VE489 Project

VE489 staff

May 2020

Due date: August 2, 8 a.m.
You have to do this project **individually**. You have to abide by JI's honor code.

1 Mininet Experiments (2 weeks recommended)

1.1 Mininet Tutorial

Please do this part as soon as possible.

Mininet should be installed in a Ubuntu Linux system, so you need a Ubuntu virtual machine to do the experiments. An older version of Ubuntu is recommended here: http://old-releases.ubuntu.com/releases/14.04.1/.

You also need a hypervisor to run virtual machines, you are recommended to install the free Oracle VirtualBox. To create a virtual machine, open VirtualBox, click "New". Input a "name" for your VM, select "type" as "Linux" and "version" as "Ubuntu 64 bit". Keep clicking "continue" or "create" until a VM is created.

Now click "start" on the VM you just created. A window will be prompted, select the Ubuntu release you just downloaded. Keep clicking "continue" and set up a user name and a password. After that, restart the machine. You may restart by directly powering off and starting again. Once you log in, please reject to upgrade your system.

You may install Mininet now. Open up a terminal and run `sudo apt-get install mininet`. After installation, first test if Mininet is working correctly. Launch a simple Mininet network by `sudo mn`. Some errors may prompt first time you launch it, you may try launching again or run `sudo mn -c` to cleanup first. After successfully lauching the Mininet, a network with two hosts (h1 and h2) and one switch (s1) will be created and the Mininet CLI will be opened. To test connectivity between h1 and h2 in the Mininet CLI, run `h1 ping h2`. You should see the round-trip time clearly, enter `ctrl C` to stop ping. Enter `exit` or `ctrl D` to quit Mininet CLI.

To get familiar with Mininet, please follow the walkthrough on Mininet website and pay attention to part 1 and 3 (excluding the Wireshark part).

The following are some useful commands for this project. Mininet can define customized networks with link bandwidth and propagation delay specified. The customized network is written in Python, you do not need to worry how to write that. To launch a customized network, run `sudo python p1_topo.py` and the Mininet CLI will be opened.

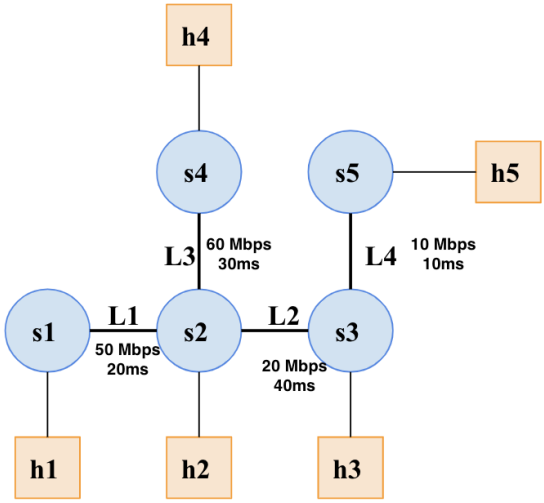
How to know the IP of a host? `h1 ifconfig -a` will display information of all interfaces of `h1`, including its IP and MAC address, you may ignore the loopback interface. However, for convenience, in the `p1_topo.py`, the host h_i will have an IP 10.0.0.i

How do I run a program on one host? `xterm h1` will open a terminal on host h1. Then in the popped out terminal, you can run program on the host. Note that if you want to run ping in the xterm, do not use `h1 ping h2`. Instead, use `ping 10.0.0.2` because outside the Mininet CLI you have to specify the destination IP address for ping.

Note: please do this as soon as possible, you have to make sure the Mininet works to work on the whole project. If you encounter any problems, please first search online and come to office hours if you fail to solve it. If you use the provided version of Ubuntu and VirtualBox, there should be no problem.

1.2 Simple Experiments

You are provided with a customized Mininet network topology in `part1/p1_topo.py`, launch it by `sudo python p1_topo.py`. If you use the provided VM and see errors like "please kill controller...", the common fix is `sudo mn -c`, specifically you may also try `sudo pkill controller`. The parameters of the topology are shown below. h_i stands for a host, and s_i stands for a switch. The links that connect switches (L_i's) have its parameter near it, e.g., L_1 has a delay of 20ms and a bandwidth of 50Mbps.



You are asked to do the following measurements.

