>

Project 1: Reliable Data Transfer

Due on Wednesday, February 19th at 11:59 pm

#Quick Links

Starter Code
Using the Local Autograder

Note: If you pulled the starter code before January 24th at 7:50 pm, please pull the newest changes.

#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	Server
`host0:~/project0 $ ./client localhost 8080 < test.bin > client.bin`	`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 tianyuan staff 192929280 Oct 8 14:48 client.bin
-rw-r--r-- 1 tianyuan staff 192977408 Oct 8 14:48 server.bin
-rw-r--r-- 1 tianyuan 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**.
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 linkable shared binaries.)
5. This project must be done in groups of up to **3 people**.
6. **Make sure to select your highest score as your active submission before the deadline.**
7. **If you're in a group, use Gradescope's group feature to add your group members. Only one member must submit.**
8. Use `./helper zip` to create your submission if you plan on submitting by ZIP file.

Setup

Clone the **Starter Code**. The `helper` tools work exactly the same as Project 0—see **Using the Local Autograder** for more info.

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.

Note: We won't be using `libsocket` for this project. Feel free to use the standard socket APIs.

Quick Info

- Synchronize sequence numbers with SYN
- Must perform an integrity check (XOR all bits, must be 0)
- Minimum/initial window size of **1012 bytes**; current window determined by receiver's Flow Window field
- Fixed retransmission delay of **1 second**
- Must retransmit after **3 duplicate ACKs**
- Maximum segment size of **1012 bytes**

Packet Layout

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

Field Descriptions

- Sequence Number (SEQ; 2 bytes)
  This number represents the packet number. When sending data, the first packet number should be randomized (e.g. 789). For example, if there are 2000 bytes in stdin, the first packet (SEQ = 789) has the first 1012 bytes and the second (SEQ = 790) has the last 984 bytes.

- Acknowledgement Number (ACK; 2 bytes)
  This number represents the next packet that the receiver is expecting. For example,

> if the receiver has packets 3, 4, 7, and 8, the receiver will send a packet with ACK 5 (missing packets 5 and 6).

- Length (LEN; 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.

- Flow Window (WIN; 2 bytes)
  This field lets the sender know the maximum flow window the receiver can handle in bytes. For example, if the receiver sends an packet with this field set to 2000, the sender should not send any more than 2000 unacknowledged bytes at a time. **Note:** the initial window size is always 1012 bytes, or the maximum segment size.

- Flags (FLA; 2 bytes)
  This field holds various packet metadata. 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 (P)arity flag is set to ensure data integrity. **Taking the XOR of all the bits in a packet should yield 0.** If not, this packet is invalid and should be dropped.

  The S flag is the least significant bit, the A flag is the second-least significant bit, and the P flag is the third-least significant bit. You may access their values like so:

  ```
  uint16_t flag; // Flag value from some packet
  bool syn = flag & 1;
  bool ack = (flag >> 1) & 1;
  bool parity = (flag >> 2) & 1;
  ```

  Note: do not make this field big endian. We're defining significance not with respect to the integer abstraction.

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

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 {
    uint16_t seq;
    uint16_t ack;
    uint16_t length;
    uint16_t window;
    uint16_t flags;
    uint16_t unused;
    uint8_t payload[0];
} packet;

// Assume the socket has been set up with all other variables
char buf[sizeof(packet) + MSS] = {0};
packet* pkt = (packet*) buf;
```

```
int bytes_recvd = recvfrom(sockfd, pkt, sizeof(packet) + MSS, 0, (struct sockaddr*) &server_a
>
uint16_t seq = ntohs(pkt->seq); // 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

