CSE 160
Computer Networks

Project #4 Report

December 3, 2018

Members

Chris DeSoto
Khushpreet Buttar

1 Design Decisions

In order to implement a client/server model for this project, we found quickly that it would require bidirectional communication. Because our TCP implementation is unidirectional, each server and client would need two TCP connections between them in order to communicate.

Our first task was to design a data structure for containing the state of the chat server/client. Each server would need to accept connections on our protocol's well known port, 41. For each connection to the server, we needed a send buffer and a receive buffer and for each buffer, we needed two pointers; one to point to the last data written to the buffer and one to point to the last data read off the buffer. We also needed to store the file descriptors for the listening socket and two for each (simulated) bidirectional connection to clients.

A server on node 1 is run on boot. This allows clients to connect to it almost immediately. Protocol commands are all sent from a single python method:

chat("hello chris 98\r\n")
chat("msg hey there!\r\n")
chat("whisper khush hi!\r\n")
chat("listusr\r\n")

This was done, rather than making an individual python command for each command, to simplify the code. The command is passed to and processed by the chatApp module and string comparisons are made. If the command is valid, the appropriate logic is applied.

For the most part, the structure of the code is very similar to the test application for project 3. The chat app is modular and interacts with the transport layer as an interface. The code and logic of TCP is abstracted away from the chat app and, for the most part, it only connects, reads, and writes. This was done, as before, to mimic a real-world application written on a unix-like system. The main difference being the asynchronous nature of TinyOS and the our unidirectional TCP implementation that increased the complexity of the application.

2 Discussion Questions

1. Like HTTP/1.0, writing the application to use a single TCP connection for each command and reply would be easier and more simple. The problem is that this works against the features of TCP. Every time a request is sent it takes a full RTT to for the set-up. Then, TCP must enter slow start as data begins to be transmitted. This creates a lot of inefficiency if there is a high frequency of requests.

2. Maintaining the transport interface as an interface, rather than mixing application layer code in with it, really helped abstract the more complicated details away. It was much easier to write code without having to worry about TCP implementation details.

   The unidirectional TCP implementation on the transport layer was a huge disadvantage in this situation. Because the chat client and server needed to transfer data in both directions, the complexity of the application increased a great deal. We estimate that about half of the code in our application could be eliminated with a duplex socket interface and a bidirectional TCP implementation.

3. Our transport protocol, in most ways, has different goals and priorities for sending and receiving data. Reliability is the common factor between them. Other than that, the needs of a web server do not exactly fit. A web page consists of many resources that need to be requested from the server. This means that if connections are not kept alive at the transport layer, a new connection will need to be created for each request. This creates a lot of unnecessary overhead with connection set-up and Slow Start/AIMD. We can decrease this overhead by using persistent parallel connections, but this requires application level multiplexing which increases the complexity of every program that uses the internet.

4. One of the limitations in our design is the max character length for the chat() method. This is an existing limitation within the code base given to us when we began. The commands are passed through TOSSIM to the CommandHandler module. There the arguments for the method are passed as a message payload, which has a fixed size. If a chat message exceeds this size it produces an error in the simulator. I think fixing this bug would make the chat app more useful and functional.