A. **Link latency using ping.** You will measure the latency between the pair (h_1, h_2) and (h_3, h_5). Let one host ping 10 times and report the round-trip time (RTT). Which link is used? Is RTT close to 2 times link latency?
Note: you may use `ping -c 10 <dest-ip>` to ping only 10 times.

B. **Path latency using ping.** Measure the latency between (h_1, h_5) and (h_3, h_4). Let one host ping 10 times and report the round-trip delay. Compute the theoretical delays by the given parameters and compare with your results.

C. **Link bandwidth using `iperf`.** `iperf` is a tool that can measure the throughput or bandwidth between two hosts. You will measure the bandwidth between hosts (h_1, h_2) and (h_3, h_5). Let one host run server mode, and another host run client mode. Let the client send data through TCP to the server and report the bandwidth results. Are they close to the link's bandwidth? How many data are transferred and received, are they equal?
Note: You have to open xterm for two hosts, then first run server mode on one host: `iperf -s -p <port number>`. Then run client mode on another host: `iperf -c <server's IP> -p <server's port number> -t <time>`. Measure at least 5 seconds. It is ok if the bandwidth calculated on server and client are different, but they should be both close to the link bandwidth.

D. **Path throughput using `iperf`.** Now measure throughput in pairs (h_1, h_5) and (h_3, h_4). Do the same experiment as 3, report the bandwidth. Which link is the bottleneck link in this path? Is the result close to its bandwidth?

E. **Multiplexing.** Now suppose two pairs of hosts (h_1, h_5) and (h_3, h_4) are sharing the same link. Measure and report the latency and bandwidth again. Compare latency and bandwidth with previous results in question 2 and 4, and explain why it changes or not. Which link(s) are shared? Are they sharing the link bandwidth fairly?

   Note: To do this measurement, you may open up terminals for each of the hosts you will use, type the commands in and quickly ENTER on each terminal. You do not need to worry too much about starting the clients at the exact same time, as long as the number of pings or iperf time is reasonably large, the effect of multiplexing should be visible.

1.3 Submission

Include results in a report (copy paste or screenshots are both fine).

2 Socket Programming (2 weeks recommended)

2.1 TCP File Transfer Server

In this part, you will write a simple file transfer server `ftrans` that allows remote clients to read a file on the server. The communication between the client and server will be achieved through TCP sockets. An example of using TCP socket will be provided on Canvas.

You do not need to worry about how TCP works now, you may use TCP sockets as a blackbox that delivers your data reliably. Later this semester, you will implement a simple selective repeat (SR) ARQ based on UDP socket (which is not reliable), then you will replace the TCP sockets in this part with your implementation of SR ARQ and test under unreliable networks.

The program `ftrans` should operate in two ways.

A. Server mode.

   it should be invoked as follows:
   `./ftrans -s -p <port>`

   For example,
   `./ftrans -s -p 8001`

   For simplicity, you can assume the arguments appear in this order, and there will be no error in arguments, and port number is in [1024, 65535].

   When running as a server, `ftrans` must always listen on `port` for incoming TCP connections from a client . When it has a connection, it must first get the requested file name by reading from the socket until it hits a null terminator. Then it will open the file in its directory (you can assume the file exists) to get the file content and its length.

   Then the server must first send the `length` (an unsigned int) to the client. After that, it may start transferring the content.

   Note: `ftrans` in server mode should never exit, it keeps listening for connections from clients. You may assume the client will never crash in the middle of connection. Your server only needs to handle one client at a time, it does not need to be multi-threaded.

B. Client mode.

   it should be invoked as follows:
   `./ftrans -c -h <server's IP/hostname> -sp <server_port> -f <filename> -cp <client_port>`

   For example,
   `./ftrans -c -h 10.0.0.5 -sp 8001 -f file.txt -cp 8002`

   Again, you can assume the arguments appear in this order, and there will be no error in arguments, and port number is in [1024, 65535]

   When running as a client, `ftrans` must first bind the client's socket on `client_port`, then establish a connection to the server's IP, which is listening on `server_port`. Then it must send the `filename` to the server. Then it must read an unsigned int `length` from the socket, which indicates the size of the file. It must finally read `length` bytes of data from the socket and write them to `filename` in its directory. After this, `ftrans` should exit.

   Note: you may assume the server's IP or host name exists and the port is open for connection. When sending `filename` to the server, include the null terminator, because the server does not know how many bytes it should read off the sockets, it relies on the null terminator to indicate the end of `filename`.

The Workflow is shown below.

