PROJECT 2

Simple Window-based Reliable Data Transfer

KE LIAO & SHARON GREWAL
KE'S UID: 904285392
SHARON'S UID: 704298423
Ke's SEASnet login: kel
Sharon's SEASnet login: grewal

High Level Description

The purpose of this project was to gain a better understanding of reliable data transfer using UDP protocol. We implemented the Go-Back-N protocol since we resend all packets within a given window in the case of a timeout.

This project was split into three files (`sender.cpp`, `receiver.cpp`, and `packet.h`) as well as a `makefile`.

In `packet.h`, we define what a Packet is. A Packet contains all the information necessary to ensure reliable data transfer such as a sequence number, a packet number, its length, as well as its type (since we distinguished ACKs from regular messages using a Boolean). The function `badPkt` checks to see if a packet is bad and/or possibly corrupted by comparing a randomly generated probability with the given threshold. The function `createPkt` initializes a packet with a given type, sequence number as well as packet number.

The file `receiver.cpp` contains `int main(int argc, char** argv)` which contains the bulk of the code for receiver's implementations.

We first parse the given command and ensure that we have the correct number of arguments. We then check to see if the given loss threshold and packet corruption probability are correctly defined (between 0 and 1). We then begin to set up the socket and look up the name of the host.

We begin by sending an initial request for the file. We send the message directly through the socket using `sendto`. We then wait to receive an ACK from the sender using `recvfrom`, if the sequence number of the initial packet we receive is -1, then the file is not found. Otherwise, that means we have received an ACK from the server and can begin receiving packets.

As we receive packets, we check against the given packet loss and corruption probabilities and print out sequence number when each event occurs. Also, we check to see whether we received the packet in order. If the packet is received in order, we extract its contents then create an ACK requesting the next packet with its sequence number and send it back to the server as well as update the number of packets we have received so far. If an out of order packet is received, we re-send an ACK for the most recent in-order packet.

Each packet contains a Boolean for the last packet, so the receiver keeps sending ACKs until it finally receives the last packet so that we can write the contents into a file named data.out. We also ensure that `data.out` can be opened.

The file server.cpp contains two functions: `bool isTimeout(timeval curr, timeval old)`,and `int main(int argc, char** argv)`.

The function `isTimeout` checks to see if a packet has timed out. In this project, we defined a packet timeout to be 0.1 seconds (or 100 milliseconds). Again, `int main(int argc, char** argv)` contains the bulk of the code especially since the sender will handle the job of sending the packets.

Like before, we first parse the given command to ensure for correctness before opening the socket. In our code, the server binds to the first address in the list of network addresses. We wait for a file request and check to see if it is a valid file we have. If so, then we send an ACK to confirm whether the requested file is valid. Once the data has been split into packets (assigning the correct packet and sequence numbers for each), we are ready to begin sending.

Once all packets have been sent, we then check for timeout, loss and corruption events. If a time out has occurred, then we resend all packets within a given window. Otherwise, we have a received an ACK from the receiver. If we receive all the ACKs (if the number of ACKs we have received is the same as the number of packets) we stop listening for ACKs. Otherwise, if an ACK is received we slide the window, and re-send the new set of packets.

Difficulties

At first we didn't know how to set up the structure of the packets – for instance we were not sure what should go inside the packets. We solved this by outlining what we were going to use the packets for – in this case the packets are used to send ACKs as well as actual data. At first we attempted to send ACKs separately from packets which only carry data but realized that would not work nearly as well. Since packets are used by both sender and receiver, we realized we just needed one header file to define packets that are used by both. This helps keeps packets consistent between senders and receivers.

Another difficulty was how to handle the retransmission in the case of a combination of timeout, packet loss and packet corruption events. For timeout we decided to use a vector to store the time that we sent the packets and compare them to the current time.