CS 6210 Project Report:

Project 3: RPC Proxy-Server

Mikhail Isaev

Taekyu Kim

Introduction

The purpose of this project is to implement an RPC Proxy-Server for HTTP requests and analyze performance of several replacement policies for request cache.

Remote Procedure Call (RPC) is a protocol that can be used to access remote server located in distributed system like a local network from the client machine. The main feature of RPC is that it completely covers all the necessary network interaction between client and server and does it in a nice smooth manner so as the server and the client were on the same machine.

Proxy-Server works as an intermediary for resources that clients request. These kinds of servers are pretty common in distributed network systems and play the role of central moderators between clients and resources. One of the most common types of proxy-servers is HTTP proxy. HTTP proxies are frequently used to perform centralized processing of HTTP requests. It can reduce the amount of contentions in network, amount of WAN traffic, serve for security reasons, or increase web access performance. The last property is achieved by caching, e.g. saving, HTTP requests on the proxy server. Local network access to proxy server can be much faster than downloading a page from the Internet. Although caching is pretty common in Web Browsers on the client machines, proxy server can incorporate significantly larger amount of cache storage and can achieve larger performance increase.

The overall performance of caching proxy server depends on performance of the cache itself, which depends on the replacement policy. The replacement policy determines which entry will be evicted from cache if it doesn't have enough space to store new request. Different cache replacement policies and their performance is the main focus of this experiment.

For the project implementation we used C++ programming language. For RPC server implementation we used **Apache Thrift**. It is an interface definition language and binary communication protocol that can be used to create various services to work over the network using Remote Procedure Calls. Thrift supports a lot of programming languages and uses code generation to build cross-language services. For HTTP request processing we used **libcurl** - a library that provides data transfer over various protocol and supports multiple programming languages.

Work Split

This project was made by two students of School of Computer Science, College of Computing, Georgia Tech: Mikhail Isaev and Taekyu Kim. The work was divided evenly between both of us for implementing RPC client-server application as well as for necessary supporting code for running experiments.

Taekyu generated RPC-server code and implemented parser for URL-lists for the client. In addition to it, Taekyu implemented FIFO replacement policy.

Mikhail implemented Random and Largest Size First replacement policies. In addition to it, Mikhail elaborated server part for more convenient test runs.

Some parts of the work such as preparing a makefile and writing a report were prepared together.

Cache Design Description

The Cache is designed as a hash table where URL is a key, and HTTP response is a data. For implementation we used STL container **std::unordered_map**. An average search time for this data structure is O(1) with the worst case is O(N), which makes the cache look-up sublinear. The size of the cache sets with **-size n** option in term of powers of 2 bytes with the default value n=20. That means that the size of cache is 2^20 bytes = 1MB.

Cache supports two methods: **get(***url***)** and **put(***url***)**. The first method performs URL cache look-up, the second one puts new URL into the cache. If cache doesn't have enough size to store a webpage, it evicts <URL, webpage> pairs until it can insert the new value. To determine eviction candidate the special **policy** class is used. It is designed as a pure virtual **ReplacementPolicy** class, and three derived classes, **RandomPolicy**, **FifoPolicy**, and **MaxPolicy**, which implement Random, FIFO, and Largest Size First replacement policies respectively. The policy type sets with **-policy n** option. Available options are 0 for no cache, 1 for Random, 2 for FIFO, and 3 for Largest Size First replacement policies.

Cache Replacement Policies Description

When cache is full and there is a new data to be added, replacement policy chooses which cache entry to evict from the cache. In this project, we have implemented three different cache replacement policies: Random, FIFO, and Largest Size First. Replacement policy is implemented separately from cache itself. It uses its own data structure and stores only URLs and necessary supporting information. It doesn't store web page data, which is Cache responsibility. All the implemented policies has the same interface and can be used interchangeably. The exact policy which is used determines on the server start from the start options.

Random

Random policy randomly chooses data to evict when it is necessary. When cache uses random policy, it keeps all the pair **<url, web page>** into a **std::vector** data structure called **url_set**. When it needs to choose data to evict, it randomly generates the vector's index number and returns a URL to evict from the cache. It also evicts this URL from the the data from **url_set** data structure.

FIFO (First-in-First-out)

FIFO policy chooses data to evict in the order the data came into cache, so the cache evicts the oldest data the cache has. We used Priority Queue data structure from STL container **std::priority_queue** to store the state of each URL presented into cache. We used the special data structure consisted of URL and timestamp of the moment it was put into the cache to store the necessary replacement information. To generate timestamp, each time when we put data into the datacache we called **gettimeofday()** function and converted its result to **unsigned long long timestamp**. We overloaded **operator<** for the data structure in order the oldest URL to be on the top of the priority queue.

