CSE 156/L

Programming Assignment 3
Due: [ ]

In this project you will develop a client and server for a reliable file transfer application using UDP. The user of the application sends a local file to the server. The server will reassemble and save the file locally from the received client packets. The packets sent by the client may get re-ordered or lost in transit to the server.

Description

1. After the **server** is started, it waits to receive packets from clients. Once a packet arrives, it reads the packet and saves the payload of the packets transmitted by the client to reassemble the file locally. The server saves the file locally using the path name given by the client. The path name may contain directory names.

   Each time the server drops a packet, it must print a line to **stdout** for it. The line must follow the format given below in the requirements. The line should have the timestamp of when it occurs in the Internet format (see RFC 3339). It indicates if the dropped packet is DATA with DROP DATA or is ACK with DROP ACK. Finally, it shows the sequence number if it's a DATA packet or the acknowledgement number if it's an ACK packet.

   The server is started by the following command giving the port number for receiving packets. Also, it takes a percentage value, between 0 and 100, to drop packets. Then, *droppc* could be use for debugging your reliability functionality. Zero means no drops, and 100 means every packet is dropped.

   `./myserver` *`port_number droppc`*

2. The **client** reads a file from disk and sends it to the server over a UDP socket. In order to send the file, the client breaks the file up into *mtu*-byte sized packets.

   The client expects the file to be saved by the server correctly and reliably. The client communicates to the server the path name for saving the output bytes. This path name is specified by the command argument *out_file_path*. The resulting new output file (on the server) must be identical to the initial read file for correctness and success. In particular, the bytes in the file must be correct, even if the packets have been re-ordered in transit to the server.

   The client can exit successfully, i.e., with exit(0), when the file has been fully and reliably acknowledged by the server. Remember that both packet loss and re-ordering may occur in-transit.

   If there are packet losses, the client must re-transmit each lost packet at least three times but no more than 5 times before giving up and exiting with failure. If there are any losses, the client must detect and print an error message "Packet loss detected" to stderr, whenever it re-transmits a packet. If it has reached the limit of re-transmitting the same packet 5 times, it must print an error message "Reached max re-transmission limit" to stderr and exit with failure.

   The client sends at most *winsz* many packets in transit that are yet to be acknowledged. There are various reliable transfer protocols that work with a window of packets, e.g., "Go-Back-N" or "Selective Repeat." Each time the client sends out a DATA packet, it must print to **stdout** the sequence number *pktsn* of the current DATA packet being sent. Similarly, it must print the acknowledgement numbers as they are received. The line must include details of the state of the window being used to control the data transfer. These must be formatted as described in the requirements section below.

   The client is started by the following command.

   `./myclient` *`server_ip server_port mtu winsz in_file_path out_file_path`*

For example, the following two commands start a server and a client, respectively. Please note that the input or output filename arguments may be paths.

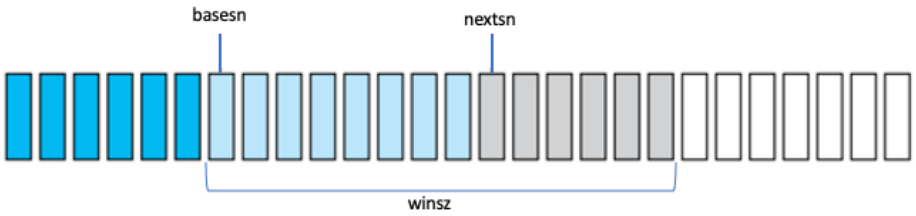
```
./myserver 9090  5

./myclient 10.10.0.1 9090 512 10 dir/testfile outfile
```

Requirements

- The reliable protocol **must not** be "Stop-and-Wait". Up to window size *winsz* of packets **must** be in transit to the server when there are available bytes of data that remain to be sent.
- The protocol between the client and server shall not specify or use any transport (TCP or UDP) port number. That is, the UDP payload between the client and the server **must not** carry any transport port number for control.
- The client log line for DATA and ACK packets must be formatted as below with these additional fields: the current *basesn*, *nextsn*, *basesn* + *winsz* used by window-based protocols. The line must show the time in the Internet date/time format (see RFC 3339). The timestamp field must be the first field, followed by DATA for data packets or ACK for acknowledgements. The log line must go to **stdout**.

  ```
  # csv format with the following fields
  # RFC 3339 time, DATA or ACK, pktsn, basesn, nextsn,  basesn+winsz
  2019-02-27T03:19:44.852Z, DATA, 7, 5, 8, 15
  2019-02-27T03:19:44.852Z, ACK, 4, 5, 8, 15
  ```



- The client **must** not send packets with payload data larger than *mtu* bytes.
- The client **must** handle input or output filename arguments that include directory paths.
- The server **must** write the final re-assembled file as communicated by the client from its command argument `out_file_path`.
- The client **must not** block indefinitely on reading from or writing to any sockets.
- The client **must not** hang if the server is down. It should print an error message "Cannot detect server" to `stderr` and exit with failure, if it has not received any packets from the server within some maximum interval (e.g., 30 seconds).
- The client must be robust and not crash if there are any socket API errors. It should instead close itself properly. A message about the error should be displayed to the `stderr`.
- The client must be able to work with input files of any size, including zero.
- The client **must** exit with zero, if successful, and non-zero otherwise.

