Investigating the benefits of Web Caching

Richard Andreas, Patrick Carpanedo, Igor Solomatin, Sean Brady

Github link: https://github.com/randreas/CS655_GENI_Project
Project on GENI: Web Caching

Introduction / Problem Statement

As the modern internet has grown, being connected to the internet has become a normal part of life. Many users expected quick and reliable access to their email, news websites, and other distributed applications at home, school and work. Oftentimes, connections to origin servers that host these vital services can experience high latency, either due to network constraints. Upgrading the network infrastructure to increase the bandwidth to the origin server is often expensive. In these cases, it is important to establish caches of data outside of the origin server, to reduce the load of any bottlenecks to the server. When a user requests a piece of data, they will first be directed to a local forward cache rather than to the origin server. If the cache does not have the requested data, it will recursively handle getting the data for the client from the origin server and then store that piece of data locally. In the future, if that piece of data is requested again, the local cache will return that data to the requesting users, without contacting the origin server.

Establishing a forward cache can speed up the performance of end user networks by caching requested data for future users, but raises interesting design challenges. In this project we seek to make the investigate the following properties of caches:

- How big should the cache be (Is bigger always better)?
- How do we design an eviction policy for data objects in the cache?
- How does an empty (cold) cache compare to a full (hot) cache?

Experimental Methodology

Determining the efficacy of web caching requires testing different permutations of environments and situations (e.g. Empty cache or Full Cache). In our project we will be comparing the efficiency of Apache Traffic Server (ATS). Figure 1 below shows our Architecture Diagram.

Clients and caches will be represented in the subnet before the internet gateway. This subnet allows for high speed communication between the client and caches. The link between the internet gateway and the origin server represents the “internet” as a low throughput, high latency link from the subnet to the server.

Both caching servers will be compared in performance and the origin server in terms of latency and throughput to the client. The client will be measuring the following metrics:

1. Origin Server RTT (Response time)
2. Cache Server RTT (Response time)
3. Cache hit percentage
4. Throughput of Each Link

The experiments will be implemented in the client node with python code to execute wget commands and scripts to track metrics. The origin server will be a simple Apache HTTP server. The links speeds will also be altered to reflect varying conditions between the client and origin server.

In this environment variables of cache-size, object-size, max-age etc. to test the caching performance in a myriad of situations. If time allows, advanced features such as “Read While Writer” and Hierarchical Caching will be included to see if the benefits of web caching situations are compounded or if there lies a further complexity in enabling such features. The control data will be collected from fetching from the same batch of websites without a cache present. Ideally, Multiple websites with varying distances (from client), complexity, and size will be used to conduct our test to simulate different scenarios that a normal user may encounter.

Experiments:

We implemented our subnet on NYSERNet InstaGENI, in the following configuration shown in figure 1. The host operating system on all nodes was Ubuntu 18.04.6 LTS. All object fetching operations were run on the Client Node. All objects were hosted on an Apache web server running on the Server Node. We configured Apache traffic server on both Cache1 and Cache2 as a forward proxy with a cache size of 256MBs.(Cache 2 was used as a manual backup to Cache1). Links were established between the router and all nodes, and all experimental traffic was routed through the router node. Unix’s tc (traffic controller) was used to control the throughput and loss percentage of the links during the experiments. The throughput of each link was measured using iperf. Iperf was used to verify and record the link throughput on each link after each. All experiments were written in python with the assistance of the wget command.

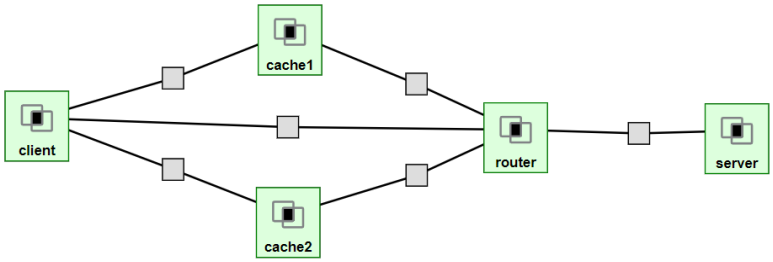

Figure 1: Geni Architecture

In our experiments, we plan to test the cache in two different ways. First we wanted to test the performance of the cache in varying Throughput to the origin server. Second we wanted to test the performance of the cache in different configurations and states. In each experiment, a predetermined fetch order of objects was established. From there python would call UNIX’s wget command and track statistics on throughput and latency of the call. The first run would only attempt to fetch the objects from the origin server creating our control. The second would attempt to fetch using the cache as a proxy, and possibly not need to contact the origin server for our object. We generate objects to fetch by using a python script which fills a text file to our particular requested size. We used this script to generate multiple files of different sizes and names.

