Geni Project-Web Caching

Zhitong Wu

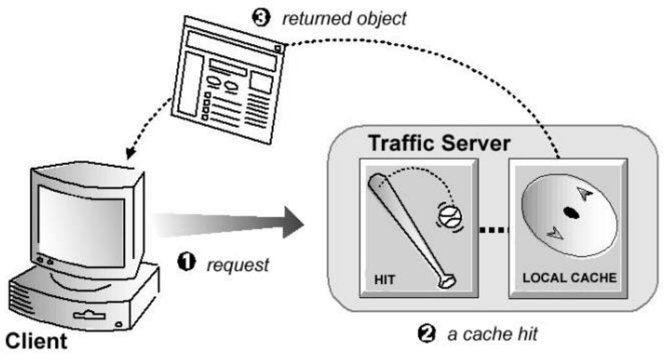
GitHub link: https://github.com/innocentgrey/GeniProject-WebCaching.git
GENI Slice name: ZhitongWu-project

1. *Introduction*

This project aims at the benefit brought by web caching. Specifically, in my experiment the primary goal is to show web caching can increase the performance of HTTP GET. The metric here is the time spent on completing 100 GET requests to www.bu.edu, and caching should provide a smaller costing time.

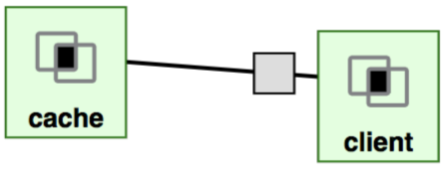
2. *Experimental Methodology*

I'm going to use strict forward proxy provided by apache traffic server, which is suffice for my goal described above.



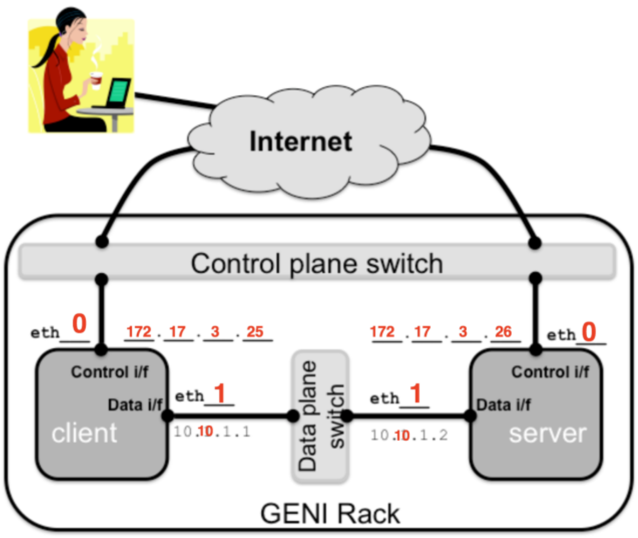
This picture from ATS website shows how the forwarding cache works. The client forwards all his request to the cache server. If it is a cache hit, the cache server will obtain the requested objects locally and return them to the client. If it is a cache miss, the traffic server will then access the original requested server, retrieve objects and cache them. The assumption is that the delay is mainly caused by the delay to outer network (my configuration), i.e. the delay between client and the cache can be ignored.

For experiment implementation, two nodes are needed.



The experiment steps include:

a. Research on the performance without web caching. In this scenario, client sends 100 requests to www.bu.edu under different delays (configured by me), each round's request time will be recorded.
To add a delay, simply add a delay to eth0 will be suffice, because this is the interface that controls the traffic between the GENI node and the outer network, as shown in our first GENI experiment.



b. Research on the performance with web caching. In. this scenario, users can manually set up the delay on cache server, and client will use cache server as a proxy, which means all requests will be forwarded to the cache server. The request time with caching mechanism will be recorded and compared with part a.

3. Results

3.1 Usage Instructions

3.1.a

For experiment part a, just run *python withoutCache.py* in the root directory of client node. This script will automatically set up different delay and plot a graph.