Largest Size First

Largest Size First policy evicts URL that corresponds to the largest web page presented in the cache. We used Priority Queue data structure from STL container

std::priority_queue to store the state of each URL presented into cache. We used the special data structure consisted of URL and the size of the web page correspondent to the URL. We overload **operator<** for the data structure in order the URL of the largest web page in the cache to be on the top of the priority queue.

Metrics Evaluation

We came up with three different metrics for the cache performance evaluation. We measured time the client waits for the response on his requests, time that server spends processing clients requests, and cache hit/miss rate.

The main goal of caching requests on proxy server is increasing web access performance for the client. The first metric is trying to measure this performance increase visible from the client side. It shows how good or bad caching, as well as exact replacement policy, is for the performance boost. However, this metrics is not very accurate, because it measures cache performance indirectly and it's affected by many other parameters. One of the most significant parameters is the Local Area Network topology and delays it adds to the measurements.

Cache hit/miss rate is an universal metric to measure cache performance. It's not affected by any unrelated parameters and purely describes the cache performance. However, this metric is quite artificial because it doesn't say anything about how much better is caching or particular cache policy in terms of requests processing time. The cache has certain size, but its entries can be any size, so the exact number of entries may vary and depend on each entry size. The number of entries affect performance of cache significantly.

To address this issue we are measuring request process time on server. This metric is not affected by any other parameters unrelated to server design and at the same time measures time performance of different caching techniques.

Experiment Methodology

We ran our server and client application on separate laptops. These laptops were both running Linux Ubuntu 14.04 and were connected to the same Local Area Network.

We ran a series of experiment using two lists of URLs and different caching techniques: no caching, Random, FIFO and Large Size First replacement policies. Hence, we have eight different cache runs. For each run we performed 5 experiments to average the value obtained and we varied the cache size from 32 KB to 2048 KB to show the performance dependency on the cache size. We changed cache size from 128KB to 2048KB running task1 and from 32KB to 512KB running task2. The number of URLs in task1 and task2 is different, as we ll as the total size of all web pages that can be put into cache, so the cache size of 1024KB ~ 2048KB results in 100% cache hit rate for task2 and 32KB ~ 256KB results in the large number of cache misses for task1.

The first run of the experiment performs with the “cold” caches, e.g there’s no data there. It affects performance because after the first run we have 0 cache hits. To eliminate it we “warmed up” caches before each run. Without warming up caches we would have had a noticeable amount of bias in our results. By testing warm caches we improve caching performance, but anyway, caches can increase performance only for repeated data requests, so we are just performing our experiments in a target environment.

Task Generating

The generic idea of generating task list is to mix blocks of URL lists which are divided by URL size. Initially we formed an URL list with 121 URL. Than we sorted it in descending order. To generate URL list in order, we first ran an experiment with the whole URL list in random order to get data size corresponding to each URL. After we had ran the experiment, we wrote a python script to generate sorted URL list, and from the result of python script, we divided the result into five pieces - A, B, C, D, and E. Each of five groups contained approximately 20-30 URLs. So, we got groups A, B, C, D, E with huge, big, medium, small, and tiny sizes of web pages respectively. We did not include URL block A to our tasks because some URLs in block A are larger than the cache size we are experimenting with, and larger than the majority of other web pages. It often caused heavy performance degradation with better results only for the size of the cache larger than the size of all pages.

From this URL lists we designed two task sets: task1 and task2. First task consisted of URLs of similar size with repeated parts. Second task consisted of mixed URL blocks

based on blocks D and E, but with addition of blocks with heavy web pages to perform cache flush. The exact block pattern for the test sets is given below.

First Task : D-E-D-B(big)-E-D-C(medium)-D-E-D-E-B(big)