We tested the efficiency of web caching in several ways. We tested how different object sizes, link speeds, and the status of the cache affect latency of the retrieval of the files. In our first experiment, we compared fetching the same 10MB file to different 10MB files. In experiment 1, the same 10MB file was fetched 9 times whereas in experiment 2, 9 different 10MB files were fetched. We expected the origin server to perform identically in each case, but the proxy server did not improve our overall performance in experiment 2. This is because in experiment 2, there should be no cache hits, whereas in experiment 1, after the first fetch, the file should be in the cache during the rest of the experiment.

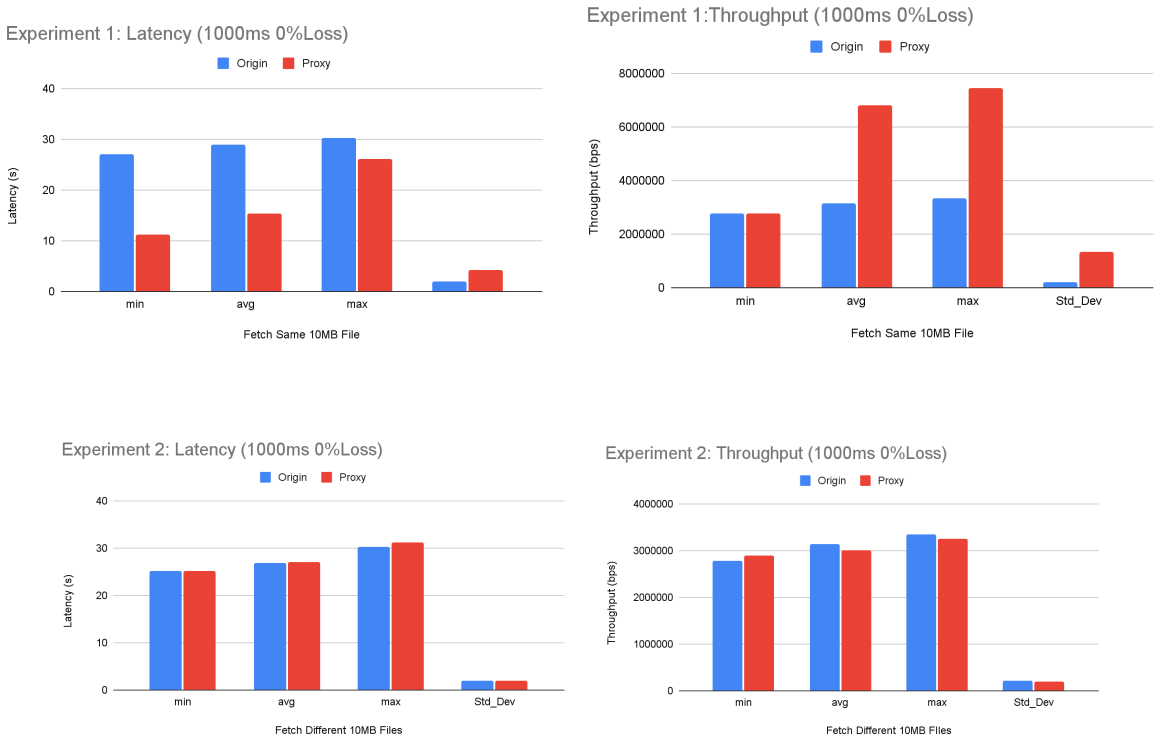

Figure 2: Experiments 1 and 2

As expected, the proxy performance was much better in experiment 1. We see that the average latency is significantly reduced in experiment 1 as well as an increased throughput. Notice that the worst the maximum latency of the proxy in experiment 1 is comparable to the average case of fetching one object from the origin server. The second experiment shows what happens in the opposite case, a 0 percent cache hit rate. The cache performed nearly identical to fetching directly to the origin server in this experiment. An important note here is that if the origin server latency is much greater than the LAN latency, the latency overhead to installing a proxy server is always negligible. This experiment not only validated the cache works but how much a cache can potentially improve the performance of web services by hosting copies of objects close to the requestor at minimum cost..

