Renee Helfert
John Villalvazo
CSE 160 - Fall 2020
Project 3
Due: 11/27/2020 @3pm

Discussion Questions

Your transport protocol implementation picks an initial sequence number when establishing a new connection. This might be 1, or it could be a random value. Which is better, and why?

By choosing the starting point of the sequence at 1, in real world applications this would not be a good choice as there is a security risk to incrementing in a known pattern when compared to starting a randomly chosen number by the application (or Operating System). By assuming we start at sequence number 1, someone with a malicious agenda could pretend to mimic an administrator role and by knowing the pattern for which number the sequence is at (or a general idea of it) that person with malicious ideas would then have the ability to inject their own packets into the datastream. With a randomly selected initial sequence number, someone from the outside (in this case say a hacker) seeing the data stream will a tcp packet is being sent can see what the sequence number is at that particular moment, but if they attempt to mimic a tcp packet, their guess of the sequence number would not match up making it almost impossible to fake a packet.

Your transport protocol implementation picks the size of a buffer for received data that is used as part of flow control. How large should this buffer be, and why?

We can hypothesize though that the best buffer size should be RTT x Channel Speed to keep packets from colliding. This buffer size should also be large enough to handle typical transmission sizes or else packets would begin cutting off needed information. When there are many more channels being used, this value should be able to be smaller.

Our connection setup protocol is vulnerable to the following attack. The attacker sends a large number of connection request (SYN) packets to a particular node, but never sends any data. (This is called a SYN flood.) What happens to your implementation if it were attacked in this way? How might you have designed the initial handshake protocol (or the protocol implementation) differently to be more robust to this attack?

If an attack like this were to happen on our code, what would firstly happen would be a lack of sockets available as the list we are using to handle the sockets location would quickly run out of available sockets. Some ways we could implement our could to help prevent against this type of attack could to have a timer of sorts that would track how many socket requests are made within a certain amount of time and if there were to many requests then our code could shut down any packet transmission or not allow more sockets to be provided until a cool down period of sorts.

What happens in your implementation when a sender transfers data but never closes a connection? (This is called a FIN attack.) How might we design the protocol differently to better handle this case?

We could design the protocol to compare the declared amount of data to be sent to the amount that was received. When all the data has been sent, we can just close the connection without receiving a FIN flag. Another solution that could have been implemented would have been a timeout feature for each clients socket connection, thus if a client does not close their connection within a timely manner, or if they have no update within a certain amount of time, then the connection to that socket would be closed so the system does not run out of sockets to provide other clients.