Simple Window-Based Reliable Data Transfer in C++

Computer Science 118

Spring 2018

Pochao Yang 204631541
Yizhu Zhang 504577340

Project Expectation

In this project, we will implement a simple congestion window-based TCP protocol. There are two hosts, the client and the server, and they communicate using the User Datagram Protocol (UDP) socket, which cannot guarantee data delivery and integrity as TCP.
After client establishes a connection using three-way handshake, the server receives the file request from the client. If the file exists, the server will divide it into multiple packets of max size 1024 bytes, with customarily designed packet header before sending to the client.
If a packet is lost during transmission, the timer of that specific packet will be triggered and followed by a retransmission of the packet. Since actual rate of packet loss or corruption in LAN may be too low to test the program, we will emulate packet loss via the tc command in UNIX.
As long as the packet loss rate is not 100%, the file transmission will eventually succeed, and we will terminate the connection using FIN/FIN-ACK procedure.

Implementation Description

Connection:

At the beginning, we use common UDP socket programming procedure to set up sockets on both sides and check for errors. Then the client side will establish a connection to the server via three-way handshake. It first sends a request with type SYN. The server replies with type SYN + type ACK. Finally, the client acknowledges back with type SYN. To shut down the connection, the server sends FIN message after it finishes sending data. The client replies with type ACK + type FIN message. If this packet is lost, the server will retransmit this FIN message until ACK is received.

Header Format:

For each header of the packet we create, there are 5 parts. The timestamp part calls timestamp() function in which we get time of day and convert it to a number in milliseconds. Time is recorded when the packet is sent. The type part is used to indicate different types of the packet, i.e. SYN and ACK. The length part is the length of data. The sequence number part is the offset of first byte in this packet and the ack number part is the next byte the sender expects to receive in the next transmission from the receiver. At last, there is a reserved area for the payload, which is not included in the header.

Messages:

Messages regarding packet status are outputted to the console. On the server side when it is sending packet to the client, the message being printed will be “Sending packet" [Sequence number] [cwnd] (“Retransmission") (“SYN") (“FIN”), and when receiving, the message will be “Receiving packet" [ACK number]. On the client side when it is sending packet, the message will be “Sending packet" [Sequence number] (“Retransmission") (“SYN") (“FIN") and when receiving, the message will be “Receiving packet" [ACK number]. Several debugging messages will also be printed out when some errors happen, or undefined behavior occurs. For example, ("---- Unexpected data packet seq = %d. Wait for packet %d. ----\n", response.seq, ack_num) will be shown whenever the seq number of the packet just received cannot match with the expected offset.

Timeouts:

We use the default value, which is 500 ms as the timeout threshold for the timer over each packet being sent. We check the timestamp of all the packets in the sending window and compare it with the current timestamp to find out if a packet is still inside the window after 500 ms. If this happens, we assume that the packet is lost during transmission, therefore a retransmission will be performed in order to preserve the data integrity and reliable transfer.

Window-Based Protocol:

We use Go-Back-N protocol to achieve reliable data transfer. A window of fixed size 5120 bytes, which is equal to 5 packets, is used on the server side. Every time when the server sends a packet, it stores this packet in the window, waiting to be acknowledged. Since GBN protocol uses cumulative acks, the server can be sure that any packet who has the sequence number smaller than the acknowledge number sent from the client is delivered successfully. So, in the process_ack() function, the server uses a while loop to find the packet who has the sequence number that is matched with the acknowledge number and deletes all the packets before that packet. If the server doesn't find the packet, which means that all packets in the window have arrived, so it simply clears the window.

Difficulties

Since packet loss can happen at any time during the connection, we need to consider if the connection request sent by the client is lost, or the following ack sent by the server is lost. In other words, the first two steps in the three-way handshake should be handled in other ways. Hence, we use retry mechanism. In the very beginning, we set ack_num to be a special value. By checking that value, we can make sure that the request is delivered successfully or not and go back to request again.

Another difficulty we faced is the blocking and non-blocking issue. When we use recvfrom(), this function can easily block and nothing can move on. To solve this, we use two functions fcntl() and setsockopt(). The second function also allows us to set the time we want the socket open to receive any more data so that the client can implement TIME-WAIT mechanism.

Compile and Run

To compile the basic version of the project, type: `make ncc`
To compile the version with congestion control, type: `make`
To run the basic version of the project, type:

```
./server [port]
./client [hostname] [port] [filename]
```

To run the version with congestion control, type:

```
./server_cc [port]
./client_cc [hostname] [port] [filename]
```

Conclusion

This project helps us to understand better how TCP provides reliable data transfer and how congestion control is implemented. We also learn how to emulate the network environment, which makes it closer to reality so that we can get a feeling about how researchers develop network.