In our next experiment we tested how the object size affected fetching from our proxy to our client. We expect that alternating the object size should not affect the performance of the cache, as long as all the objects seen can fit into the cache. In the following 3 experiments, we request the same 3 objects at the client 3 times. The first iteration should have to go to the origin server to fetch the unique object, but the second and third should be contained at the proxy. We should expect a large standard deviation for the proxy latency, as the alternating objects fetch order should have a cache hit rate of 0.33 percent. The first experiment we set the object size to 10MB, the second experiment to 1MB, and last to 500KBs

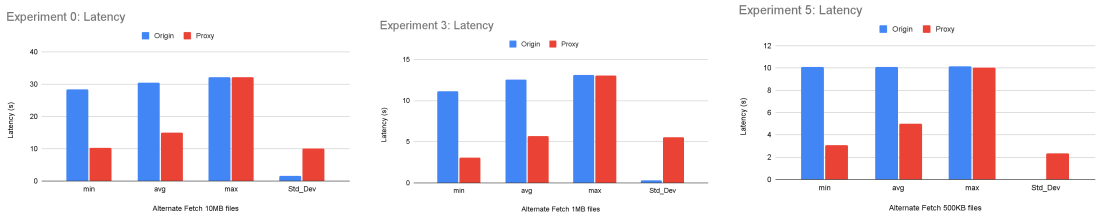

Figure 3: Alternating Files, 10MB,1MB,500KB

As expected, other than the total latency of fetching smaller files, the performance rate of the cache remains relatively the same despite the object being fetched. However it's important to note that the gains are relatively consistent, this chart shows how important it is to attempt to sort the costly objects in our cache as compared to the most recent ones. Our improved latency is much higher when fetching large objects from the cache.

In our final experiment, we test to see how adjusting the link speed of the connection to the server could affect performance. One of the more natural solutions to increasing fetching speed is to increase the bandwidth of the connection to the origin server. In the following experiments, we adjust the delay of the server link, further improving its request latency.

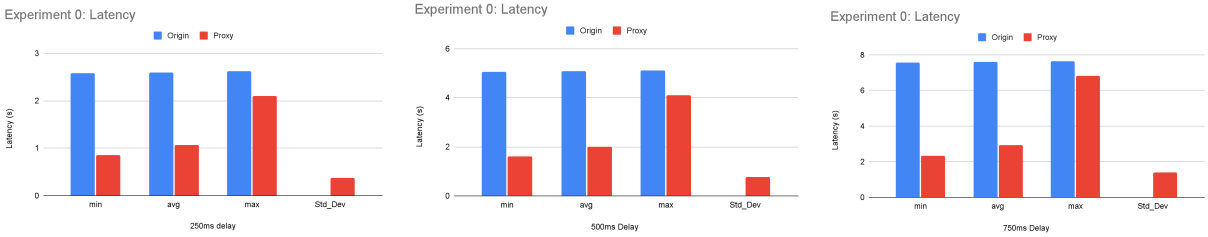

Figure 4: Alternating Link speed

As we increase the link speed, the need for a cache becomes less and less, sometimes only saving a few seconds in request time. This goes to show that if you are close to the actual origin server, there isn't really a need to store copies of data in your local cache.

Conclusions and Future work:

In this project we have shown the benefits of the cache in multiple different scenarios. Caches normally are the economical solution to getting users their data faster. However we have shown some interesting results when adjusting some of the basic parameters of the network and objects being fetched. It makes more sense to store objects which are large in size, and have a large latency in our cache. Perhaps these ideas could be explored in a unique eviction policy from the cache, improving overall performance by tracking these data points when ingesting data into the cache server.

Division of Labor

1. Richard Andreas: Creating Custom Data and run experiments
2. Patrick Carpenedo: Writing Base Code, Data Engineering and Data Analysis
3. Igor Solomatin: Systems Architecture and Environment Setup
4. Sean Brady: Writing Base Code, Data Engineering and Data Analysis