Project 3: Evaluation of Caching Policies in RPC Systems

Aman Mangal, Flavio Castro

March 28, 2015

Contents

1 Introduction

Caches have been extensively used to improve the response time for applications these days, may it be a general purpose computer or a web proxy server. In this project, we have implemented a proxy server using RPC with the help of *thrift* as well as implemented various caches each having different replacement policy. We have also designed workloads to compare the Random, FIFO and LRU cache replacement policies and evaluated them. We discuss in details the implementation and the evaluation results in this report.

2 Cache Design

We have defined an abstract class (*Cache* class) in C++ which defines the interface for all the cache we have designed. All the cache classes inherits from the *Cache* abstract class and implements the methods defined in it. We describe the functionality of the methods defined as follows-

Method	**Functionality**
type	returns a string specifying the replacement policy
contains	returns true if a document for a url is present in the cache
get	retrieves the document present in the cache, this function should only be called if the document is present in the cache
put	puts a document in the cache. If cache is full, it replaces an existing document with the current document. Should only be called if document is already not present in the cache

Table 1: Interface Defined in *Cache* Class

We need to maintain a key value pair from url to the document present at the url to implement the *get* function. We have used the *map* STL library of C++ which enables a $O(\log(n))$ access to the document for a given url in worst case. We also need to maintain a list of all the url for which the document is present in the cache. This list will help us in implementing the replacement policy for a specific cache. The implementation should be such that a replacement candidate can be chosen in constant time. Note that we keep removing documents from the cache unless there is enough space to put the new document. Assuming that we remove constant number of documents in each call to *put* function, the time complexity of all the methods, therefore, will be as follows-

[1] we may want to modify the list of urls such as in LRU cache replacement policy

Method	Asymptotic Complexity
contains	$O(\log(n))$
get	$O(\log(n)) + O(1)$[1]
put	$O(\log(n)) + O(1)$

Table 2: Asymptotic Time Complexity of Methods of *Cache* Class

We have decided to implement 3 cache replacement policies in this project. We discuss their implementation in the following section.

3 Caching Policies

3.1 Random Replacement Policy

In this cache replacement policy, we implement the list of urls using *vector* STL library. we choose the candidate for replacement by choosing a random index of the vector in the range ($0 \leq index \leq vector.length()$). While removing the document from the cache, we delete the key value mapping as well as the corresponding url entry from the vector. We fill this empty space in the vector using the last element so that the cost of shifting all the elements in the vector can be avoided.

Random replacement policy doesn't require us to maintain any access history and hence, it is easiest to implement. Such algorithms may perform well when web pages accessed by the client doesn't have any co-relation among them. Whereas, they will not perform well when same page is accessed frequently as random replacement doesn't take into consideration any access history.

3.2 FIFO Replacement Policy

In this replacement policy, we pick the replacement candidate as the document which was present for the longest time in the cache. To find such a document in constant time, we use the queue STL of C++. We insert the url at the end of the queue, whenever we put the corresponding document in the cache. The oldest document will always, therefore, be at the head of the queue.

The advantage of this algorithm is that it is simple and fast. It can be implemented using just a queue data structure. The disadvantage is this is a naive algorithm. it doesn't make any efforts to keep more commonly used pages in cache.

3.3 Least Recently Used Cache Replacement Policy

We use a doubly linked list to implement LRU cache replacement policy. We maintain the invariant in the linked list that the head is always the most recently used document whereas tail points to the least recently used document.

To implement this algorithm, we keep the pointer to the node present in the linked list, in the map data structure. The linked list node contains the document as well as other meta data. Whenever a page is accessed (a call to *get*), we first find the pointer to the node from the map. We, then, remove the node from the middle (if at all) and put it at the head of the linked list.

In case of a call to *put*, we assume that the document doesn't exist already. We create a new node and store a url to node mapping. We add this node at the head of the linked list as it was accessed most recently. In case, there is not enough space in the cache, we remove document from the tail of the linked list. We also invalidate the mapping from url to document while deleting it.