Second Task : D-C-D-E-D-C-D-E

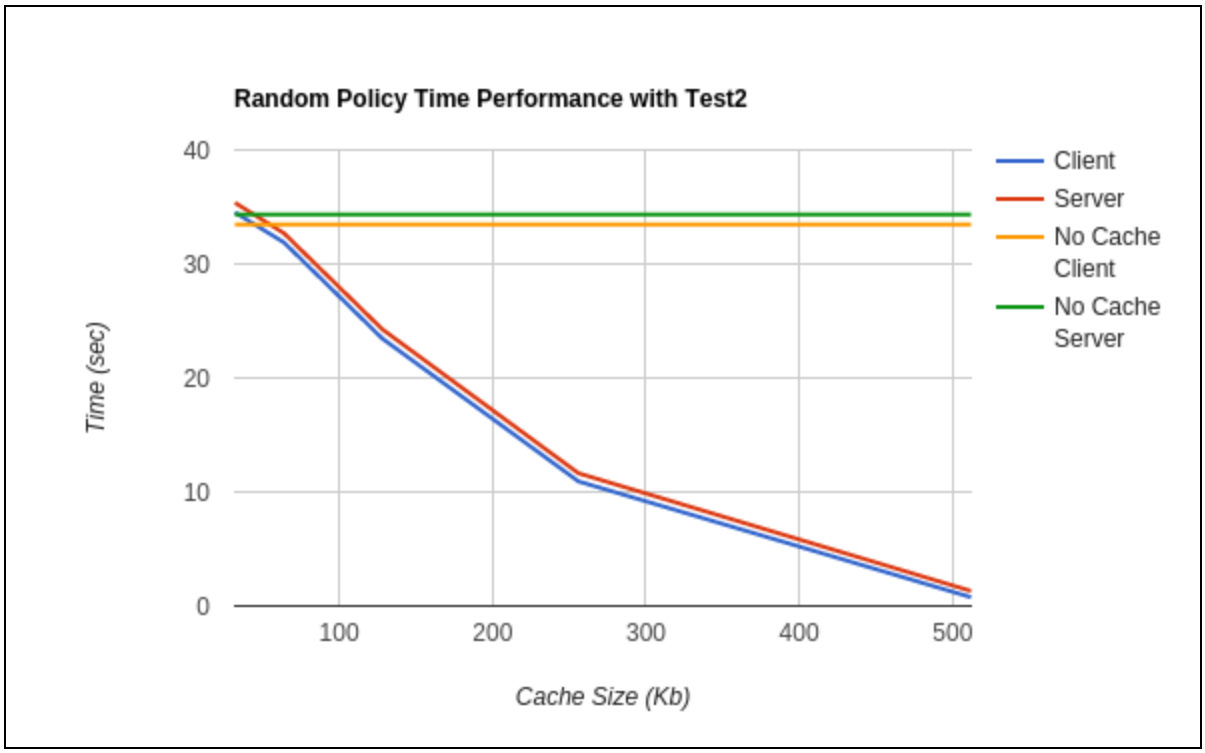
Here for the first task we picked 10 same URLs from each of blocks presented in pattern. For the second task, we picked all the URLs from block D, and 5 random from blocks E and C where they are presented.

Both of these test sets are based on the repeated tasks, but the first one focuses on the workload where user “switches context” between blocks of pages he’s using, whereas for the second test sets focus more on repeated webpages with some extra web pages that are not frequently used. That can corresponds to using web browser for several weakly connected types of activities (working on several different projects, plus social networks, plus procrastination) for the first test, and