Project 1: Transport

Due on Friday, November 8th, 2024 at 11:59 pm (Week 6)

Thank you to Professor Lixia Zhang; former CS 118 TAs Xinyu Ma, Varun Patil, Tianyuan Yu; and former CS 118 LA Paul Serafimescu for the original version of this project.

Quick Links

Starter Code (preferably use your Project 0 solution instead):
https://github.com/CS118F24/project0starter
Project 0 Solution (only use the files in the `src` directory for reference):
https://github.com/CS118F24/project0

Overview

In Project 0, you implemented a bidirectional pipe between two processes over the network. Congrats! Here's a fun challenge: try sending over 200 MB of data using your programs.

Firstly, I would create a file containing 200 MB of random data:

```
host0:~/project0 $ head -c 200000000 /dev/urandom > test.bin
```

Then, I would run both my client and server and pass `test.bin` into each of them while saving each program's output.

Client

```
host0:~/project0 $ ./client localhost 8080 < test.bin > client.bin
```

Server

```
host0:~/project0 $ ./server 8080 < test.bin > client.bin
```

Once I'm confident that both processes have finished sending to each other (~ 5 seconds), I can use the `diff` command to make sure that `client.bin` and `server.bin` are the same as `test.bin`.

```
host0:~/project0 $ diff test.bin client.bin
Binary files test.bin and client.bin differ

host0:~/project0 $ diff test.bin server.bin
Binary files test.bin and server.bin differ
```

What's going on? If we look at the current directory, we can see that not only do the bytes differ, but the file sizes are completely different.

```
host0:~/project0 $ ls -la
-rw-r--r--   1 varghese  staff  192929280 Oct  8 14:48 client.bin
-rw-r--r--   1 varghese  staff  192977408 Oct  8 14:48 server.bin
-rw-r--r--   1 varghese  staff  200000000 Oct  8 14:48 test.bin
```

Why does this happen? In Project 0, we used the User Datagram Protocol (UDP) to send data over the network. This protocol attempts *best effort delivery*—meaning that UDP will attempt to send your data over the network, but has no guarantee that it will actually get to the destination. *Most* of the time, the data makes it through (as seen in Project 0), but this is clearly not the case *all* the time (especially for larger amounts of data). Note that this is even over the same host—imagine how much worse the error would be over the actual internet with many routers across the network.

In this project, you'll be creating a reliability layer on top of UDP. At a high level, this layer will split data into chunks called packets and label them with numbers (so we know which go in what order). This layer will attempt to guarantee 3 main things:

1. packets will be sent from one end to the other,
2. dropped packets will be retransmitted, and
3. packets will be exported (in this case, written to standard output) in order.

Notices

1. If you're using GitHub or a similar tool to track your work, please make sure that your repository is **set to private** until the due date. You're welcome to make your solution public after the matter.
2. As per the syllabus, the use of AI assisted tools is not allowed. Again, we can't prove that such tools were used, but we do reserve the right to ask a couple questions about your solution and/or add project related questions to the exam.
3. In the provided autograding Docker container, there are reference binaries. Please do not reverse engineer the binaries and submit that code—which would be obvious and clear academic dishonesty. Remember, we do manually inspect your code.
4. This project is two of three in this class. They will all build on top of each other. **It's important that you finish this project in order to work on the next one.** (If you're not able to finish by the deadline, we do plan on releasing the reference solution to individuals that need it to start on the next projects.)

Setup

Since Project 1 builds on Project 0, we recommend duplicating your Project 0 directory and using it for Project 1. **Only use the Project 0 Solution if you had no submission for Project 0.**

The local autograder works exactly the same. However, there is one change you have to make. **In your root `Makefile`, change `IMAGE` from `sockets-are-cool` to `reliability-is-essential`.**

All the commands are the same; see the Project 0 specification for more details.

Specification

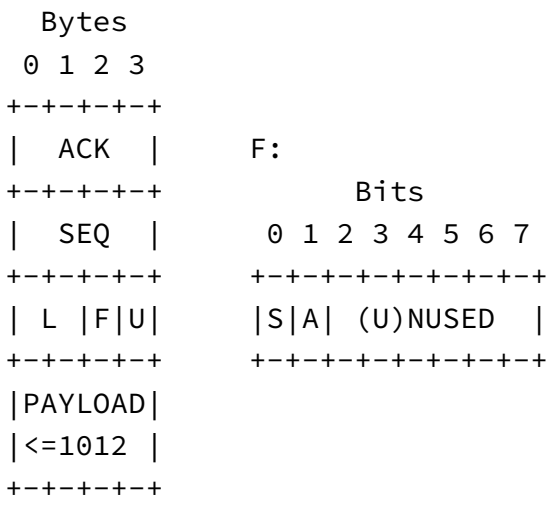
Create two programs (a server and a client) written in C/C++ (up to version 17). Both programs will use BSD sockets using IPv4 and UDP to listen/connect. The expected behavior is the exact same as Project 0, but with different inner logic. See the Project 0 specification for more information.

