Chris VanBlargan Spencer LaValle

CS305 Networks

Prof. Amir Sadovnik

Writeup

**Introduction:**

Our Network simulator attempts to fully simulate packets/ payloads from the application layer all the way down through the physical layer. Each layer disregarding the transport and application layer are responsible for propagating the byte array (payload) through each layer. Only once this payload reaches the physical layer does a socket get opened, and is the data transmitted.   
**Description (HTTP Implementation)**

Our HTTP implementation includes HTTP 1.0 and 1.1. The only differences between these two is the concept of persistent connections meaning: once a connection has been established between the client and server, the connection remains open unless a timeout of inactivity occurs. The advantage of keeping the connection open is less overhead as the three-way handshake (syn, ack, synack) doesn’t need to be executed for every payload of data, rather, only a single handshake is needed to establish a connection, and the payload can be sent freely.

These messages start at the client with a prefix: “IP: ” where we tokenize said string via prefix and read the next token after the colon. The server is setup via command line interface (CLI), with the propagation delay, transmission delay (per byte), and a similar fashion for starting the client (appending 1.0/ 1.1).

Example: “java Server 10 2” would invoke the ServerApp.java file with the propagation delay of 10ms, and a transmission delay of 2ms. The client could be created as such: “java ClientApp 1.1 which would invoke the ClientApp.java with HTTP protocol 1.1. The client could then send: “IP: test.txt and that would request a page (GET), at the IP (localhost While it isn’t the cleanest, it’s functional. To parse these, we have a switch statement responsible for parsing the original prefix, and sending the “suffix” of a command through to be interpreted.

The expected result of these messages is a payload with the code (200/ 304, 404) and the data appended to the beginning (assuming it’s not a 404). Similarly, in our client we have a switch to “split” the header and payload from each other, and print the two to the client’s console window.

**Description (Markup Implementation)**

Our current markup implementation reads in a .txt file, utilizing the line parsing feature of BufferedReader and FileReader to remove the need to create a delimiter for new line within our file. We did, however, create a delimiter of “\nc” surrounding a file name in order to embed a text file within a text file. This causes the client to open a new connection, and request the “branched” text file from the server “off” of the original text file.

**Description (Code Design & Data Structures)**

Our project uses byte arrays as the primary data structure for message passing. If we were to recreate this project from scratch, rather than using the archaic byte array with its encoding, we’d much prefer passing an Object with member functions to be called through the layers, rather than relying on string to byte array conversions whenever the payload data needs to be verified/ modified/ cached. To further aid in the conversion from string literals (the messages to be transmitted), we invoke a stringEncode() method which converts the desired string into a byte array (byte[]) for cleanliness’ sake.

Regarding Data structures, the only ones used were strings, byte arrays, Booleans, and text-files that function as the web-page to be sent from server to client when the page is requested.

**Correctness Results**

Due to time-constraints, creating a cache system was not possible. However, we can show the correctness for persistent versus non-persistent.

Using the values of 10ms and 2ms as were in our tests, for each connection, the 10ms propagation delay will occur, followed by a 2ms transmission delay. In the case of test2.txt for example, there is one handshake, which will create 12ms of travel time, 10 from the propagation, and 2 from the syn-ack byte. The message is translated into 28 characters, and sent along with the propagation delay, creating a total Round-Trip-Time of 88ms. This shows that there is a margin of error due to the Operating System, however our results are very acceptable.

Here we can see the results for the algorithm which show a RTT of 89ms for HTTP 1.0 and 87ms for HTTP 1.1, which averages to 88ms, which is what we calculated above. This RTT calculation is used for all tests, with differences only in Handshake quantities for HTTP 1.0 vs. 1.1, accounting for the faster HTTP 1.1 connection only when embedded files are present.

**Results Analysis**

To stress-test our algorithm we manufactured various test files with differing “difficulties” from embedded files, large line length, to large line number. While doing this, we also changed the propagation delay, transmission delay, and HTTP protocol in a varied number of situations. On the next four pages, are tests providing RTT, and the average difference between the two HTTP protocols.

Test one utilizes a propagation delay of 10ms, and transmission delay of 2ms. This first line-graph shows that HTTP 1.1’s persistent connection reduces the average RTT by 29.4ms. This average difference, which would be the same time as the handshakes to open the embedded connections in HTTP 1.0, consisting of 10ms for each because of the Propagation Delay, and 2ms each for the Transmission Delay. The last 4ms can be attributed to Operating System margin of error.

With test two, we see a very small difference between the two protocols, even having the SAME RTT for two test-runs. The propagation and transmission delay are the same as test one. test2.txt only has two lines, which would help to explain why the average difference between HTTP 1.0 and HTTP 1.1 is 1.2ms. The largest overhead is the 3-way handshake, due to the small file-size, said overhead is almost negligible.

Similarly, in test three there is a very small difference between the two protocols with HTTP 1.0 being *faster* than HTTP 1.1 in a single test-run. Like previously, the propagation delay and transmission delay are 10ms and 2ms respectively. The average difference between the HTTP protocols is a mere 0.2ms. This would also indicate that the relatively-small file size doesn’t benefit from the added efficiency of a persistent connection.

Lastly, we examine test four which is our “behemoth” at a *whopping* 35 lines… In this test, we changed the propagation delay and transmission delay to 13ms and 5ms respectively. The average delay between these two HTTP protocols was 2.6ms (in the favor of HTTP 1.0). This test shows that HTTP 1.0 was *faster* than HTTP 1.1 for a varied amount of reasons. One can attempt to explain this unexpected result due to the increase delays, and secondly by examining the larger file size. This test appears to be an outlier. However, one must show this data to better illustrate the test-cases and their expected or unexpected test-results.

**Conclusion**

This project was a great learning opportunity and helped to facilitate more applicable knowledge with the client-server relationship. As far as algorithms, we found our HTTPS1.1 protocol to be more efficient in every test case other than test four in which the HTTPS1.0 surpassed that of HTTPS 1.1 by 2.6ms. While we were unable to complete both caching and an additional improvement to the client-server interaction due to time-constraints, a lot of thought went into what we would have done given an extra 24/48 hours. For example, one could reduce overall latency theoretically by compressing the payload. In doing so, the transmission delay would be reduced by a significant percent. This supposed improvement would only have benefits where the total transmission delay exceeds that of the propagation delay.

**References**

Java API and Stack Overflow for various syntactical mishaps.

**Team Contribution**

Spencer:

* Converting to and from byte arrays
* Helper methods for client/server to aid in sending/receiving of payload
* Argument parsing/ utilization
* User/client prompt interaction
* Various Null-checks
* Switches
* Code-cleanup, code-style enforcer
* String appending server-side for 200,304,404 and string splitter on client-side
* Bug-finder
* Performed some tests

Chris:

* File-reading server-side
* 3-way handshake
* RTT calculations and appends
* File embedding/ embedded file parsing
* HTTP 1.0/ HTTP 1.1 functionality
* Bug-maker
* Performed some tests