Project 2: Reliable Transport Protocol over UDP

Overview

In this project you will need to implement a basic version of reliable data transfer protocol, including connection establishment and congestion control. You will implement this protocol in context of server and client applications, where client transmits a file as soon as the connection is established (same as in project 1).

All implementations should be written in C++ using BSD sockets. No high-level network-layer abstractions (like Boost.Asio or similar) are allowed in this project. 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, multi-threading. We will also accept implementations written in C, however use of C++ is preferred.

The objective of this project is to deepen your understanding on how TCP protocol works and specifically how it handles packet losses and reordering.

You are required to use `git` to track the progress of your work. The project can receive a full grade only if the submission includes git history no shorter than 3 commits FROM ALL PARTICIPANTS OF YOUR GROUP. If commit history includes commits made only by one group member, other group members will receive no credit.

You are encouraged to host your code in private repositories on GitHub, GitLab, or other places. At the same time, you are PROHIBITED to make your code for the class project public during the class or any time after the class. If you do so, you will be violating academic honestly policy that you have signed, as well as the student code of conduct and be subject to serious sanctions.

Task Description

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 acknowledgements, and basic version of the congestion control.

Protocol Specification

Header Format

- The payload of each UDP packet sent by server and client MUST start with the following `12-byte` header. All fields are in network order (most significant bit first):
-
```
  0                   1                   2                   3
   0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                         Sequence Number                       |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |                      Acknowledgment Number                    |
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  |         Connection ID         |         Not Used        |A|S|F|
  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Where
  - `Sequence Number` (32 bits): The sequence number of the first data octet in this packet (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.
  - The sequence number is given in the unit of bytes.

    - `Acknowledgement Number` (32 bits): If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.
    - The acknowledgement number is given in the unit of bytes.
    - `Connection ID` (16 bits): A number representing connection identifier.
    - `Not Used` (13 bits): Must be zero.
    - `A` (ACK, 1 bit): Indicates that there the value of `Acknowledgment Number` field is valid
    - `S` (SYN, 1 bit): Synchronize sequence numbers (connection establishment)
    - `F` (FIN, 1 bit): No more data from sender (connection termination)

Requirements

- The maximum UDP packet size is `524 bytes` including a header (`512 bytes` for the payload)
- The maximum sequence and acknowledgment number should be `102400` and be reset to zero whenever it reaches the maximum value.
- Packet retransmission (and appropriate congestion control actions) should be triggered when no data was acknowledged for more than `0.5 seconds` (fixed retransmission timeout).
- Initial and minimum congestion window size (`CWND`) should be `512`
- The maximum congestion window size (`CWND`) is `51200`
- Initial slow-start threshold (`SS-THRESH`) should be `10000`
- If `ACK` field is not set, `Acknowledgment Number` field should be set to 0
- `FIN` should take logically one byte of the data stream (same as in TCP, see examples)
- `FIN` and `FIN | ACK` packets must not carry any payload

Server Application Specification

The server application MUST be compiled into `server` binary, accepting two command-line arguments:

