CS 6210 Advanced Operating System

RPC-Based Proxy Server Project Report

1. Introduction

A proxy server is a server that acts as an intermediary for requests from clients seeking resources from other servers. A client connects to the proxy server, requesting some service, such as a file, connection or other resource available from a different server, and the proxy server evaluates the request and retrieves the resource.
There are different types of proxy servers depending on the supported content like web proxy, file proxy etc. In this project, we concentrate on web proxy server where the server serves web pages. A caching web proxy server accelerates web page requests by retrieving content saved from a previous request made by the same client or even other clients. Caching proxies keep local copies of frequently requested resources, allowing large organizations to significantly reduce their upstream bandwidth usage and costs, while significantly increasing performance. Most ISPs and large businesses have a caching proxy. Due to the limited cache space, replacement of the cache entries is inevitable when there are new resource requests. There are various well known replacement policies which pick the cache entry to be replaced based on a criteria.
In this project, we develop a caching web proxy server which exposes RPC interface to query the contents of web pages. We also implement three replacement policies and evaluate their performance on two datasets. We use Apache Thrift to implement the RPC interface and libcurl to fetch the web pages.
Apache Thrift is an interface definition language (IDL) which is used as a RPC framework to implement a variety of applications. 'libcurl' is a powerful library for communicating with servers via HTTP (and FTP, LDAP, HTTPS, etc.). It supports HTTP GET/PUT/POST, form fields, cookies, etc.

2. Division of Work

Chenhao: Implemented LRU cache
Srinivas: Implemented FIFO and Random caches
Joint-efforts: Learned how to use Apache Thrift framework, implemented the RPC-based proxy client and server using Apache Thrift, created test data sets and report write-up.

3. Cache Design & Policy Description

3.1 LRU Cache

Least Recently Used (LRU) is a family of caching algorithms, which discards the least recently used items first. This algorithm requires keeping track of when the item was used, which is expensive if one wants to make sure the algorithm always discards the least recently used item.

General implementations of this technique require keeping "age bits" for cache-lines and track the "Least Recently Used" cache-line based on age-bits.

3.1.1 Operations

The main operations that the LRU cache implementation supports are *string get(string key)* and *void put(string key, string data)*.
To be specific, the get method takes in the key (i.e. url) and returns the corresponding webpage body stored in the cache as a string. If the key does not exist in the cache, then an empty string is returned.
The put method simply stores the key-value pair of url and webpage body in the cache.
There are other public methods defined and implemented internally to facilitate the implementation of the LRU cache. For example, the getSize() method returns the current size of the cache in byte, which is used to help check whether the cache is full or not. However, auxiliary operations like this are not critical ones in the interface of the LRU cache.

3.1.2 Data Structures

The main data structures that the LRU cache uses include a customized node struct, and a combination of a doubly linked list and a STL map.
To be specific, the node struct is defined in a C++ template as follows:

```
template<class K, class T>
struct Node{
        K key;
        T data;
        Node *prev, *next;
};
```

Upon using this template in the LRU cache context, K and T are both instantiated as std::string, with key being the url and data being the webpage body. prev and next are pointers to the predecessor node and successor node.
The doubly linked list is used to keep the nodes in order. In this context, the order refers to the last time the nodes were accessed. The reason to use doubly linked list is to make sure that the complexity of locating the tail node is O(1).
The STL map simply maps the key to the corresponding linked list node containing the key. Using the map data structure reduces the complexity to look up the node from the linked list.

3.1.3 Algorithms

Once the data with key K is queried, the function get(K) is first called. If the data of key K is in the cache, the cache just returns the data and refresh the data in the linked list. To refresh data T in the list, we move the item of data T to the head of the list. Otherwise, if the data T of key K is not in the cache, we need to insert the pair into the list. If the cache is not full, we insert into the hash map, and add the item at the head of the list. If the cache is already full, we get the tail of the list and update it with , then move this item to the head of the list.