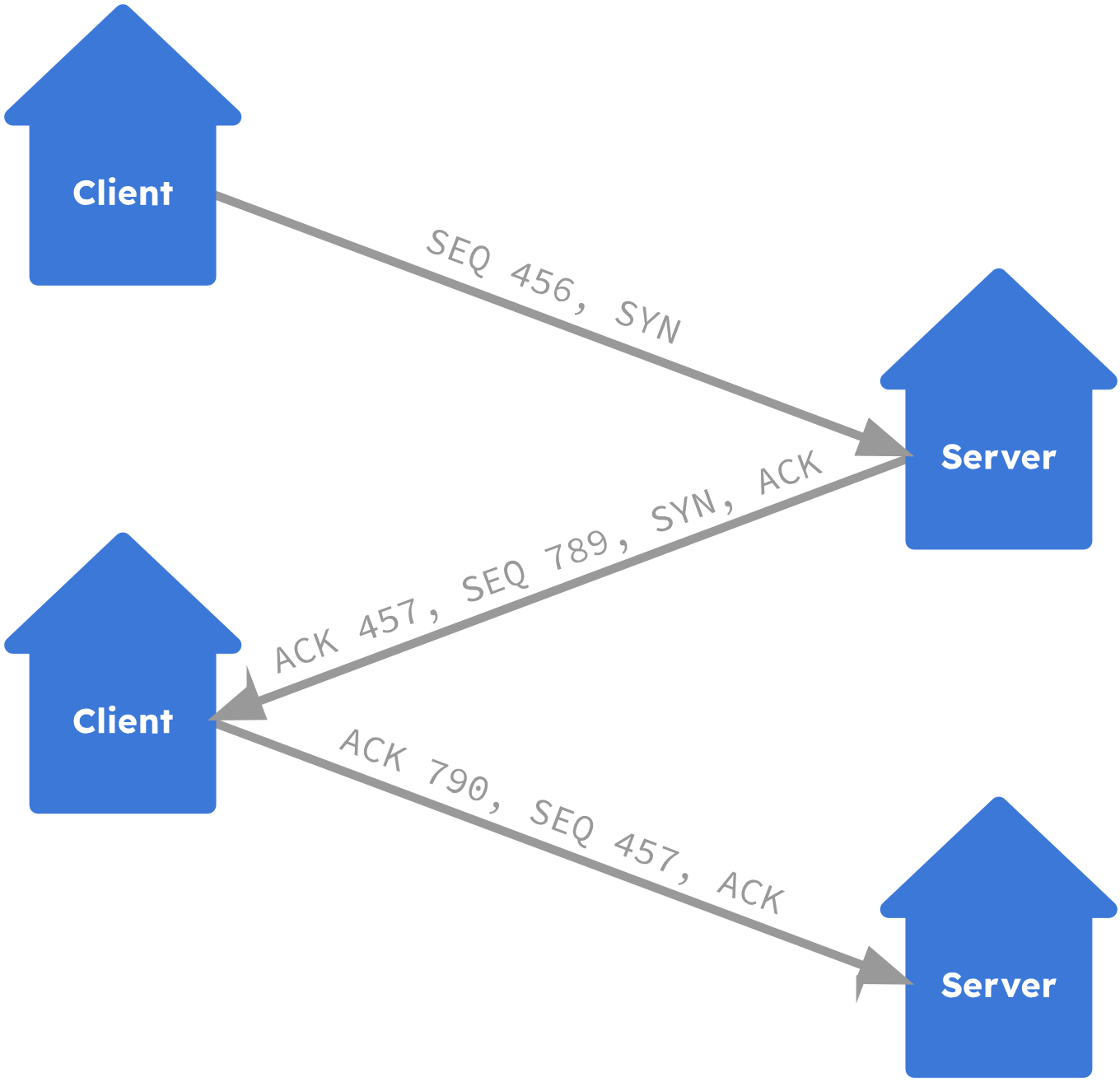
The client and server will perform a "three way handshake." This three way handshake is to ensure that both initial sequence numbers are in sync.

1. Once the client connects, it sends a packet with the SYN flag set and a randomly chosen initial sequence number in SEQ.
2. The server responds with another packet with 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 set to 0 (unless there's a real payload, then increment SEQ by 1 from the original client's SEQ number).