Quick Info

- Sequence numbers are in bytes
- Must perform a 3 way handshake (with a SYN flag)
- Maximum Window Size of **20240 bytes** (20 full packets)
- Fixed retransmission delay of **1 second**
- Must retransmit after **3 duplicate ACKs**

Packet Layout

Data sent/received from/to the socket will be in the following byte format:



Field Descriptions

- Sequence Number (SEQ; 4 bytes)
  This number represents the *first* byte number in this packet. For example, if the previous packet's SEQ is 0 with length 1000, this packet's SEQ is 1000.

- Acknowledgement Number (ACK; 4 bytes)
  This number represents the *next* byte that the receiver is expecting. For example, if the receiver has bytes 0-999 and 2000-2999, the receiver will send a packet with ACK 1000 (missing bytes 1000-1999).

- Length (L; 2 bytes)
  This field represents the length of the payload field. If the payload is 1000 bytes long, this field will store a value of 1000.

- Flags (F; 1 byte)
  This single byte marks the packet as having different purposes. If the (S)YN flag is on, this means that this packet seeks to synchronize sequence numbers (by providing an initial one in the SEQ field). The (A)CK flag is on when the packet is carrying an acknowledgement.

  The S flag is the least significant bit, and the A flag is the second-least significant bit. You may access their values like so:
  ```
  uint8_t flag; // Flag value from some packet
  bool syn = flag & 1;
  bool ack = (flag >> 1) & 1;
  ```

- Unused (U)
  We don't use this space in the packet. Keep it as null bytes.

Reading/Sending Packets

Note that the layout suggests that the largest packet size is 1024 bytes (or maximum length 1012 bytes). In order to read in a packet, you may do the following:

```
#define MSS 1012 // MSS = Maximum Segment Size (aka max length)
typedef struct {
      uint32_t ack;
      uint32_t seq;
      uint16_t length;
      uint8_t flags;
      uint8_t unused;
      uint8_t payload[MSS];
} packet;

// Assume the socket has been set up with all other variables
packet pkt = {0};
int bytes_recvd = recvfrom(sockfd, &pkt, sizeof(pkt), 0, (struct sockaddr*)
&server_addr, &s);
```

```
uint32_t ack = ntohl(pkt.ack); // Make sure to convert to little endian
```

In short, instead of passing in a pointer to some `char` buffer, you're more than welcome to pass in a pointer to a packet struct. You may also perform something similar with `sendto`.