Figure 1: Linked List for LRU

LRU algorithm tries to adapt to the data access pattern unlike FIFO replacement algorithm. The data accessed more frequently is less likely to be removed from the cache. The main disadvantage of LRU replacement algorithm is that it can still get filled up with items that are unlikely to be accessed again soon. In particular, it can become useless in sequential scans of larger number of pages all of which do not fit in the cache. Nonetheless, this is by far the most frequently used caching algorithm.

4 Experimental Evaluation

4.1 Metrics for Evaluation

In our proxy server, we evaluate the cache schemes using two metrics: Hit Rate and Response Time.

In theory, the hit rate is a good performance metric for the cache. A higher rate should incur in a better overall performance, but there are exceptions. For example, the complexity of the cache comes into place too: if a cache has a higher hit rate but can't implement a scalable lookup procedure,

the additional latency may impact in the response time, which is the actual performance experienced by the users.

The response time is a good metric to show the effectiveness of the cache. A rough estimation of the response time is given by the following formula : (r)*ProxyRTT + 1-r*(InternetRTT). A proxy may or may not be effective in reducing the response time given its performance parameters. Additionally, the proxy increases the original delay of Internet requests, all this should be taken in account.

4.2 Workloads

The workload design for this experiment is a huge challenge. On one side some algorithms might perform better depending on the workloads chosen. On the other side real workloads are hard to simulate. Ideally we would obtain a big trace file from real HTTP requests and *rerun* it over our proxy server. Unfortunately, that was not feasible for this experiment given time constraints.

Nevertheless some probability models can still represent real traffic with enough accuracy. We used a uniform distribution and a exponential distribution.

4.2.1 Uniform distribution

We, repeatedly, choose pages randomly from a small set of 10 urls.

4.2.2 Exponential distribution

Here, we choose pages from a big set of 100 urls, but using an exponential probability distribution with average 10. In that way a small set of pages recur several times, and a big set of pages are still occasionally visited. We expect this model of thick-tail to be closer to the real world.

4.3 Experiment Description

We ran two separate experiments one to evaluate the response time and another to calculate the cache hit rate.

Our setup includes two different virtual machines, one running the RPC client and one running the RPC server. The network connection between the virtual machine is provided by the VM manager, and it does not add a lot of overhead to the communication, making the effects of caching even more visible.

To calculate the response time for any given combination of: (1) cache size, (2) workload type, and (3) cache type, we executed the experiment 1000 times to guarantee statistical reliability of the measurement. The experiment consists of: starting from an empty cache, execute GET requests to the proxy via RPC, while measuring the time difference between beginning and end of each request from the client point of view.

When running the experiment described above we noticed that it only represents a single observation of cache hit ratio, thus we faced the challenge of running this long experiment (20 minutes) multiple times to guarantee statistical reliability of the measurements. We decided to skip the HTTP get requests performed by the CURL library in order to reduce the overall length of each experiment.

A new challenge arose from that, which was how to take the page size in account when caching. We decided to fetch all pages in the list, storing it page size in a file; in the next experiments we fetched the page sizes from this file and then ran the cache hit experiments multiple times. To make the measurement reliable we ran 5 experiments of 1000 repetitions for each combination of cache size, workload type, and cache type.

We ran the experiment for the following cache sizes: [0, 64, 128, 256, 512, 1024, 4096].

4.4 Analysis of results

Figure 2: Random Workload - Average Hit Rate(%) vs Cache Size(KB)

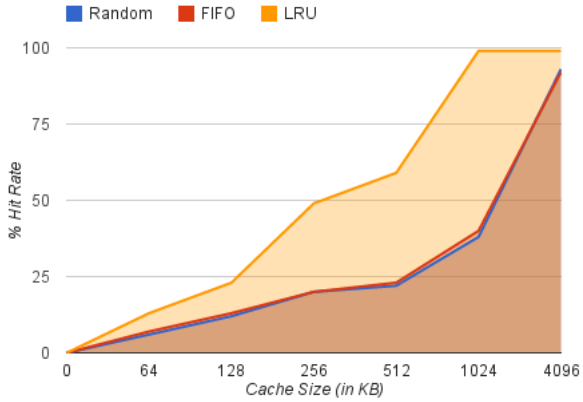


Fig. 2 illustrates a curve for each cache type representing its cache hit rate. Notice that LRU outperforms FIFO and RANDOM cache replacement policies.