```
$ server <PORT> <FILE-DIR>
```

- `<PORT>`: port number on which server will "listen" on connections (expects UDP packets to be received). The server must accept connections coming from any interface.
- `<FILE-DIR>`: directory name where to save the received files.

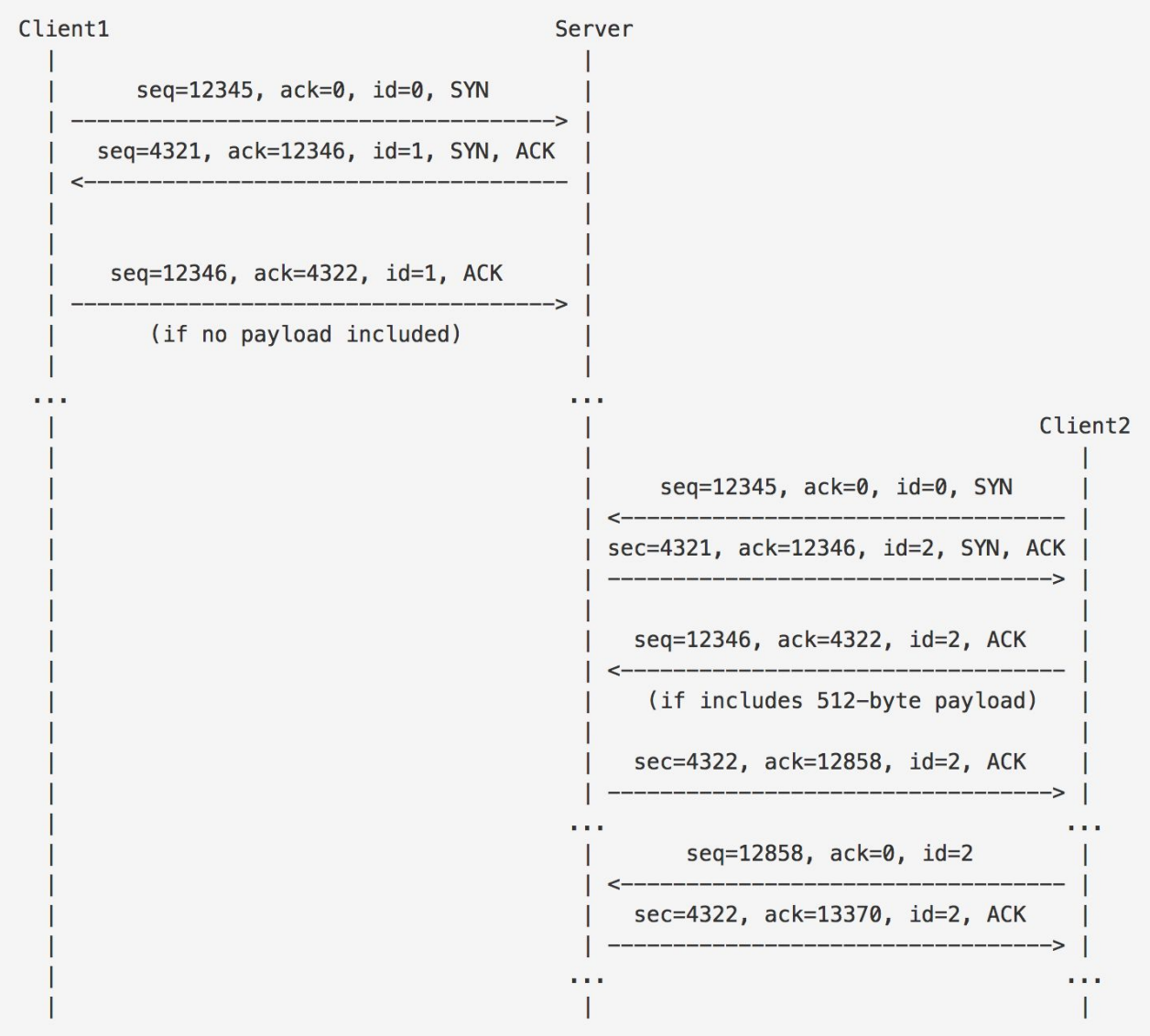
For example, the command below should start the server listening on port `5000` and saving received files in the directory `/save`.

