COM SCI 118 Spring 2019

Project 2: Simple Window-Based Reliable Data Transfer
Due date: June 7th, 23:59 PT

1 Overview

The purpose of this project is to implement a basic version of reliable data transfer protocol, including connection management and congestion control. You will design a new customized reliable data transfer protocol, akin to TCP but using UDP in C/C++ programming language. This project will deepen your understanding on how TCP protocol works and specifically how it handles packet losses.

You will implement this protocol in context of server and client applications, where client transmits a file as soon as the connection is established. We made several simplifications to the real TCP protocol and its congestion control, especially:

- You do not need to implement checksum computation and/or verification;
- You can assume there are no corruption, no reordering, and no duplication of the packets in transmit; The only unreliability you will work on is packet loss;
- You do not need to estimate RTT, nor to update RTO using RTT estimation or using Karn's algorithm to double it; You will use a fixed retransmission timer value;
- You do not need to handle parallel connections; all connections are assumed sequential.

All implementations will be written in C/C++ using BSD sockets. You are not allowed to use any third-party libraries (like Boost.Asio or similar) other than the standard libraries provided by C/C++. You are allowed to use some high-level abstractions, including C++11 extensions, for parts that are not directly related to networking, such as string parsing and multi-threading. We will also accept implementations written in C. You are encouraged to use a version control system (i.e. Git/SVN) to track the progress of your work.

2 Instructions

The project contains two parts: a server and a client.

- The server opens UDP socket and implements incoming connection management from clients. For each of the connection, the server saves all the received data from the client in a file.
- The client opens UDP socket, implements outgoing connection management, and connects to the server. Once connection is established, it sends the content of a file to the server.

Both client and server must implement reliable data transfer using unreliable UDP transport, including data sequencing, cumulative acknowledgments, and a basic version of congestion control.

2.1 Basic Protocol Specification

2.1.1 Header Format

- You have to design your own protocol header for this project. It is up to you what information you want to include in the header to achieve the functionalities required, but you will at least need a `Sequence Number` field, an `Acknowledgment Number` field, and `ACK`, `SYN`, and `FIN` flags.

- The header length needs to be exactly 12 bytes, while if your design does not use up all 12 bytes, pad zeros to make it so.

2.1.2 Requirements

- The maximum UDP packet size is 524 bytes including a header (512 bytes for the payload). You should not construct a UDP packet smaller than 524 bytes while the client has more data to send.
- The maximum Sequence Number should be 25600 and be reset to 0 whenever it reaches the maximum value.
- Packet retransmission and appropriate congestion control actions should be triggered when no data was acknowledged for more than RTO = 0.5 seconds . It is a fixed retransmission timeout, so you do not need to maintain and update this timer using Karn's algorithm, nor to estimate RTT and update it.
- If you have an ACK field, Acknowledgment Number should be set to 0 if ACK is not set.
- FIN should take logically 1 byte of the data stream (same as in TCP, see examples).
- FIN and FIN-ACK packets must not carry any payload.
- You do not need to implement checksum and/or its verification in the header.

2.2 Server Application Specification

2.2.1 Application Name and Argument

The server application MUST be compiled into a binary called server , accepting one command-line argument:

```
$ ./server <PORT>
```

The argument ⟨PORT⟩ is the port number on which server will "listen" on connections (expects UDP packets to be received). The server must accept connections coming from any network interface. For example, the command below should start the server listening on port 5000 .