3.1.4 How evictions are chosen

When an existing node is accessed, this node is moved to the head of the doubly linked list, because we can reasonably assume that this node might need to be accessed again in the short future due to locality. With the same assumption, when adding a new node to the linked list, the new node is also added to the head of the linked list. In other words, the youngest node is always at the head of the linked list, and the oldest (i.e. least recently used) node is always at the tail of the linked list. By keeping a reference of the tail node, the eviction of the least recently used node can be done in O(1) complexity.

3.1.5 Pros of LRU

LRU saves the most recently used data, so it responds quickly to what's happened recently. In other words, LRU is more responsive to short term patterns such as those caused by user activity.

3.1.6 Cons of LRU

Given its design, LRU cache can be typically polluted by long scans.

3.2 FIFO policy

First-in-First-out (FIFO) is a well known policy used in a variety of contexts, mostly related to queuing and buffering. As a cache replacement policy, FIFO works by keeping track of the entry that was put in first into the cache. This is the entry that will be chosen in the face of eviction. Typically, FIFO algorithms demand the use of a linked list, as it preserves the order in which nodes are inserted and the ‘head’ would always point to the first entry in-order of insertion.

3.2.1 Implementation

As mentioned above, typically FIFO policies mandate the use of some ‘ordering’ to keep track of insertion. This can be achieved using dynamic linked list. An alternative solution is to use static arrays, in which case the node at index ‘0’ is the first node and new nodes are pushed at the end of the array. When the ‘head’ node is removed, the list has to left shifted. This overhead when deleting a node can be avoided using a linked list.

The FIFO web proxy cache in this project is implemented using a STL map structure with the key-value pairs being strings, where the ‘key’ is the URL to be fetched. It is to be noted that using the hash map provides a O(1) access to the cached URLs and thus the cache access overhead is minimal in the proxy server.

3.2.2 Data Structures

Internally, the elements in the STL map are sorted from lower to higher key value following a specific strict weak ordering criterion set on construction. However, this ordering does not necessarily reflect the ordering criterion required for FIFO, which is basically the order in which requests arrive at the server.

To facilitate this ordering, an array based list is maintained in which entries are placed into the array, the array index of that URL being the order in which it arrived. There are two such arrays. One array, *URLs,* holds the string names of the URLs and another, *URL_size*, holds the size (in bytes) of these URLs body. This helps keep track of the sizes of individual URLs in the cache, while at the same time preserving their order of arrival. There are other variables to keep track of the current cache length (number of URLs in the cache), the available capacity of the cache etc. There are few other variables to facilitate the collection of statistics like hit rate etc.

3.2.3 Algorithm

The following pseudocode captures the program flow for the FIFO policy based proxy cache.

1. At the server, when a URL is requested, check if it is available in the cache, by indexing the hash map with the URL
   1.1. If available, return the cached body of that URL.
2. If the cache does not contain the URL, fetch the URL using libcurl and compute the size of the URL body.
3. Check if the URL body is smaller than or equal to the overall cache size (not the available cache size)
   3.1. If not, the URL is too big to be cached. Simply return the URL body to the client.
4. Check if the URL body is smaller than or equal to the available cache size.
   4.1. If so, insert the body into the cache using the <URL, body> as the <key, value> pair.
   4.2. Append the URL to the *URLs* array and the size of the URL body to *URL_size* array.
5. If not, recursively erase the 'first' URL inserted into the cache until there is enough space to accommodate the new URL. This is done by using the indexing the hash map with the head of the *URLs* array..
   5.1. The available cache size is then incremented by the head of the *URL_size* array, since it corresponds to the URL that is erased.
   5.2. Both the *URLs* and the *URL_size* arrays are left shifted to update their heads and the URL body is returned to the client.

3.3 Random replacement policy

As the name suggests, the random replacement arbitrarily chooses the entry to replace until there is enough space in the cache to accommodate the requested URL. Implementation of the Random replacement policy is very similar to the FIFO policy. Most of the data structures and variables are reused among the FIFO and random policies.