browsing web sites from your regular or daily pattern (social networks, news, etc) with loading external resources (external links from facebook newsfeed) for the second one.

Experiment Results

Random Policy


Random Policy Time Performance with Test1
60
45
30
15
0
Time (sec)
400
800
1200
1600
2000
Cache Size (Kb)
Client
Server
No Cache Client
No Cache Server


Random Policy Time Performance with Test2
40
30
20
10
0
Time (sec)
100
200
300
400
500
Cache Size (Kb)
Client
Server
No Cache Client
No Cache Server

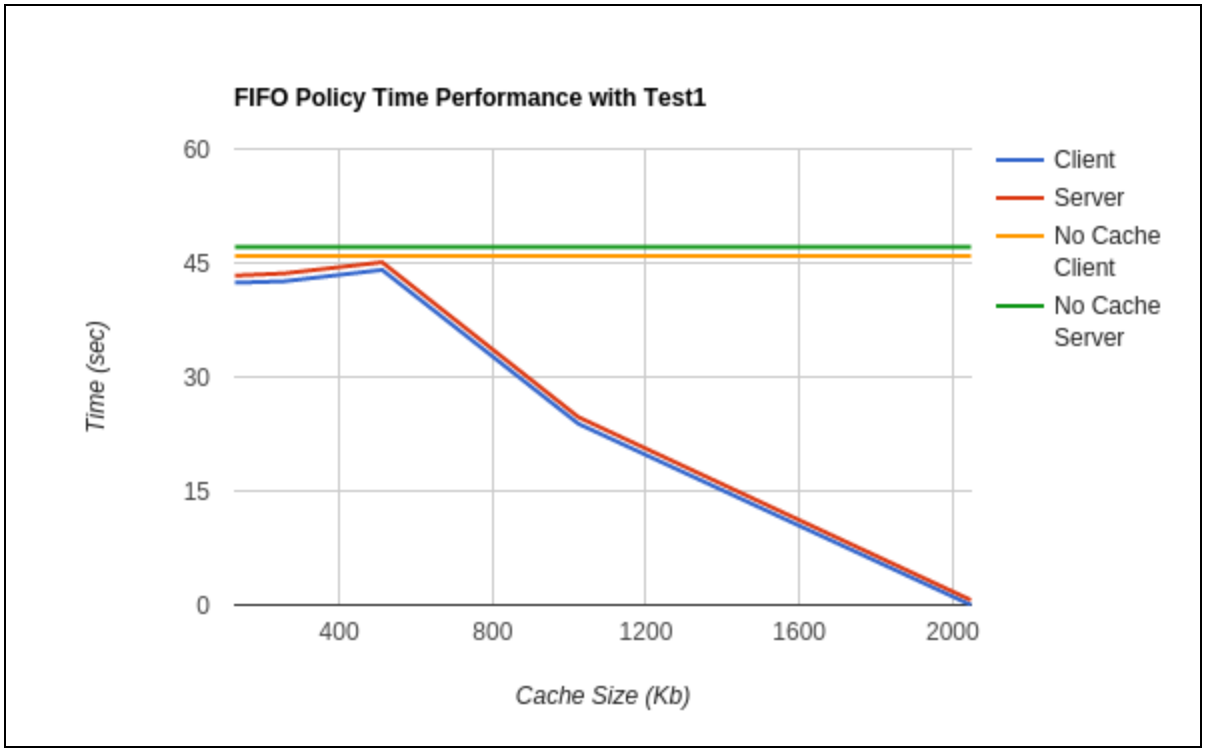
On both Test1 and Test2, Random policy shows a tendency to reduce execution time almost linearly with the increasing size of cache, which means both of them have better performance when the cache size increases.

