RPC Based Proxy Server

Anish Khale
`anish_khale@gatech.edu`

Nikita Gupta
`ngupta71@gatech.edu`

Abstract

Remote Procedure Calls (RPC) are powerful, commonly-used abstractions for constructing distributed applications. One such modern RPC technology is Apache Thrift. Thrift allows for scalable, cross-language services development by combining a software stack with a code generation engine to build services that work efficiently and seamlessly between C++, Java and a host of other languages. Caching in a distributed system environment further helps to improve efficiency of accessing the required data. In this project, we combine the powerful abstraction of RPC from Apache Thrift with the efficiency and high performance of various caching schemes in a distributed application.

1 Introduction

To build a RPC-based proxy server, a modular approach was taken beginning with generating a C++ server skeleton code using Apache Thrift. Next, the ability to make curl requests, using the `libcurl` library, was added to this server to fetch the HTML content from a particular website pointed to by its URL, passed as a parameter by a requesting client over RPC. Having the RPC portion of the project taken care of, we built in the code libraries for caching policies that included Random Eviction, First In First Out (FIFO) and Least Recently Used (LRU) as the cache replacement policies. Sections 3 and 4 explain these cache designs and policies, respectively, in detail.

To measure the performance of the three caching policies, two performance metrics were selected, viz. hit ratio and average memory access time, and are described in detail in section 5. Experiments were run under two different workload patterns, as described in section 6, and the experimental setup and methodology is described in detail under section 7. Results obtained from these experiments indicate that the decision of choosing the right cache replacement policies has a significant performance impact on cache performance. The results, documented in section 8 and analysed in section 9, prove that under varying workloads, LRU performs the best while the performance obtained from random eviction of cache entries is the least.

2 Task Division

The tasks associated with this project can be broadly classified into three main categories: Code, Test and Documentation. The breakdown of tasks along with the team members they were assigned to respectively are shown in table 1.

3 Cache Design

Generally application developers want to reduce expensive operations and, therefore, wish to reuse some results to improve application performance. Hence caching comes into play. In caching, pages referred by the user are stored in a cache, so when the same page is requested it can be restored from the cache instead of going to the main memory and fetching that page. This saves time and improves performance of the application. In this project cache memory is used to store copies of web pages to exploit localities of reference. When client requests a web page, server first checks the presence of web page in the cache, if present it will fetch the web page and send it to the client. If not present, the server gets the web page from the internet, updates the cache entry and sends it to the

Category	Task	Anish K.	Nikita G.
Code	RPC using Apache Thrift	100%	0%
	Fetching HTML from URL using `libcurl`	100%	0%
	Random Eviction policy	0%	100%
	FIFO policy	0%	100%
	LRU policy	0%	100%
	Test Harness	50%	50%
	Integration	100%	0%
Test	Integration	50%	50%
	Performance	50%	50%
Documentation	Writeup	50%	50%