3.4 FIFO vs Random vs LRU replacement policy

The primary difference between FIFO, LRU and random replacement policy is that for random placement, instead of choosing the 'first' or "least recently used" cache entry for eviction, a random entry is chosen every time. Using a standard random number generator like *rand(),* an index is generated in the range (0, cache length). The URL corresponding to that index in the *URLs* array is then erased from the cache, the size of that URL as determined from the *URL_size* array is added to the available cache size. The arrays are then left shifted starting from the chosen index till the end.

Other than this step, the random replacement policy follows the algorithm described above. The primary advantage of the random replacement policy is that it avoids most of the bookkeeping overhead as compared to FIFO and LRU, especially if the cache is implemented in hardware. Although, the simplicity of random replacement is blurred when software caches are involved as the bookkeeping data structures are almost the same.
Also, it is to be noted that the random index chosen is' truly' random i.e it could potentially vary across different runs of the program . This introduces some variability in the tests but the random replacement policy as such is built on top of this variability.

4. Metrics for Evaluation

We measure performance of the replacement policies in two dimensions: by total time taken to access the complete dataset and cache hit rate. We varied the cache size to measure the effect of the cache on various performance metrics. We also disabled the cache completely and measured the total access time for all URLs in the workloads.

4.1 Total Access Time on Server

For this metric we measure the total time the server takes to finish fetching the complete list of URLs in each dataset. The time is measured as the time difference between the client initiating the RPC call to server, and the time at which the server finished processing all the URLs in the dataset. We used the *gettimeofday* function in <sys/time.h> to facilitate the measurement.
Total time taken by the proxy server is a good metric for evaluating cache performance because generally latency is a major factor in user experience. We chose to measure and present the time taken to complete the entire workload instead of measuring time taken to fetch individual URLs because of the variabilities described in section 6. Metrics such as avg. access time for each URL in the workload would not be accurate as some URLs may hit and some may miss in the proxy cache. Also, the dynamics of the internet affect the response time to fetch a web page. On the other hand, total access time on the server minimized these variabilities and we can study the extent to which the replacement policy chosen helps speed up the overall processing of workloads.

4.2 Cache Hit Rate

For this metric we measure, on the server side, the percentage of URLs that “hit” in the cache when processing the complete list of URLs in each of the workloads.
This metrics gives us more insight about how the cache performs with respect to the specific datasets provided. The higher the cache hit rate is, the more effective the cache is in terms of helping speed up the URL response.

5. Workloads Description

A workload is a set of URLs, requested at the client in a particular order, used to test the performance and functionality of the web proxy. We have used the following workloads in our test harness. In both the workloads we use a set of 12 popular URLs and fix the total number of URL requests as 48 (this empirical choice does not affect any aspect of the semantics of the program).

5.1 Workload 1 -- Random Selection

In the random selection workload, a URL is randomly chosen from the list of 12 URLs. The temporal reuse of the same URL is a function of the random distribution. We have used a uniform random distribution and so all URLs are likely to be issued at almost the same rate. An important point to note is that variability in this random selection makes it very difficult to compare the performance of different caching policies with this workload. Ideally to compare different policies using this workload, we'd like the workload to be deterministic to the extent that across different runs, we request the URLs in the same order while at the same time having enough randomness in choosing which URLs to request.

Contrasting this with the random replacement policy itself, where the entire algorithm is built around true randomness in choosing the entry to be replaced, in this case we only require an initial random order of URLs and then we preserve that order by using the same seed for the random number generator. That initial degree of randomness is both necessary & sufficient to test the performance of different caching policies and compare them against one another.

5.2 Workload 2 -- Grouped Selection

In the grouped selection workload, we divide the list URLs into groups of 'n' URLs. Each group is repeated 12/n times, before the next group is issued, thereby keeping the total number of requests the same as workload 1. The reason behind such a workload is that conceptually, it is orthogonal to the previous workload i.e, there is a lot of determinism in the ordering of URLs while Workload 1 is mostly random.

This determinism, we hope, would play an important role in distinguishing the performance of various caching policies, especially random replacement which doesn't exploit the determinism in the order of URLs arrival.

