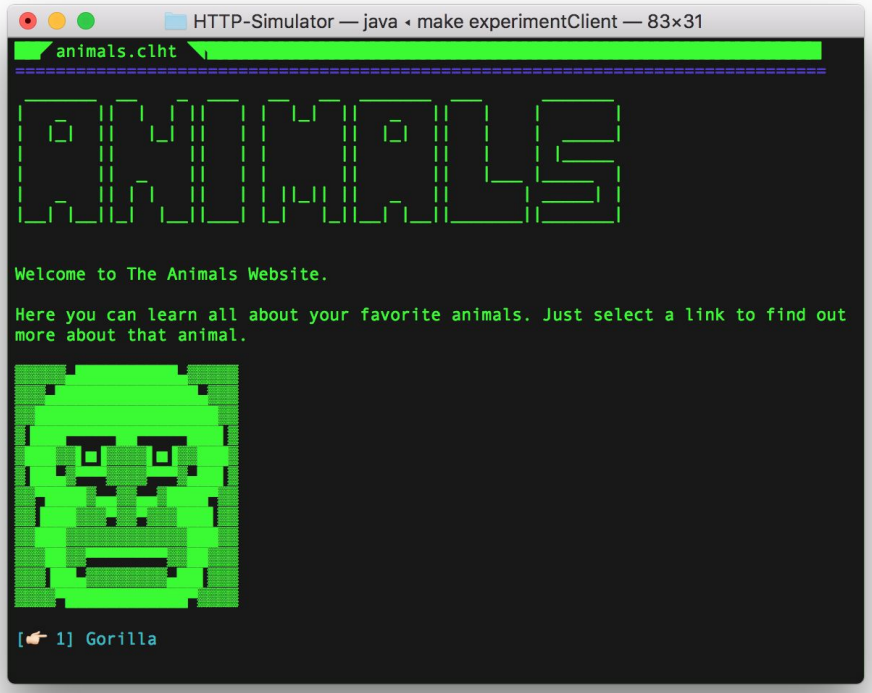
February, 23rd 2018

Professor Sadovnik

CS 305 : Computer Networks

Wassim Gharbi - Erik Laucks - Zurabi Mestiashvili

Project: Network Simulation



I . Introduction

This project is a simulation of a browser, proxy, and server over a locally hosted network. The code structure models the network layers of the Internet and pushes data between different

ports on the host computer between the virtual clients and servers. The lower layers of the network model don't do much other than pass data to and from the other layers, but the application and transport layers contain significant modifications to the provided codebase.

The transport layer includes the 3 way handshake of TCP on all new connections. The sender sends a synchronize packet, to which the receiver must send back an acknowledgement, before sending the actual request. The requests are wrapped in a TCP segment with header information related to its host and the state of the connection. The transport layer also has the capability of using persistent TCP connections, instead of the default non-persistent connection, and allows clients to disconnect from and reconnect to servers with ease and without server restart. Additionally, the network layer implements a transmission delay on all sent data that buffers by a set amount of time per byte of data, as well as a constant propagation delay on all received data.

The application layer contains most of the modifications to the given code. There is a browser that can request and receive pages and images by name and using our own implementation of HTTP. The protocol supports GET requests and returns either 200, 404, or 304 response codes. The browser locally caches files and, when it doesn't have a file, requests a proxy server, which sends its own cached version of files or requests them from the main server. This allows for the use of the If-Modified header in HTTP and the 304 response code. The browser parses received files and display pages with embedded images using Command Line HyperText. When it finds links, it provides link numbers that allow the user to navigate through them.

II . The Protocol

Syntax/Semantics

Requests can only use the GET method. Supported headers include “Host”, which is where the request came from, “Connection”, which tells the server whether or not to maintain the TCP connection after the request finishes, and “If-Modified”, which tells the proxy server how recent a cached version of a file must be for it to respond with the cached file instead of the actual file. Here is the format of our HTTP requests:

```
GET <url> HTTP/<1.0 or 1.1>
Host: localhost:<port>
Connection: <keep-alive or close>"
If-Modified: <date of oldest acceptable cached object>
```

Responses can use the 200, 404, and 304 error codes. Supported headers include “Connection”, which tells the client whether or not the connection is about to be terminated, “Date”, which is the time the response was sent, “Last-Modified”, which is the time the requested file was last changed, “Content-Length”, which is the length in bytes of the response data, and “Content-Type”, which is the custom MIME type of the response data. Here is the format of our HTTP responses:

```
HTTP/<1.0 or 1.1> <200 or 404 or 303> <Ok or Not Found or Not Modified>

Connection: <keep-alive or close>

Date: <date of response>

Last-Modified: <date object was last changed>

Content-Length: <length in bytes>

Content-Type: <text/clht or text/art>

<empty line>

Data
```