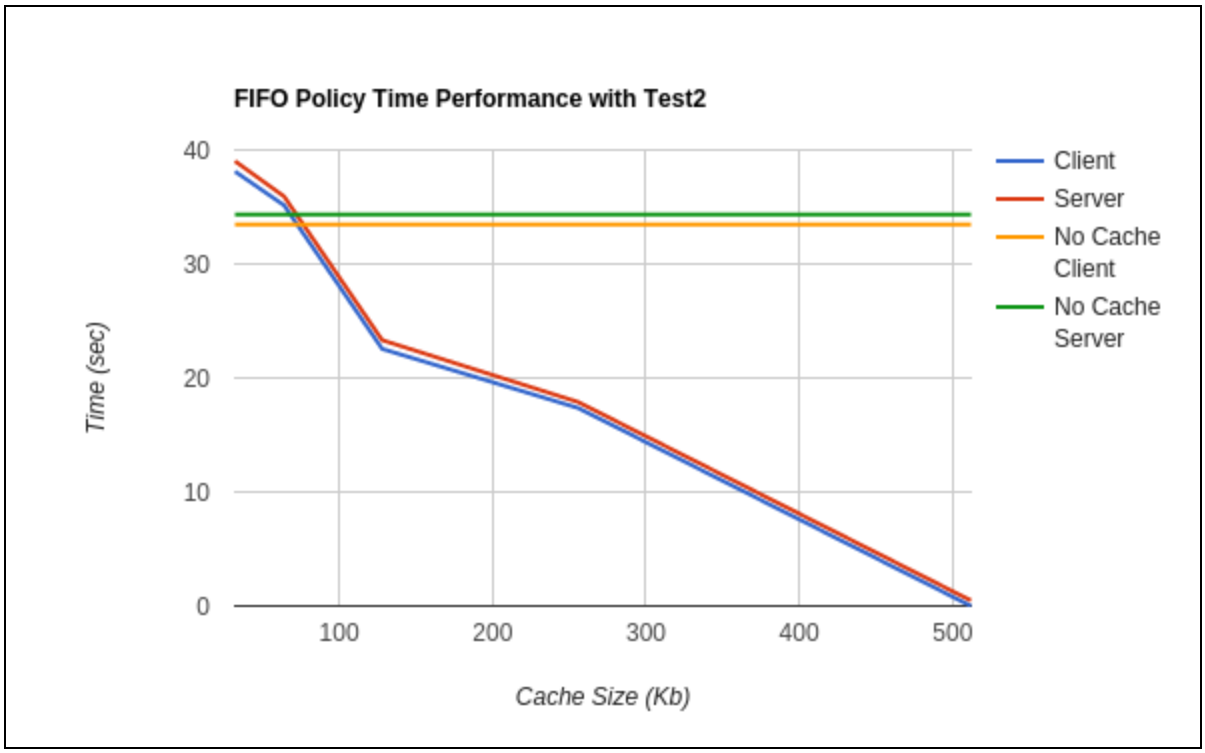
On Test2 we see that when cache size is small the execution time is larger for the server with cache than without it. It might be explained by the change in web page access time due to some effects in Wide Area Network or the change in signal quality of laptops wireless adapter.