Table 1: Task Division for Project 3

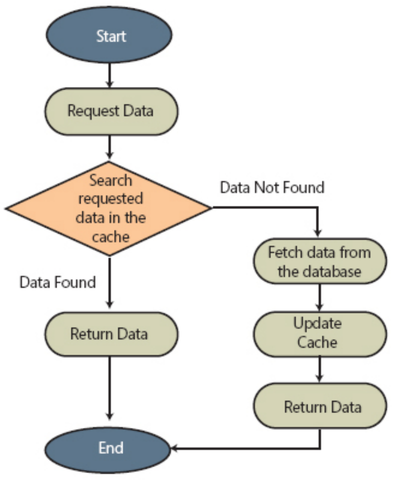

Figure 1: Working of a cache based system

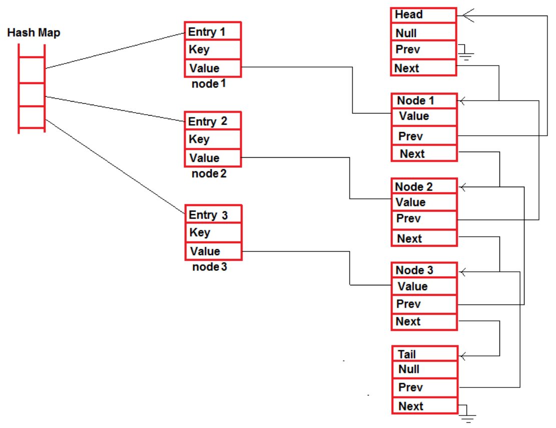

Figure 2: Cache implementation

client. So, in this project, a web cache is implemented at the server which stores URLs which have been accessed by the client. Implementing this caching policy reduces time that the server takes to process clients request if it is a cache hit, if it is a cache miss normal time is taken to get the web page from the internet. This will in turn improve application performance. The URL and data are stored in a Key-value container, key being the URL and value being the HTML present in that URL. Cache implementation, in this project, is done using hashmap and doubly linked list, as shown in fig 2. Hashmap stores Key-Value pairs and doubly linked list is used to index the pairs in order of data age. There are two functions which are implemented one is `search_cache()` and the other is `insert_into_cache()`. These two functions are self descriptive by their names. When client request a URL server calls `search_cache()` to find the URL in the cache, if it is present, it is a hit else a miss. In case of hit the URL content is returned to the client, in case of miss server searches for that URL on the internet and updates the cache entry and returns the content to client. For updating cache entry `insert_into_cache()` function is called. First this function checks whether the size of cache is big enough to add new entry or not. If memory is available to add new entry life is good, else using a cache replacement policy one of the entry is removed and in place of that new entry is added. Hashmap is used for storing key-value pairs and searching in hashmap is O(1) which is very efficient, hence making the caching algorithm efficient.

4 Caching Policies

One of the important factors in selecting an effective caching policy is the lifetime of the resource. In order to store a new resource in cache, an existing cache resource needs to be swapped out if the cache becomes full; this is known as paging. There are different paging algorithms, viz., LRU(Least Recently Used), LFU(Least Frequently Used), FIFO(First In First Out), MRU(Most Recently Used). Caching policy is selected with the goal to minimize paging and maximizing cache hits. In this project, we have implemented Random, FIFO and LRU cache replacement policies.

4.1 Random

In Random paging an entry is selected at random to be evicted. When client requests a web page and entry is not present in the cache and cache size is full, then randomly a entry is selected and that entry is evicted. In case of cache hit, the entry is returned to the client and no change in the cache is done. While in the case of cache miss, randomly selected entry is deleted from the doubly linked list.

4.1.1 Pros

- Searching complexity is O(1) because to search the entry hashmap is used.
- Deletion complexity is O(n), where n is the random number which is generated using `rand()`. While loop is implemented which iterates till n and then that entry is evicted, hence complexity is O(n).

4.1.2 Cons

- In this policy a pseudo random number generator is required, In this project we are using a standard C++ `rand()` function.
- This policy makes no attempt to benefit from temporal or spatial localities.
- Cache is not thread safe, i.e cache is not safe for concurrent modifications, when multiple clients call the same server.

4.2 First In First Out (FIFO)

In this paging policy, the page which has been in the cache the longest is evicted, i.e the page which was stored in the cache first gets swapped out first. So to implement this policy we have maintained a queue using doubly linked list. New cache entries are added at the end of the queue and entries to be deleted are deleted from the staring i.e head of the queue. Whenever an entry is present in the queue and is requested by the client, then server returns the entry as a response to the client request and no change is made in the queue.

4.2.1 Pros

- Search complexity of this algorithm is same as that of random algorithm, O(1).
- Deletion complexity is O(1), because entry which is at the head of the doubly linked list is evicted.
- This algorithm tries to take benefit from temporal locality of cache.

4.2.2 Cons

- It requires more space, as queue is used to maintain First In First Out policy.
- Cache is not thread safe, i.e cache is not safe for concurrent modifications, i.e when multiple clients call the same server.

4.3 Least Recently Used (LRU)

When cache capacity is full and new content needs to come in, the previous content has to be swiped out to make space for new content. In LRU known as Least Recently Used, the content which is not used since a long duration of time i.e the content which has been used the least will be replaced. Thereby making space for new content. In this project we have implemented LRU as the third caching policy. In the doubly linked list new cache entry is added at the front of the list just after the head, so this makes the end or tail of the list as the least recently used entry. So when an entry has to be deleted the last entry is deleted and the new entry is added at the head of the list. Now if the entry is already present in the cache and is requested by the client, then this entry is deleted from that position and the entry is again added to the front of the list, making it the most recently used entry. In this way LRU policy is implemented.