```
$ ./server 5000 /save
```

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 (you can assume that the folder is always correct). In addition to exit, the server must print out on standard error (`std::cerr`) an error message that starts with `ERROR:` string.
- The server should exit with code zero when receiving `SIGQUIT`/`SIGTERM` signal
- The server must count all established connections (1 for the first connect, 2 for the second, etc.). The received file over the connection must be saved to `<FILE-DIR>/<CONNECTION-ID>.file` file (e.g., `/save/1.file`, `/save/2.file`, etc.). If the client doesn't send any data during gracefully terminated connection, the server should create an empty file with the name that corresponds to the connection number.
- The server should be able to accept and process multiple connection from clients at the same time
    - After receiving packet with `SYN` flag, the server should create state for the `connection ID` and proceed with 3-way handshake for this connection. Server should use `4321` as initial sequence number.

    - After receiving packet without `SYN` flag, the server should lookup the connection using `connection ID` and proceed with appropriate action for the connection.



- The server must assume error if no data received from the client for over 10 seconds. It should abort the connection and write a single `ERROR` string into the corresponding file.
- The server should be able to accept and save files up to 100 MiB

Client Application Specification

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

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

- `<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.

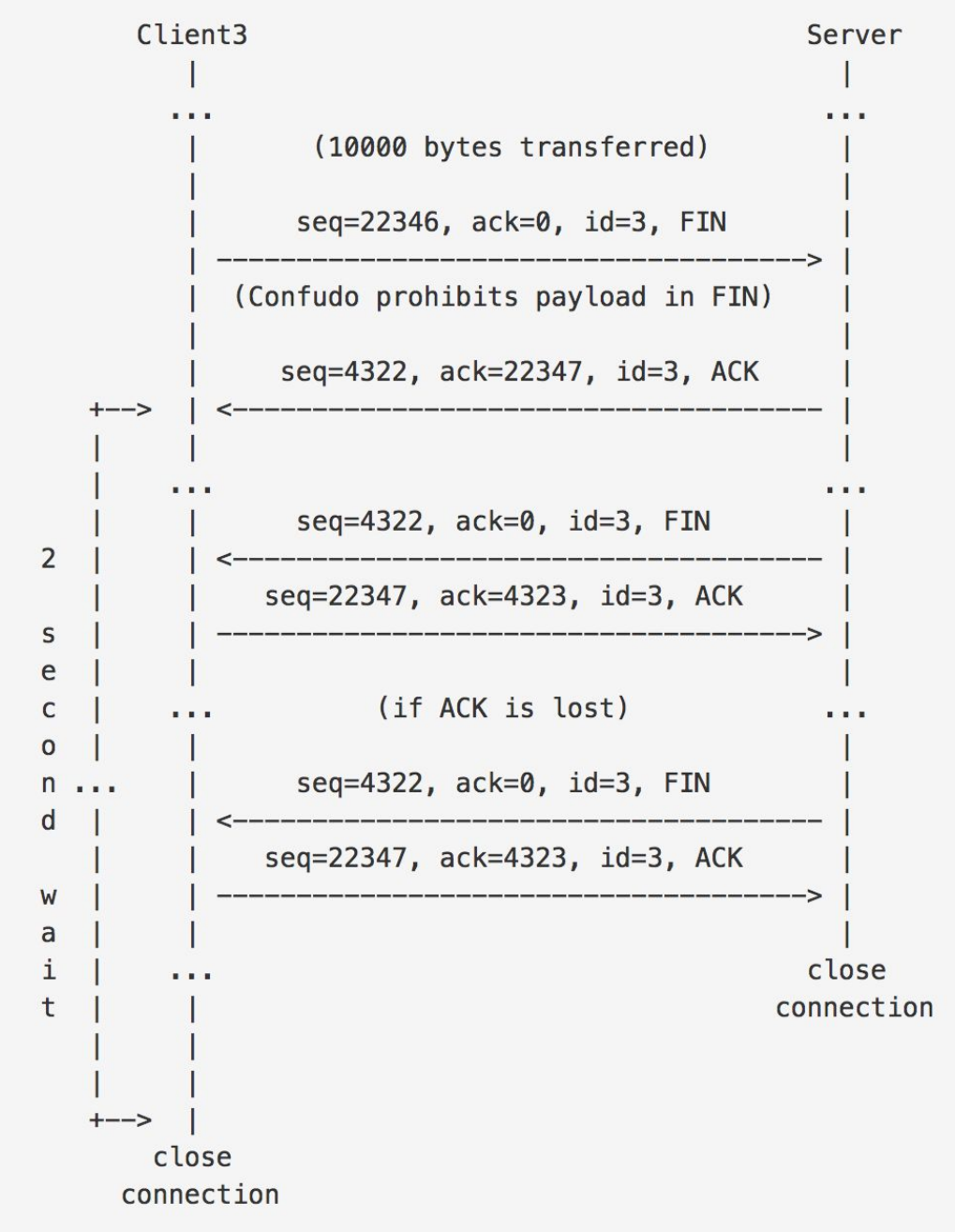
For example, the command below should result in connection to a server on the same machine listening on port 5000 and transfer content of `file.txt`:

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

Requirements:

- The client must open a UDP socket and initiate 3-way handshake to the specified hostname/ip and port
    - Send UDP packet `src-ip=DEFAULT, src-port=DEFAULT, dst-ip=HOSTNAME-OR-IP, dst-port=PORT` with `SYN` flag set, `Connection ID` initialized to `0`, `Sequence Number` set to `12345`, and `Acknowledgement Number` set to `0`
    - Expect response from server with `SYN | ACK` flags. The client must record the returned `Connection ID` and use it in all subsequent packets.
    - Send UDP packet with `ACK` flag without including any payload. Then send another UDP packet that has 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 exit, the client must print out on standard error (`std::cerr`) an error message that starts with `ERROR:` string.
- After file is successfully transferred file (all bytes acknowledged), the client should gracefully terminate the connection
    - Send UDP packet with `FIN` flag set

- Expect packet with `ACK` flag
- Wait for `2 seconds` for incoming packet(s) with `FIN` flag (`FIN-WAIT`)
- During the wait, respond to each incoming `FIN` with an `ACK` packet; drop any other non-`FIN` packet.
- Close connection and terminate with code zero.



- Client should support transfer of files that are up to `100 MiB`.

- 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)

Congestion Control Requirements

Client and server (mostly client) is required to implement TCP Tahoe congestion window maintenance logic (without 3-dup ACK loss detection):

- After connection is established, the client should send up to `CWND` bytes of data without starting the wait for acknowledgements
- After each `ACK` is received
  - (*Slow start*) If `CWND < SS-THRESH`: `CWND` += 512
  - (*Congestion Avoidance*) If `CWND >= SS-THRESH`: `CWND` += (512 * 512) / CWND
- After timeout, set `SS-THRESH -> CWND / 2`, `CWND -> 512`, and retransmit data after the last acknowledged byte.
- 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 acknowledgement).

Common Requirements

- The following output MUST be written to standard output (`std::cout`) in the exact format defined. You will get no credit if the format is not followed exactly and our test script cannot automatically parse it. If any other information needs to be shown, it MUST be written to standard error (`std::cerr`)
