Assignment 2

Computer Networks (CS 456)

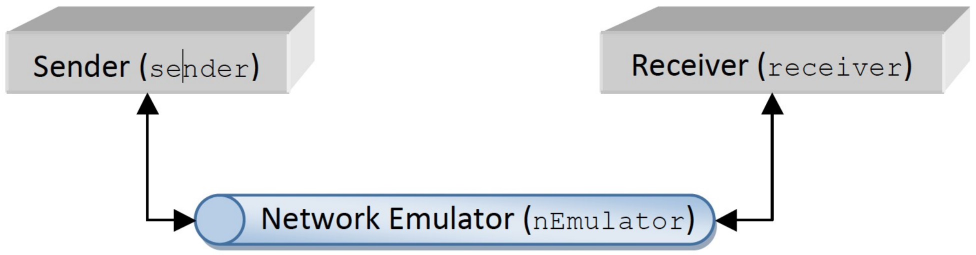
A Congestion-controlled Pipelined RDT Protocol over UDP

Due Date: Monday March 20, 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 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 network errors (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 selective acknowledgements (SACKs) 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 SACKs to the sender.

Packet Format

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

```
integer type;             // 0: SACK, 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 a SACK, 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 SACK 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 SACK 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 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 SACKs from the emulator>`,
- `<timeout interval in units of millisecond>`, and
- `<name of the file to be transferred>`

Upon execution, the sender program should be able to read data from the specified file and send it using a congestion-controlled pipelined (selective repeat) RDT protocol to the receiver via the network emulator. *The initial window size should be set to N=1 packet*. After all contents of the file have been transmitted successfully to the receiver (and **corresponding SACKs 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 SACKs for all data packets it has sent and received an EOT from the receiver. To keep the project simple, *you can assume that the EOT packet never gets lost in the network*.

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

If the sender has a packet to send, it first checks to see if the window is full. If the window is not full, the packet is sent, the appropriate variables are updated, and a timer for the packet is started. The sender will use a timer per packet. If the window is full, the sender will try sending the packet later.

When the sender receives an acknowledgement packet with `seqnum n`, it will be taken to be a selective acknowledgement, indicating that the packet with the sequence number n has been correctly received at the receiver. *If a timeout occurs and the timer is associated with the packet that is at the base of the window, the sender sets N=1 (which causes the window to shrink and only contain the packet at the base) and* ***retransmits that packet***. Otherwise, if a timeout occurs and the timer is associated with **any other packet,** the sender sets N=1 (thereby shrinking the window and forcing the packet outside the window), and the packet's retransmission is delayed **until it re-enters the window**. (Note: A packet can exit and re-enter the window within one timeout interval. A packet's timer should **not** be reset upon exiting the window.) Once a packet is retransmitted, the timer of that packet is reset.

*If a **new** SACK (and not a duplicate SACK) is received, N is incremented by 1 up to a maximum of 10* (N cannot exceed 10).

The first packet of the file must be transmitted with `seqnum 0`, the second packet 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 SACK packets and the EOT packet that the sender receives during the entire period of transmission (including duplicate/old SACKs). 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 updated. *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 a SACK/EOT, timeout, or delayed retransmission). The timestamp t=0 is reserved for initialization, and the only event that happens during this is the window size N being 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 *seqnum.log* should record t=105 EOT. Similarly, if an EOT is received by the sender from the receiver at t=106, then *ack.log* should record t=106 EOT.

Be careful, some actions are executed at the same timestamp, e.g., if the packet at the base of the window times out 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 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 by the sender via the network emulator, the receiver should execute the following:

- Check the sequence number of the packet.
- If the packet is an EOT packet, send an EOT packet back and terminate the program.

- Else, if the sequence is within the receiver window:
    - Send a SACK for the packet
    - If the packet was not previously received, add the packet to the buffer, then, if the received packet is at the base of the window, write the data in the packet and any previously buffered and consecutively numbered packets to the file and remove those packets from the buffer

- Else, if the sequence number is within the last 10 consecutive sequence numbers of the base of the window
    - Send a SACK for the packet, then discard the packet
- Otherwise, ignore the packet.

Output

The receiver program is also required to generate a log file, named as *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 format for the log file is one number per line (**no timestamp**) or EOT for the EOT packet. You must follow the format to avoid losing marks.*

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 SACKs 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 SACKs 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, one per line, e.g. receiving Packet `seqnum` /SACK `seqnum`, discarding Packet `seqnum` /SACK `seqnum`, forwarding Packet `seqnum` /SACK `seqnum`).

Hints

- The protocol is somewhat similar to Selective Repeat as described in Chapter 3 in your textbook and lecture slides, but keep in mind that the protocol we are implementing uses a congestion control mechanism with dynamic window size as opposed to the static window size of Selective Repeat.
- *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, March 20th, 2023, 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 for your sender and receiver
- *Makefile (if applicable)*: if your program requires compilation, 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 submission may not include any extraneous files (e.g. test files, log files), and the README and Makefile should be in the top-level directory. 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).

<u>You **will** lose marks if your code is unreadable, sloppy, and inefficient</u>.