CS 6210 Project 3 - RPC-based Proxy Server

>> Team Members
Rakesh Surapaneni (903088569)
Aishvarya Krishnan (903036986)

>> Introduction

In this project we have created a proxy server using '*Apache Thrift*', a Remote Procedure Call (RPC) framework for creating services on various platforms. A proxy server serves as a mediator between the clients and the server resources. When a client requests a particular service, the proxy server examines the request, fetches the desired data from the server resource and gives the data to the client. The clients send RPC requests to the proxy server for fetching the content of a given web page url. For the purpose of fetching the webpage content, we use *'libcurl'*, an open-source API for transferring data from given url.

To improve the speed of request-response and efficiency of the server, our proxy server is backed by a web cache of a limited size. The web cache is used to store a local copy of the webpage documents from previous requests made by the client. When a requested url content is not found in the web cache, it results in a cache miss which means that, the web page content has to be fetched from the internet. If the cache is full, based on the cache replacement policy followed, one or more of the existing pages in the cache is evicted and is replaced with the recent web page content.

We have implemented three cache replacement policies for the web cache and evaluated them based on cache hits-misses and access time for a set of urls.

We observe that for small number of url requests the cache replacement algorithms perform almost similar to each other and much better than no cache server. The performance seems to be affected by a marginal factor by the cache size.

>> Cache policies description

We have implemented the following cache replacement policies:

- **Random replacement**

In random replacement policy, we randomly select one of the stored pages from the cache and evict it. We repeat this process until there is enough free memory in the cache to accommodate the new page.

Pros

→ This type of cache replacement does not require storing details of access history for the pages in the cache.

→ When equal probability of the page to be evicted is ensured, over a period of time, the older pages are more probably going to be evicted.

Cons

→ Since the pages are evicted randomly inspite of the page being frequently accessed or used, this policy usually results in higher cache miss ratio when compared to other policies.

- **First-In-First-Out (FIFO) replacement**

In FIFO cache replacement, the page that has been in the cache for longest period of time is evicted first. Thus, the oldest page in the table is evicted irrespective of the frequency of page request or how recently the page is used. The order in which the page was stored in the cache is required in this policy.

Pros

→ This approach is a simple and low-overhead algorithm that involves maintaining only the order of insertion in the cache which can be done at ease using a linked list.

→ Deletion in the linked list is easy and is an O(1) operation since we just need to evict the page at the head of the linked list. When using a chained hash table, deletion will involve O(hash_table_size) operation, we have chosen hash_size in order of sqrt(N), where N is expected number of cache entries when cache is full.

Cons

→ This approach does not take into account the frequency of page requested or how recently the page was used, hence, there is less optimal use of cache compared to LRU.

- **Least-Recently-Use (LRU) replacement**

LRU replacement strategy involves evicting the page from the cache that is least recently accessed. The details of the page access time is required to be maintained in this approach.

Pros

→ This method takes into consideration how recently the page is used and hence usually results in better use of cache resources thereby leading to low cache miss ratio.

→ Insertion and Eviction is low-cost operation in LRU since it can be implemented similar to FIFO.

Cons

→ Whenever there is a cache hit, the page has to be moved to the end of the list to show that it is most recently used page. This is an additional overhead when compared to Random or FIFO replacement policies.