- **Packet received:**
  - `Client: "RECV" <Sequence Number> <Acknowledgement Number> <Connection ID> <CWND> <SS-THRESH> ["ACK"] ["SYN"] ["FIN"]`
  - `Server: "RECV" <Sequence Number> <Acknowledgement Number> <Connection ID> ["ACK"] ["SYN"] ["FIN"]`
    - `[xx]` means that `xx` value is optional
    - `"xx"` means `xx` string should appear on the output
    - `<yy>` means that value of `yy` variable should appear on the output

- **If received packet is dropped (e.g., unknown `connection ID`):**

    - `"DROP" <Sequence Number> <Acknowledgement Number> <Connection ID> ["ACK"] ["SYN"] ["FIN"]`

- **Packet sent:**
    - **Client:** `"SEND" <Sequence Number> <Acknowledgement Number> <Connection ID> <CWND> <SS-THRESH> ["ACK"] ["SYN"] ["FIN"] ["DUP"]`
    - **Server:** `"SEND" <Sequence Number> <Acknowledgement Number> <Connection ID> ["ACK"] ["SYN"] ["FIN"] ["DUP"]`

A Few 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 server send a packet, the receiver 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 congestion window size.
- Third, introduce packet loss. Now you have to add a timer at the first sent and unacked packet. There should be one timeout whenever data segments are sent out. Also congestion control features should be implemented for the successful file transmission.

Submission Hints

Checklist:

- Are your cout statements correct? You should not have any other `std::cout` statements besides the one in the spec, they should be in the right format, printed at the right time with the right optional outputs, and with the correct `cwnd` and `ssthresh` values.
    - No other cout statements, others should be to `std::cerr`
    - `std::cout` statements are in the exact correct format
    - You are adding in the `[ACK]` `[SYN]` `[FIN]` `[DUP]` optional output correctly. DUP in particular is easy to miss.
    - Are you printing a statement every time you should be? Is there a case where you receive something but don't print it, etc.
- Are your timers correct?
    - Is your 0.5 second retransmission working properly?
    - Are you waiting the 2 seconds before closing the client?
    - Do you have the 10 second timer working properly?
    - Is there a place where you should be updating your timer threshold (e.g., restarting when it begins to countdown) but you aren't?
- Is there a memory error? We had a couple and are not sure if this error is directly related but fixing these will help you chase down this error more easily. Most of these should be caught

    - Are you ever dereferencing a pointer that isn't pointing at anything?
    - Do you properly free your memory for all of your allocated objects?
    - Do you ever invalidate an iterator by mistake?
    - Do you somehow allocate so much memory that the kernel has to send a SIGKILL to terminate your program? This really shouldn't happen realistically but you never know…
- Do you ever somehow go into an infinite loop by accident?

Emulating packet loss

If are using the Vagrantfile provided in project-2 skeleton, you can automatically instantiate two virtual machines that are connected to each other using a private network (`enp0s8` interface on each). You can also run preinstalled `/set-loss.sh` script to enable emulation of 10% loss and delay of 20ms in each direction (need to run on each VM separately).

You can use the following commands to adjust parameters of the emulation:

- To check the current parameters for the private network (`enp0s8`)
- `tc qdisc show dev enp0s8`

- To change current parameters to loss rate of 20% and delay 100ms:
- `tc qdisc change dev enp0s8 root netem loss 20% delay 100ms`

- To delete the network emulation:
- `tc qdisc del dev enp0s8 root`

- If network emulation hasn't yet setup or you have deleted it, you can add it to, e.g., 10% loss without delay emulation:
- `tc qdisc add dev enp0s8 root netem loss 10%`

- You can also change network emulation to re-order packets. The command below makes 4 out of every 5 packets (1-4, 6-9, …) to be delayed by 100ms, while every 5th packet (, 10, 15, …) will be sent immediately:
- `tc qdisc change dev enp0s8 root netem reorder 100% gap 5 delay 100ms`

  `# or if you're just adding the rule`

  `# tc qdisc add dev enp0s8 root netem reorder 100% gap 5 delay`

