CS5450-HW1

Shunzhe Yu & Xiangru Qian

October 5, 2018

1 General Description of the Protocol

1.1 Basic Design of Protocol

Our implementation of simplified TCP followed the diagram below.

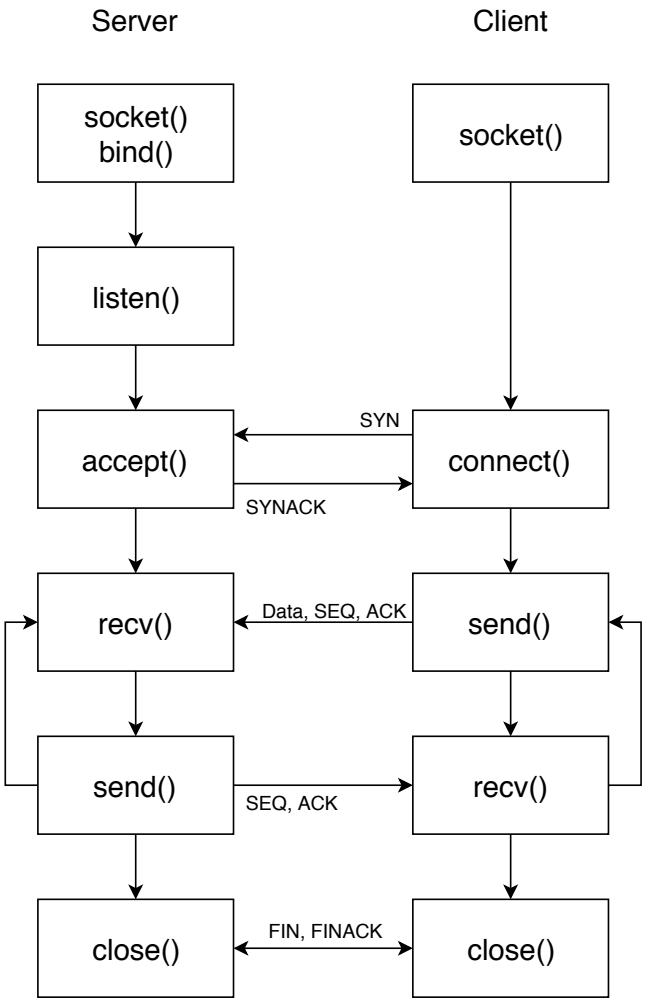

Figure 1: Diagram of TCP logic

After a server is started, it will bind the socket to an assigned port. Then it listens to any connection request and waits in that state. Once a SYN packet is captured, the server will reply a SYNACK packet and move to the state of receiving data. Each time it receive a packet in this state, it will reply a DATAACK to client and write the data to a buffer. If packet loss or timeout or data corruption happens, duplicated DATAACK will be sent, until the next expected packet arrives. If a FIN packet is received, the server would reply a FINACK and shutdown it self, since it's a two hand shakes process.

As for the client, it will open the socket and try to connect to the assigned IP and port by sending SYN packet to it. It will wait until a SYNACK is received. Then it moves to the next stage. The client would cut the file into pieces and wrap each piece in a packet. N packets will be sent at a time, depending on the window size. The client will move on to send next packets if the expected next DATAACK is received. Otherwise, if duplicated packet is received or timeout occurs, it will resend the packets, until receiving the right DATAACK. After sending all packets, client will initiate closing the connection by sending a FIN packet. It will shut down itself once a FIN/FINACK is received.

1.2 Some comments

Here are some clarifications about our protocol implementation.

- Only two hand shakes needed for establishing and closing connection.
- The program runs in a single thread, i.e., only one connection to the server are allowed and the server won't fork new process upon a connection request
- The window size is either 1, 2 or 4. When a packet is lost or a timeout occurs, window size will drop to 1. When a packet is successfully received, the windows size will increase by 1, if it is not at maximum value 4.
- The TIMEOUT is set to be 1 second. The connection will close if 5 consecutive time out happened.

2 Tricky Parts of Implementation

1. checksum:
   The original checksum function takes uint16 as input, however, the data is in uint8 (char), which would lead to errors. We modified the given checksum function to adapt to uint8 data..
2. The global state contains the last sent packet. When a timeout happens, handler will resend this packet. The tricky part is, this global state should be update only when an ACK is received, instead of changing it right after sending packets in a window.
3. The packet loss in maybe_recv is tricky. Since even the packet is lost, the function still returns a legitimate data length, with an empty buf (random string in uninitialized memory). We can only determine whether the packet is lost.
4. logic of close():
   Either client or server can initiate closing the communication by sending a FIN packet and wait for a FINACK or FIN. If FINACK is received, exit directly. If FIN is received, reply

a FINACK and exit (this is in case of FINACK lost and the target client/server is already exited). We did this instead of simply waiting for FINACK, because it's impossible that the client and server close at exactly the same time. So the sencond one to close may never hear from the first one for FINACK (the FINACK it sent before closing could be lost).

5. Logic of seq_num:
   During the implementation of seq_num, we found that there actually two plausible methods to assign values to seq_num. First, which is also the method we implement in our code, is to use the position in the sender's byte stream. For example, if the sends has sent a packet of seq_num = 1 and body_len = 1024, and has successfully received the acknowledgement of that packet, the next packet's seq_num it is about to send is 1025(1 + 1024). The number refers to the position of the sender's data byte streams, thus is not consecutive numbers. The other pattern is to use the consecutive index number to identify each packet. For example, if the sender has sent the packet of seq_num = 1, the next packet it is about to send will have seq_num = 2. These two methods of assigning seq_num both works under the project's context. However, we prefer the first option as it aligns closely with the TCP transmission in reality.

6. Data type cast:
   for **ssize_t maybe_recvfrom(int s, char *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)**. It takes string for buf, but actually we need struct gbnhdr, which includes the header. However, casting to char* won't affect the function of it, i.e., **maybe_recvfrom(sockfd, (char *) &buf, sizeof(buf), 0, client, socklen)** works fine. Additionally, as for the boundary of seq_num and ack_num, we noticed that they are previously unit8_t type, which can easily go overflow and be incompatible with other 32-bit integer. Thus, we modified the header file to assign both of them to have unit32_t type.

7. Address or value:
   At the early stage of development, it is easy to make mistake by misusing pointer and real value, especially in the case where **maybe_recvfrom(int s, char *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen)** and **sendto(int, const void *, size_t, int, const struct sockaddr *, socklen_t)** for the last argument, they have different type, thus be careful when calling these apis.

3 Instructions for Running the Program

1. Compile the files gbn.h, gbn.c, sender.c, receiver.c by placing them in the same directory with Makefile, and use the commands:
   make clean
   make all

2. Run the program by:
   receiver <port> <filename>
   sender <server IP> <port> <filenam>

3. Sample commands:
   ./receiver 1234 recv.dat
   ./sender localhost 1234 send.dat

4. We highly recommend you to pipe the output into log file, since it could be verbose and long.
   ./receiver 1234 recv.dat ¿ receiver.log
   ./sender localhost 1234 send.dat ¿ sender.log