Project 3 Document

Devanshu Kumar and Keerthana Madadi

Design:

We built our project 3 from the foundation of our project 2 codebase that we demoed. Our major design involved using two new timers and calling and stopping them at the appropriate places.

We used the provided command handler functions for changing the state of server and issuing a 3-way handshake connection from client to server. The command handlers setTestServer changes the state to listening and setTestClient initializes the client's port with appropriate values. To demonstrate reliability, we wired a new timer called TCP_Timer and TCP_Timeout, which retransmits the packets after timeout accordingly based on the state of the node's port. The setCloseClient function closes the client's connection by just changing the client's state to CLOSED.

We implemented the sliding window and flow control as follows. During the 3-way handshake, we sent the effective window during the SYN_ACK protocol. Once this packet arrives at the client side, we initialized the effectiveWindow instance variable from the effective window we received from server. Upon initializing, we send TCP packets with unique sequence number as long as we had space in the effectiveWindow buffer. Once we received the ACK packet for the corresponding sequence number. Before sending the next packet, we checked to see if the ack we received was the next sequence number that was being expected by comparing it to the lastAck instance variable. If the difference between the two is one, we accepted the ACK packet received and we send one more packet while waiting on other packets. Otherwise, a timeout will be called and we restate the effective window back to the original capacity the receiver can take and retransmit the packets that it is next expecting and whose ACK received was rejected.

We had the opportunity to implement the extra credit for congestion control as follows. Following up in the previous discussion of sliding window and flow control, our main objective was to demonstrate that the client should figure out the effectiveWindow of the receiver using the slow start + AIMD method. We kept most of our setup for sliding window and flow control the same. Once the connection is setup, we sent packets at a slow rate, 1 packet, and then as we received the packet, we update the lastAck and

increment by 1 or divide by 2 the effective window size demonstrating the slow start and decrease aggressively.

Discussion Questions

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?

   Even though we can choose 1 or any random value, it is better to choose a random value when establishing a new connection to ensure that the first packet is not easily accessible and therefore also makes it easier to access consequent packets. Choosing a random packet also avoids confusion on the server end when a connection is closed and another connection is opened, it can detect the changes based on the random sequence number.

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 needs to be able to accommodate at least the amount of data transmitted and also store remaining packets in case of lost packets. So, the receiver's buffer size should be larger than bandwidth * RTT *2 to allow enough data to be stored without dropping packets frequently.

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?

   By flooding SYN, the attacker is exploiting the three way handshake. The server will temporarily open multiple ports after sending the SYN-ACK while waiting to receive the ACK's from the attacker. This will leave the server un functionable until all the port connection timeouts. To avoid such an attack, we can limit the number of half - open connections in the server and have a backlog of the SYN requests. We can recycle the older half open connections when we have new half open connections in the backlog.

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?

   We can call a timeout if a connection is not closed within a certain time while sending the packets. The client can again request to reopen the connection after it is closed if the client needs to send more packets to the same server.