6. Experiment Description

The test harness used to measure and compare the performance of FIFO, random & LRU caching policies is as follows. We vary the cache size from 128 KB to 3MB, with each of the caching policies and measure the two metrics explained above viz. execution time and cache hit rate for both the workloads. We've tested the following scenarios;

A. Server and Client on the same machine
B. Server and Client on different machines

The results presented below are collected from tests with the server and client on the same machine. We expect a similar behavior when the server and client are on different machines.

6.1 Variabilities in the experiment

Before presenting the results, a word of caution on the variabilities involved in the measurement. As mentioned in section 3, the 'random' cache replacement policy makes it difficult to get persistent results even without any other variability in the harness. The results discussed below are average values to account for this randomness.

The dynamics of the internet also play a role in the measurements, as due to factors like server load balancing or fluctuations in the ISP's network, the time taken to actually fetch a page across the internet may not necessarily remain persistent. Also, ideally applications involving caching operations tend to employ an expiration timer for the cached entries. Since the use of timers is beyond the scope of this project, we always return the cached copy of a URL if it is present in the cache and the validity of such URLs is not a concern for this project.

7. Experimental Results & Analysis

Figures 7.1 and 7.2 show the variation in Hit rate of the proxy cache as a function of cache size, for FIFO, random and LRU policies. From the graphs, clearly cache size has a major impact on the hit rate, the bigger the cache the better. However, as we see towards the end, the performance converges to a hit rate of 75%. This is a typical pattern observed in caching systems, which is due to the phenomenon of ‘compulsory misses’ i.e, the 25% misses are compulsory. They are the misses faced when the URL is requested for the first time. In other words, this 25% penalty is the time taken to ‘warm up’ the cache and it HAS to be paid as initially the cache is cold/empty. Increasing the cache size beyond a certain saturation point does not contribute to speeding up the system.

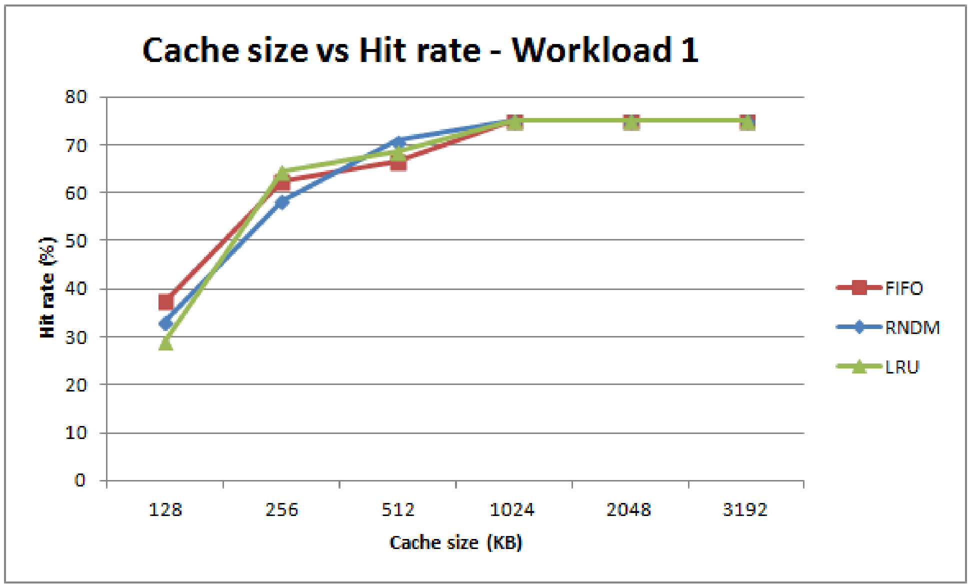

Figure 7.1 Cache size vs Hit rate for Workload 1

The difference in performance across different workloads is actually clear when we compare Fig. 7.1 and 7.2. It can be seen that workload 2 actually achieves hit rate saturation faster (at a smaller cache size) than workload 1. To reiterate the difference between the workloads, in workload 1 URLs are chosen at random from a predefined list whereas in workload 2, groups of 4 URLs are repeated before other URLs are tested. These repetitions in workload 2 face minimal cache pollution by other URLs (which might not be the case in workload 1 as the entropy of random URLs typically pollutes the cache)