A. Client sends `filename` to server.

B. Server reads `filename` and open file.

C. Server sends `length` to client.

D. Client reads `length`.

E. Server sends `file_data` to client.

F. Client reads `file_data` (`length` bytes in total) and writes to `filename`.

G. Client exits, server keeps running.

You will find `ftrans.cc` and `Makfile` under `part2/`, implement in `ftrans.cc` and compile by running `make`.

To test your code, launch `p1_topo.py`. Use xterm to open terminals at h1 and h2 and let one be server and another one be client. Let client request the file `Makefile` (please run client in a different directory as it will write to the file `Makefile`), make sure the file is delivered correctly. Create a couple of files with various size ranging from roughly {100B, 100KB, 10MB} yourself and transfer them. You may use `diff` to check the difference between delivered and original files. To check your correctness, when transferring the `Makefile`, capture the packets sent and received on client with `tcpdump`, and indicate the packets that contain filename, length and data. Report it.

2.2 Submission

Include results in a report (copy paste or screenshots are both fine). Submit your `ftrans.cc` with `Makefile` too.

3 Reliable Transmission (2 weeks recommended)

In this section, you will implement a selective repeat protocol (`sr`) sender and receiver to transfer files reliably. You will test the correctness of `sr` by sending files of various size in a "lossy" network in Mininet. The purpose of `sr` is similar to `ftrans`, however, to simplify the protocol and focus on the reliability, the `sr` is different than `ftrans` in the following ways.

A. `sr` no longer has the concept of client and server, instead, it has a receiver and a sender.

B. `sr` receiver no longer requests a particular file from the sender. Instead, it is **always** ready to receive **any** file from the sender and write to a file in the specified directory. `sr` sender transfers the file name specified in the command line argument.

You must implement `sr` on top of UDP sockets (recall that UDP is unreliable). Use `sendto()` and `recvfrom()` to write and read on a UDP socket. Consider using `MSG_DONTWAIT` in `recvfrom()`, otherwise the `recvfrom()` may block when the packet gets lost. An example of using UDP can be found on Canvas. You must NOT use TCP sockets, doing so will result in a zero for this part.

Protocol Overview

The SR delivers data in packets, and each packet must have a header `SRHeader` (defined in `SRHeader.h`). The header contains the following fields: `flag, seq, len, crc`. The `flag` indicates the type of packet (see below), the `seq` is the sequence number. The `len` is the length of the SR payload, it should be 0 for packets without payload (such as an ACK packet).

`flag` can be one of the following: `SYN, FIN, DATA, ACK, NACK`, all of them are defined in `SRHeader.h` too. To start the file transmission, the `sr` sender must send a `SYN` packet with a random `seq` value, and wait for an `ACK` for this `SYN` packet. After the handshake is done, the sender can start sending `DATA` packets with proper `seq`. The sender will use 0 as its initial `seq` when sending packets, i.e., the first `DATA` packet has `seq` 0, the second one has `seq` 1, etc. After th transmission is done, the sender must send a `FIN` packet with the same `seq` as the `SYN` packet, and wait for an `ACK` for this packet.

On the receiver side, when receiving `SYN` or `FIN` packets, the `seq` value in its `ACK` must be equal to the `seq` set by the sender. When it receives a `DATA` packet, it must send back a cumulative `ACK`, i.e., an `ACK` with `seq = N` means every packet with `seq < N` is received correctly.

`crc` is the CRC-32 value for the payload, please use the `crc32` function found in `crc32.h`, it takes in arguments a pointer to payload and length of payload. For packets without payload, simply call `crc32(NULL, 0)`, which will return a zero.

The `NACK` will be introduced in section 3.3.

The program `sr` operates in two modes.

A. Receiver mode.

   It should be invoked as follows
   `./sr -r <port> <window_size> <recv_dir> <log_file>`

   For example, `./sr -r 8002 10 recv_dir/ recv.log`

   The `log_file` is used to check your correctness and help you debug, its format will be introduced in section 3.1.

   Note: you can assume the arguments appear in this order exactly and do not have errors. You can also assume that the directory `<recv_dir>` exists.

B. Sender mode.

   It should be invoked as follows
   `./sr -s <receiver's IP> <receiver's port> <sender's port> <window_size> <file_to_send> <log_file>`

   For example, `./sr -s 10.0.0.2 8002 8001 10 picture.jpg send.log`

   Note: you can assume the arguments appear in this order exactly and do not have errors.