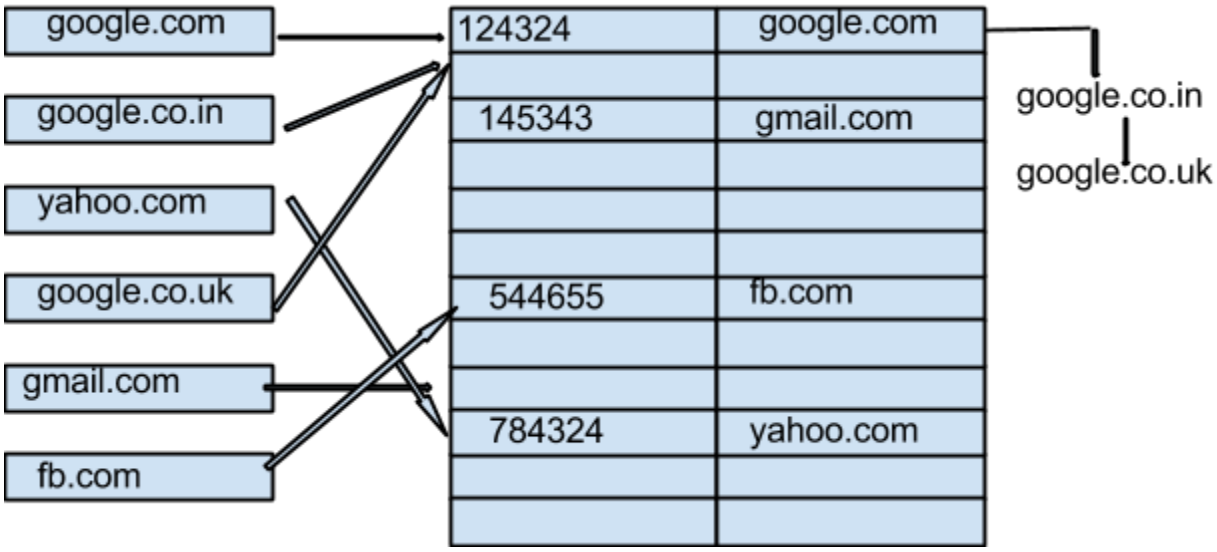
>> Cache design description

We have designed the web cache to be used in common for all the cache replacement policies.

- **Data Structures**

We have used a chained hashtable with linked list data structure to manage the web cache. It stores linked list of pages at each index to which the urls are mapped, to ensure

minimal insertion complexity and search complexity. The hash function we have used involves converting the url string to chars, converting it to int32 and then taking modulus HASH_SIZE. The urls indexed to same value are stored in a linked list.



→ HashMapList.cpp

It consists of the chained hashtable datastructure with the values consisting of linked lists of pages (implemented in LinkedList.cpp). We have implemented our own version of the Hashtable to customize the functionality tailored to our needs.

For example, we have implemented the *free()* function that selects the page to be evicted based on the *cache_scheme* to be used. Similarly, we can fetch a cache entry using it's url(key) as well as the timestamp(index).

→ LinkedList.cpp

The linked list node consists of: string *url*, char* *val*, List* *next* and int *counter*. The *counter* value is maintained for determining the page to be evicted in case of FIFO and LRU policies. It's value is assigned to *curr_counter* value during insertion of the page to the table. The *curr_counter* value is incremented after every insertion to the table. The *min_counter* value maintained in Hashtable is used for FIFO and LRU policies to search and evict the pages speedily.

We have customized functions implemented in the linked list to enable fast operations such as *remove_head(), remove(by_id).*

- **Operations**

  The operations used for cache replacement are:

→ fetch(string& _return, string url)

When a page is requested by the client, the server first checks if the page content is present in the cache. To fetch from cache, the *fetch()* function is invoked, which checks the cache for the page's content, and if *_return* value is *not null*, cache hit has occurred and the value in *_return* is the page content to be provided to client.

→ insert(string content, string url)

When a requested page is not found in the web cache, the *_return* value obtained from the fetch operation would be *null*. In this case, the content is fetched using '*libcurl'* and since it is most recently used, it has to be stored in the cache. The *insert()* function is used to store the url and web page content as key-value pair in the hashtable. If the cache cannot accommodate the new url and content values, cache replacement process is executed.

Cache replacement involves evicting pages until the required amount of memory is freed for the new content to be stored. For cache replacement, we have the *free()* function which checks the cache scheme that needs to be performed based on which it evicts pages.

- **Algorithms**

  We have implemented three cache replacement algorithms as follows:

  1. Random replacement

When there is a cache miss and the new page content has be fetched, if random cache replacement policy has to be performed, in the *free()* function, a random page is selected from all the entries in the hashtable and the page is evicted repeatedly until there is enough free space for the new page to be stored. Fairness is ensured in selecting a random page to be evicted by first selecting an index based on the total number of entries in the hashtable and assigning probability of a page to be selected from a list based on the size of the lists.

2. First-In-First-Out (FIFO) replacement

In case of FIFO cache replacement, if the *min_counter* of the linked list at a particular index is equal to the global *min_counter* value, the *remove_head()* function is called at that index to evict the page at the head of that list. The global *min_counter* value is incremented by 1 to find the next oldest page in the table. This process is repeated until there's enough free memory for new page to be stored in the cache.

3. Least-Recently-Use (LRU) replacement

In case of LRU cache replacement, whenever there is a cache hit, that is, the page requested by the client is found in the cache, the page is removed from the current position and reinserted at the end of the list with an updated *counter* value to denote that it is most recently used. When there is a cache miss, and cache cannot accommodate the new page cache replacement process is initiated. In the *free()* function, if the *min_counter* of the linked list at a particular index is equal to the global *min_counter* value, the *remove_head()* function is called at that index to evict the page at the head of that list. The global *min_counter* value is incremented by 1 to find the next oldest page in the table. This process is repeated until there's enough free memory for new page to be stored in the cache.

