Computer Science & Engineering 160

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?

Choice of changing initial sequence numbers (ISNs) minimizes the chance of hosts that crash getting confused by a previous incarnation of a connection.
Also, the sequence number benefits from being random rather than being 1. If the sequence number begins at 1 rather than a random value, the number can be easily deduced and the connection intercepted. If a random value is generated, standard is 0 to 4,294,967,295, then the likelihood of the number being deduced and the connection intercepted, decreases immensely.

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 should be large enough so that it can handle a regular transmission size. Since we haven't implemented the application layer yet, data is never actually removed from the receive buffer, so it must be sufficiently large enough to handle most transmissions otherwise the transmission will stop as soon as the buffer is full. We to have a size of 16 unsigned integers.

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 our implementation was attacked, the maximum amount of sockets would be used up immediately since each SYN forces the server to allocate a new socket to handle the new connection. The server then checks in socket list to see if the node trying to connect to it already has an open connection with the server. The server could reject this new connection and avoid running out of sockets. What could happen was that the server could set a maximum sockets per host value and make sure no host has more than that amount of open connections to the server.

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 is closed once the transmission is over. When the client has sent its entire transfer size of data, the protocol automatically closes the connection. When a client sends data but doesn’t finish, that is another problem. If we don’t close the connection is the socket will never be freed to host a new connection. We could use a timeout to fix this.

Write up

Neighbor Discovery/Flooding

In order to discover the sockets and use them as part of the project, we had to start with the connection. Once a client connects to a server, we copy the listening socket and create a new one for the connection to take place and add it to the list of sockets on the client and server side. Sockets are now identified when we just find a socket with corresponding source and destination port values. We don't really use file identifiers because we're pretty lazy.

Data transmissions were changed a bit. Since we are send numbers that are already ordered, it doesn't matter. Therefore, buffering the data is easy because everything corresponds to each other. Also, since we never actually remove from the client/receiving end's buffer, our sliding window only checks if the buffer has enough space. Since we haven't reached the max size, therefore we haven't reached the max payload size. Each window has a set amount of data it can take. Since we check before we send, we should be okay. However, since nesC is different, we can't have dynamic timers for each and every packet. Instead, we just sent an array of data to be sent to the client/server that each window can take. Since we can't differentiate which packet is with which timer that nesC cannot do, we just resent the packet if its lost. In addition, sending multiple packets at a time is harder without the timer. Since we use a sequence number to keep track of which ones are lost and which ones make it, we just resend the lost packets instead of trying to backtrack.

For connection teardown, we just did a simple three way handshake, but in reverse. The big downside of such is that we might lose a packet here and there, but that should be okay since 99% of it make it. We might also have offset to help with this. In addition, the FIN signal shouldn't really cut off since the packet already made it. As told previously, building project 3 properly in order for future projects is key. In addition, we realized that nesC doesn't have the same functions as C, so we were limited by the scope of what we could do. So far we've modularized it to the point in which it somewhat works seamlessly, and removing/adding things is much easier.