Expected Exchange

The expected exchange of messages in our HTTP protocol is the same as it is in regular HTTP. Thus, the request/response diagrams for persistent and non-persistent diagrams are the same as in the image below (*Figure 1.*):

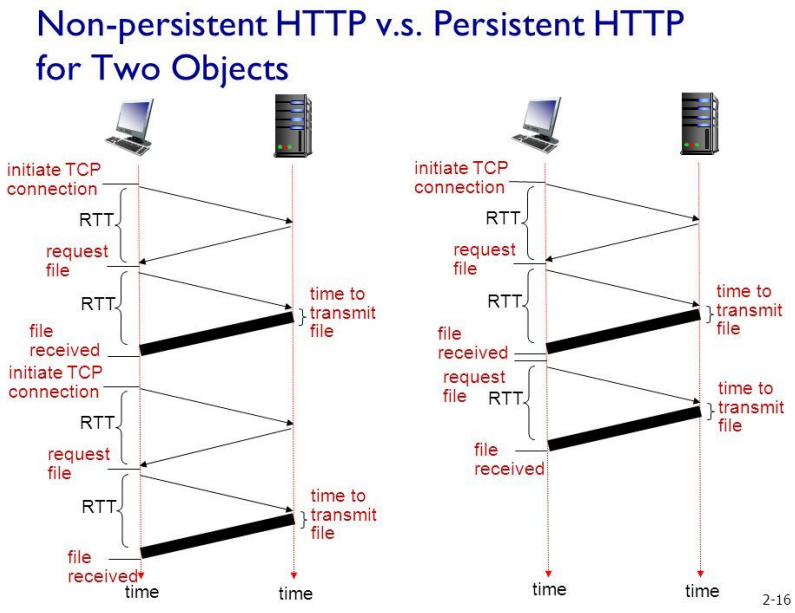
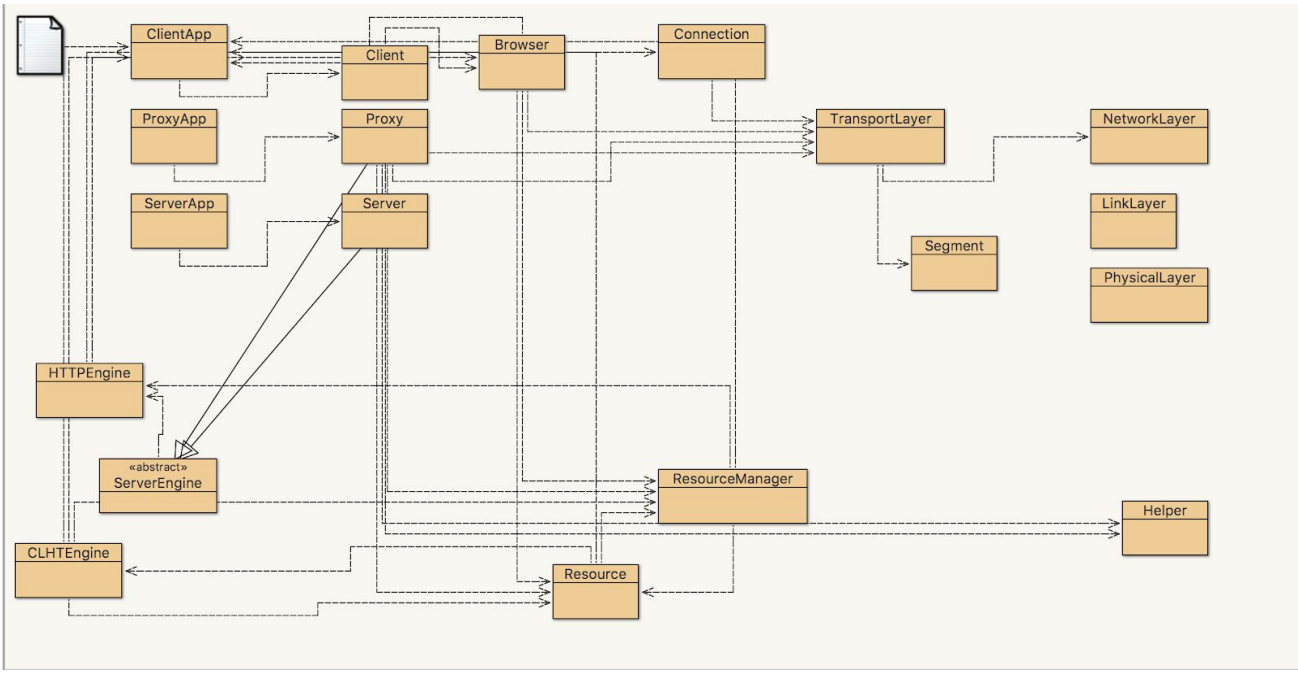


