received an EOT packet from the receiver. For simplicity, you can assume that the EOT packet from the receiver never gets lost in the network.

To ensure reliable transmission and congestion control in the data transmission stage, your program should implement the **congestion controlled pipelined RDT** protocol as follows:

2

If the sender has a packet to send, it first checks to see if the window is full, i.e., whether there are N outstanding, unacknowledged packets. If the window is not full, the packet is sent, the appropriate variables are updated, and a timer is started if not done before. The sender will use only a single timer that will be set for the oldest transmitted-but-not-yet-acknowledged packet. If the window is full, the sender will try sending the packet later.

When the sender receives an acknowledgement packet with `seqnum` n, the ACK will be considered a cumulative acknowledgement, indicating that all packets with a sequence number up to and including n have been correctly received at the receiver. If a timeout occurs, the sender sets N=1 and **retransmits the packet** that caused the timer to timeout (i.e., only that one packet and **not** all the non-ACKed packets). If a packet is retransmitted, the timer is reset.

If a **new** ACK (and not a duplicate ACK) is received, but there are still additional transmitted-but-yet-to-be-acknowledged packets, the timer is restarted. If there are no outstanding packets, the timer is stopped. Also, if a **new** ACK is received, N is incremented by 1 up to a maximum of 10 (N cannot exceed 10). The first data packet is transmitted with `seqnum=0`, the second packet is transmitted with `seqnum=1`, and so on. After the packet with `seqnum=31` is transmitted, the next packet is transmitted with `seqnum=0`.

Output

For both testing and grading purposes, your *sender* program should generate three log files, named seqnum.log, ack.log, and N.log. Whenever a packet is sent, its sequence number should be recorded in *seqnum.log*. The file *ack.log* should record the sequence numbers of all the ACK packets that the sender receives during the entire period of transmission. For SYN and EOT packets, the sequence number written to the file should be "SYN" and "EOT", respectively. *N.log* should record the initial value of N, as well as every time the value of N is changed. The format for these log files is one timestamp, space, and one sequence number per line.

Timestamps are recorded as "t=X", where X is the timestamp of the current action. The timestamp is a number that is incremented by one at every new **event** (i.e., a new packet to be sent, receiving an ACK, or timeout). In the connection establishment stage, the timestamp remains t=-1. The timestamp t=0 is reserved for stage 2 (data transmission) initialization, and the only event that happens during this initialization is that the window size N is initialized to 1. The subsequent data transmission beings at t=1.

Therefore, for all the SYN packets, your program should write T=-1 SYN in *seqnum.log*. *N.log* will have, t=0 1 as the first line in the log. For the first data packet, your program should write t=1 0 in *seqnum.log* for packet #0 sent at t=1. If an EOT is sent by the sender at t=105, then the log should record t=105 EOT.

Similarly, if an EOT is received by the sender from the receiver at t=106, then the *ack.log* should record t=106 EOT.

Be careful, <u>some actions are executed at the same timestamp</u>, e.g., if a timeout occurs at t=T, there should be an entry t=T in *seqnum.log* for the retransmission as well as in *N.log* for (re)setting N to 1. You must follow this format to avoid losing points.

Receiver Program (`receiver`)

You should implement the receiver program, named as `receiver`. Its command line inputs include the following, in the given order:

- `<hostname for the network emulator>`,
- `<UDP port number used by the link emulator to receive ACKs from the receiver>`,
- `<UDP port number used by the receiver to receive data from the emulator>`, and
- `<name of the file into which the received data is written>`.