```
100ms
```

- More examples can be found in [Network Emulation tutorial]

If you are not using the provided Vagrant, you should adjust network interface to one that matches your experiments.

Environment Setup

The best way to guarantee full credit for the project is to do project development using a Ubuntu 16.04-based virtual machine.

You can easily create an image in your favourite virtualization engine (VirtualBox, VMware) using the Vagrant platform and steps outlined below.

Set up Vagrant and create VM instance

Note that all example commands are executed on the host machine (your laptop), e.g., in Terminal.app (or iTerm2.app) on OS X, cmd in Windows, and console or xterm on Linux. After the last step (`vagrant ssh`) you will get inside the virtual machine and can compile your code there.

- Download and install your favourite virtualization engine, e.g., VirtualBox
- Download and install Vagrant tools for your platform
- Set up project and VM instances
    - Clone project template
    - ```
      git clone https://github.com/CS118W19/winter19-project2.git ~/cs118-proj2
      cd ~/cs118-proj2
      ```
    - Initialize VMs, one to run the client app and the other to run the server app
    - ```
      vagrant up

      # Or you can set up them individually
      # vagrant up client
      # vagrant up server
      ```
    - Do not start VM instance manually from VirtualBox GUI, otherwise you may have various problems (connection error, connection timeout, missing packages, etc.)

    - To establish an SSH session to the created VM, run
    - To ssh to the client VM
    - `vagrant ssh`

      ```
      # or vagrant ssh client
      ```

    - To ssh to the server VM
    - `vagrant ssh server`

- If you are using Putty on Windows platform, `vagrant ssh` (`vagrant ssh server`) will return information regarding the IP address and the port to connect to your virtual machine.
- Work on your project
- All files in `~/cs118-proj2` folder on the host machine will be automatically synchronized with `/vagrant` folder on both virtual machines. For example, to compile your code, you can run the following commands:
- ```
  vagrant ssh
  cd /vagrant
  make
  ```

Notes

- If you want to open another SSH session, just open another terminal and run `vagrant ssh` (or create a new Putty session).
- The client and server VMs are connected using the private network `10.0.0.0/24` (`enp0s8`)
    - client's IP address: `10.0.0.2`
    - server's IP address: `10.0.0.1`
- Note that these addresses do not mean that your server and client must support only this environment. The server and client should be able to work a linux host with any other IP addresses as well.
- If you are using Windows, read this article to help yourself set up the environment.
- The code base contains the basic `Makefile` and two empty files `server.cpp` and `client.cpp`.
- ```
  $ vagrant ssh
  vagrant@client:~$ cd /vagrant
  vagrant@client:/vagrant$ ls -a
  ```

