-

CSCI 1680: Network Protocols and Web Service

Connor Shaw

Faculty Sponsor: Nicholas DeMarinis

Spring 2022

1 Abstract

Over the course of my studies in Computer Networks during the Spring 2022 semester, I was tasked with implementing various protocols for link layers pertaining to Internet (IP), Transmission Control (TCP), and Hypertext Transfer (HTTP). Furthermore, I had to utilize their unique design features to further augment my knowledge on how each of the designated protocols are used to support the growing advancements of technology along with the importance of scalability in relation to storage concerns over time. Each of these projects were written entirely in the C programming language, although HTTP also allowed the opportunity to write JavaScript application code for test cases necessitating local service of web pages.

2 Projects

2.1 IP - Traceroute

The Internet Protocol project allowed me the opportunity to connect nodes (hosts) using IP header information to send data between interfaces, which were represented by separately run executables, by forwarding data packets in a network-safe manner. I also had to make use of Routing Information Protocol (RIP) to setup a routing table that would track the connections between each of my virtual i.p. addresses to ensure data would reach its intended target. This routing table and forwarding mechanic would play a role in writing an implementation for *Traceroute*, which is a command which returns the ip addresses and "next hops" of all nodes found in a path as a data packet is sent from one ip to another. My implementation for this consisted of generating a struct which would hold the entries of each node along the path then sending this struct as a data packet across the link layer. At each node, the local ip and cost to reach both the incoming and outgoing ip in the path were recorded as an entry, and later sent back to the originating node to be printed to the terminal.

2.2 TCP - Congestion Control [TCP Tahoe]

The Transmission Control Protocol project utilized the foundation from the IP project in order to transmit information read from files between hosts. Additionally, sockets were used to track two-way connections and denote whether a specified nodal connection was able to read or write to certain files in a thread-safe manner. Initially, a sliding window protocol was utilized to ensure that data allowed in the connection would be allowed to fit into the specified send and received buffer sizes. However, my capstone also entailed an additional component: *TCP Tahoe*, which is a congestion control algorithm which incorporates knowledge about data loss to reduce the amount of unacknowledged data on a given connection. The send rate of the TCP protocol was enhanced by taking the minimum of the advertised receiving window size and the congestion window size such that that the sent data would not exceed the computed threshold to reduce the amount of potential data loss. Data loss was determined by tracking the amount of duplicate ACKs sent across the TCP network, and if there were 3 or more, it was safe to conclude that the threshold for congestion control should be reset, and if data was not lost, the amount of data transmitted would be added to the current running total to determine if a full window of data was sent successfully. If there was no data loss, then the congestion window size was doubled to raise the current threshold until an equilibrium was reached.

2.3 HTTP Web Server

The *HTTP* Project gave me the opportunity to implement web-based design in an architecturally focused manner. This Web Server was written with the intent to follow RFC 7231 protocols and therefore was required to respond to GET and HEAD requests. In addition, the optional POST, PUT, and DELETE protocols were added as they were the most reasonable to include when adjusting files, even though, among these three, normal web design generally facilitates only POST (e.g. for form submissions or queries). The Web Server was run from an executable that allowed for servicing a web-page on the *localhost:*8000 URL, without the use of additional tools such as Express. The server also supported the following status codes: $200, 201, 204, 403, 404, 405,$ and 409, which were utilized to denote if requests were successful, failed, or in general were not supported by the current implementation. For example, all calls that tried to open valid directories returned a 405 status code indicating that support to modify, create, or remove directory information was not supported, although files within those directories were adaptable. As an additional security measure, a directory named "public" was utilized to indicate information which a user could adjust, and allowed for organization of other directory files to remain private and inaccessible unless necessitated by the server, such as when responding with an HTML page along with an error-induced status code.