When receiving packets sent from the sender via the network emulator, it should execute the following:

- If it is a SYN packet, send a SYN packet back, otherwise:
  - o Check the sequence number of the packet.
  - o If the sequence number is the one that it is expecting:
    - ▪ If the packet is an EOT packet, send an EOT packet back and terminate the program. Otherwise, write the data from the packet to the output file. Then check if the packet with the next sequence number is in the buffer.
    - ▪ If the packet exists, remove the packet from the buffer, write the data of the packet to the output file, then repeat the previous step.
    - ▪ If the packet does not exist, send an ACK packet back to the sender with the `seqnum` equal to the `seqnum` of the last packet written to disk and set the expected `seqnum` to the `seqnum` of the missing packet.
  - o Otherwise, if the sequence number is not the one that it is expecting:
    - ▪ If the sequence number is within the next 10 sequence numbers, store the received packet in a buffer if the packet is not already stored.
    - ▪ In all other cases (e.g., duplicate/old packet), discard the received packet.
    - ▪ For both the cases above, send an ACK packet for the most recently received in-order packet.

Once the receiver has received all data packets and an EOT from the sender, it should send an EOT packet and then exit.

Output

The receiver program is also required to generate a log file, named *arrival.log*. The file *arrival.log* should record the sequence numbers of all the data packets that the receiver receives during the entire period of transmission. The sequence number for SYN and EOT packets should be logged as "SYN" and "EOT", respectively. The format for the log file is one number per line (**no timestamp**). You must follow the format to avoid losing marks.

Network Emulator (`network_emulator`)

The network emulator is provided to you as a Python 3 program. When the network emulator receives a data packet from the sender, it will discard it with the specified probability. Otherwise, it will store the packet in its buffer, and later forward the packet to the receiver with a random amount of delay (i.e., less than the specified maximum delay). The same behaviour applies to ACKs received from the receiver. An EOT packet from the receiver is never discarded.

To run `network_emulator`, you need to supply the following command line parameters in the given order:

- `<emulator's receiving UDP port number in the forward (sender) direction>`,

- `<receiver's network address>`,

- `<receiver's receiving UDP port number>`,

- `<emulator's receiving UDP port number in the backward (receiver) direction>`,

- `<sender's network address>`,

- `<sender's receiving UDP port number>`,

- `<maximum delay of the link in units of millisecond>`,

- `<packet discard probability>`, and

- `<verbose-mode>` (Boolean: If set to 1, the network emulator will output its internal processing).

Hints

- The protocol is somewhat like a simplified version of TCP, but it is not TCP! Notably, we buffer out-of-order packets, and sequence/ACK numbers are different.
- You must ensure your programs run in the CS student environment
- Experiment with network delay values and sender time-out to understand the performance of the protocol.

- Please ensure that your implementation works even if the three programs run on separate machines within the CS student environment.
- The assignment package provides a skeleton code in python to start your implementation. The code assumes a multi-threaded implementation. You can use this skeleton code or write your own loop-based implementation if you do not feel comfortable with multi-threading.
- The assignment package also provides sample log files created when the sender and receiver are connected directly. Please follow the same format to avoid losing marks.

Example Execution

1. On the host **host1**: `network_emulator 9991` **host2** `9994 9993` **host3** `9992 1 0.2 0`

2. On the host **host2**: `receiver` **host1** `9993 9994 <output File>`

3. On the host **host3**: `sender` **host1** `9991 9992 50 <input file>`

Note: Use different ports than the ones listed here, as these might be in-use by your classmates.

Procedures

Due Date
The assignment is due on **July 4th 2023, at midnight (11:59 PM)**. Late submission policy: 10% penalty every late day, up to 3 late days. Submissions will not be accepted beyond 3 late days.

Hand in Instructions
Submit all your files in a single compressed file (.zip, .tar etc.) using LEARN. The filename should include your username and/or student ID.

You must hand in the following files / documents:
- *Source code* files.
- *Makefile (if applicable)*: your code **must** compile and link cleanly by typing "*make*" or "*gmake*".
- *README* file: this file **must** contain instructions on how to run your program, which CS student environment machines your program was built and tested on, and what version of *make* and *compilers* you are using (if applicable).

Your implementation will be tested on the machines available in the **CS student environment**.

Documentation
Since there is no external documentation required for this assignment, you are expected to have a reasonable amount of internal code documentation (to help the markers read your code).
You **will** lose points if your code is unreadable, sloppy, and inefficient.