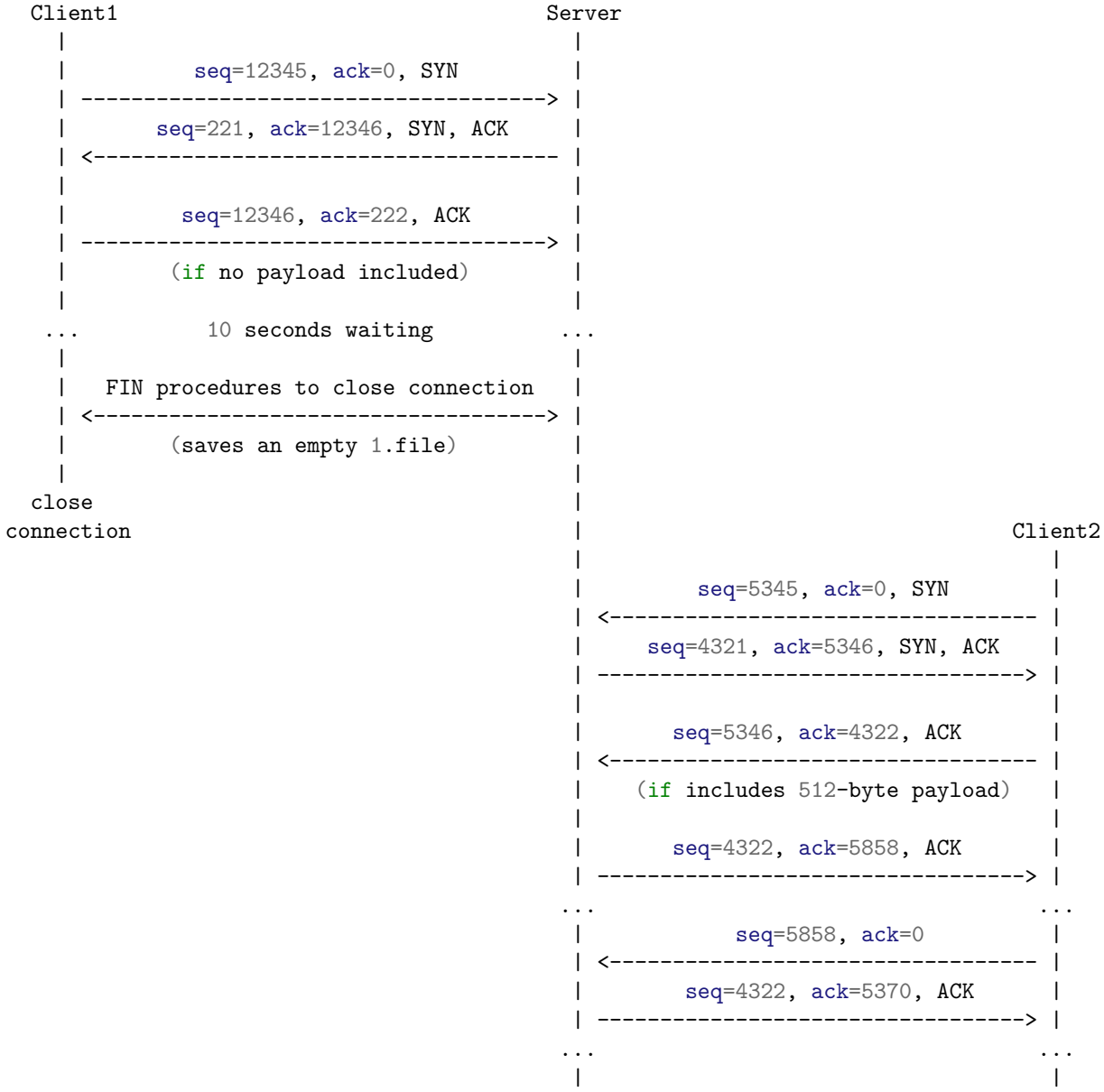
```
$ ./server 5000
```

2.2.2 Requirements

- The server must open a UDP socket on the specified port number.
- The server should gracefully process incorrect port number and exit with a non-zero error code. In addition to exiting, the server must print out on standard error (std::cerr) an error message that starts with ERROR: string.
- Once the server receives correct port number, it will remain running forever. The only condition that it exits is upon receiving SIGQUIT / SIGTERM signal. It should exit with code 0 .
- When server is running, it should be able to accept multiple clients' connection requests. It is guaranteed that a new client only initiates connection after the previous client closes its connection. In other words, connections from clients are initiated *sequentially*, so you do not need to handle parallel connections.
- The server must save all files transmitted over the established connection and name them following the order of each established connection at current directory, as ⟨CONNECTION_ORDER⟩.file . For example, 1.file , 2.file , etc. for the file received in the first connection, in the second connection, and so on.

- The server should be able to accept and save files, where each file's size could be up to 100 MB.
- Handling edge cases when receiving files:
  1. If the server is gracefully terminated (because of receiving SIGQUIT / SIGTERM signal) during an established client connection, write only INTERRUPT in the corresponding file. For example, the server should create a 2.file containing only INTERRUPT string if the server was terminated while having a connection from Client2, while Client2 had transmitted some data already.
  2. If the server establishes a connection but receives no data from the client after 10 seconds since the connection establishment, it should create an empty file according to the connection order (e.g. 1.file) at current directory, and close this connection.
  3. If the server establishes a connection but receives no data from the client after 10 seconds since the last successful data packet transmit, it should save all the data received (e.g. into 1.file) at current directory, and close this connection.

