COM SCI 118 Spring 2022

Project 2: Simple Window-Based Reliable Data Transfer
Due date: June 3th, 23:59 PT

1 Overview

The purpose of this project is to implement a basic version of reliable data transfer protocol. 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 will provide skeleton code for connection management. The following functions should be realized:

- Large file transmission with pipelining.
- Loss recovery with Go-Back-N (GBN) or Selective Repeat (SR). If you implement SR, you can get up to 15% bonus. Implementation with GBN will not get the bonus.

We made several simplifications to the real TCP protocol, 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.
- You do not need to realize congestion control in your project.

We require you implement code in a Linux environment so we can test it with simulated packet loss. All implementations need to 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++14 extensions, for parts that are not directly related to networking, such as string parsing and multi-threading.

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. We will give you the skeleton code including header setup and connection setup.

2.1 Basic Protocol Specification

You can find the skeleton code from the github repo: https://github.com/CS118S22/Project2. The given skeleton code cover the implementation of header, basic connection between client and server, and printout function. Specifically, there are four files:

- server.c implements a server that accepts connection setup and store received files.
- client.c implements a client that initiates connection setup and send a file to the server.
- Makefile helps to compile the server and the client.
- test_format.py helps to check the format of client side output.

So you will find the skeleton code covers all functions mentioned in this section. Please read through to make sure your final implementation STILL conforms to the specification.

2.1.1 Packet Format

- The given header format includes a Sequence Number field, an Acknowledgment Number field, and ACK, SYN, and FIN flags.
- The header length is exactly 12 bytes, while the design does not use up all 12 bytes, pad zeros to make it so.

2.1.2 Server Application Specification

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). You can assume that the specified port is always **correct**. 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
```

2.1.3 Client Application Specification

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 testfile:

```
$ ./client localhost 5000 testfile
```

You can assume that the specified inputs are always **correct**.

The client open a UDP socket and initiate 3-way handshake to the specified hostname/IP and port. It shall

- 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.

After file is successfully transferred (all bytes acknowledged), the client gracefully terminate the connection following these steps:

- Send UDP packet with FIN flag set.
- Expect a packet with ACK flag and another packet FIN flag.
- Respond to each incoming FIN with an ACK packet; drop any other non-FIN packet.
- After receive the FIN packet, wait for 2 seconds to close connection and terminate.

Please check the appendix for examples.

2.1.4 Common Printout Format Requirements

The following output MUST be written to standard output (std::cout in C++ or stdout in C) in the **EXACT** format defined. If any other information needs to be shown for your own debugging, it MUST be written to standard error (std::cerr) or commented out before your submission.

- Packet received:
  RECV ⟨SeqNum⟩ ⟨AckNum⟩ [SYN] [FIN] [ACK]
- Packet sent:
  SEND ⟨SeqNum⟩ ⟨AckNum⟩ [SYN] [FIN] [ACK] [DUP-ACK]
- Packet retransmition:
  RESEND ⟨SeqNum⟩ ⟨AckNum⟩ [SYN] [FIN] [ACK] [DUP-ACK]
- Timeout:
  TIMEOUT ⟨SeqNum⟩

Note the convention:

- **After the TCP three-way handshake and before the TCP tear-down**, since the ACK number in packets sent by the client will always be the same (because the server does not send payload), therefore, the ACKNum in SEND and RESEND can be replaced by 0 and there is no need to append ACK (Check the sample output below).
- When a packet is being retransmitted, the client should print RESEND instead of SEND. Notice that unless in the TCP tear-down stage, a server will never print RESEND because duplicate ACK transmission is not considered to be retransmission.
- There is a white space between any two pieces of information in the output line.
- ⟨yy⟩ means that value of yy variable should appear on the output (in decimal). Undefined value should be 0. For example, in the first SYN packet in TCP three-way handshake, the AckNum should be 0.

- [xx] means that xx string is optional. If the packet belongs to one or more of the following cases, the host needs to follow the corresponding printout:
  - Acknowledgment packet: [ACK]
  - Duplicate ACK packet: [DUP-ACK]. When [DUP-ACK] shows up, [ACK] **must not** show up at the same time.
  - If SYN flag is set: [SYN]
  - If FIN flag is set: [FIN]

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. You can use the parsing script test_format.py to check the format with the following steps.

First, start server at port number 5000, then start another terminal to run the client and collect output: ./client localhost 5000 file100 > output_file

Finally, run the script to check the output with python3 test_format.py output_file
You are supposed to see output like:
Client ISN: 2254
Server ISN: 25157
Check result: True

2.2 Instructions on Pipelining and Loss Handling

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

- Introduce a large file transmission and pipe-lining. This means you must divide the file into multiple packets and transmit the packets based on the specified window size.
- Introduce packet loss. Now you have to add a timer for last sent packet (Go-Back-N) or several timers for each unacked packets (Selective repeat). 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. Please implement the project incrementally.

2.2.1 Requirements on pipelining

Here are some general 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 25600 + 1.
- Packet retransmission 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 the ACK flag is false, Acknowledgment Number should be set to 0 .
- SYN and FIN packets must not carry any payload. But these packets should take logically 1 byte of the data stream (same as in TCP, see examples).

Here are the requirements on the client and server:

- Client should send file in pipeling scheme with a fixed window size 5120 (10 packets in a window) if packets are not retransmissions. For retransmissions, the client initiate retransmissions based on Go-Back-N or Selective-Repeat.
- Client should support transfer of files that are up to 10 MB.
- 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 10 MB.

2.2.2 Requirements on loss handling

- Select Go-Back-N scheme OR Selective Repeat scheme for loss recovery. You'll get bonus up to 15% for implementing Selective Repeat.
- Generally, Go-Back-N scheme keeps ONE timer for the least recent unacknowledged packet and retransmits all unacknowledged packets upon timeout. RTO=0.5 seconds. Duplicate ACKs should not trigger retransmission. Procedures at the sender are as follows (see pseudo code in section 3.33 of textbook):
  - After the sender sends out a packet, the timer is started if not running.
  - After the sender receives a new ACK (not duplicate), the timer is restarted if there is any unacknowledged packet.
  - Upon timeout, the sender transmits all unacknowledged packets in the window; The timer is started for the same packet again.

  Procedures at the receiver are as follows:
  - Upon receiving a packet, send back a cumulative ACK with the sequence number of the next in-order byte expected from the sender. For example, the receiver has got bytes 0 - 511 and 1024 - 1535; The next in-order byte is 512. So when the segment of byte 1536 - 2047 arrives next, the receiver will acknowledge this packet with ACK number 512.
  - Any out-of-order packet and duplicate packet (which has been received before) is directly discarded.
- For Selective Repeat scheme:
  - The sender keeps a timer for each unacknowledged packet. Retransmission of a certain packet is triggered only when the corresponding timer is expired.
  - The receiver acknowledges every packet with ACK number = sequence number of the last byte of that packet+1. The ACK is not cumulative. For example, when the packet of bytes 0 - 511 is received, the receiver will send back ACK with number 512.

2.2.3 Sample Output

Please check the sample output here or directly copy past the URL:
https://docs.google.com/document/d/1N1Vpp7jkCBEDKFTbNyCDcLukKYvIA1pMFSoUAHMuNjI/edit?usp=sharing

2.2.4 Loss emulation for testing

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 or local loopback may be too low to test your applications. Therefore, we are going to emulate packet loss. We provide two ways for emulation, you can freely choose one according to your settings. Regardless of the simulation method you choose, there are two requirements:

- 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).

Method 1: Using the `tc` command in Linux.

Note: this method requires a linux system with root permission, and it has to be applied on the proper network interface.

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. `lo0` for localhost; The interface name of localhost may differ on your laptop, you can figure it out by command `ifconfig`):

   ```
   tc qdisc show dev lo0
   ```

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 lo0 root netem loss 10%
   ```

3. To delete the network emulation:

   ```
   tc qdisc del dev lo0 root
   ```

Method 2: Using the python script `rdproxy.py`.

This method is intended for students without access to Linux system with root permission. It creates a userspace "proxy" between the client and the server. Instead of letting the client talk to the server directly like the previous method, the client communicate with the script, which automatically forwards the packets in between and randomly drops packet in the process. To use the script:

1. Start the server normally on a port. For example, suppose the server runs on port 5000.
2. Decide the port of the proxy (you need to pick a different one from the server) and the loss rate. For example, if the proxy port is 9999 and the loss rate is 10%, you need to run:

   ```
   python3 rdproxy.py 5000 9999 0.1
   ```

3. Launch the client. In order to emulate the packet loss, the client will communicate with the proxy script (port 9999) instead of using port 5000 to directly communicate with the server.

