CS118 Project 2 Report

Joshua D. Sambasivam
404619618

Jaron Mink
904598072

Description

A simple UDP socket provides no guarantees of reliability. However, utilizing this unreliable socket, one can make a protocol on top to ensure that if any packets are dropped (assuming the socket will not drop every packet of course) we will eventually send all data reliably.

Using our own protocol which we call JJP, we are able to ensure that data is successfully and safely received. We closely follow a Selective Repeat protocol that acknowledges when individual packets have reached there destination. Adding timers, we are able to ensure that any dropped packets will be resent and eventually received. Below we discuss the basic structure of the program built in order to ensure this does in fact function as intended.

Structure

We wished to make JJP as similar as possible to an actual socket. Essentially, the only capabilities JJP can do besides setting up and closing the socket is reading and writing. However, when we read and write, our socket will perform multiple operations such as storing the data until enough room is available for it to be sent, encapsulating it into a packet with predefined headers when room is available, ensuring the once sent, we receive an ACK or it will be resent, and when receiving data, we ensure that we only read data which we successfully demultiplexed in the correct order. In order to simply this, we structured JJP into 3 classes with their own unique responsibilities. A Packer, a Sender, and a Receiver. Additionally, JJP is responsible simply for the logic combining these three packets.

The Packer is in charge of storing data until it is ready to be sent which JJP will eventually determine. Once JJP does decide that data is ready to be sent, it will request a packet with the following headers set, and a max size of x bytes. Packer will then create the header and append up to x bytes - header length to the packet, which it will then hand off to the JJP.

The Sender is in charge of ensuring that all packets given to it is sent successfully, Additionally, if any packet fails, it will reattempt to send any of the packets back out. Only when it is notified by JJP that a packet was successfully received will it discard the packet.

Lastly, we have the Receiver which is in charge of receiving packets that are possibly out of order and storing them until it is ready to be read by the JJP. It will only give back bytes that are in order otherwise, nothing will be read. Once they are read, they will be deleted from the Receiver.

Header Structure

When we began building this protocol we had to consider the additional details a packet would need to provide to be utilized. Thus we came up with the needed information: Packet Len, Sequence Number, Acknowledgement Number, Fin flag, ACK flag, SYN flag, and the data itself. Packet Len, Sequence Number, Receiver Window and Acknowledgement number all have somewhat small maximums (defined by the spec) and thus we were fit each of them within a 16 bit unsigned int (however we did chose to put data len in a 32 bit int for testing large packets). Additionally, each of the flags could be represented by a single bit and thus they were also put into a 16 bit int (with room to spare for future flags). Thus the entire header was 12 bytes long, allowing 1012 bytes of actual payload. We thus used the following structure (each row being 32 bits long)

data_length	
seq_num	ack_num
rwnd	flags
data	data
data	data
...	...
data	data

JJP Implementation

JJP utilized a packer, a sender, and a receiver to complete it's tasks. The following the general algorithm used by a background thread in JJP during connection to send and receive packets correctly.

```
While(not_closed) {
packet = read_in_single_packet
    Receiver.store_and_process(packet)
    if(data)
        Sender.send_ack()
    if(ack)
        Sender.notifyack(packet)
    if(fin)
        beginFinHandshake()

    Sender.resend_expired_pack()
    avaliable_space = Sender.get_min_cwnd_rwnd
    sending_packet = Packer.give_packet(avaliable_space)
    Sender.send(sending_packet)
}
```

Thus this would take care of the background tasks needed to be carried out. Actually using the public methods write() and read() would simply read ordered bytes out of the Receiver and store bytes into the Packer, but everything else is run on a background thread.

Difficulties

There were many unexpected difficulties in dealing with this lab. Specifically, we had difficulties in maintaining the correct sequence number within the different classes and it would have been beneficial to have that stored in a single location. Additionally, syncing up the background thread and foreground thread was more difficult than expected due to issues with blocking while reading/writing. Large files also became problematic due to the inefficiencies it took to make this program work.

Usage

In order to run the program, simply type ”Make” to compile the various classes. It will then generate two executables, client and server. To begin the server, simply type

```
./server portNum
```

and to request a file, simply type

```
./client ip_addr portNum fileName
```

It will then place the file in received.data