CSE 160 Computer Networks

Project 2

Discussion Questions & Report

By: Abdias Tellez & Renato Millan

Design Decisions

The design process for our Project #3 began with implementing the method/commands that were provide in the Transport.nc interface and the socket.h file. We added news fields to the RTT field that was provided; this includes the implementation of EWMA for the round trip-time, RTX for retransmission time, and RTO for the retransmission timeout. We used a simple array for our queueing structure to enqueue and dequeue connections to listening sockets. Since the 3-way handshake is unidirectional, we left the TCP implementation unidirectional because we know if the socket was sending or receiving data in the transport.

In addition, we decided to separate the application/server/client and transport layer because we learned from project 2 that relying on other file receiver created problems. These problems consist of the receiver receiving the same packet on both files, thus there were many duplicates. Also, we separated the 2 layers just in case need we just the application layer for project 4 in the sake for convenance. The ClientP.nc contains the implementations of the cmdTestServer/cmdTestClient commands and the TransposrtP.nc contains the TCP implementations.

Furthermore, we created an IP and TCP packet that are an array of Boolean values that avail to track the port in use and an array of sockets. We used a HashMap to save the indices of our sockets and their connections; we needed a null response, so we left an empty key of zero value to represent the null response.

Moreover, we chose an alpha value of 0.8 for the following equation: RTT = α X CurrentRTT+(1-α) X EstimatedRTT. The estimated RTT was calculated by comparing the time when a packet was sent and when the ACK packet arrived; the currentRTT was hard coded with 1024 ticks = 1 seconds in all nodes. The value was recommended by RFC 6298. Both transmissions and retransmission are handled by a single Timer component with a period length about 0.5 seconds. We had a problem with calculating the RTT on retransmission because of the ambiguity regarding which packet received ACK was acknowledged, thus we disregard it. We have yet prevented deadlock because the server/receiver of the data will detect instances when its advertised window has shrunken to zero and the client/sender has a problem checking if its window has reopened. Therefore, it will sometimes send an ACK even if the advertised window has shrunk to zero.

.

1. 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?

When generating a random value, the transport protocol picks a number between 0 to (2^32 – 1). The downside from having 1as the begging of the sequence is the higher likelihood of the connection being intercepted. Having a random value as opposed to 1, allows for a much more secure connection.

2. 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?

The buffer size should be capable of handling a transmission size. With this in mind, we decided to make our buffer size of 16 unsigned integers. In this case, we don't have to worry about needed a bigger buffer size, as project 3 does not include an application layer.

3. 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 the attacker sends multiple SYN packets to a particular node, practically flooding, our implementation will result in filling up all the sockets designed to create the connection. To improve this and be more robust to this attack, our implementation would have to detect multiple connection requests if the connection to that node already exists and reject every new connection request to that node.

4. 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?

The connection SHOULD close after transferring the data, this is part of the protocol. However, in the case that the data was not fully transferred, the connection will stay open. To prevent this, we can always add a timeout to ensure that the connection closes, regardless of FIN attacks.