See below for an example illustration of clients establishing connections and transmit files to the server.



2.3 Client Application Specification

2.3.1 Application Name and Argument

The client application MUST be compiled into a binary called client, accepting three command-line arguments:

```
$ ./client <HOSTNAME-OR-IP> <PORT> <FILENAME>
```

Their meanings are:

- ⟨HOSTNAME-OR-IP⟩: hostname or IP address of the server to connect (send UDP datagrams).
- ⟨PORT⟩: port number of the server to connect (send UDP datagrams).
- ⟨FILENAME⟩: name of the file to transfer to the server after the connection is established.

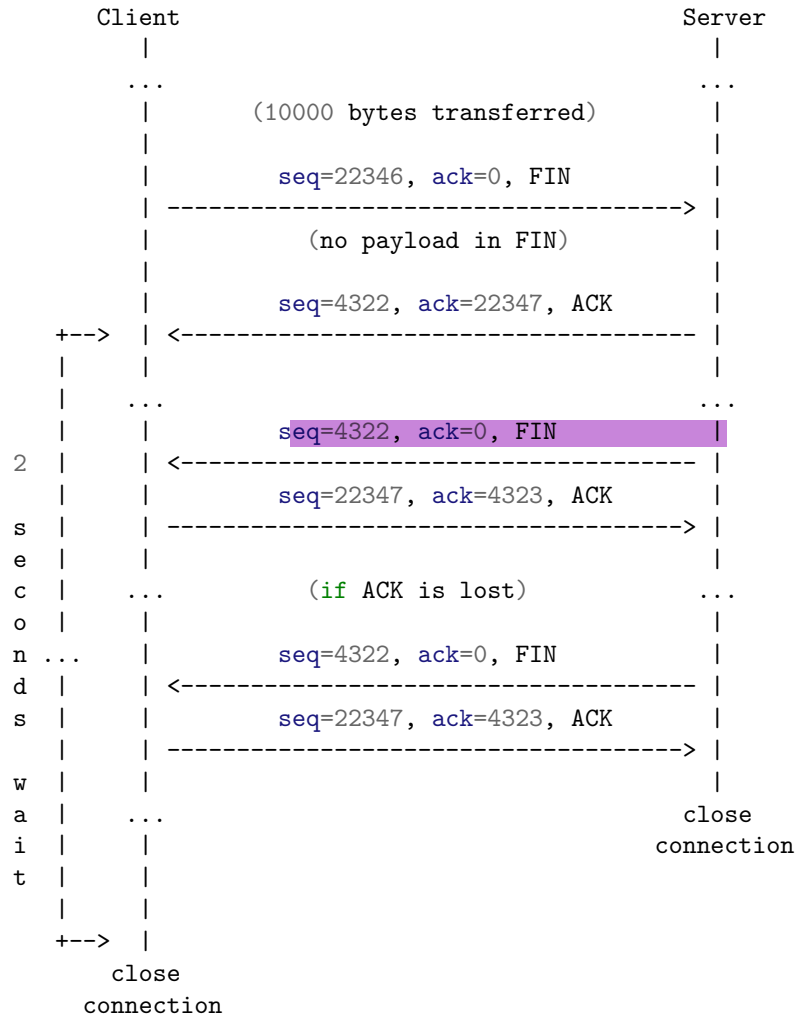
For example, the command below should result in connection to a server on the same machine listening on port 5000 and transfer content of test.txt:

```
$ ./client localhost 5000 test.txt
```

2.3.2 Requirements

- The client must open a UDP socket and initiate 3-way handshake to the specified hostname/IP and port.
  - Send UDP packet with SYN flag set, Sequence Number initialized using a random number not exceeding MaxSeqNum = 25600, and Acknowledgment Number set to 0.
  - Expect response from server with SYN and ACK flags set.
  - Send UDP packet with ACK flag including the first part of the specified file.
- The client should gracefully process incorrect hostname and port number and exist with a non-zero exit code (you can assume that the specified file is always correct). In addition to exiting, the client must print out on standard error (std::cerr) an error message that starts with ERROR: string.
- Client should support transfer of files that are up to 100 MB.
- Whenever client receives no packets from server for more than 10 seconds, it should abort the connection (close socket and exit with non-zero code).
- After file is successfully transferred (all bytes acknowledged), the client should gracefully terminate the connection following these steps:
  - Send UDP packet with FIN flag set.
  - Expect packet with ACK flag.
  - Wait for 2 seconds for incoming packet(s) with FIN flag.
  - During the wait, respond to each incoming FIN with an ACK packet; drop any other non-FIN packet.
  - Close connection and terminate with code 0.

