CSE 156/L

Programming Assignment 4
Due: [ ]

In this project you will extend the reliable transfer to add redundancy for a file using UDP. The user of the application replicates a local file to multiple servers. Each 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 under the folder *root_folder_path*. The path name for the file may contain directory names.

   Additionally, the server takes a new command line argument *root_folder_path* for the root directory under which the files will be saved. Thus, file path is prepended with this folder name on the server.

   Multiple server can be started to run at the same time on the same or different end hosts. Each server is started by the following command giving the port number *port_number* for receiving packets. Also, it takes a percentage value *droppc*, between 0 and 100, to drop packets, either data or acknowledgment. Then, *droppc* could be used for debugging your reliability functionality. Zero means no drops, and 100 means every packet is dropped. The last argument is as given above.

   `./myserver` *`port_number droppc root_folder_path`*

2. The **client** reads a file from disk and sends it to each server over a UDP socket. In order to send the file, the client breaks the file up into *mtu*-byte sized packets. The client must be concurrent in sending packets to all of the servers using threads.

   The client expects the file to be saved by the servers correctly and reliably. The client communicates to the servers 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 a server.

   The client must exit successfully, i.e., with exit(0), when the file has been fully and reliably acknowledged by **all** of the servers. 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 five 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 **stderr** the sequence number *thissn* 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. The first argument *servn* defines how many servers to replicate to. The next argument is the server configuration file name *servaddr.conf*. The remaining arguments are as defined before.

`./myclient` *`servn servaddr.conf mtu winsz in_file_path out_file_path`*

The server configuration file has the IP address and port number for each available server, one server per line.

```
# comment line starting with #
# IP address  port
10.1.0.2 9090
10.1.0.3 9091
192.71.62.5 9092
```

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

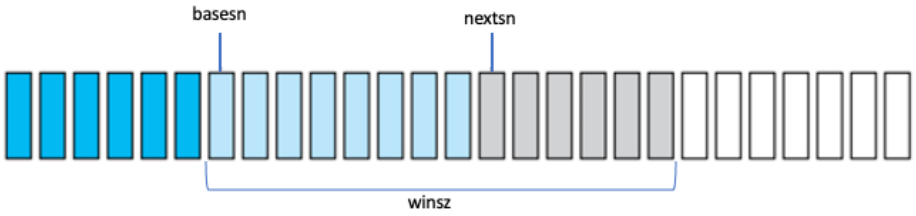
```
./myserver 9090  0  serv0_folder
./myserver 9091  0  serv1_folder
./myserver 9092  0  serv2_folder

./myclient 3  serv.conf  512 10 dir/testfile outfile
```

Requirements

- The client **must** send packets concurrently to all replica servers using only threads within one process.
- 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 start with the timestamp, and these additional fields in order: the local port *lport*, the remote IP address *rip*, and the remote port *rport*. The rest of the line follow the same format as the previous assignment.

  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
  # time, lport, rip, rport, DATA or ACK, thissn, basesn, nextsn, basesn+winsz
  2019-02-27T03:19:44.852Z, 9000, 10.1.0.2, 9090, DATA, 7, 5, 8, 15
  2019-02-27T03:19:44.852Z, 9000, 10.1.0.2, 9090, ACK, 4, 5, 8, 15
  2019-02-27T03:19:45.852Z, 9000, 10.1.0.3, 9093, DATA, 8, 6, 9, 16
  2019-02-27T03:19:45.852Z, 9000, 10.1.0.3, 9093, ACK, 8, 6, 9, 16
  ```



- 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, thissn
2019-02-27T03:19:44.852Z, DATA, 8
2019-02-27T03:19:44.852Z, ACK, 5
```

- 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.

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 files should include

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`)
3. The source files should be in the `src` sub-directory.
4. A `Makefile` that can be used to build the client program from the required source files. It should be in the top directory.
5. You should not include the compiled program. The `Makefile` will be used to generated the executable program. The make step should put the executable program in the `bin` sub-directory.
6. Documentation of your application in plain text or pdf. 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. *The documentation should list 5 test cases done by you to validate the functionality.* This file should be in the `doc` sub-directory.
7. Name your file lab4-CruzID.tar.gz where CruzID is your UCSC email username, and lab4 is the name of this assignment. 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

- 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.
- The maximum size for *winsz* is 1024 for this lab.
- The *droppc* is applied independently to received and sent packets by the server. Therefore, the effective round-trip drop rate is higher.
- Consider if the server becomes unreachable in the middle of transferring a file. 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.
- Optionally, it might helpful for debugging to log the dropped packets on the server as well. For such a log line, update the DATA field to be DATADROP and the ACK to be ACKDROP. These lines are printed by the server for the purposeful drops done by it.
- The maximum number of clients could be up to five.

Tests to consider

- The packets sizes are never larger than the given *mtu* size.
- A moderate sized file is transferred correctly with the presence of loss; e.g. 100MB.
- A large binary file transfer with presence of 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.