4.3.1 Pros

- Searching complexity of this algorithm is O(1), which is also the complexity of FIFO and Random.
- Deletion complexity is O(1), because every time the entry at the end of the doubly linked list is evicted.

4.3.2 Cons

Cache is not thread safe, i.e cache is not safe for concurrent modifications, i.e when multiple clients call the same server.

5 Performance Evaluation Metrics

The chosen metrics for evaluating the performance of the three caching policies detailed in the previous section are indicative of the time saved by not having to access the memory and the correctness of the policies in having those pages in the cache that fulfil a very high number of URL requests coming in.

5.1 Hit ratio

Hit ratio (h) is a basic performance metric that measures the ratio of the number of memory references that hit in the cache and the total number of memory references. Mathematically,

$$h = \left(\frac{hits}{hits + misses}\right) \tag{1}$$

This metric was chosen because it can accurately indicate the number of times a request, that is made, can be found in the cache (a hit) versus the number of times it has to be served by fetching the data from the physical memory. Ideally, the highest value possible for h is 1. Practically, higher the value of h, the better the concerned caching policy is.
An equivalent performance metric is Miss ratio (m), where,

$$m = (1 - h) \tag{2}$$

5.2 Average memory access time

The average memory access time (t_{avg}) accurately conveys the amount of time the server spends in the cache to serve an incoming request versus that spent to access the same from the main memory. It is computed as

$$t_{avg} = (h * t_{cache} + m * t_{memory}) \tag{3}$$

where, h and m are calculated as per equations (1) and (2) respectively, and t_{cache} and t_{memory} are the times required to access the cache and main memory respectively. Lower the value of t_{avg}, better is the particular caching policy.

6 Performance Evaluation Workloads

To obtain the most accurate results from the performance evaluation of the caching policies, we choose the following two patterns of workload distribution, assuming four URLs: A, B, C and D. In our experiments, we include these patterns directly in their individual text files such that the URLs can be individually read by the client.

6.1 Workload 1: Uniform Distribution

There is no fixed order of querying URLs in this workload. The pattern signature can be depicted as BDADACAABCBBADDABCD....

6.2 Workload 2: Statistical Distribution

In this distribution, each unique URL is queried a specific number of times consecutively. The pattern signature can be depicted as ABBCCCCADDDDBBCCCA. . . .

7 Experimental Setup and Methodology

The experimental setup for this project includes two laptops, one running the client on a 32-bit version of Ubuntu with 3 GB RAM and the other running the server and hosting the cache on a 64-bit version of Ubuntu with 8 GB RAM. Both devices consist of Intel's quad core i5 processor and are networked to each other via a common WiFi interface over port 8080.

As discussed in section 5, we evaluate the performance of the three caching policies by measuring the hit ratio and the average memory access time for the caches on the server machine. We write a test harness that includes a set of two URL list files, one for each workload, on the client that it makes request to the server from, and a code on the server to measure the aforementioned parameters for each workload.

Based on the workload patterns described in section 6, we hypothesize that, with the cache present in our experiments, the random eviction policy should perform better for workload 1 since there is no predictable pattern of querying URLs to the server, while LRU should perform better for workload 2 due to the doctored pattern signature. In either case, the no-cache policy should yield equivalent results. The actual results of this experiment are documented in the graphical format in section 8.