Figure 1. Non-persistent vs Persistent HTTP Connections

III . Code Design & Data Structures



The project is mainly split in 6 packages: *endsystems, engines, extra, helpers, layers and runners*. EndSystems package contains all the endsystem classes used for this project: Client, Server and Proxy. Client represents a real-world client who tries to browse a page, sends request to the server and waits for the response. Server has the same functionality as a real-world server - receives requests and returns response to the client. Proxy class in this case represents a server but also implements part of client functionality, such as sending request to the server and receiving the response. Engines package includes *CLHTEngine, HTTPEngine and ServerEngine.* CLHTEngine is responsible for rendering pages in the browser and replacing all the links in received response with right files. Finally it generates a String that later loads in the browser.

HTTPEngine implements *http's* functionality. It implements *get* method and keeps a HashMap data structure to store all the header information. For instance, following get request headers

GET Request

```
GET cats.clht HTTP/1.0
Host: localhost:8889
Connection: close
```

would be stored as:

Stored GET Request

```
Key: "Host", value: "localhost:8889",
Key: "Connection", value: "close"
```

HTTPEngine uses RegEx to parse the headers:

HTTP Raw Data Parser

```
/HTTP/(\\d.\\d) (\\d{3}) ([a-zA-Z ]+)\\n((.+:.+\\s*)+)\\n([\\S\\s]+)/
```

ServerEngine provides the server's functionality: a loop that keeps it running all the time, and generating response for received request. This class provides another layer of abstraction and simplifies the Proxy and Server classes' implementation, as they both inherit from ServerEngine.

The “extra” package contains all the classes that help client, proxy and server to handle the sent and received data. Connection class enables end systems to setup a connection with the server and initializes Transport Layer. Also it contains such information about the endsystem as the port and the type of the endsystem. Resource and ResourceManager classes represent a type of storage, that keep track of all the files the got fetched or are about to be fetched from the server. They provide such information about those files as the last time they were requested, loaded, url, type, the content. Finally, package Extra implements a Segment class that represents TCP segment. It contains as 32 element byte type array, where the TCP segment header gets stored, keeps track of the source and destination ports and generates initial sequence number. It also provides ability to correctly parse ACK number from the segment header.

Helpers package contains class Helper. Helper class provides helpers functions that are used at various places throughout the project(reading files, formating the url).

Layers package contains all the 5 layers - Application(represented by Browser class), Transport, Network, Link and Physical. Browser class represents a browser where user can insert url and request a page. It keeps track of current page, renders the received pages and handles 404 responses. Transport Layer sets up TCP connection with the server using three way handshake. It receives data from Network Layer and passes to Application Layer and vice versa. Network Layer passes data to Link Layer and then Physical Layer and vice versa.

Runners package starts each endsystem based on provided arguments. This contains - ClientApp, ServerApp and ProxyApp, executing Client, Server and Proxy respectively.

IV . Usage

Building the simulation

Build the project using the included Makefile as follows :

```
$ make
```

then run the server application, the proxy application and the client application (in that order)

```
$ make server
$ make proxy
$ make client
```

Running the simulation

The simulation runs in three different modes: browser mode, experiment mode and verbose mode.

Browser mode

Browser mode is the default mode for running the client. It provides a browser with rendering capabilities without showing any information about the network. To run the browser mode, type:

```
$ make client
```

Experiment mode

In experiment mode, the client app adds the total time it took to render a page to the end of the screen. Run experiment mode through:

```
$ make experimentClient
```

Verbose Mode

In verbose mode, pages are not rendered. Instead, the browser will display the ongoing/outgoing HTTP requests and responses and the time each of them took. Run verbose mode using:

```
$ make verboseClient
```

V . Correctness Results

The following sections demonstrates the accuracy of our simulation. The average request and response messages are shown here for size calculations, and the size in bytes of the objects being used to test is listed in the table below. The tests are structured according to the following: The client requests to load cats.clht, which has 4 image dependencies that are loaded in series. The propagation delay is 100ms and the transmission delay is 1ms per byte.

Average GET Request

```
GET cats.clht HTTP/1.0
Host: localhost:8889
Connection: close
```

Average GET Response

```
HTTP/1.0 200 Ok
Connection: close
Date: xxxxxxxxxxxx
Last-Modified: xxxxxxxxxxxx
Content-Length: xxx
Content-Type: text/clht"
```

Test Objects and Their Sizes

Item	Size (bytes)
cats.clht	2212
cat_logo.art	195
cat.art	403
cat2.art	574
cat3.art	717
TCP Packet Headers	32
Average GET request	64
Average GET response	126

Delay - 100 ms + 1ms per byte

Time to send packet - T(x) = (x + 32) + 100 = x + 132 (32 bytes is TCP packet)

Theoretical Results

The following 4 scenarios demonstrate the various functionalities of our simulation: non-persistent HTTP, persistent HTTP, local caching, and proxy caching. All of the calculations except for proxy caching multiply the sum of all the parts by 2 - this is because the request has to go through the proxy server, which has to re-request everything from the server, doubling the time.