3.1 Implement a simple SR

Implement your program in `sr.c`

3.1.1 Receiver

The receiver must always be ready to accept any files from a sender, i.e., it does not exit. However, you can assume there is at most one sender in your network. The receiver must store the ith file in `recv_dir/file_i.txt`.

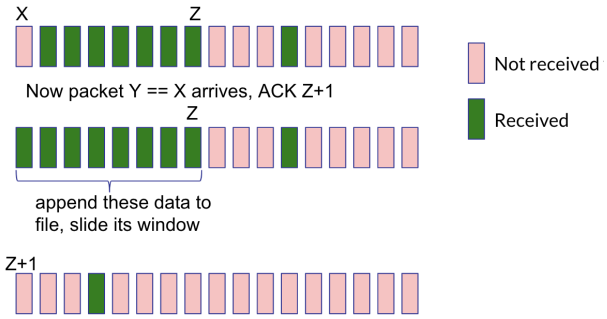
The receiver must maintain a window of size `window_size`, each slot of the window buffers one packet from the sender. To illustrate how the sliding window should work, let us say the beginning of the receiver's window is `seq = X`, now it receives a `DATA` packet with `seq = Y`. You should consider the following scenarios (and think about why it behaves like this!):

A. `Y >= X + window_size`. Receiver must drop the packet (this happens if sender's window is larger than receiver's).

B. `Y < X`. The receiver must `ACK X`, however, it does not need to buffer it.

C. `X < Y < X + window_size`. The receiver must `ACK X` and buffer it in its window.

D. `Y == X`. The receiver must look for the highest in-order sequence number `Z` and send back an `ACK` with `seq = Z + 1`. In other words, all packets with `Y <= seq <= Z` are received without gaps. In this case, the receiver must also advance its window to start at `seq = Z + 1`, and append the data in packet from `X` to `Z` to file `file_i.txt`. See figure below.



When receiving a packet, the receiver must compute the `crc` value for the payload, and compare it with the `crc` field in `SRHeader`. If they do not match, it must drop the packet, i.e., do not send back an `ACK`. You may find the crc function in `crc32.h`

For each packet received or sent, the receiver must log its header in `log_file` Each line should have the following format: `<flag> <seq> <len> <crc>`, each field is separeated by a white space. For example, `DATA 0 1456 77777` and then `ACK 1 0 0`. Note that your log will be used to check the correctness, so make sure the format is correct. The receiver logs any packet's header, including the corrupted one.

Hint: in a "lossy" network, all packets have equal chance to get lost (not just `DATA` packets). For example, when the receiver `ACK` the `SYN` packet from a sender, the `ACK` may get lost so the sender may resend the `SYN` packet and thus the receiver needs to `ACK` again. Make sure you consider such cases in you implementation.

3.1.2 Sender

The sender should first read `file_to_send`, then split and send the data by chunks, each chunk (except the last one) has a size of `MAX_PAYLOAD_SIZE` (defined in `SRHeader.h`). It must compute `crc` for each payload and put it in the header. The `seq` in the header must increment by one upon sending a new packet.

The sender must maintain a sliding window. It must wait for cumulative `ACK` from the receiver. When receiving an `ACK`, the sender must check if it can advance its window. If its window can be advanced, it must do it immediately and send out the new outstanding packets as soon as possible.

In order to detect and retransmit lost packets, in this part the sender will simply set a 400ms timeout on its current window. The timeout is reset when its window is advanced. If after 400ms the window is still not advanced, the sender simply resends all packets in its window and resets its timeout. You may notice that the sender here is not resending packets "selectively", however, you will make it real "selective repeat" in the next two parts by duplicate ACK and NACK.

The sender also logs every packet header sent/received in `log_file` with the same format as described in receiver part.

Hint: think carefully how sender should advance its window and send new packets upon receiving an `ACK`. Think about various possible cases (similar to the ones in receiver). A correct implementation is able to cope with loss of **any** types of packets, errors in `DATA`, reordering of **any** packets, delay or reordering of `ACK`, etc. Make sure your code works fine in all possible cases. Think before you code!

3.1.3 Test sender and receiver

The testing and debugging for this part could be tricky sometimes. If your program does not behave as you wish, you are recommended to look into the logs and find out what happens.