>> A Comparison of the cache replacement policies

We used normal distribution and uniform random distribution of list of urls. Among the chosen cache replacement policies, ideally LRU should perform well in normal distribution (which is expected to be similar to real-world usage). This is because, in real-world scenarios some pages are more frequently accessed when compared to others and hence we expect to see better performance in terms of both cache miss ratio and server access time with LRU.

In uniform random distribution, it is hard to predict which cache replacement policy would perform better, so ideally the results should be comparable.

>> Evaluation Metrics

We have chosen the following two metrics to evaluate the performance of our implementation of the cache replacement policies:

- **Cache miss ratio**
  C.M.R = Number of cache misses/ Total number of urls accessed

→ This metric is essential to evaluate the performance of cache replacement algorithms since it tells us information about average cache misses when using a particular policy.
→ Cache miss ratio should be influenced by two factors, namely, the efficiency of cache replacement algorithm and size of the cache. As the size of the cache increases, the likelihood of finding a page in the cache increases.
Similarly, if the cache replacement algorithm is smart, the cache miss ratio will be minimal.

- **Average Server access time per url**
  Avg.S.A.T = Total server access time to access all the urls/ Number of urls accessed

→ This metric gives us information about the time taken to serve the requests when using a particular policy. This metric is also essential since it takes into consideration the utilization of cache memory.
→ Similar to cache miss ratio the efficiency of cache replacement algorithm and size of the cache should have effect on this metric. In other words, this metric is dependant on cache miss ratio since fetching a page from local memory would result in lower access time.
Also, if the cache replacement algorithm is smart, the average server access time will be reduced.

>> Workloads

Workload is important in order to evaluate different cache replacement algorithms, since each cache replacement algorithm performs differently for different order of page requests. A workload, here, is a list of urls generated from a large set of urls in a text file. For example, there maybe some popular urls or some pattern in which urls are accessed across

internet by various users. Hence, it is very important to capture these patterns using some form of distributions in our workload while evaluating our algorithms.

We have used two different probability distribution functions to generate two workloads in order to evaluate our algorithms.

- **Uniform random distribution**

→ In uniform random distribution, we have a list of 10 urls which we are randomly querying 50 times with repetitions. We have used uniform random distribution to simulate selection and repetition of urls with equal probability.
→ Since each url is likely to be queried with equal probability, we expect to see high cache miss ratio in all the algorithms. This because, there is no pattern which can be effectively exploited to improve any of the cache replacement policies' performance.

- **Normal distribution**

→ In normal distribution, we want to simulate real-world scenario of page requests where some urls are more frequently accessed over others.
→ This gives us a scope of optimization in cache replacement policies since we have some information about the frequency of usage of various urls. Thus, it is important to actually evaluate at least some aspects of various cache replacement algorithms.
→ To implement this, we have used *variate_generator* function of the boost library over normal distribution. This function provides us with a random number generator which can be used to generate a normal distribution.
→ We have used mean = number of urls/2 and standard deviation = number of urls/5 for the random number generator.

>> Experiments description

We have run our experiments to compare and evaluate the performance of FIFO, LRU and Random cache replacement policies as well as No cache server.
For our experiments, we also wanted to factor the cache size of the web cache and analyze the performance for various scenarios. We have allocated different cache sizes as follows: 512KB, 1MB, 2MB, 4MB, 6MB and 8MB.