Scenario 1 - Non-Persistent, No Caching

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Load initial file - Request + Response - T(64) + T(126 + 2212) = 2666ms

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Cat Logo - Request + Response - T(64) + T(126 + 195) = 649ms

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Cat 1 - Request + Response - T(64) + T(126 + 403) = 857ms

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Cat 2 - Request + Response - T(64) + T(126 + 574) = 1028ms

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Cat 3 - Request + Response - T(64) + T(126 + 717) = 1171ms

Total: 2 * (264 + 2666 + 264+ 649 + 264 + 857 + 264 + 1028 + 264 + 1171)

= 15382ms

Scenario 2 - Persistent, No Caching

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Load initial file - Request + Response - T(64) + T(126 + 2212) = 2666ms

Cat Logo - Request + Response - T(64) + T(126 + 195) = 649ms

Cat 1 - Request + Response - T(64) + T(126 + 403) = 857ms

Cat 2 - Request + Response - T(64) + T(126 + 574) = 1028ms

Cat 3 - Request + Response - T(64) + T(126 + 717) = 1171ms

Total: 2 * (264 + 2666 + 649 + 857 + 1028 + 1171) = 13270ms

Scenario 3 - Persistent, Local Caching (Objects in local cache)

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Load initial file - Request + Response - T(64) + T(126) = 454ms

Cat Logo - Request + Response - T(64) + T(126) = 454ms

Cat 1 - Request + Response - T(64) + T(12) = 454ms

Cat 2 - Request + Response - T(64) + T(126) = 454ms

Cat 3 - Request + Response - T(64) + T(126) = 454ms

Total: 2 * (264 + 454 + 454 + 454 + 454 + 454) = 5068ms

Scenario 4 - Persistent, Proxy Caching (Objects in proxy, not local cache)

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Load initial file - Request + Response - T(64) + T(126) = 454ms

Cat Logo - Request + Response - T(64) + T(126) = 454ms

Cat 1 - Request + Response - T(64) + T(12) = 454ms

Cat 2 - Request + Response - T(64) + T(126) = 454ms

Cat 3 - Request + Response - T(64) + T(126) = 454ms

Handshake - SYN + ACK - T(0) + T(0) = 264ms

Load initial file - Request + Response - T(64) + T(126 + 2212) = 2666ms

Cat Logo - Request + Response - T(64) + T(126 + 195) = 649ms

Cat 1 - Request + Response - T(64) + T(126 + 403) = 857ms

Cat 2 - Request + Response - T(64) + T(126 + 574) = 1028ms

Cat 3 - Request + Response - T(64) + T(126 + 717) = 1171ms

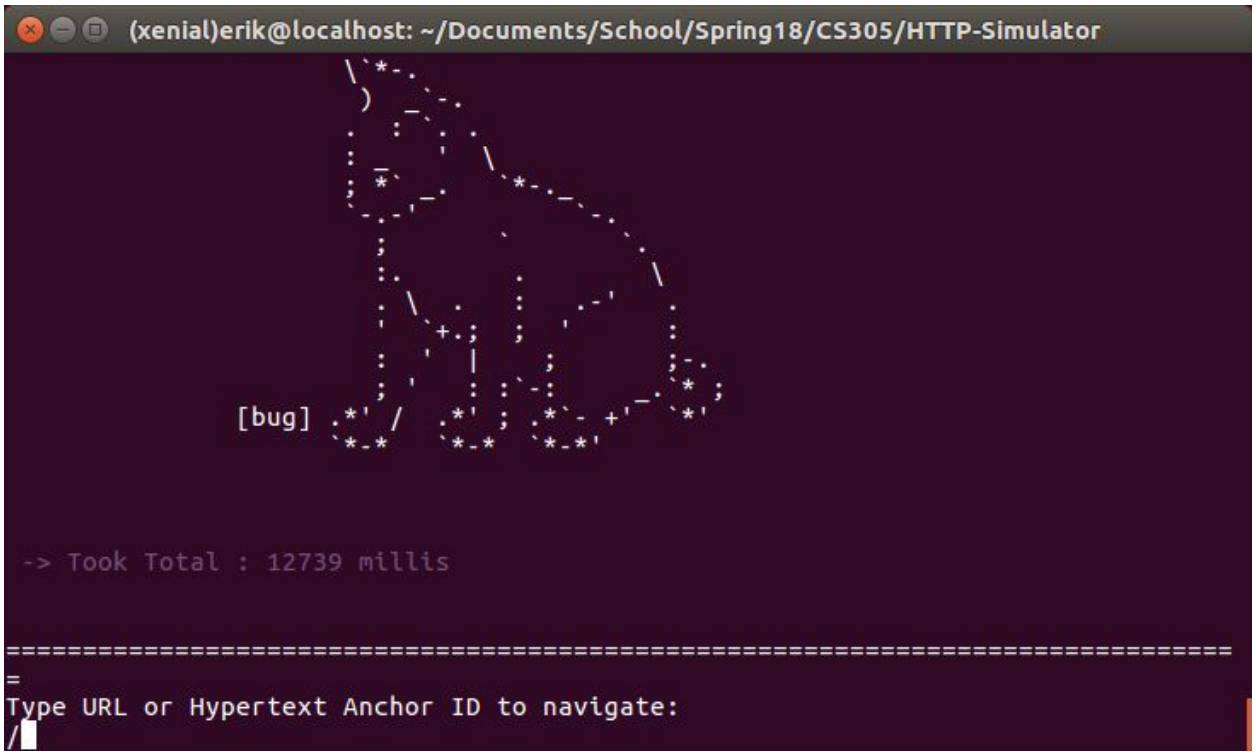
Total: 264 + 454 + 454 + 454 + 454 + 454 + 264 + 2666 + 649 + 857 + 1028 + 1171 = 9169ms