  See below diagram for an example illustration.



2.4 Congestion Control Requirements

Client and server are required to implement TCP congestion window maintenance logic following RFC5681. The congestion control will be mostly done at the client, since the client sends data to the server. TCP Slow Start and Congestion Avoidance are mandatory to implement in this project. Fast Recovery/Fast Retransmit with 3rd-duplicate ACK loss detection is left as optional part with bonus. You may find an electronic version of RFC5681 posted on CCLE or at https://tools.ietf.org/html/rfc5681.

2.4.1 Mandatory: Slow Start and Congestion Avoidance

You should read the Slow Start and Congestion Avoidance description specified in Section 3.1 of RFC5681. We made some simplifications to the RFC and use our own parameters, see summary below.

- Initial and minimum congestion window size (cwnd) should be 512 . The maximum congestion window size (cwnd) should be 10240 .
- Initial slow-start threshold (ssthresh) should be 5120 .
- After connection is established, the client should send up to the initial cwnd = 512 bytes of data without starting the wait for acknowledgments.
- After each ACK is received:

(**Slow Start**) If `cwnd < ssthresh`: `cwnd += 512`

(**Congestion Avoidance**) If `cwnd ≥ ssthresh`: `cwnd += (512 * 512) / cwnd`

- After timeout, set `ssthresh := max (cwnd / 2, 1024)`, set `cwnd := 512`, and retransmit data after the last acknowledged byte. After retransmission, **Slow Start** is performed.
- For each valid packet of the connection (except packets with only `ACK` flag and empty payload), the server responds with an `ACK` packet, which includes the next expected in-sequence byte to receive (cumulative acknowledgment).

2.4.2 Optional: Fast Retransmit/Fast Recovery (Extra Credit)

It is optional to implement TCP Tahoe (with Fast Retransmit) or TCP Reno (with Fast Retransmit/Fast Recovery). If you implement only Fast Retransmit, you can get up to 5% extra credits of this project. If you also implement Fast Retransmit/Fast Recovery, you can get up to 10% extra credits. For details of Fast Retransmit/Fast Recovery, you can refer to descriptions specified in Section 3.2 of RFC5681. We made some simplifications to the RFC and use our own parameters, see summary below.

- (**Fast Retransmit**) After receiving 3 duplicate ACKs in a row, set `ssthresh := max (cwnd / 2, 1024)`, set `cwnd := ssthresh + 1536`, and retransmit data after the last acknowledged byte. After retransmission, set `cwnd := ssthresh`, and enter **Congestion Avoidance**.
- (**Fast Recovery**) After receiving 3 duplicate ACKs in a row, set `ssthresh := max (cwnd / 2, 1024)`, set `cwnd := ssthresh + 1536`, and retransmit data after the last acknowledged byte. Each time another duplicate ACK arrives, set `cwnd += 512`. Then, send a new data segment if allowed by the value of cwnd. Upon a new (i.e., non-duplicate) ACK, set `cwnd := ssthresh`, and enter **Congestion Avoidance**.

2.5 Error Handling: Loss

We will test your reliable transport protocol in unreliable conditions. However, we will only test under the packet loss. You can safely assume there are no corruption, no reordering, and no duplication of the packets in transmit.

Although using UDP does not ensure reliability of data transfer, the actual rate of packet loss or corruption in LAN may be too low to test your applications. Therefore, we are going to emulate packet loss by using the `tc` command in Linux. Note that you need to have root permission to run `tc` command, and it has to be applied on the proper network interface. There are two requirements for the network loss emulation using `tc`:

- The file transmission should be completed successfully, unless the packet loss rate is set to 100%.
- The timer on both the client and the server should work correctly to retransmit the lost packet(s).