Mininet can simulate a link with loss, you may specify the loss rate by adding a `loss` term to the link. However, it cannot directly simulate packets re-ordering and errors. Therefore, you are provided with a man-in-the-middle proxy `mitm`, which can shuffle the packets randomly, drop packets randomly, or add errors to packets randomly. The `mitm` should stand between your sender and receiver and it will relay the packets. The `mitm` makes the following assumption and operates as follow.

(a) The `mitm` is invoked as follows

```
./mitm <shuffle> <loss> <error> <receiver's ip> <receiver's port> <sender's ip> <sender's port>
```

For example,
`./mitm 0 10 0 10.0.0.2 8002 10.0.0.1 8001`

`shuffle` is either 0 or 1 and indicates if `mitm` would shuffle packets between sender and receiver randomly. `loss/error` is an integer between 0 and 100, and indicates the chance that a packet will be dropped or corrupted. Note: when corrupting, `mitm` only corrupts the payload in `DATA` packets. When dropping packets, all packets have equal chance to be dropped. `mitm` leverages multi-threading to shuffle the packets, it might not be purely random but should be sufficient for testings.

(b) To use `mitm`, launch the `normal_topo.py`, launch `mitm` at a host (say 10.0.0.3) and it will listen on port 8003. Suppose your sender will listen at 10.0.0.1:8001 and receiver will listen at 10.0.0.2:8002, then your `mitm` should be launched like `./mitm 0 10 0 10.0.0.2 8002 10.0.0.1 8001`. Note that when launching the sender, specify the receiver's IP and port as the `mitm`'s (in this case, 10.0.0.3 and 8003). In other words, sender should transmit files to `mitm`, and `mitm` will relay packets between sender and receiver. If you use `mitm`, you do not need to specify `loss` in Mininet because `mitm` can do so.

You should test your program at least in the following cases (and these are the cases your program will be tested against), and transfer files of various sizes, e.g., about 100KB, 1000KB, 10MB. Make sure the file is delivered correctly by calling `diff` on original and received files. Consider the following cases. In each of these cases, set window size to be 10 on both sender and receiver (but make sure your program works with different window sizes). Report the results in each case in the writup (writeup only requires transferring one file for each case, but the file needs to be larger than 1000KB). However, you are encouraged to test your program more extensively, not limited to the following ones. Possible outputs may be found at Appendix.

A. **No reordering, no loss, no error**.
Report the first few lines of log on sender and receiver, and explain how your window is slided on both sides upon receiving an ACK or DATA packet. Is it correct?

B. **10% loss, no reordering, no error**
In sender's log, find packets that are not ACKed due to packet loss, find evidence that the whole window is retransmitted and ACKed later. On receiver's log, look for gaps in its window due to packet loss, and check what ACK number it sends back to the sender. You have to find at least one packet drop in your log and explain it.

C. **10% error, no reordering, no loss**
In sender's log, find packets that are not ACKed due to packet corruption, find evidence that they are retransmitted later. On receiver's log, look for such packets that are received but not ACKed (those are the corrupted ones). You have to find at least one packet corruption in your log and explain it.

D. **Reordering, no error, no loss**
Report the first few lines of sender and receiver's logs, until the point where the first 10 packets are surely delivered. Explain how your sender and receiver cope with packet re-ordering. Does it seem correct?

E. **Reordering, 5% loss, 5% error**
Mix all possible scenarios, make sure the file is delivered correctly. Nothing to be reported here.

3.1.4 Submission

Implement your program in `part3-1/`, write your report in a PDF.

3.2 Make sender more efficient – leverage duplicate ACK on sender side

3.2.1 Implement duplicate ACK

As you may have noticed, the sender in 3.1 only relies on timeout, and resends **all** packets in the window after timeout. This is not truly "selective" repeat. In this part, you will explore "duplicate ACK" to 'selectively' resend the lost packets. Duplicate ACK is actually used in the TCP to trigger fast retransmission.

Duplicate-ACK-retransmission works as follows. Suppose sender has a window of size 10, and packet 0-9 are just sent out and packet 3 is lost on the way. Then, the sender will receive ACK 1, 2, 3, 3, 3, 3,... The duplicate ACK number (3 in this case) usually indicate that the packet is lost, and the sender should retransmit it immediately. In this part, your sender counts the duplicate ACK number (or the sequence number at the beginning of its window), and when it sees 3 duplicate ACKs, it has to resend the packet immediately and reset the counter.