Experimental Results

Scenario 1 - Non-Persistent, No Caching

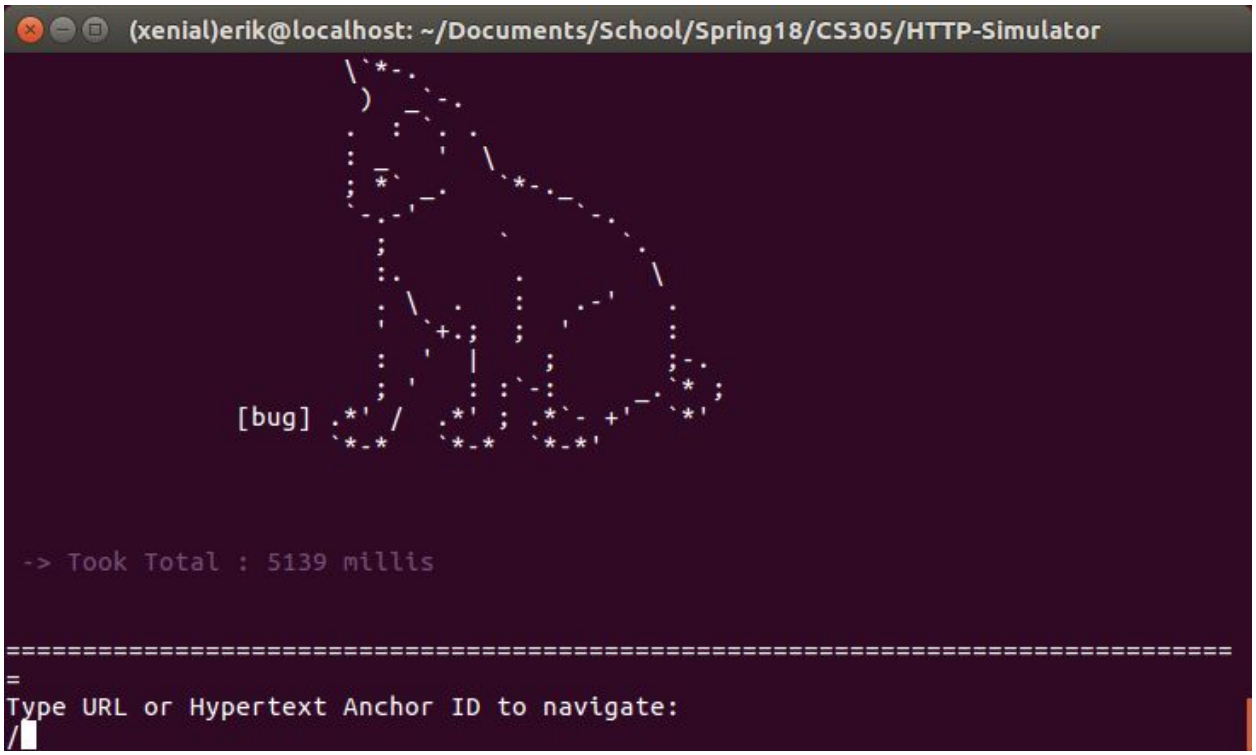
Trial 1	Trial 2	Trial 3	Average	Theoretical
15223ms	15082ms	15082ms	15129ms	15382ms