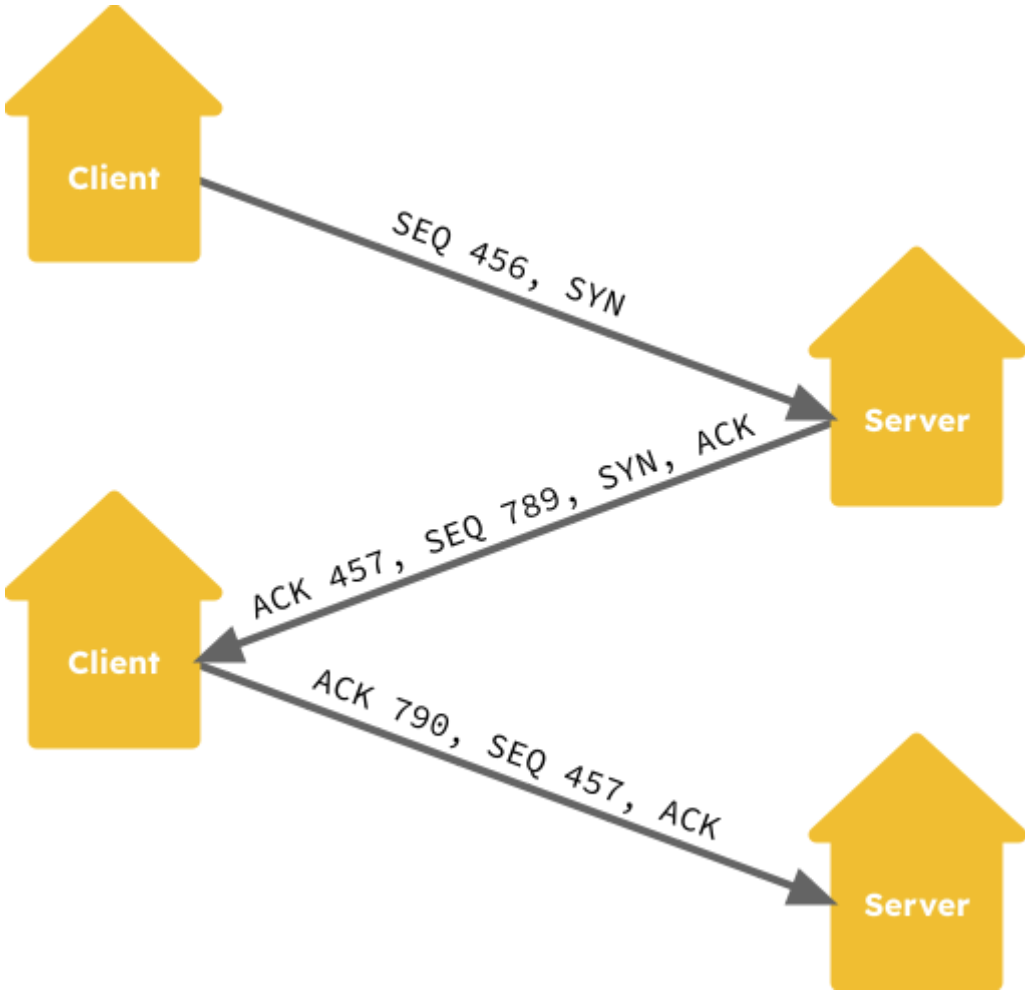
General Operation

Handshake

Before any data is transferred, both the client and server will perform a "three way handshake." This three way handshake is to ensure that both initial sequence numbers are in sync.

1. Once the client connects, it sends a packet with a 0 length payload, the SYN flag set, and a randomly chosen initial sequence number in SEQ. (Keep all other fields 0).
2. The server responds with another packet with a 0 length payload, the SYN flag set, and a randomly chosen initial sequence number in SEQ. However, this packet also has the ACK flag set, and the ACK number set to the client's SEQ + 1.
3. The client finally responds with another packet. This packet has the ACK flag set, the ACK number set to the server's SEQ + 1, and the SEQ incremented by 1 from the very first packet. (The client may also have a real payload in this packet.)

Once the handshake is complete, both sides are ready to send data over.



Sending Data

Read as much as you can from standard input and place it in a new packet's payload. Send the packet over the socket, but keep it in a *sending buffer*. Keep doing this until you've reached the **maximum window size of 20240 bytes (20 full packets)**. You may only keep sending once you receive acknowledgement of packets (which reduces the amount of packets in your window).

Receiving Data

Once you receive a new packet, check if the ACK flag is set. If it is, do a linear scan of all the packets in the sending buffer. Any packet with a SEQ less than the packet's ACK is assumed to be acknowledged. You may remove them from the sending buffer. This reduces the amount of packets in your window and should let you send more packets if needed.

Also, place the packet in a *receiving buffer*. Do a linear scan of all the packets in the receiving buffer starting with the next SEQ you expect and write their contents to standard output.
For example, if you have packets:

- SEQ 1000, length 1000;

- SEQ 2000, length 1000;
- SEQ 5000, length 1000; and
- SEQ 6000, length 1000;

you'd write out the contents of packets SEQ 1000 and 2000 and remove them from the receiving buffer. Your new ACK number is 3000 (since that's the next byte you expect); 5000 and 6000 remain in the receiving buffer.

Finally, send an acknowledgement. You may do this by sending it with another data packet (with ACK set and flag), or you may send a dedicated ACK packet (if you have no more data to send or reached your maximum window size). Dedicated ACK packets should not go in the send buffer. The ACK number you use is the SEQ that you expect next (after performing a linear scan of the receiving buffer).

Retransmission

After **1 second** of not receiving any *new* ACKs from the receiver (or any sort of data at all), retransmit the packet with the lowest sequence number in the sending buffer. Reset this timer as soon as you receive a new ACK.

After **3** duplicate acknowledgements (all with the same ACK number), retransmit the packet with the lowest sequence in the sending buffer. Do not reset the timer.

Potential Improvements

In Project 2, you'll be adding a security layer on top of this reliability layer. Currently, the reliability layer gets its data directly from the socket/standard output. Can you think of a way to modularize this (e.g. using function pointers to change what you input into the reliability layer)?