For a large cache size (higher than 1024) the hit rate does not represent too much, because the total size of pages is small. Notice that FIFO and Random policies still present a reasonable performance reaching about 25% hit rate on a chache size of 512.

Figure 3: Exponential Workload - Average Hit Rate(%) vs Cache Size(KB)

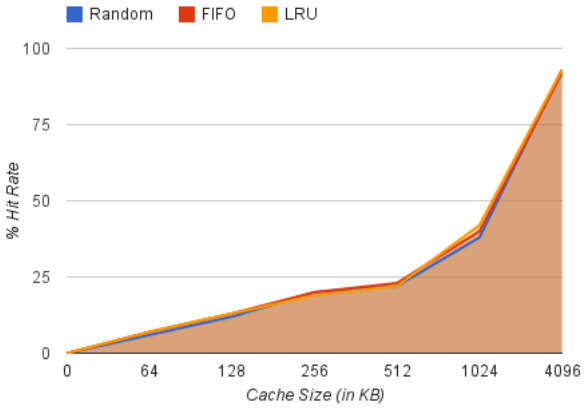


Fig. 3 illustrates a curve for each cache type representing its cache hit rate. Notice that all caches present a reasonable increasing performance for small cache sizes. LRU slightly supper FIFO and RANDOM cache replacement policies for 1024 cache size.

Figure 4: Random Workload - Average Response Time(ms) vs Cache Size(KB)

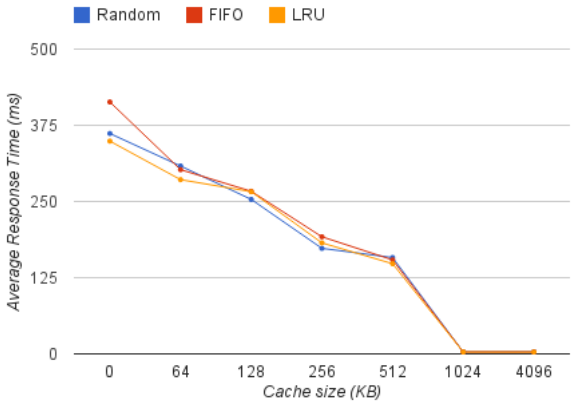
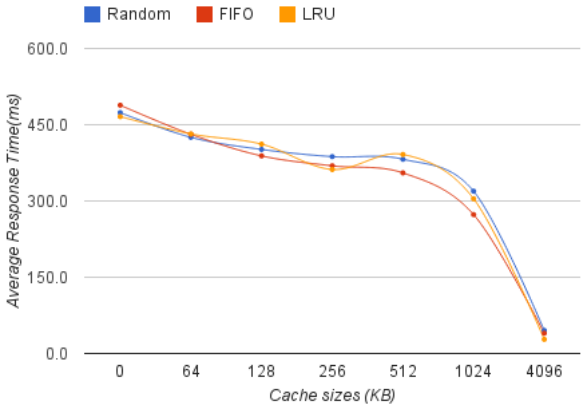


Fig. 4 illustrates a curve for each cache type representing its response time. Here we can clearly correlate the cache hit rate with response time. Notice that response time goes down as cache hit rate goes up. Also notice

Figure 5: Exponential Workload - Average Response Time(ms) vs Cache Size(KB)



that the overall response time decreases significantly showing the effectiveness of the cache.

Fig. 5 illustrates a curve for each cache type representing its response time. In this case we can see that Fifo slightly outperforms its peers.Again we can notice that the overall response time decreases significantly showing the effectiveness of the cache.

5 Conclusion

We have discussed the implementation of Random, FIFO and LRU cache replacement policy. We saw that they all have $\log(n)$ overhead for all the operations. We concludes that the implementation of LRU cache is comparatively complex than the Random and FIFO cache. We observed the response time and hit rate for both the caches for uniformly and exponentially distributed urls.

We could see how the presence of a web proxy with relevant cache replacement policies can improve the overall latency of the web, by executing a load model close to the real world we could see the effectiveness of a proxy system. Our system decreased the average latency from roughly 460ms to 300ms under an exponential load using a cache size of 1024KB.