```
  .  ..  client.cpp  confundo.lua  .gitignore  Makefile  README.md
server.cpp  .vagrant  Vagrantfile
```

- You are now free to add more files and modify the Makefile to make the `server` and `client` full-fledged implementation.

Submission Requirements

To submit your project, you need to prepare:

1. A `README.md` file placed in your code that includes:
    - Name and UID of each team member (up to 3 members in one team) and the contribution of each member
    - The high level design of your server and client
    - The problems you ran into and how you solved the problems
    - List of any additional libraries used
    - Acknowledgement of any online tutorials or code example (except class website) you have been using.
2. If you need additional dependencies for your project, you must update Vagrant file.
3. Update Makefile, updating UID variable to the list of UIDs for all members of the group separated by underscore. For example,
4. `UID=123456789_987654321_111223333`

5. All your source code, `Makefile`, `README.md`, `Vagrantfile`, `confundo.lua`, and `.git` folder with your git repository history as a `.tar.gz` archive (and any files from extra credit part).
6. To create the submission, use the provided Makefile in the skeleton project. Just update `Makefile` to include your UCLA ID and then just type
7. `make tarball`

8. Then submit the resulting (e.g., `123456789_987654321_111223333.tar.gz`) archive to Gradescope.

Before submission, please make sure:

1. Your code compiles
2. Client and server conforms to the specification
3. `.tar.gz` archive does not contain temporary or other unnecessary files. We will automatically deduct points otherwise.

> Submissions that do not follow these requirements will not get any credit.

Grading

Your code will be first checked by a software plagiarism detecting tool. If we find any plagiarism, you will not get any credit.

Your code will then be automatically tested in some testing scenarios. If your code can pass all our automated test cases, you will get the full credit.

We may test your server against a “standard” implementation of the client, your client against a “standard” server, as well as your client against your server. Projects receive full credit only if all these checks are passed.

Grading Criteria

1. **Miscellaneous tests**

   1.1. (2.5 pts) At least 3 git commits (at least one from each group member)

   1.2. (1.25 pts) Client handles incorrect hostname

   1.3. (1.25 pts) Client handles incorrect port

   1.4. (2.5 pts) Server handles incorrect port number

2. **Client tests**

   2.1. (2.5 pts) Client initiates three-way handshake by sending a SYN packet with correct values in its header

   2.2. (2.5 pts) Client has correct initial values for CWND, SS-THRESH, and Sequence Number

   2.3. (5 pts) Data segments that client sends are not exceeding 512 bytes and on average larger than 500 bytes (for 1~MByte file)

   2.4. (2.5 pts) Client should reset its sequence number to zero when the sequence number reaches the maximum value

   2.5. (2.5 pts) Client sends a FIN packet after transmitting a file

   2.6. (2.5 pts) After finishing connection, client responds with ACK for incoming FINs for 2 seconds, dropping packets for this connection afterwards

   2.7. (5 pts) Client successfully transmits a small file

2.8. (5 pts) Client aborts the connection if no incoming packets for more than 10 seconds

2.9. (5 pts) Client properly increases congestion window size in slow start phase

2.10. (5 pts) Client properly increases congestion window size in congestion avoidance phase

2.11. (5 pts) Client detects and retransmits lost data segments

2.12. (5 pts) Client sets SS-THRESH and CWND values properly after timeout

3. **Server tests**

3.1. (2.5 pts) Server responses with SYN-ACK packet with correct connection ID

3.2. (2.5 pts) Server has correct initial values for CWND, SS-THRESH, and Sequence Number

3.3. (5 pts) Server responds with ACK packets, which include the next expected in-sequence byte to receive (cumulative ACK)

3.4. (5 pts) Server able to receive a large file (10 MiB bytes) and save it in 1.file without delay, loss, and reorder

3.5. (5 pts) Server able to receive a large file (10 MiB bytes) and save it in 1.file over lossy and large delay network

- We will use `tc` command with reorder, gap, and delay to generate reordered and delayed packets
- Test need to pass under different packet delays (50 ms ~ 100 ms) and reorder rates
- We will not test timeout on the server side

3.6. (5 pts) Server able to receive 10 small files (1 MiB bytes) in 1.file, 2.file, ..., 10.file without delay, loss, and reorder (sequentially)

3.7. (5 pts) Server able to receive 10 small files (1 MiB bytes) in 1.file, 2.file, ..., 10.file without delay, loss, and reorder (in parallel)

3.8. (7.5 pts) Server able to receive 10 small files (1 MiB bytes) in 1.file, 2.file, ..., 10.file over lossy and large delay network (sequentially)

- We will use `tc` command with loss and delay to generate the lossy and large delay network
- Test need to pass under different packet delays (50 ms ~ 100 ms) and packet loss rates (1% ~ 10%)
- We will not test timeout on the server side

3.9. (7.5 pts) Server able to receive 10 small files (1 MiB bytes) in 1.file, 2.file, ..., 10.file over lossy and large delay network (in parallel)

- We will use `tc` command with loss and delay to generate the lossy and large delay network

- Test need to pass under different packet delays (50 ms ~ 100 ms) and packet loss rates (1% ~ 10%)
- We will not test timeout on the server side