Common Problems (list will be frequently updated)

- Make sure that **every** number (ACKs, SEQs, lengths, etc.) is in network order/big endian.
- Choose your initial sequence number randomly, but below half the maximum sequence number so you don't have to worry about wraparound (the reference binaries will do this as well).
- There's no need to preserve the packet boundaries; if you read in 1024 bytes, and you receive a packet that's only 1000 bytes, you don't need to save the bottom 24 bytes to place at the beginning of the next packet.

  UDP only sends datagrams—if the datagram is less than the amount of bytes you read in, that's all you receive. The next packet will be read in the next time you call `recvfrom`.

  (The same goes for reading in less than the packet size. If you receive a packet that's 1024 bytes long, and you only allocate a buffer that's 500 bytes, those extra 524 bytes are gone forever; the next call to `recvfrom` will move on to the next packet).

- Remember to use non-blocking calls so you don't need to multithread your solution.
- **Update (10/28)**: I updated the autograder so that it doesn't drop any handshake packets. Correct solutions should still retransmit the handshake if needed, but technically you don't need to anymore.

Autograder Test Cases

A word on the local autograder: after Project 0, it's very clear that even if it uses the same image as Gradescope, it by no means offers an identical testing environment. While this is unfortunate, the local autograder can still be useful. For Project 1, these test cases are somewhat performance based—which can vary a lot from computer to computer. An ideal testing strategy would be to see if you can pass a couple of the drop test cases on your computer. If you can, try your solution on Gradescope. In the next coming days, I'll be updating the autograder's image with more useful testing utilities.

0. Compilation (`class TestCompilation`)

This test case passes if:

1. Your code compiles,
2. yields two executables named `server` and `client`,
3. and has no files that aren't source code/text.

1. Reliable Data Transport (`class TestRDT`)

1. **Reliable Data Transport (Your Client <-> Your Server): Small file (20 KB): `test_self` (10 points)**
   This test case runs your server executable then your client executable. It will then input 20 KB of random data in both programs' stdin and check if the output on the respective other side matches.

2. **Reliable Data Transport (Your Client <-> Your Server): Medium file (100 KB), Drop (5%): `test_self_drop` (5 points)**
   Same as 1, but passes your solution through a proxy. This proxy will drop 5% of packets and may reorder the rest. **Timeout: 30 seconds.**

3. **Reliable Data Transport (Your Client <-> Reference Server): Small file (20 KB): `test_client` (10 points)**
   Same as 1, but uses the reference server instead.

4. **Reliable Data Transport (Reference Client <-> Your Server): Small file (20 KB): `test_server` (10 points)**
   Same as 1, but uses the reference client instead.

5. **Reliable Data Transport (Your Client <-> Reference Server): Medium file (100 KB), Drop (5%): `test_client_drop` (10 points)**
   Same as 2, but uses the reference server instead. **Timeout: 30 seconds.**

6. **Reliable Data Transport (Reference Client <-> Your Server): Medium file (100 KB), Drop (5%): `test_server_drop` (10 points)**
   Same as 2, but uses the reference client instead. **Timeout: 30 seconds.**

7. **Reliable Data Transport (Your Client <-> Reference Server): Large file (500 KB), Drop (1%): `test_client_drop_large` (5 points)**
   This test case uses a substantially larger file, so we decrease the drop rate to 1%. **Timeout: 60 seconds.**

8. **Reliable Data Transport (Reference Client <-> Your Server): Large file (500 KB), Drop (1%): `test_server_drop_large` (5 points)**
   Same as 7, but uses the reference client instead. **Timeout: 60 seconds.**

9. **Reliable Data Transport (Your Client <-> Reference Server): Ginormous file (2 MB): `test_ginormous` (25 points)**
   We don't want to keep the autograder running too long, so this test case only tests the client. **Timeout: 60 seconds.**

Description of Work (10 points)

This is new for Project 1. We ask you to include a file called `README.md` that contains a quick description of:

1. the design choices you made,
2. any problems you encountered while creating your solution, and
3. your solutions to those problems.

This is not meant to be comprehensive; less than 300 words will suffice.

Submission Requirements

Submit a ZIP file, GitHub repo, or Bitbucket repo to Gradescope by the posted deadline. You have unlimited submissions; we'll choose your best scoring one. As per the syllabus, remember that all code is subject to manual inspection.

In your ZIP file, include all your source code + `README.md` + a `Makefile` that generates two executables named `server` and `client`. **Do not include any other files; the autograder will reject submissions with those.** If you're using the starter repository, the default `Makefile` inside the `project` directory includes a `zip` directive that automatically creates a compatible ZIP file.