- The server log line for dropped DATA and ACK packets must be formatted as below. The line must show the time in the Internet date/time format (see RFC 3339). The timestamp field must be the first field, followed by DATA for data packets or ACK for acknowledgements. The log line must go to **stdout**.

```
# csv format with the following fields
# RFC 3339 time, DATA or ACK, pktsn
2019-02-27T03:19:44.852Z, DATA, 8
2019-02-27T03:19:44.852Z, ACK, 5
2019-02-27T03:19:44.852Z, DROP ACK, 6
2019-02-27T03:19:44.852Z, DROP DATA, 9
```

- The server **must** handle receiving packets from one or more clients.
- The server may block to receive packets from clients, i.e., it doesn't have to do any book keeping but to wait for and process packets.
- You are free to output any needed debug messages on `stderr`.
- The documentation should describe the protocol you define to provide ordering and reliability of the data to transfer the original file correctly.
- The documentation should include a graph of the sequence numbers and the acknowledge numbers as seen over time by the client. It must be based on the output log of the client. The y-axis is the sequence number and the x-axis is the time. The sent sequence numbers would be plotted as one set and the received ack numbers would be plotted as the other set on the same graph, based on the log at the client.

Honor Code

All the code must be developed independently. All the work must be your own. You can re-use code that you have written in your own assignments in the class. Also, please do not cut-and-paste code blocks off of the Internet. Any violation of the above will result in getting a zero for the test.

What to submit?

You must submit all files in a single compressed tar file (with `tar.gz` extension). The tar file name should start with your ucsc login id, e.g., **loginid-lab3.tar.gz**. The submission must follow the following steps.

1. A README file including your name, student ID, and a list of files in the submission with a brief description. It should be in the top directory.
2. Organize the files into sub-directories (`src`, `bin`, and `doc`). The source files should be in the `src` sub-directory. The documents should be in `doc`.
3. A `Makefile` that can be used to build the client program from the required source files. It should be in the top directory. The compilation should create the two named applications, myserver and myclient.
4. You should not include the compiled program. The `Makefile` will be used to generate the executable program. The make step should put the executable program in the `bin` sub-directory.

5. Documentation of your application in plain text or pdf. *The documentation should list 5 test cases done by you to validate the functionality.* Do not include any Microsoft Word files or other formats. The documentation should describe how to use your application and the internal design, as well as any shortcomings it might have. This file should be in the `doc` sub-directory.

6. Name your submission as above **loginid-lab3.tar.gz**, where loginid is your CruzID and UCSC email username. Submit this file.

Grading

The assignment should compile and run on the campus linux timeshare (unix.lt.ucsc.edu). The grading will be done on that cluster. Each submission will be tested to make sure it works properly and can deal with errors. Grades are allocated using the guidelines below.

Basic Functionality:	60%
Dealing with Errors:	30%
Documentation:	10%

Note that 30% of the grade will be based on how well your code deals with errors. Good practice includes checking all system calls for errors and avoiding unsafe situations such as a buffer overflow.

The files must be submitted before the due date and time. Please make sure to submit on time. Late policy commences immediately after the due date/time. Reminder: Late policy deducts 10% per day from your grade.

The basic functionality is that the file created on the server is identical to the original file the client sent to the server. Pay attention to how well your code deals with errors. Good practice includes checking all system calls for errors and avoiding unsafe situations such as a buffer overflow.

The files must be submitted before the due date and time. Please make sure to submit on time.

FAQ

- What happens if the output file path does not exist? If the file path does not exist then you must create it before writing to the output file.

- The *mtu* represents the maximum size of the UDP payload. If the size given is less than the overhead of the sequencing protocol implemented, there should be an error message printed to **stderr** "Required minimum MTU is X", where X is the size of the added header in bytes by your protocol. The client should return a non-zero exit value.

- Consider if the server becomes unreachable in the middle of transferring a file. This is when there an ICMP error has been received for the server. The client should exit with failure in such a case and print "Cannot detect server" to **stderr**.

- What should be the maximum sized packet that the server can read and process? For this, let's assume a maximum size of 32k.

- Look at strftime() and gmtime() for displaying Internet formatted timestamps, i.e., RFC 3339.

Tests to consider

- The packet sizes are less than given *mtu*.

- A moderate sized file is transferred correctly with loss.

- A large binary file transfer with loss.
- Client behavior if the server is unreachable.
- Client behavior if there are too many losses.
- Server drops packets according to the specified loss rate.