Lab 2: Reliable Transport

Kyle Pontius

Please note: This report will be largely incomplete. I appreciate you taking the large amount of effort spent on the code and implementation itself into consideration when reviewing my work.

1-Second Timer:

Reliable transport really relies on TCP to govern the way different states in the process are handled by the program. My implementation followed the basic TCP premise of using Sequence Numbers, ACKs, and other logic to create a reliable system for sending data.

First, I started implementing reliable transport by modifying the provided tcp.py file, loading in all the data that was available (up to the window size) designated by the program. In fact, my *send_next_packet_if_possible()* method all of this automatically. Each time an ACK came back, this method was called to handle any outstanding data, based on availability.

```
def send_next_packet_if_possible(self):
    while self.send_buffer.available() > 0 and
              self.send_buffer.outstanding() < self.window:
        new_data, new_sequence = self.send_buffer.get(self.mss)
        self.send_packet(new_data, new_sequence)
        self.restart_timer()
```

Notably, when a new packet is sent, the *self.restart_timer()* method is called. This prevents TCP from timing out prematurely, which would create unnecessary overhead from re-sending packets that had already been sent and potentially received. Scenarios where the timer is modified include: First, anytime a new packet is sent the timer is started (restarted). Second, after all data has been acknowledged the timer is canceled. Third, when an ACK comes back acknowledging new data has arrived at the receiver the timer is also restarted. This last scenario specifically handles the case where all data has been sent, and the sender is just waiting on ACKs.

If timeout occurs, all outstanding packets are essentially disregarded and the "next" packet in line is resent. The "next" packet is determined to be the packet sequence number designated by ACKs coming

back from the receiver, or in other words, the earliest segment not yet acknowledged by the receiver.

When the loss was turned on for the simulator, it was critical to handle packet loss gracefully. There were at least two ways loss occurred. First, loss could occur in the packet failing to arrive at the sender. Second, an ACK packet could be lost when returning to the sender. Additionally, packets can arrive out of order, throwing another curve ball into the situation. Obviously TCP handles these situations using clearly-defined steps. First, any time a packet is received by the receiver an ACK is sent for the largest in-order packet received at that point. This is system is cumulative ACKs. However, if several out of order packets are already in the receiver's buffer, and an ACK is received that is the last piece required to make all those packets "ordered", then the ACK sent back then jumps several packets to, again, the largest in-order packet received.

The receiver has important functionality in this system as well. The receiver must take all the in-order packets it receives and send those to the application. However, in situations where there are packets that are not in order, but still in the buffer, the receiver will simply hold on to those packets until they're in order. At that point, the receiver sends the group of packets over to the application.

Dynamic Timer:
The dynamic timer is a fairly straight forward set calculations that enable the round trip time of packets to govern the retransmission timer. This effectively reduces unnecessary traffic in the network, avoiding congestion and lost packets. There are several variables used to calculate the dynamic timer: First RTO, which represents the retransmission timout calculation. Second SRTT, which is the smooth round-trip time. Third RTTVAR, which is the round-trip time variation. For the first calculation of RTO: First, RTO is set to 3 seconds by default. Second, SRTT is set to R (first RTT measurement). Next, RTTVAR is set to R/2. Finally, RTO is calculated by SRTT+(K*RTTVAR). K has already been set to 4 at this point. Granularity (G) is mentioned in the RFC2988, but it is noted as being unnecessary in the calculation. All subsequent calculations use, in the following order:

```
RTTVAR = (1-beta) * RTTVAR + beta * abs(SRTT - R')\\
SRTT = (1-alpha) * SRTT + alpha * R'\\
RTO = SRTT + (K*RTTVAR)\\
```

It's also worth pointing out that a maximum value can be set on the timer. The RFC recommended at least 60 for the ceiling value.

Final Note:
Again, this report is incomplete, I recognize that. Getting any feedback you can offer on the quality of what is written would be incredibly helpful. Thank you!