Assignment 2

Computer Networks (CS 456/656)

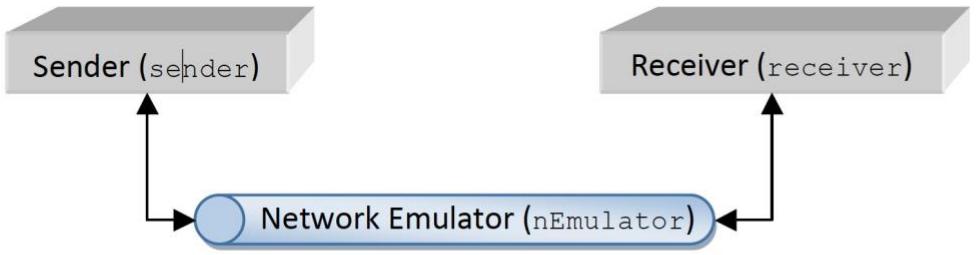
A congestion controlled pipelined RDT

Due Date: Monday July 12, 2021, 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) in the opposite direction. To implement this protocol, you will write two programs: a sender and a receiver, with the specifications given below. To test your implementation, we will provide a third program, the network emulator, that will emulate an unreliable network link.



When the sender needs to send packets to the receiver, it sends them to the network emulator instead of sending them directly to the receiver. The network emulator then forwards 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
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 (see the definition and use of an end-of-transmission packet below). For data packets, *`seqnum` is the modulo 32 sequence number of the packet*. The sequence number of the first 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 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”.

Sender Program (`sender`)

You should implement a sender program, named `sender`. Its command line input includes the following: `<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>` in the given order.

Upon execution, the sender program should be able to read data from the specified file and send 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. After all content of the file has been transmitted successfully to the receiver (and **corresponding ACKs have been received**), the sender should send an EOT packet to the receiver. The EOT packet is in the same format as a regular data packet, except that its `type` field is set to 2 and its *length* is set to zero. 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 from the receiver. To keep the project simple, *you can assume that the* end-of-transmission packet never gets lost in the network.

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

If the sender has a packet to send, it first checks to see if the window is full, that is, 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 taken to be 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* (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 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 be able to *generate three log files*, named as *seqnum.log*, *ack.log, 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 EOT packets, the sequence number should be written to the file as "EOT". *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). The timestamp t=0 is reserved for initialization, and the only event that happens during this is the window size N is initialized to 1. Thus, N.log will have t=0 1 as the first line in the log. Packet transmissions begin at t=1. For the first 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 log should record t=106 EOT.

Be careful, *some actions are executed at the same timestamp*, 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 marks.

Receiver Program (`receiver`)

You should implement the receiver program, named as `receiver`, on a UNIX system. Its command line input includes the following: `<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>` in the given order.

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

- Check the sequence number of the packet.
- 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.
- 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 then exit.

Output

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

Network Emulator (`nEmulator`)

The network emulator is provided to you as a Python 3 program. When the emulator receives a data packet from the sender, it will discard it with the specified probability. Otherwise, it stores the packet in its buffer, and later forwards the packet to the receiver with a random amount of delay (less than the specified maximum delay). The same behaviour applies to ACKs received from the receiver. EOT packet from the sender is never discarded. It is forwarded to the receiver once there are no more data packets in the buffer. EOT packet from the receiver is also never discarded. It is forwarded to the sender once there are no more ACKs in the buffer.

To run `nEmulator`, 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>`,
- `<verbose-mode>` (Boolean: Set to 1, the network emulator will output its internal processing).

Hints

- The protocol is somewhat similar to a simplified version of TCP but it is not TCP! Notably, we do not implement fast retransmit, we buffer out-of-order packets, and sequence/ACK numbers have different.

- *You must ensure your programs run in the CS Undergrad Environment*
- Experiment with network delay values and sender time-out to understand the performance of the protocol.
- To ensure the programs connect properly, you should run `nEmulator`, `receiver`, and `sender` *in this order*. Please ensure that your implementation works even if the three programs run on separate machines within the CS Undergrad Environment.

Example Execution

1. On the host **host1**: `nEmulator 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>`

Procedures

Due Date

The assignment is due on Monday, July 12th, 2021, at midnight (11:59 PM).
Late submission policy: 10% penalty every late day, up to 3 late days. Submissions are not 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 undergrad 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 **undergrad 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 marks if your code is unreadable, sloppy, and inefficient.