Please note that all of these packets may have real payloads. Pay attention to the length field to determine whether or not the packet contains data. While you should make the handshake retransmittable, for the sake of testing, we won't drop any handshake packets. **On the same note, even if there's no data available in standard input, you must send all 3 packets with empty payloads.**

Also, when choosing your initial sequence number, try to keep it low (so you don't have to worry about sequence number wraparound. The reference solution starts at sequence numbers equal to or below 1000).

>



(Assumes there are payloads in each packet.)

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 receiver's window specified in the Flow Window field. Note that the default window size is 1012 bytes (the MSS), and it will never decrease in size (watch out for reordered packets). You may only keep sending once you receive acknowledgement of packets (which reduces the amount of bytes 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 as-

sumed 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 30
- SEQ 31
- SEQ 33
- SEQ 34
- SEQ 35

you'd write out the contents of packets SEQ 30 and 31 and remove them from the receiving buffer. SEQ 33, 34, and 35 remain in the 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, 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.

Integrity Checking

Occasionally, the autograder may choose to flip a random bit (and only one bit) when proxying packets between the server and client. (See the Autograder Test Cases for the exact rates.) You'll need to detect this and drop the packets accordingly.

For the receiver, XOR all the bits (including the parity bit) in each received packet. If this operation doesn't result in **0**, drop the packet.

For the sender, with the parity bit 0, XOR all the bits in your packet (including the parity bit). If the operation results in **0**, send the packet. If not, flip the parity bit and send the packet.

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)

- What should I send as my flow window size? We recommend keeping it as `MAX_WINDOW` (defined in starter code as 40 times `MAX_PAYLOAD`) to keep it simple.
- The reference solution sets the flow window to `MAX_PAYLOAD`. Each time it receives a packet, it increments it by 500 bytes until it reaches `MAX_WINDOW`.
- Make sure that **every** number (SEQs, ACKs, lengths, etc.), **except for** `flags` is in network order/big endian.
- 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.

Autograder Test Cases

Note: Gradescope is the ultimate source of authority for your score. While the local autograder tries its best to replicate Gradescope's environment, it doesn't always match up (computer performance, kernel configuration, etc). This is especially true for Project 1, where testcases are terminated after a certain interval.

0. Compilation

This test case passes if:

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

1. Handshake

1. **Client SYN:**
   `handshake test_client_syn` **(5 points)**
   Runs your client against the reference server. If the reference server returns its own SYN-ACK packet, this test case passes. Note: no data is sent in these test cases.

2. **Server SYN-ACK:**
   `handshake test_server_synack` **(5 points)**
   Same as 1, but with the reference client. If the client sends its own ACK packet, this test case passes.

3. **Client ACK:**
   `handshake test_client_ack` **(5 points)**
   Same as 2, but in response to the server SYN-ACK. This test case passes if some sort of response was sent after the SYN-ACK.

2. Reliable Data Transport

1. **Reliable Data Transport (Your Client <-> Your Server): Small file (10 KB):**
   `rdt test_self` **(20 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. You'll receive partial credit based on the percent difference.

2. **Reliable Data Transport (Your Client <-> Your Server): Medium file (100 KB), Drop (5%):**
   `rdt test_self_drop` **(20 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 (10 KB):**
   `rdt test_client` **(5 points)**
   Same as 1, but uses the reference server instead.

4. **Reliable Data Transport (Reference Client <-> Your Server): Small file (10 KB):**
   `rdt test_server` **(5 points)**
   Same as 1, but uses the reference client instead.

5. **Reliable Data Transport (Your Client <-> Reference Server): Medium file (100 KB), Drop/Corrupt (5%):**
   `rdt test_client_drop` **(5 points)**
   Same as 2, but uses the reference server instead. Randomly chooses packets to flip one bit. (Drop and corrupt events are independent.) **Timeout: 30 seconds.**

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

7. **Reliable Data Transport (Your Client <-> Reference Server): Large file (500 KB), Drop (1%):**
   `rdt 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%):**
   `rdt 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):**
   `rdt test_ginormous` **(5 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 accept your best scoring one (please do select it as your active submission). If you're in a group, use Gradescope's group feature to select your group members (only one member must submit). 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 `helper` tool has a zip method that automatically creates a compatible ZIP file.