FIFO Policy

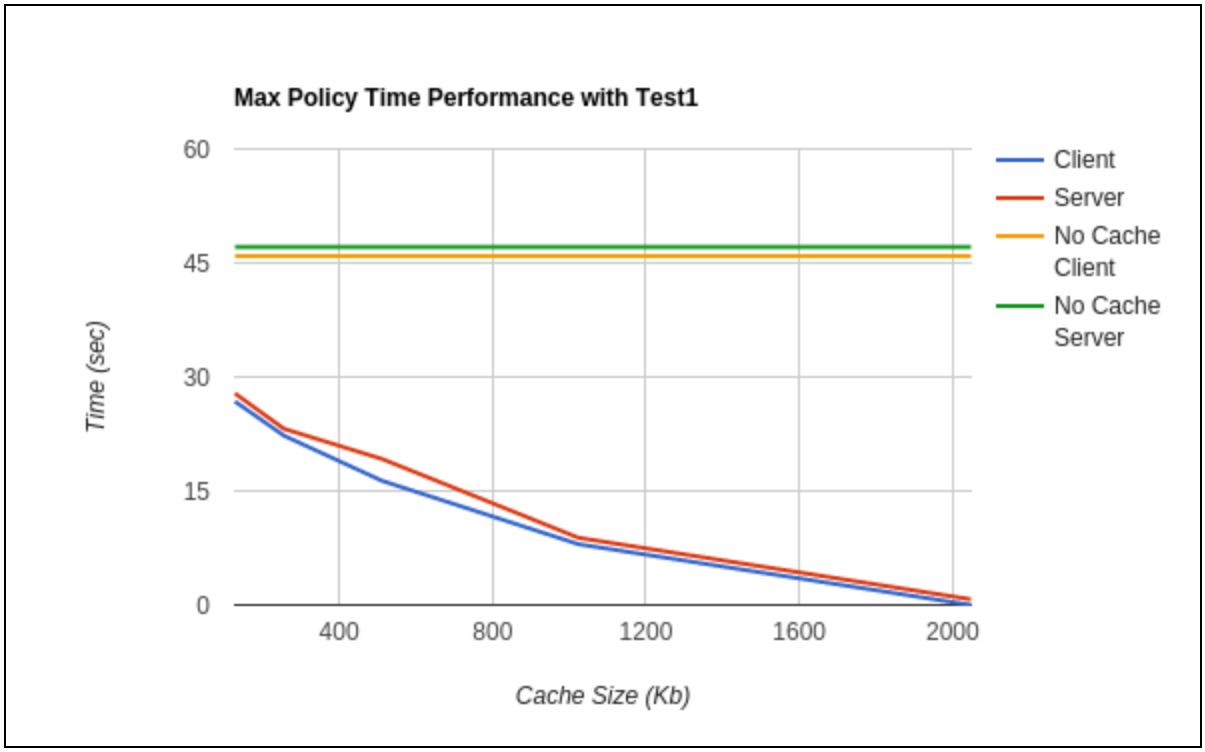
On both Test1 and Test2, FIFO policy shows a tendency to reduce execution time linearly to the increasing size of cache which means both of them are having better performance when they have larger cache size. However, FIFO policy longer preserves the performance of no cache policy. It happens because the cache is too small to include any of the repeated patterns from the tests. With the increase of the cache size the performance of FIFO replacement policy increases faster than for random replacement policy. It heavily depends on the test set and easily noticeable for repeated workloads. The more repeated parts there are in workload, the better would be FIFO performance.

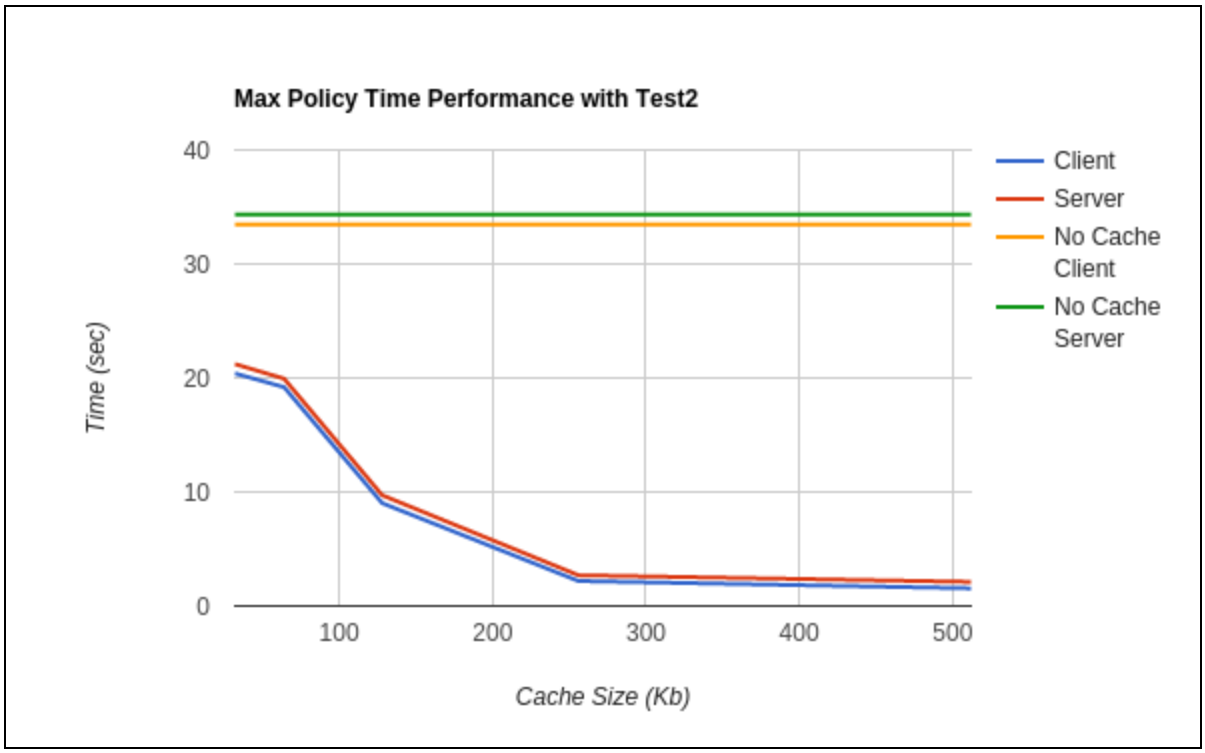
Similar with the result of the Random policy, we see that when cache size is small the execution time is larger for the server with cache than without it. We have the same explanation here.