Copy your program from `part3-1/` to `part3-2/` and start modifying it.

3.2.2 Test duplicate ACK

Set window size to 10 on both sender and receiver.

A. 10% loss, no reordering, no error.
In sender's log, find the headers which indicate that your sender receives three duplicate ACKs and resends the corresponding packet immediately. Report at least one such case.

B. Efficiency.
Transfer a large file (more than 1MB) under 10% loss link, time your sender (you may use `time` in the terminal). Compare the transferring time in part 3.1 and 3.2, does duplicate ACK make the sender more efficient? Report your results.

C. Make sure it transfers files reliably.
Nothing needs to be reported here. But you need to make sure your `sr` transfers files of various size successfully under various delay, error, or loss. Your program will be tested in such cases.

3.2.3 Submission

Implement your program in `part3-2/`, write your report in a PDF.

3.3 Make sender more efficient – send NACK on receiver side

3.3.1 Implement NACK

The other way to make the sender repeat selectively is the negative acknowledgement (NACK) from the receiver. As you have seen in lecture slides, in this part, upon receiving a packet, the receiver must look for gaps in its window, and actively NACK the first gap that has not been NACKed before. On the sender side, upon receiving a NACK, it must retransmit the corresponding packet immediately.

Copy your program from `part3-1/` to `part3-3/` and start modifying it. Note that since you are using NACK to achieve selective repeat here, you do not need the duplicate ACK in your program.

3.3.2 Test NACK

Set window size to 10 on both sender and receiver.

A. 10% loss, no reordering, no error.
In sender's log, find the headers which indicate that your sender retransmits the packet immediately after it receives a NACK. In receiver's log, find the headers which indicate that your receiver sends back a NACK when there is a gap in its window. Report it.

B. Efficiency.
Transfer a large file (more than 1MB) under 10% loss link, time your sender (you may use `time` in the terminal). Compare the transferring time in part 3.1 and 3.3, does NACK make the sender more efficient? Report your results.

C. Make sure it transfers files reliably.
Nothing needs to be reported here. But you need to make sure your `sr` transfers files of various size successfully under various delay, error, or loss. Your program will be tested in such cases.

3.3.3 Submission

Implement your program in `part3-3/`, write your report in a PDF.

3.4 Throughput, delay and window size

In this part you will observe the relationship between throughput and link delay. You will transfer the file `file3.jpg` (about 4.2MB) from host h_1 to h_2 and measure the time on the sender side and compute its throughput. You will use your program in part 3.3 and your network has no loss, no error, no packet reordering.

In `nromal_topo.py`, find the link between h_1 and h_2, change the argument `delay` to {0.01, 10, 50, 100} milliseconds. Fix the window size as 10, measure the time on the sender side. You may use the command `time`, e.g., `time ./sr -s 10.0.0.2 8002 8001 10 file3.jpg sender.log`. After it finishes, find the `real` time printed on the terminal. This would be the total time your sender spends. Make sure you do not have too much unnecessary I/O on the sender side (e.g., remove the `printf()`s for debug), since those would add extra time and make your measurement inaccurate.

Record your measured time and compute throughput for each `delay`. Then, change the window size to 50 and do the same measurements. Include the results and your findings in how throughput changes with delay and window size in the report.

3.5 Grading

Since this part is a coding assignment, part of your points will be graded through testing your program. Make sure your program can compile by `Makefile` (you may write your own `Makefile` if necessary). The test cases will be similar to the ones described above (with various loss, error, reordering, etc.), your logs will be checked to determine if your program behaves correctly, especially in the duplicate ACK and NACK.

Again, you must NOT use TCP sockets, doing so will result in a zero for this part. You must use UDP sockets only.

4 Logistics

Report results from different sections in one PDF report (handwritten is not permitted). Remember to backup your code in case your machine crashes. All the submitted code will be compiled and run against the tests described in the instructions above, points may be deducted if the program functions incorrectly. Make sure your code can be compiled with the Makefile.