3 Submission and Demo

3.1 Submission

This project is due June 3th, 2022 at **23:59 PT**. Late submission is allowed by Sunday, June 5th (20% deduction on Saturday, 40% deduction on Sunday). Put all your files into a directory, compress the directory and generate a UID.tar.gz where UID is your UCLA ID. Your submission should include the following files:

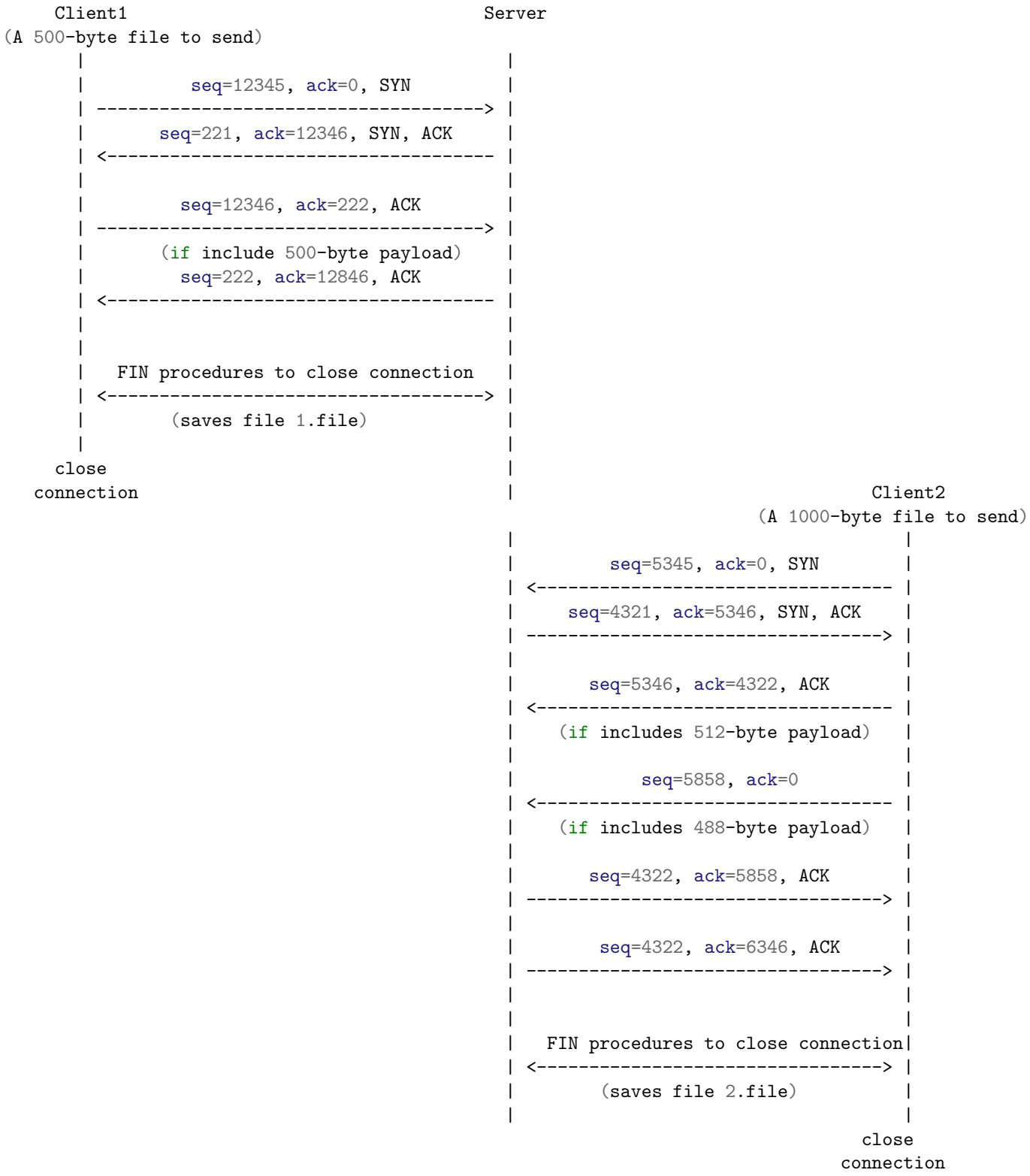
- Your source code (e.g. server.c, client.c). The code of the server and the client can be several files.
- A Makefile. TAs will only type make to compile your code.
- A README.md which contains:
  - Your name, email, and UCLA ID;
  - The high level design of your server and client (one paragraph);
  - The problems you ran into and how you solved the problems (up to three paragraphs);
  - List of additional libraries / online tutorials / code examples (except class website) you have been using.

3.2 Demo Requirements

1. Sign up for demo: TA will distribute the demo sign-up sheet in Week 8.
2. Testing files and scripts: TA will release testing files and scripts after the due date. So as the details of demo. You need to download the testing files and scripts and learn how to run it before the demo.
3. Demo day:
   - 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. You will be asked to run test scripts. 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 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.

A Example of connection setup and teardown

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



See below diagram for illustration for FIN procedures during connection teardown.