The following commands are listed here for your reference to adjust parameters of the emulation. More examples can be found in `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

1. To check the current parameters for a given network interface (e.g. `eth0`):

   `tc qdisc show dev eth0`

2. If network emulation has not yet been setup or have been deleted, you can add a configuration called root it to a given interface, with 10% loss without delay emulation for example:

   `tc qdisc add dev eth0 root netem loss 10%`

3. To change current parameters to loss rate of 20% and delay 100ms:
   `tc qdisc change dev eth0 root netem loss 20% delay 100ms`
4. To delete the network emulation:
   `tc qdisc del dev eth0 root`

2.6 Common Printout Format Requirements

The following output MUST be written to standard output (`std::cout`) in the **EXACT** format defined. If any other information needs to be shown, it MUST be written to standard error (`std::cerr`)

- Packet received:
  `RECV ⟨SeqNum⟩ ⟨AckNum⟩ ⟨cwnd⟩ ⟨ssthresh⟩ [ACK] [SYN] [FIN]`
- Packet sent:
  `SEND ⟨SeqNum⟩ ⟨AckNum⟩ ⟨cwnd⟩ ⟨ssthresh⟩ [ACK] [SYN] [FIN] [DUP]`

Note the convention:

- `[xx]` means that `xx` string is optional. However, if the packet belongs to one or more of the following cases you cannot omit the corresponding printout:
  - Acknowledgment packet: `[ACK]`
  - SYN, or SYN-ACK packet: `[SYN]`
  - FIN, or FIN-ACK packet: `[FIN]`
  - Duplicate ACK packet: `[DUP]`
- `⟨yy⟩` means that value of `yy` variable should appear on the output (in decimal). Note, since the server does not need to implement congestion control, you should print `0 0` for `⟨cwnd⟩ ⟨ssthresh⟩` fields.

Please make sure the printout of your client and server matches the format. You will get no credit if the format is not followed exactly and our parsing script cannot automatically parse it. The parsing script will be released to you at the end of the 8th week for you to check the format.

3 Hints

The best way to approach this project is in incremental steps. Do not try to implement all of the functionality at once.

- First, assume there is no packet loss. Just have the client send a packet, and the server respond with an ACK, and so on.
- Second, introduce a large file transmission. This means you must divide the file into multiple packets and transmit the packets based on the current window size.
- Third, introduce packet loss. Now you have to add a timer on each sent and un-ACKed packet. If a timer times out, the corresponding (lost) packet should be retransmitted for the successful file transmission.

The credit of your project is distributed among the required functions. If you only finish part of the requirements, we still give you partial credit. So please do the project incrementally.

4 Due Date and Demo

4.1 Project Due

This project is due June 7th, 2019 at **23:59 PT**. You will submit an electronic copy of the project on CCLE before due. See the Project Submission section for details on submission requirements. After submission, you are required to demo your program on 6/11 and 6/12 (tentative schedule).

4.2 Demo Requirements

1. Sign up for demo: TA will distribute the demo sign-up sheet in Week 8.
2. Preparing for the demo: You need to **prepare at least 3 slides** to present. One slide for your design and considerations, one slide for experiences you gained,and one slide for lesson learn from project and suggestions.
3. Demo day:
   - Demo time is 15 minutes for each group.
   - TAs will ask you to demo the function step-by-step on an up-to-date Linux system.
   - You will use `make` to compile your programs, run your programs to deliver a test file from the client side to the server side. Your program should print out operations according to the defined format, and you should also be able to explain the delivery process. TAs may ask you to use different values for `tc` command to test your program. TAs will ask you to compare the received files with the sent ones using the Linux program `diff`.
   - After the demo, TAs will also ask you to walk through the slides. TAs may ask you a few questions and you need to answer them. All question will be related to your project implementation. Q&A and slides are counted towards the total credit of this project.

5 Project Submission

NOTE: Late submission is not allowed, WITHOUT EXCEPTION.

1. Put all your files into a directory called `project2_UID1_UID2`, where UIDs are your UCLA ID numbers. Compress the directory and submit a `project2_UID1_UID2.zip` to CCLE course website before the deadline.
2. The `project2_UID1_UID2.zip` should contain the following files:
   - Source codes (can be multiple files);
   - A `Makefile`; TAs will only type `make` to compile your code, make sure your Makefile works on Linux system.
   - README file that includes your group information. The README should contain:
     - Students name, and UID.
     - Brief description about how you divide the workload.
   - A PDF format report file (.pdf) no more than 5 pages. The report should contain:
     - Student names and UIDs at the very beginning (**2 students per-group**).
     - Implementation description (header format, messages, timeouts, window-based protocol etc).
     - Difficulties that you faced and how you solved them.
3. The first person in each group (according to the group sign-up sheet) submits the project files to CCLE.