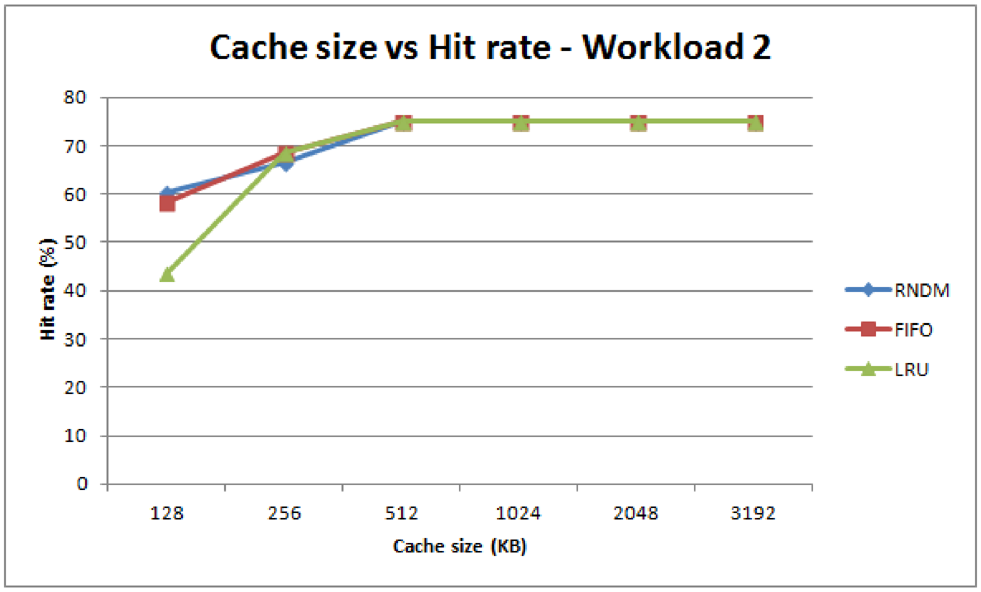

Figure 7.2 Cache size vs Hit rate for Workload 2

Figures 7.3 and 7.4 show the variation in the execution time of the entire workload (the total access time on the server) as a function of cache size. Due to the variabilities mentioned earlier, it is difficult to get a smooth 'ideal' curve in this experiment. Nevertheless, The decreasing trend of execution times with bigger caches due to increased hit rates is the key take away from these graphs. Also, as we can see, there is a significant speedup in both workloads as compared to time taken with 'no caching'.