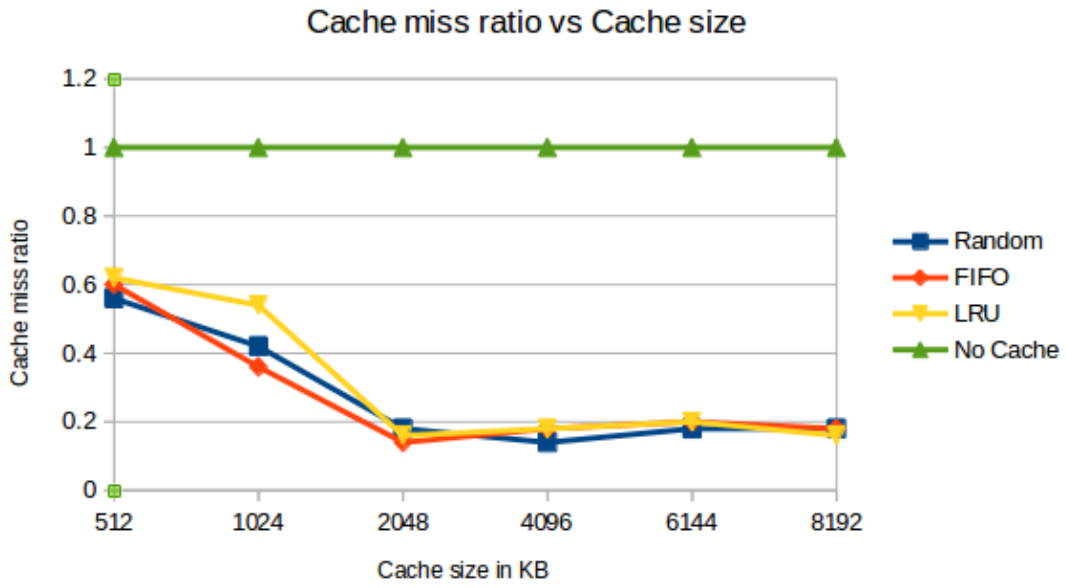
>> Experiments results and analysis

Here, we have evaluated the cache miss ratio and average server access times for each cache policy we have implemented and no cache server using line graphs. This helps us understand performance of the various caching policies.

- **Cache miss ratio vs Cache size for normal distribution workload**

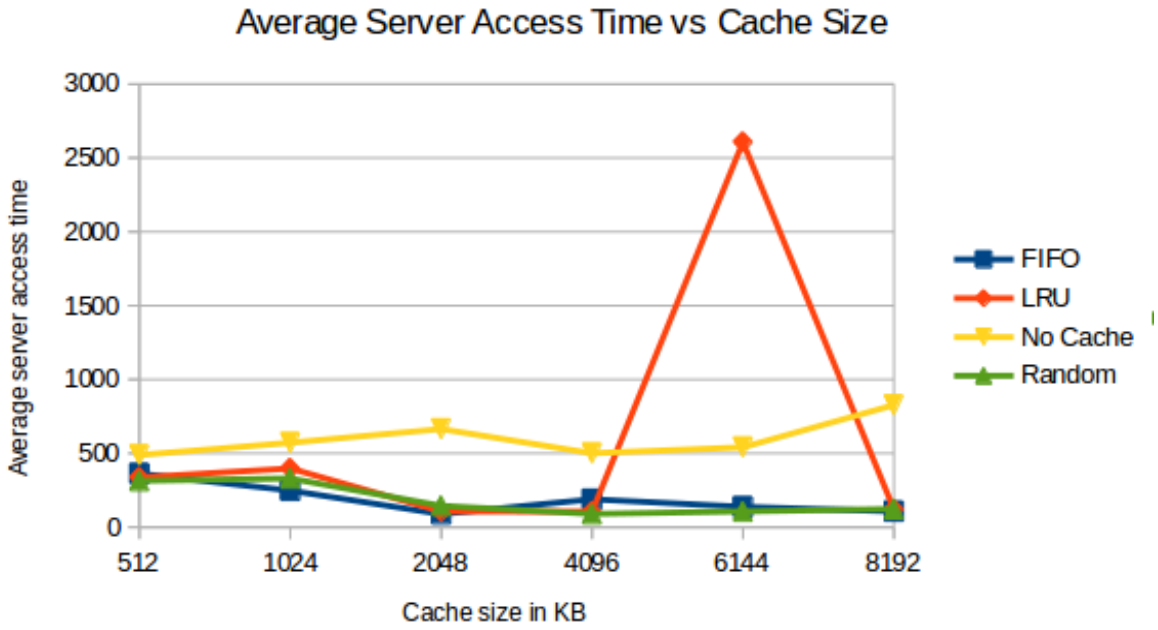
Since the number of urls we have used are less we need to evaluate the caching policies for lower cache sizes as well. We see that the cache miss ratios are comparable and almost similar for all caching policies and all cache sizes with LRU performing slightly worse

for lower cache sizes which is unexpected. This is probably due to lower number of url requests.



