Jeffrey Shu (004124780)Vincent Wong (604139038)

**CS 118 Project 2 report**

***Introduction***

For this project, we implemented a **Go-Back-N** reliable data transfer protocol that is designed to transfer single files between a server and multiple clients. As an RDT protocol, our design can tolerate a high rate of packet loss, corruption, or delay without compromising the transfer operation. Our server is also capable of supporting multiple concurrent transfers. However, our protocol does not support congestion control.

***Packet Structure***

The packet structure for our protocol looks as follows:

4

3

2

1

Bytes:

Sequence number

FIN

SYN

ACK

*Unused*

Length

Data

The size of the header is 8 bytes. There is a 4-byte sequence number, a 2-byte length field, and three 1-bit flags for ACK, SYN, and FIN. The remaining 13 bits are unused. The length field denotes the size of the data payload that follows. Our packet, including the header, has a maximum size of **1024 bytes**.

Because our protocol is layered on top of UDP, error checking is implicitly handled by the kernel using the UDP checksum. As well, the UDP header provides us the source port and destination port, while the IP header provides us the source IP and destination IP, removing any need for us to include those fields into our protocol.

***Basic operation***

Because we're following the GBN model, this means that our file sender maintains a sending window denoting which packets are in transit, while the client only needs to maintain a single packet buffer for receiving data.

To illustrate our protocol process, we first examine the sequence of steps that will happen in a completely successful file transfer (i.e. no packet loss/corruption):

1. The client requests a file by sending the server a packet with **seq 0** and the **syn flag on**. The payload of the packet must be the filename of the file to transfer.
2. The server then immediately sends the file's contents. For the *i*th data packet, **seqi must be seqi-1 + leni-1**, where seqi-1 and leni-1 is the sequence number and length of the previous packet. The first data packet has a seq of 0. The sender window (i.e. unacknowledged data packets) can be arbitrarily sized.
3. For each data packet the client receives, the client sends a packet with **seq <expected\_seq>** and the **ack flag on**. The *expected\_seq* is the sequence number of the next data packet that the client expects to receive. If the data packet's number matches that, then *expected\_seq* is increased by the packet's length before the ACK is sent out. Otherwise, *expected\_seq* remains the same as the ACK is sent (implicitly denoting a NAK). For the first data packet, *expected\_seq* is 0.
4. For each ACK packet that the server receives, the server moves its window forward by the length of the acknowledged packet. After moving the window, the server reads from the file again and sends another data packet. If there are no more data to read from the file and all packets are acknowledged, then the server sends a single packet with the **fin flag on**.
5. The client sees the FIN packet, and initiates a 1-second timer before responding with a packet with the **fin and ack flags on**.
6. The server sees the FIN ACK packet, and sends its own packet with **fin and ack flags on** before terminating.
7. The client sees the FIN ACK packet, and terminates.

***Reliable data transfer***

Now we introduce packet loss and packet corruption. Our program is capable of simulating the two scenarios independently, but in essence they are handled in exactly the same way. To understand our mechanisms for dealing with them, we will review the procedures above in light of possible packet loss.

1. When the client first sends a file request packet, it also begins a 100-millisecond timer. If this timer runs out before the client receives any response from the server, the client will resend the request packet and restart the timer.
2. As the server is sending data packets, it is also waiting for ACKs as well. It has its own 100-millisecond timer that counts down starting from the moment it sent its last data packet. If the timer runs out without the server receiving any *useful ACKs* from the client, the server will resend all of the data packets in its window. Here, a useful ACK means the **seq of the ACK packet is within the window**, allowing the window to advance forward. Note that a useful ACK doesn't have to directly acknowledge the leftmost packet in the sender window. Because the client's ACK sequence number is known to be cumulative, an ACK can be used to advance multiple packets. This follows the GBN model.
3. As mentioned above, the client only updates its *expected\_seq* when it receives the data packet with the correct sequence number. This also follows the GBN model.
4. Once the server is done transferring data, it sends the FIN packet and activates a 25-millisecond timer. If the server does not receive a FIN-ACK before the timer is done, it will resend the FIN packet and restart the timer. If it does receive a FIN-ACK, it will send its own FIN-ACK back and immediately end the request without waiting for a response.
5. When the client receives the first FIN, it begins a 1-second timer before sending the FIN-ACK. The purpose of this timer is to give the server time to resend its FIN if the FIN-ACK gets lost. If the timer expires, even if the client never got the server's FIN-ACK, it will assume that the server has received its FIN-ACK, and thus terminate.

***Difficulties***

The most difficult part of the protocol to design is the connection termination. Whichever side terminates first has a chance to leave the other in a stuck state, due to the possibility of a packet loss. The problem comes down to how to let both the server and the client know at the same time that the other will terminate. We approached this in a similar way to TCP, in which one side sends a FIN signal and the other side acknowledges it, but continues to wait for a small period of time to allow the possibility of the FIN-ACK not getting through. In our case, we set our timer to 1 second. This is a fairly long time, and so we allow the server to send a final FIN-ACK so that the client can immediately close. If that FIN-ACK doesn't reach the client, then the client simply must wait the full 1 second.

Another difficulty is in maintaining the state machine along with the window for the sender. Every ACK that arrives must be checked against the window one packet at a time. The window is implemented as a ring buffer, with packets being allocated and deallocated as appropriate. A simple function can be used to iterate through the window and resend packets. The state meanwhile is simply represented as an integer. Both the server and the client ended up with three distinct states, representing the connection establishment, data transfer, and connection termination stages. These states determine what to do when receiving a packet, or when a timer expires.

The final difficulty was a bit of a self-inflicted one. It is not a requirement that the server needs to handle multiple concurrent requests, but we managed to implement that anyway. Because UDP is connectionless, there are no sockets that each child process can hold in order to read from. If multiple children read at the same time from the same UDP socket, there will be a data race where they consume packets intended for other requests. Thus, there must be a system where the parent multiplexes the datagrams for each of the children. Internally, this is achieved using a table of source IP + source port to PID mappings. The parent receives datagrams, looks up the IP/port, and forwards it to the appropriate child that is handling that request. The forwarding is done via a Unix datagram socket established during pre-fork set-up, which allows atomic message retrieval just like a UDP socket.