8 Results

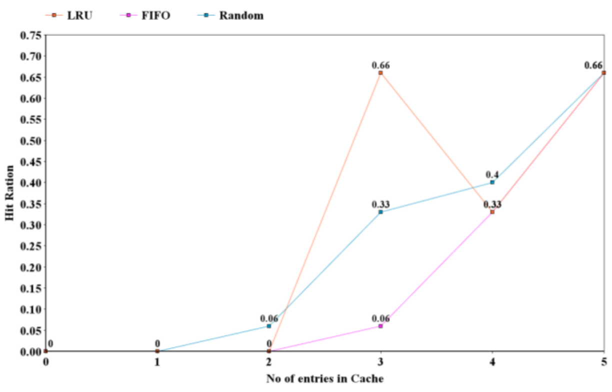

Figure 3: Workload 1 Hit Ratio

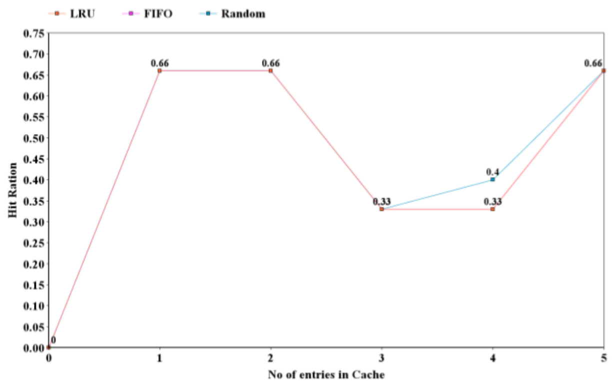

Figure 4: Workload 2 Hit Ratio

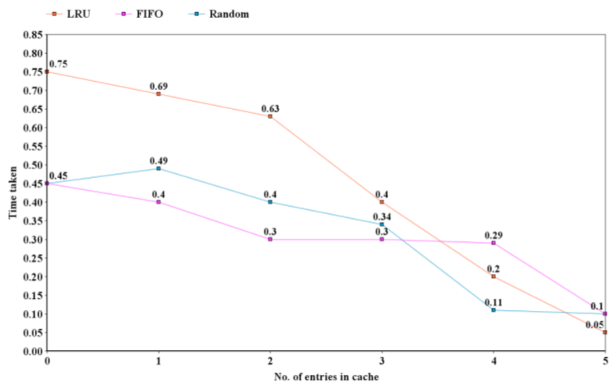

Figure 5: Workload 1 Average Memory Access Time

9 Analysis of Results

The graphs in section 8 are organized per workload, i.e. workload 1 and workload 2, and per performance metric. Thus, in total, there are four graphs, viz. Hit ratio for workload 1, Hit ratio for workload 2, Average memory access time for workload 1, and Average memory access time for workload 2. The X-axis depicts the increasing cache sizes, i.e. the number of entries that

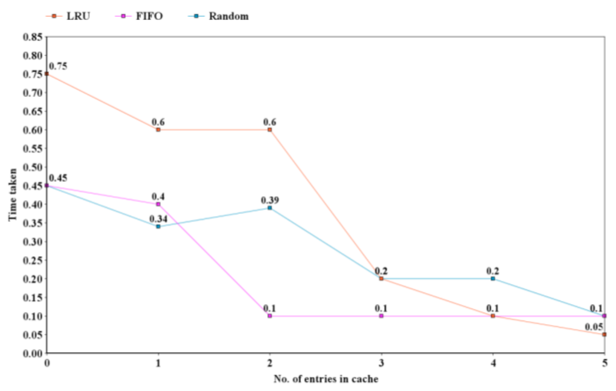

Figure 6: Workload 2 Average Memory Access Time

we store in our defined hashmap. The Y-axis depicts the increasing hit ratio or the increasing average memory access time, depending on the graph. All three caching policies are compared against each other on each graph.

For each caching policy, the extreme values, i.e. 0 and 5, result in almost equivalent performance values. This is due to the fact that 0 implies that the cache is not present and, therefore, the hit ratios will be 0 and the average memory execution time will be higher than when the caches are present. For workload 1, i.e. uniformly random distribution of workload, it can be seen than Random Eviction scheme performs better than the other two, since the probability distribution of hits is even in this case. For workload 2, i.e. statistical distribution of workload, LRU performs significantly better than FIFO and FIFO performs better than Random Eviction. This is because the workload distribution is heavily inclined towards reusing the same URLs consecutively, thereby favoring LRU over FIFO over Random.

10 Conclusion

The above results obtained from the experiments performed confirm our hypothesis stated in section 7. For extreme values of cache sizes, i.e. 0 and maximum unique URL requests, all three caches perform almost equally, while for intermediate values, random eviction performs better for workload 1 than the others and LRU performs better for workload 2 than the other two. Therefore, we can conclude that no cache policy can be the best for every request pattern. Each cache policy has its own unique signature that, when matched with the incoming workload, can produce the best performance results. Overall, caching reduces the average memory access time comparatively to a ”no-cache” policy, which is a significant performance boost.