FIFO Policy Time Performance with Test1
60
45
30
15
0
Time (sec)
400
800
1200
1600
2000
Cache Size (Kb)
Client
Server
No Cache Client
No Cache Server


FIFO Policy Time Performance with Test2
40
30
20
10
0
Time (sec)
100
200
300
400
500
Cache Size (Kb)
Client
Server
No Cache Client
No Cache Server

Max Policy



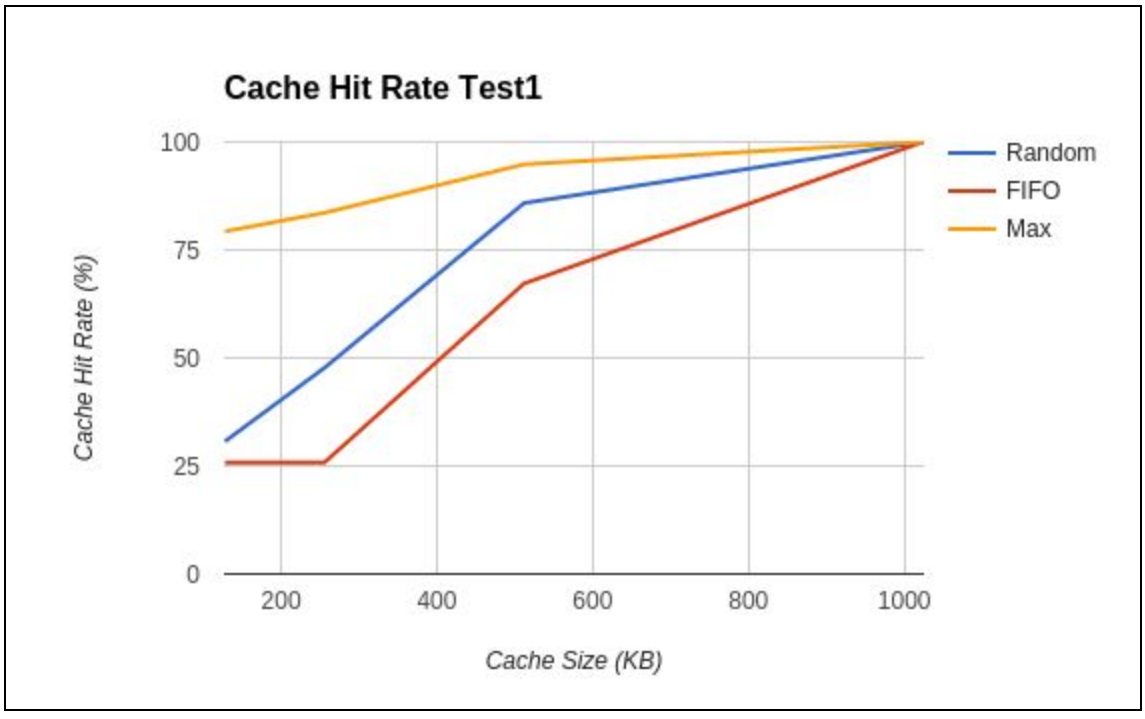


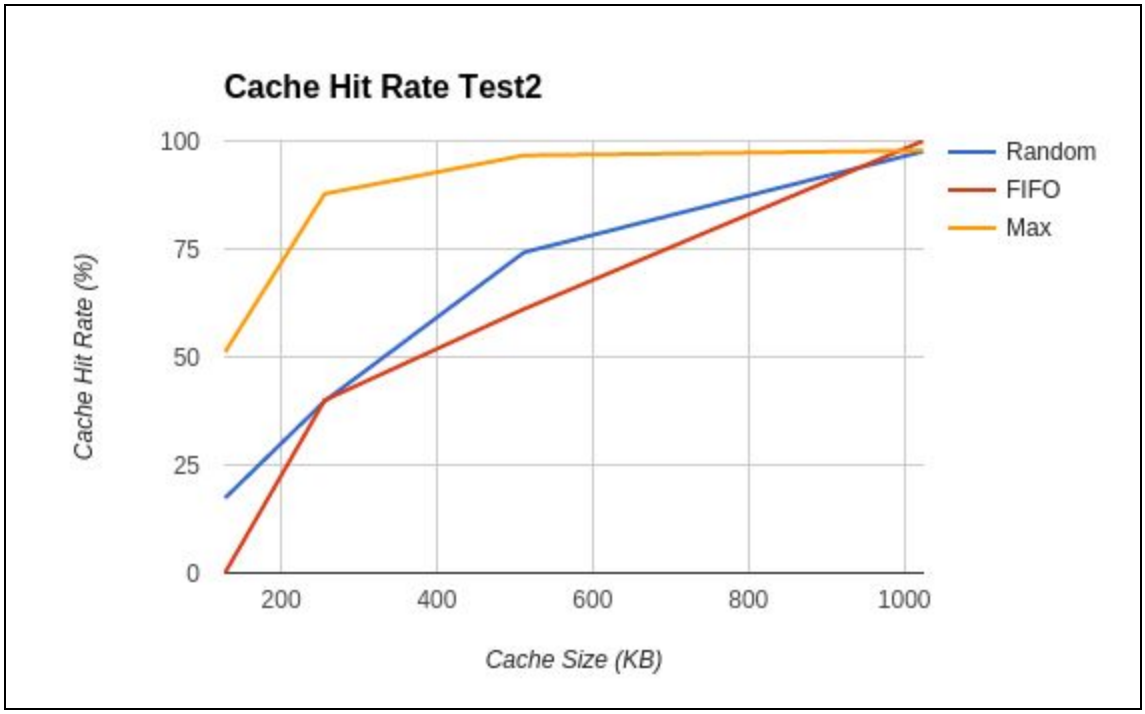
On Test 1 we see that the execution time reduces linearly as cache size increases. It happens because of the test set design. We have a lot of large page, but the overall size

of web pages decrease evenly from derived from list B to derived from list E. Hence, we always have a lack of space for larger pages, but pages don't form obvious clusters by size, and cache increase always allow to store more pages. We have good performance because we have a lot of smaller pages cached.

On Test 2, The execution time reduces linearly to a certain point, and the execution time tends to preserve even though the cache size increases. It can be explained by the fact Test2 has large number of small repeated pages, and a certain number of larger web pages, and during the experiment, they are replacing each other. We quickly achieve the volume of cache that enough to store small repeated subset of URLs. After that we need significant increase of cache to store all the pages. The cache hit rate proof this by having close to 100% hit rate from a certain point but not reaching to 100%.