Scenario 2 - Persistent, No Caching



Trial 1	Trial 2	Trial 3	Average	Theoretical
12763ms	12739ms	12757ms	12753ms	13270ms

Scenario 3 - Persistent, Local Caching (After loading all objects into cache)



Trial 1	Trial 2	Trial 3	Average	Theoretical
5139ms	5122ms	5126ms	5129ms	5068ms

Scenario 4 - Persistent, Proxy Caching (After loading all objects into cache)

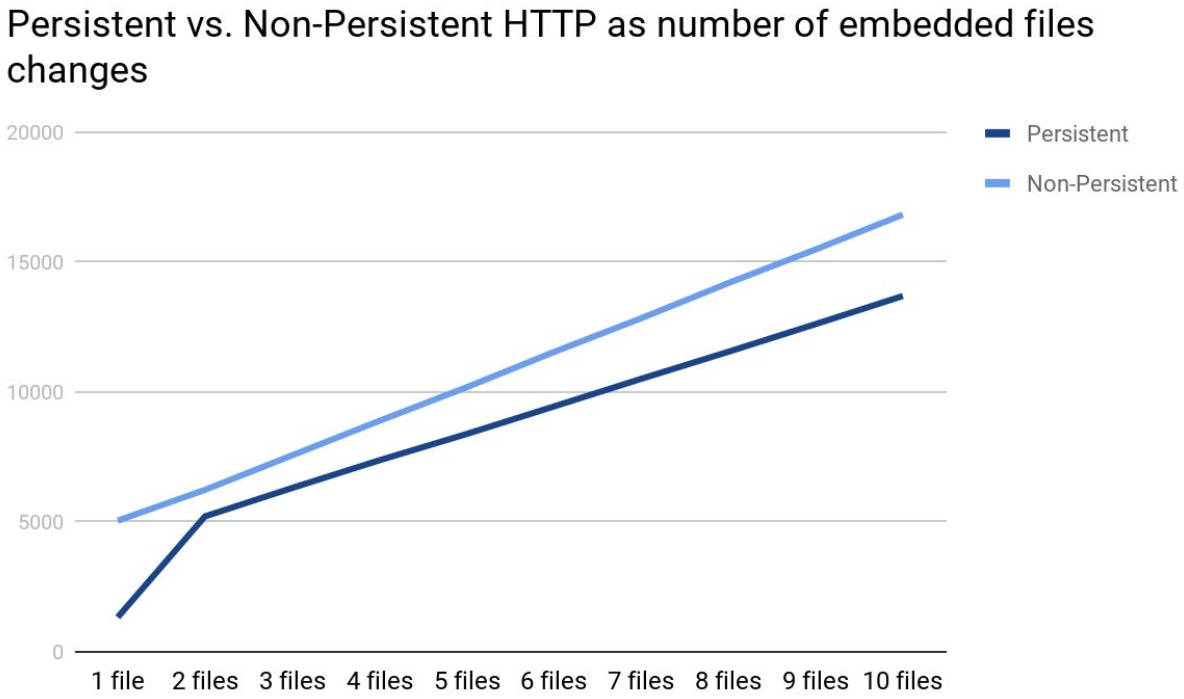
Trial 1	Trial 2	Trial 3	Average	Theoretical
9195ms	9197ms	9195ms	9196ms	9169ms

VI . Results Analysis

Experiment 1 - Persistent vs. Non-Persistent HTTP

The first experiment aims to understand the performance difference between persistent and non-persistent HTTP connections. 10 different CLHT files were created that embedded between 1 and 10 image files. The 10 image files were all identical, but differed in name to avoid caching issues affecting the time. Then, each of the 10 CLHT files was loaded in experiment mode in the browser using both persistent and non-persistent modes. The times for each of these trials was recorded and the data is presented below.

It can be seen from the graph that our simulation works exactly as expected. Persistent connections are across the board faster than non-persistent connections, which makes sense because persistent connections only have to perform one handshake, as compared to a handshake for each request. This is shown definitively in the graph because as the number of files grows, the number of requests go, and the larger the gap between the number of handshakes becomes, which increases the gap in the load times.