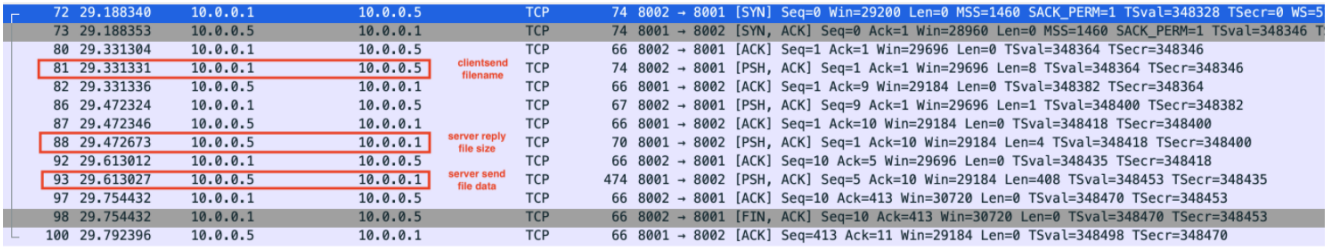
All your work is subject to JI's honor code. Additionally, the results in your report must match what your program produces. In other words, do not fabricate any result in your report.

5 Appendix

This section displays sample outputs for some of the previous sections for your reference. The purpose of these samples is to show what are needed for each section, but your results may look similar or different.

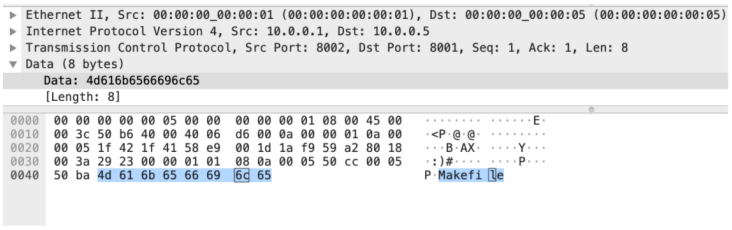
5.1 TCP File Transfer Server

The packets captured at client are shown below. Client is at 10.0.0.1:8002 and server is at 10.0.0.5:8001

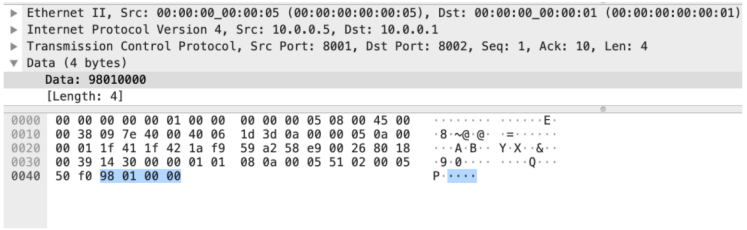


72	29.188340	10.0.0.1	10.0.0.5	TCP	74	8002 → 8001 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=348328 TSecr=0 WS=5
73	29.188353	10.0.0.5	10.0.0.1	TCP	74	8001 → 8002 [SYN, ACK] Seq=0 Ack=1 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=348346 T
80	29.331304	10.0.0.1	10.0.0.5	TCP	66	8002 → 8001 [ACK] Seq=1 Ack=1 Win=29696 Len=0 TSval=348364 TSecr=348346
81	29.331331	10.0.0.1	10.0.0.5	TCP	74	8002 → 8001 [PSH, ACK] Seq=1 Ack=1 Win=29696 Len=8 TSval=348364 TSecr=348346
82	29.331336	10.0.0.5	10.0.0.1	TCP	66	8001 → 8002 [ACK] Seq=1 Ack=9 Win=29184 Len=0 TSval=348382 TSecr=348364
86	29.472324	10.0.0.1	10.0.0.5	TCP	67	8002 → 8001 [PSH, ACK] Seq=9 Ack=1 Win=29696 Len=1 TSval=348400 TSecr=348382
87	29.472346	10.0.0.5	10.0.0.1	TCP	66	8001 → 8002 [ACK] Seq=1 Ack=10 Win=29184 Len=0 TSval=348418 TSecr=348400
88	29.472673	10.0.0.5	10.0.0.1	TCP	70	8001 → 8002 [PSH, ACK] Seq=1 Ack=10 Win=29184 Len=4 TSval=348418 TSecr=348400
92	29.613012	10.0.0.1	10.0.0.5	TCP	66	8002 → 8001 [ACK] Seq=10 Ack=5 Win=29696 Len=0 TSval=348435 TSecr=348418
93	29.613027	10.0.0.5	10.0.0.1	TCP	474	8001 → 8002 [PSH, ACK] Seq=5 Ack=10 Win=29184 Len=408 TSval=348453 TSecr=348435
97	29.754432	10.0.0.1	10.0.0.5	TCP	66	8002 → 8001 [ACK] Seq=10 Ack=413 Win=30720 Len=0 TSval=348470 TSecr=348453
98	29.754432	10.0.0.1	10.0.0.5	TCP	66	8002 → 8001 [FIN, ACK] Seq=10 Ack=413 Win=30720 Len=0 TSval=348470 TSecr=348453
100	29.792396	10.0.0.5	10.0.0.1	TCP	66	8001 → 8002 [ACK] Seq=413 Ack=11 Win=29184 Len=0 TSval=348498 TSecr=348470