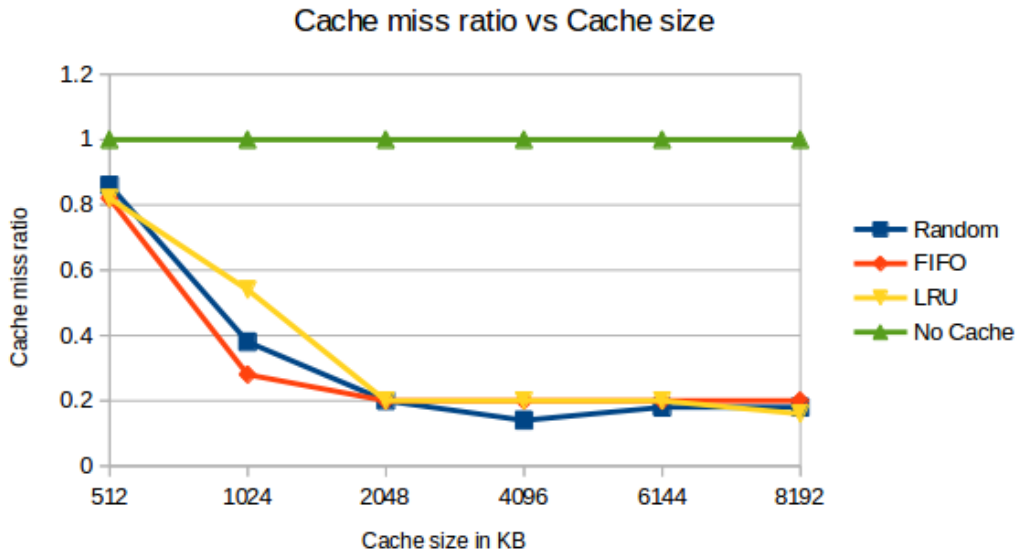
- **Cache miss vs Cache size for normal distribution workload**

As expected, we see that no cache server is performing worse compared to cache backed servers. The sudden peak in LRU is probably because of temporary congestion in the network ignoring which at all other times the results are comparable and almost similar for the caching policies.



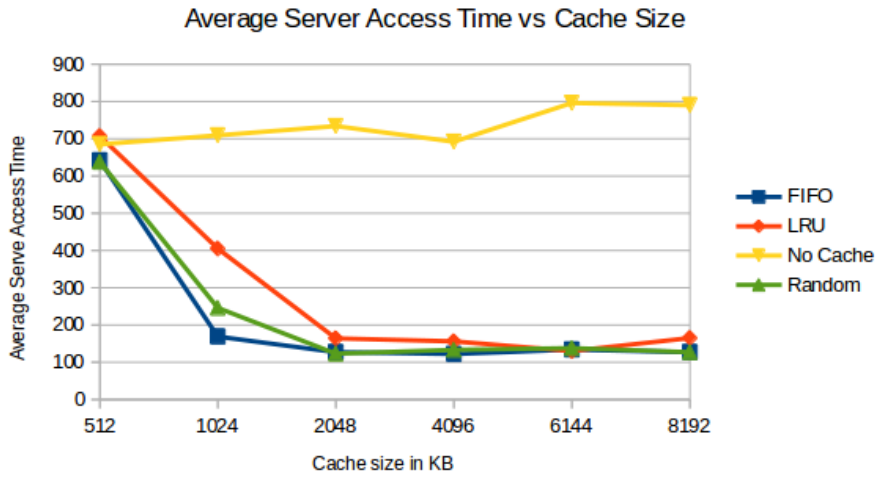
- **Cache miss vs Cache size for uniform random distribution workload**

Since the number of urls we have used are less we need to evaluate the caching policies for lower cache sizes as well. We see that the cache miss ratios are comparable and almost similar for all caching policies. The only factor that seems to affect the performance is the cache size.



- **Cache miss vs Cache size for uniform random distribution workload**

As we have expected, we see that no cache server is performing worse compared to cache backed servers. However, in lower cache sizes such as 512 KB, performance of cached servers is similar to that of no cache server since there is no pattern in the url requests which can be exploited by the cache servers. At all other times the results are comparable and almost similar for the caching policies. The only factor which affects is the cache size.



>> Conclusion

Through this project we have been able to observe and analyze the working of an RPC-based proxy server under various scenarios. We have understood the importance of cache replacement algorithms in a web proxy server and the impact it has on server access times which plays a very important role in real-world scenarios. We have also observed that the workload distribution of url access plays a very important role in designing a system such as proxy server.
For example, for lower cache sizes in uniform distribution, the underlying implementation of cache replacement policy, in fact caching itself is observed to be immaterial for the evaluation metrics we have used - which is unlikely in real world situations that is demonstrated using normal distribution.
For calculating the statistics we have averaged over multiple runs however some of the results are still unexpected probably due to smaller set of url requests used. For instance, we expected LRU to perform significantly better over other caching policies but we have not observed this behaviour in our experiments.
Also, we observe as expected that the performance is directly proportional to the cache size and using bigger cache size results in significantly better performance to some extent.