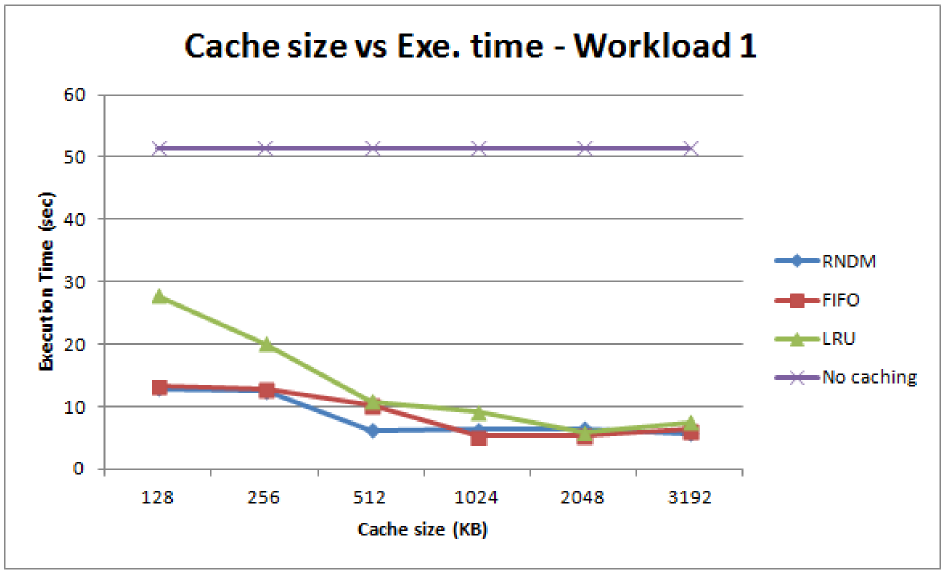

Figure 7.3 Cache size vs execution time for Workload 1

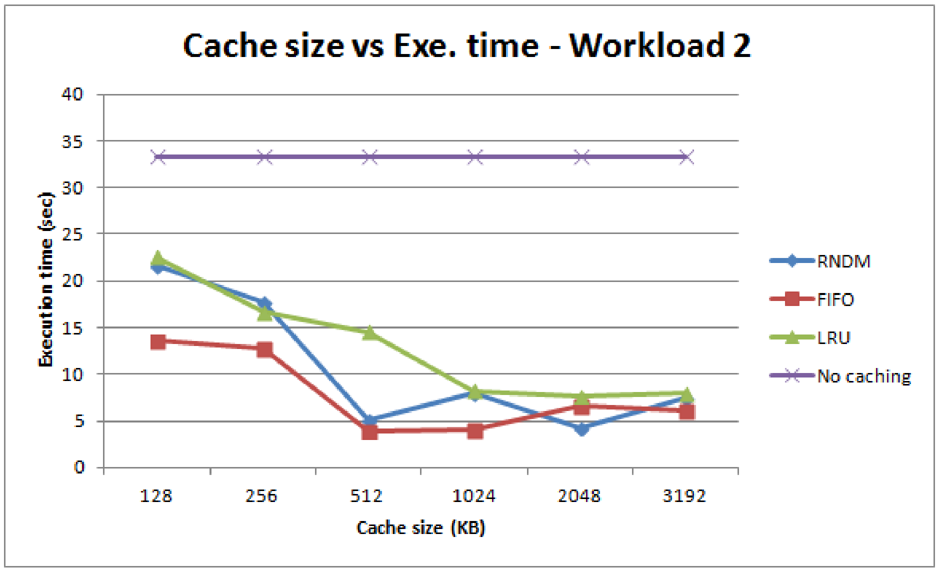

Figure 7.4 Cache size vs execution time for Workload 2

Another observation to be noted from the above graphs is that for some cache sizes, the total access time on the server is pretty low (i.e, the server is fast in completing the entire workload) although the cache hit rates corresponding to those cache sizes from the previous graphs are low. This counter-intuitive behavior is, what we believe, because of the 'asymmetry' in the load time for webpages. For example, the time taken to load a simple html webpage such as google.com is certainly less than the time taken to load a 'content-heavy' websites such as newegg.com. We can only deduce the fraction of websites that missed in the cache from the 'hit rate' metric but it does not tell us anything about the nature of the website. For the case considered specially here, it could very well be possible that the workload faced several misses in 'lighter' websites (thereby lowering the hit rate) while at the same time, since they can be fetched faster, the total access time on the server for the workload is low.

8. Conclusion

Given the importance of user-perceived end-to-end latency in serving a HTTP request, web proxies are critical internet infrastructure that account for a better browsing experience. Also, Automatic code generation tools are becoming increasingly popular in the software fraternity and we've explored one such tool, Apache thrift, for generating the skeleton code for the RPC based proxy server.
In this project, we have implemented a RPC based proxy web server using various cache management policies such as FIFO, LRU and random replacement. We've analyzed the performance of these caching policies keeping in mind the dynamics of the internet architecture and the inherent variabilities in the system. Each of these policies have their own pros and cons and depending on the requirements, resource availability of the application, and the one policy would make a better choice than the other.