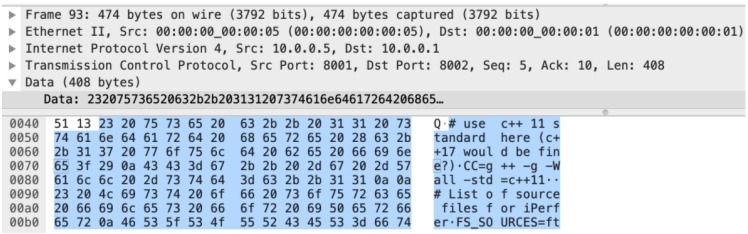
All packets captured by the client.



Packet that contains the requested filename.



Packet that contains the file size, a 4-byte int



Packet(s) that contains the file data, 408 bytes in total.

5.2 Reliable Transmission

5.2.1 Simple SR

A. No reordering, no loss, no error.

A possible result with sender and receiver's `window_size = 5` is shown below. For each line, everything after `#` is added afterwards.

```
# sender log
SYN 1804289383 0 0 # initiate a connection
ACK 1804289383 0 0 # receive ACK for the SYN (same seq as SYN)
DATA 0 1456 1048559825 # send out 5 packets...
DATA 1 1456 2133123180
DATA 2 1456 3091848712
DATA 3 1456 3741717153
DATA 4 1456 1259056260
ACK 1 0 0 # receive ACK for packet 1
DATA 5 1456 3249902976 # window is slided, packet 5 is thus sent.
ACK 2 0 0 # receive ACK for packet 2
DATA 6 1456 3794759566 # window is slided, packet 6 is thus sent.
...
FIN 1804289383 0 0 # tear down the connection (same seq as SYN)
ACK 1804289383 0 0 # receive ACK for the FIN
```

```
# receiver log
SYN 1804289383 0 0 # receive a connection request
ACK 1804289383 0 0 # ACK with the same seq
DATA 0 1456 1048559825 # receive packet 0
ACK 1 0 0 # ACK 1 (and slide window)
DATA 1 1456 2133123180 # receive packet 1
ACK 2 0 0 # ACK 2 (and slide window)
...
FIN 1804289383 0 0 # receive connection close request
ACK 1804289383 0 0 # ACK the FIN
```

B. 10% loss, no reordering, no error
A possible result with sender and receiver's `window_size = 5` is shown below. For each line, everything after `#` is added afterwards.

```
 # sender log
...
DATA 0 1456 1048559825
DATA 1 1456 2133123180
DATA 2 1456 3091848712
DATA 3 1456 3741717153
DATA 4 1456 1259056260
ACK 0 0 0
ACK 0 0 0 # still receive ACK 0
ACK 0 0 0
ACK 0 0 0
DATA 0 1456 1048559825 # timeout is triggered, resend whole window
DATA 1 1456 2133123180
DATA 2 1456 3091848712
DATA 3 1456 3741717153
DATA 4 1456 1259056260
ACK 5 0 0
DATA 5 1456 3249902976 # receive ACK, proceed.
DATA 6 1456 3794759566
DATA 7 1456 561850557
DATA 8 1456 4176746620
DATA 9 1456 2763283951
...
```

```
 # receiver log
...
DATA 1 1456 2133123180 # expect packet 0 but receive packet 1
ACK 0 0 0 # ACK 0
DATA 2 1456 3091848712 # expect packet 0 but receive packet 2
```

```
ACK 0 0 0 # ACK 0
DATA 3 1456 3741717153
ACK 0 0 0
DATA 4 1456 1259056260
ACK 0 0 0
DATA 0 1456 1048559825 # finally receive packet 0
ACK 5 0 0 # ACK 5 now
DATA 1 1456 2133123180
ACK 5 0 0
DATA 2 1456 3091848712
ACK 5 0 0
...
```

5.2.2 Duplicate ACK

Similar to simple SR.

5.2.3 NACK

Similar to simple SR.