3.1.b

For experiment part b, you need to first set up the delay on the **cache node** (if you don't set, default will be 0 ms) by running command

```
sudo tc qdisc add dev eth0 root netem delay <your choice>ms
```

then go to the traffic server fold and run traffic server
(the configs are in fine state)

```
cd /usr/bin
sudo ./traffic_server start
```

If you can not acquire the lock file, run the following command:

```
cd /var/run/trafficserver/
```

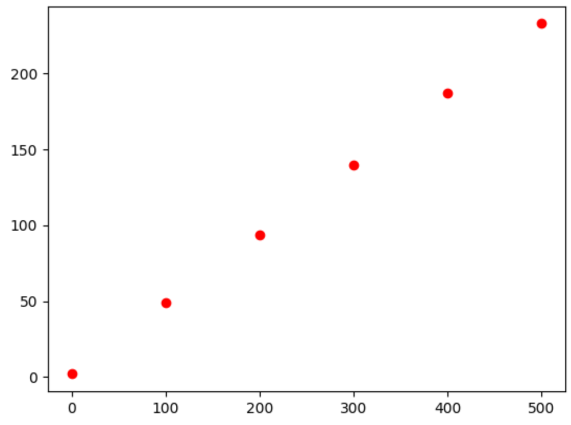
sudo rm server.lock (is asked, press y)

The last step is to run withCache.py on the **client node**

python withCache.py

3.2 Analysis

As the graph below shows, when not using cache, the total time to finish all the requests increases linearly as the delay to outer network grows.



total time(seconds) VS delay(ms)

However, the total time of experiment part b stays between 1 second to 2 second, no matter what delay we're adding to the cache server. This is because the cache objects bas been cached in the cache server, so that there is no traffic between the cache server and the outer network.

4. Conclusion

The result of my experiments show that caching can significantly increase the performance under bad network circumstance to the original requested server. What's more, caching also reduces network traffic significantly since much traffic is handled in the local network, which helps improve the capacity utility if the capacity to outer network is a bottle neck.

5. Research Summary

5.1 Headers that related to caching

In fact these headers have good explanation in RFC 7234 which I'm going to summarize, but I'm going to talk a little about the most common headers here.

Expires and ***max-age*** header explicitly define how long the object can be cached. ***Last-modified*** and **Date** are used to calculate the freshness of the object. By ***no-cache*** the client informs that it should not be served with cached objects. ***min-fresh*** and ***max-stale*** informs the client's requirement/tolerance of freshness of cached objects.

5.2 RFC 7234 Hypertext Transfer Protocol Caching

This is an internet standard document which defines HTTP caches and its cache-related headers. It standardized several things, like error handling and syntax notations, when and how responses should be stored in Caches, and how a response is constructed from Cache. The cache related headers can be divided into several categories, including Age, Cache-Control, Expires, Pragma, and warning. At the end of this document it talks about security considerations.

Basically it is standards, and it can also be a useful manual guide consisting of rules of how a HTTP cache should be implemented.

5.3 xCache

xCache is a distributed web caching architecture that aims at improving performance on edge nodes of the network. Suffering from high server response time, limited bandwidth and web page complexity, these areas have high page load times, and xCache wants to lessen the time by increasing the cache hit rate.

In the architecture of xCache, there're two level of caches. First level is called "Edge Caches", which are the counterpart of the conventional caches. Second level is called "Cloud Controller", which acts like a brain which communicates with the Edge Caches and update their cache content in a way that optimizes and trades off between improving hit rate and minimizing the bandwidth used to update them. The Cloud Controller contains two components, the Page Profiler to determine the list of active pages to profile and to learn the object-group representation of each active page, and the Cache Manager that maintains the content of Edge Caches.

The author also provides some evaluation of the performance that xCache can bring. They have proved that the improvement can be around 15% over conventional caching.

6. Division of Labor

I did this project on my own