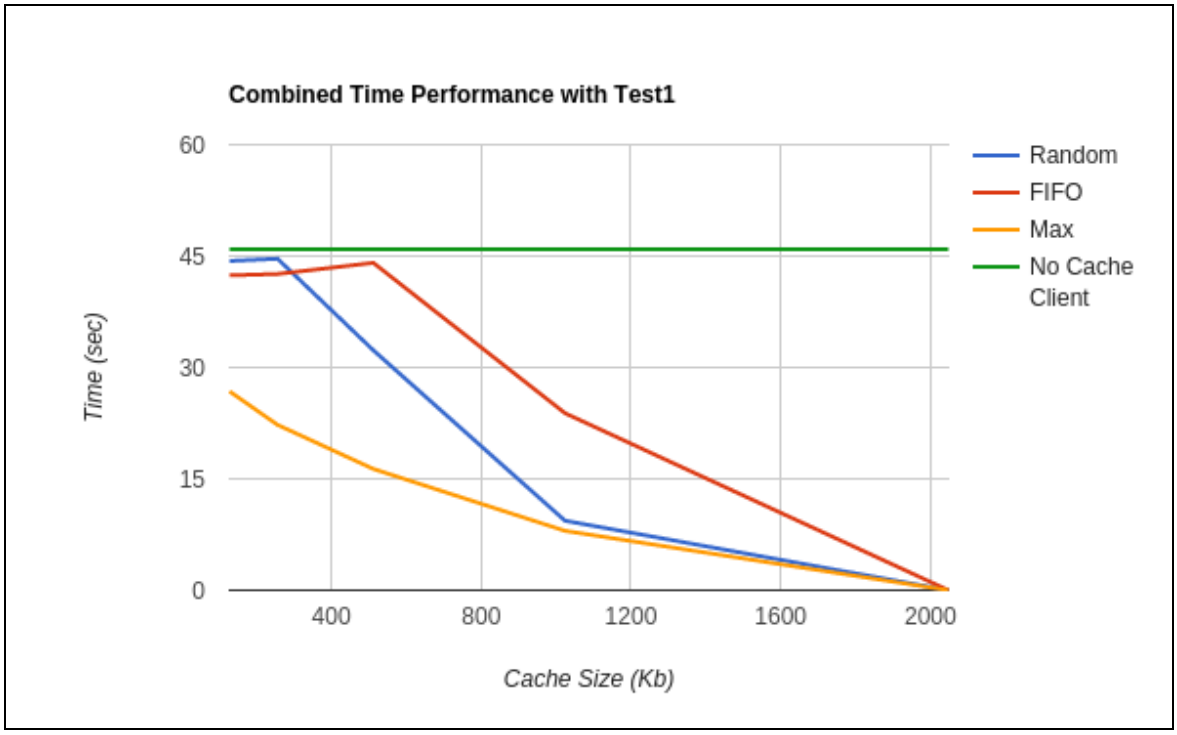
Cache Hit Rate

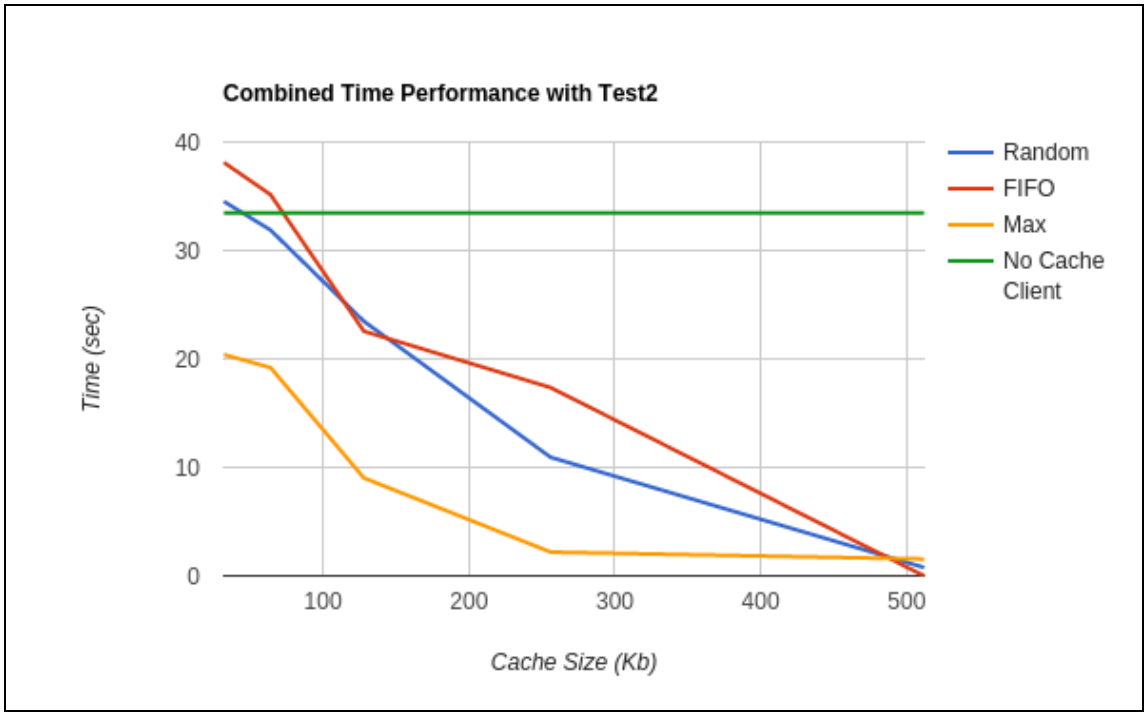




For both Test1 and Test2, Max showed the best cache hit rate among the cache policies. On Test2, Max policy shows dramatic improvements on the cache hit rate, and it is because Test2 is consisted with URLs hardly based on URL list D, so when cache has a enough size to store most of data from D, it only replaces a certain amount of data.

Compare Performance of three policies





Both on Test1 and Test2, Max policy shows the best performance among the cache policies. On Test1, FIFO cache tends to work well later than other polices, and it is because when cache size is small, cache only stores the small amount of repeated data.

Conclusion

As part of the project, we implemented RPC Proxy Server using Apache Thrift to generate C++ RPC server and libcurl library to handle HTTP traffic. We implemented cache for web pages and several replacement policies for it using C++ STL containers.

We made cache performance evaluation and evaluated several replacement policies. Our results demonstrate that using caching for HTTP traffic increases performance dramatically for repeated requests. Among several policies, there's none that is the best choice. Random policy can be efficient in case of small data cache and large number of requests, because for this case Large Size First and especially FIFO performs poorly. For workloads with small amount of large web pages and significant amount of smaller ones that are repeated, Large Size First makes better job because it preserves more webpages in the cache. FIFO performs well on the cyclically repeated requests.

However, all the policies need half of the workload size to achieve good performance, so the amount of cache available is probably more important than exact replacement policy, especially considering that all the policies increase the performance of certain type of workload.