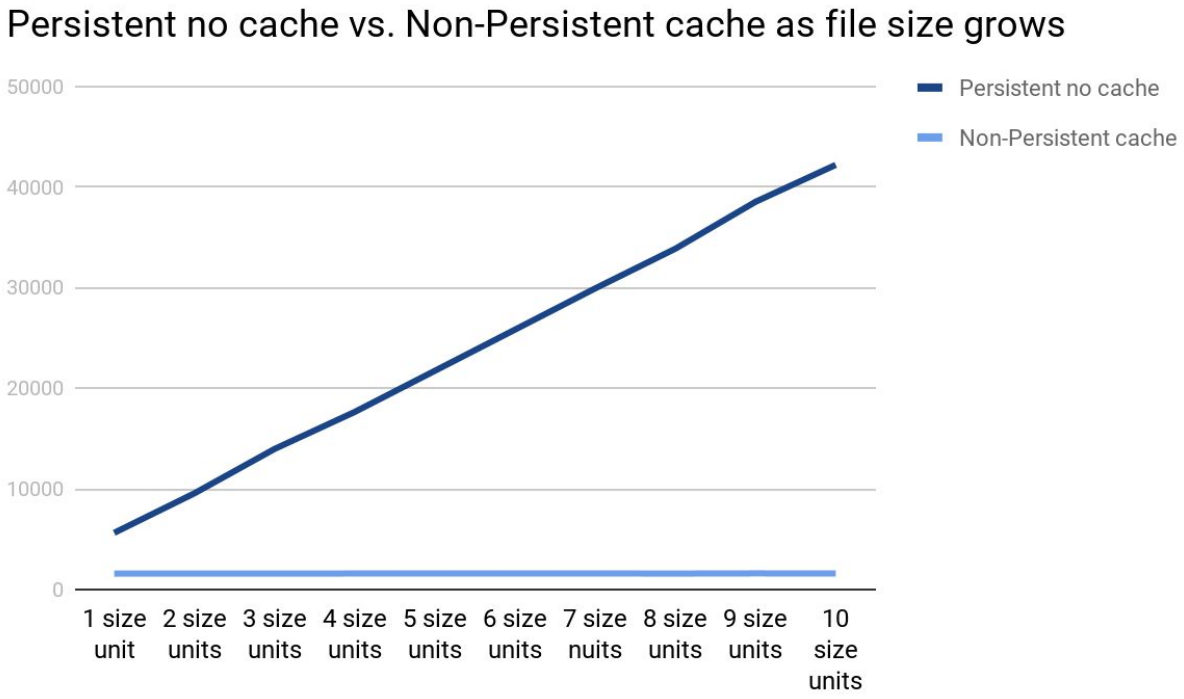


	Persistent	Non-Persistent
1 file	1334	5051
2 files	5214	6236
3 files	6310	7566
4 files	7375	8893
5 files	8387	10186
6 files	9445	11541
7 files	10509	12841
8 files	11551	14196
9 files	12616	15490
10 files	13698	16830

Experiment 2 - Persistent no cache vs. Non-Persistent cache

The second experiment aims to understand the performance difference between using a persistent connection without a cache and using a non-persistent connection with a cache. 10 different CLHT files were created that had the same block of text (the text content of cat.clht) repeated between 1 and 10 times. Each of the 10 files was loaded over a persistent connection and then twice over a non-persistent connection, with the second time recorded (the first non-persistent connection was to load the file into the cache). The times for each of these trials was recorded and the data is presented below.

The original expectation was that these would provide a good comparison as the times would be closely related, but that was proven to be an incorrect assumption. The non-cached load times increased linearly, which was expected, but the cached times were much faster than anticipated. The combination of the local and proxy caches provided a constant load time for cached files, regardless of their size. This shows that the benefits of persistent connections are far outshadowed by those of caching.

