Joe Abi Sleiman
Feston Kastrati
Lab 2
Networked/Distributed Systems
Slip days used: 2

Protocol description

In this report we describe our implementation of a simplified version of the *Go-back-n* protocol, *Go-back-2*. The protocol was implemented in C and is a slight modification to the actual protocol, since our implementation alternates between Go-back-1 and Go-back-2 depending on certain conditions, as opposed to remaining on one *n*.

Initialization

The protocol starts with a 2-way handshake. The client sends a SYN packet, the server receives this packet and sends back to the client a SYNACK. On this acknowledgement, the connection is official and the client starts to send data packets.

Data transfer

The client has a window size of 1 or 2, but the server always has a window size of only 1. When the client's window size is 1, we say the protocol is in *slow mode*, and when its window size is 2 it is on *fast mode*.

Slow mode: Slow mode is the simpler of the two and it is what the client starts out with. It sends a packet with seqnum 0 and waits until it receives the ack with the same seqnum. After a packet is sent, a timer is started and if within the timer the ack is not received the client sends the same packet again and resets the timer. If the ack is received in time, the next packet is sent and the timer reset as well. Thus, sequence numbers alternate between 0,1 to avoid duplicate packets.

Fast mode: In fast mode, the client sends two packets without having to wait for an ack. The timer is set when the first packet is sent out and on timeout both packets are resent. The sequence numbers range during fast mode are [0,1,2]. The server expects the last unacknowledged sequence number. If the expected sequence number arrives, it gets acked. If the seqnum is out of order the server resends the ack for the last acknowledged packet. This means that the acks the client receives are cumulative. So, if the client receives acks out of order, for each ack it receives it is guaranteed that all previous packets were also successfully delivered.

The protocol switches between slow mode and fast mode on certain conditions. Every time an ack is received within the timeout period, the protocol switches (or remains) to fast mode. On the other hand, every time the timeout expires for a packet, the protocol switches (or remains) in slow mode.

Finalization

After the client is finished sending data, it sends a FIN ack to the server, and the server responds with a FINACK. At this point the connection between the two is terminated.

Difficulties

We decided to implement the protocol gradually. First we implemented the easy parts of initialization and termination, with SYN, SYNACK, FIN, and FINACK methods. Next we decided to implement the data transfer. We started off with small files that fit inside the 1024 data length limit. And then moved up to files greater than 1024 by splitting up the packets manually.

One main difficulty we found out during testing is that we could successfully transfer text files of arbitrary size but binary files likes compressed ones, images, or songs. This was due to the fact that we used function like **strncpy**, **strlen**, etc. which only read arrays up to the “EOF” or “\0” character. So we refactored our code to use **memcpy** and manually passed the length of the file. This resolved the problem for binary files and we could successfully transfer arbitrary size files.

Next, we decided to implement the timeout and slow-mode. The timer was a relatively easy concept and easy to implement. We were splitting the packets and transferring them successfully. To validate our code, we switched to using *maybe_sendto* to simulate packet drops and corruptions. To do so, we implemented the sequence numbers for accepting packets in order (caused by dropped packets), we implemented resending on timeout (caused by dropped packets), and ignoring packets/acks when checksums didn’t match (caused by corrupted packets). We successfully finished and all test files were successfully transmitted using slow mode. Test file Giant4.test, takes around ~3:30 to complete.

Next we decided to implement fast mode and the protocol to switch between the two. This is where we had our greatest difficulties. The source code for slow mode we created was extremely hard to refactor to include fast-mode. So, we decided to start from scratch for gbn_send and write fast-mode with slow-mode together. By this point it was late enough (hence our 2 late day slips) and we were running into different bugs. Yet, we managed to complete it and transfer files with both slow mode and fast mode, even after introducing *maybe_sendto*. For reference, we provided two copies of our solution, solution 1 has more documentation, solution 2 is final and complete.
Solution 1: Slow-mode only, with complete congestion control and successful file transfer.
Solution 2: Slow-mode and fast-mode, with congestion control and successful file transfer.
Again, Solution 2, should is the complete one, but we included both anyways.
We learned a great deal from this project (a much better understanding of C)