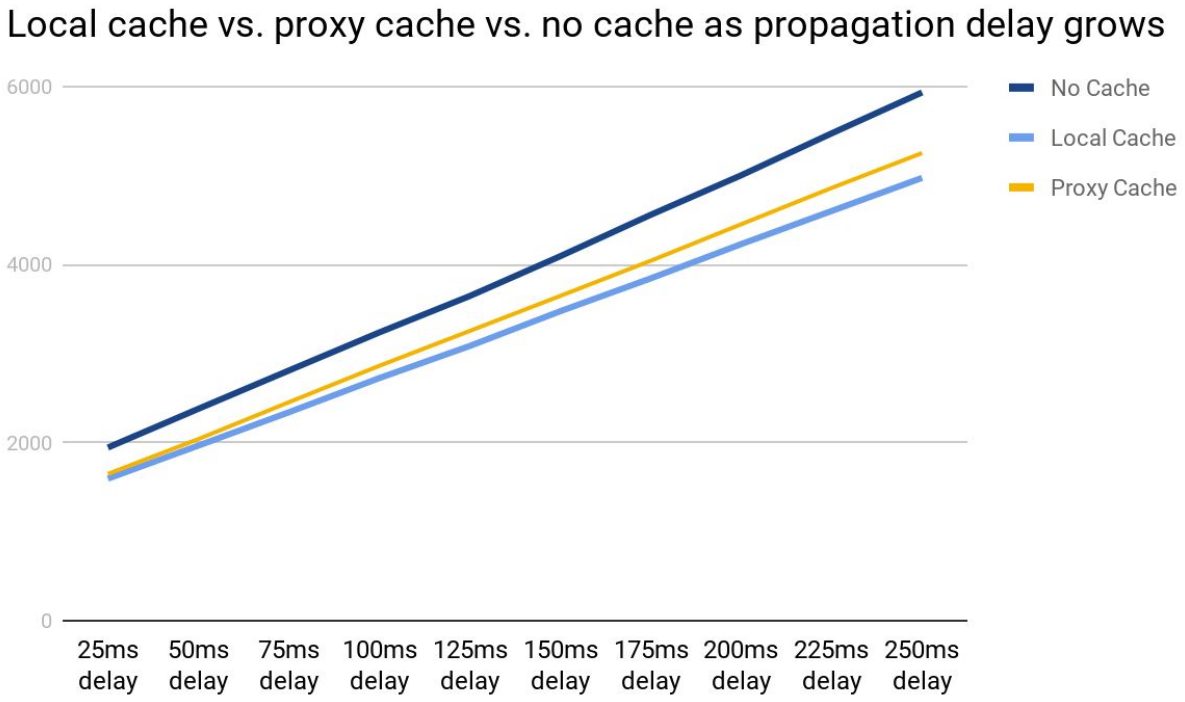


	Persistent no cache	Non-Persistent proxy cache
1 size unit	5635	1586
2 size units	9561	1585
3 size units	13972	1583
4 size units	17652	1592
5 size units	21743	1600
6 size units	25838	1598
7 size nuits	29935	1597
8 size units	33880	1586
9 size units	38544	1605
10 size units	42212	1593

Experiment 3 - Local cache vs. proxy cache vs. no cache

The third experiment aims to understand the performance difference between using the different forms of caches as delay changes. A single file, cat.clht, was loaded in the following process: the server and proxy were started with a given propagation delay, the client connected and requested the file for the no cache time, loaded the page again for the local cache time, and then exited, reopened, and requested again for the proxy cache time. This was done across 25 millisecond increments of propagation delays up to 250 milliseconds. The times for each of these trials was recorded and the data is presented below.

The results align with our original expectations, with the local cache being the fastest, followed by the proxy cache, and finally no cache. The non-cached results actually see a slight increase in the rate at which they become slower, which was interesting to see. Additionally, a gap starts to form between the proxy cache and local cache as the delay increases in the higher range, which is also interesting to note. Ultimately, this demonstrates the common sense that caching increases the speed of the requests.



	No Cache	Local Cache	Proxy Cache
25ms delay	1945	1596	1649
50ms delay	2380	1967	2040
75ms delay	2810	2341	2454
100ms delay	3238	2725	2863
125ms delay	3647	3086	3255
150ms delay	4094	3476	3648
175ms delay	4559	3845	4043
200ms delay	5003	4230	4453
225ms delay	5474	4602	4861
250ms delay	5934	4974	5253

VII . Conclusion

After examining the results of our experiments and considering the theoretical calculations, it has become clear that the optimal arrangement of parameters for efficiency from a client's perspective is to use persistent connections whenever possible and to utilize both local and middleman proxy caching. This is similar to the structure of the internet, with ISP-level and institution-level web proxies reducing traffic on end system servers and stress on the broader Internet infrastructure. Our simulation doesn't capture the full picture, given that we're dealing with a single client and single end system server with one simultaneous connection instead of many clients and many servers and many connections with real links and routers connecting them. However, it captures the essence of the principle of persistence and caching in computer networks.

VIII . References

[1] *Figure 1*. <http://images.slideplayer.com/35/10385881/slides/slide_16.jpg>.

[2] Kurose, James F., and Keith W. Ross. Computer Networking: A Top-Down Approach. Pearson, 2013.

IX . Member Contributions

- Wassim Gharbi

Contributed parts of the the simulations' architecture, creating abstractions such as the HTTPEngine and the ResourceManager. Designed and implemented the client-side interaction (the browser and its underlying abstractions and multiple running modes) as well as local caching, linking, resource management and rendering of assets.

- Erik Laucks

Contributed to various parts of the project, such as the Physical Layer, Transport Layer(persistent & non-persistent connections), generating the request and response headers for http, implemented get method, provided Transmission and Propagation delays for Network Layer. Ran the experiments, analyzed and presented in the report.

- Zura Mestiashvili

Provided abstractions such as ServerEngine, packaging, and separating end systems from runners. Implemented Caching on Proxy, If-Modified, Last-Modified and Date headers. Designed and developed TCP segments and 3 way handshake. Built the patterns for clht, request headers and response parsers. Provided code design analysis for the report.