**CS 6210: Advanced Operating Systems**

Project 3: RPC - Based Proxy Server

Yash Thadhani
Vivian Dsilva

## INTRODUCTION

Remote Procedure Calls (RPC) are a powerful and commonly-used abstraction for constructing distributed applications. Apache Thrift is a modern technology that is used for making Remote Procedure Calls. This project which involved the use of RPC had two basic goals: First, to introduce us to programming with a real remote procedure call system. Second, to explore the principles and performance of different caching schemes in a distributed application.

In order to achieve these goals, we did the following:

1. Implemented a 'web proxy' server using Apache Thrift.
2. Investigated and implemented 3 different caching mechanisms in the server.
3. Evaluated the performance of the service under different load conditions using different caching schemes.

## CACHE DESIGN DESCRIPTION

Using the RPC framework, the client invokes get_data(url). When the server receives this request, it first checks its cache to see if the page is present. If it is not present, it uses commands defined in the libcurl software package to obtain the requested webpage and store it in the cache.

Data structures and cache parameters:

1] The cache designed by us uses the std::map data structure. The std::map data structure is basically a hash table which is implemented as a self balancing tree and hence provides lookup, insertion and deletion of elements in sublinear time with a time complexity of O(logN). We populated the map with <key, value> pairs. The 'key' was the URL of the webpage requested by the user whereas the 'value' was a pointer to a structure representing a cache entry. This structure (called cache_entry_t) had various fields including the actual HTML content of the website, the size of the webpage, a timestamp denoting the time that URL was last accessed and a unique identifier which was assigned to it.

2] A variable corresponding to the total memory used was created and this only took into account the size of the webpage content and not the aforementioned variables declared within our structure.

3] We use a variable called min_entry_size to denote the smallest possible block that can be added to the cache. The fundamental purpose of this variable is to estimate the maximum number of entries our cache can accommodate.

4] We use a dynamic array with a size corresponding to the maximum number of possible cache entries to store the cache_entry_t structures corresponding to all URLs accessed by the user. Each cache_entry_t structure is assigned a unique identifier which is used to index into the array.

5] Furthermore, we store all available identifiers in a deque. The reason for using a deque was that pushing and popping elements from the front of the deque are constant time operations. At the start, when the cache is empty, the deque contains all possible identifiers from zero to the maximum possible cache entries minus one. Whenever we wish to add a new entry to our cache, we simply pop the identifier at the front of the deque and assign it to our new cache entry.

6] We used a minimum indexed priority queue as our eviction queue for our Least Recently Used (LRU) and Largest Size First (MAXS) (involved a slight modification which is explained in the next section) cache eviction policies. This data structure is implemented as a binary min heap such that it rebalances itself at each insert, update and delete operation to ensure that the smallest element (element with the smallest "key") is at the top of the heap thereby ensuring that eviction/deletion takes constant time. This structure is implemented using three dynamic arrays namely 'pq' which stores the identifier that is used to index into the data structure, 'keys' stores the timestamp at the position corresponding to the identifier and 'qp' is used to identify whether a particular index is valid/currently present in the data structure (initialized to -1). When an element is inserted at an "index", qp[index] is initialized to the current size of the queue.

7] For our random cache eviction policy, we tried out two different data structures for our eviction queue, a vector and an indexed random queue. The indexed random queue comprises of two dynamic arrays, 'pi' which is used to store the identifier and 'ip' which is used to ascertain the validity (whether it is present or not) of an entry.

<u>Caching policies and algorithms</u>:

Since the cache capacity is limited, there was a need to evict elements from the cache so as to make space for new webpages. To achieve this we implemented 3 cache eviction schemes:

1. Least Recently Used (LRU): remove the item that has been requested least recently.
2. Largest Size First (MAXS): remove the item which has the largest size, assuming that users are less likely to re-access large documents because of the high access delay associated with such documents.
3. RANDOM (RAND): remove an item at random.

**LRU:**

Once the get_data function of the server is invoked by the client through an RPC call, the first thing we do is to check whether our cache has the webpage corresponding to the requested URL.

**1] On a hit in the cache:**

- The timestamp (calculated using the gettimeofday() method) corresponding to the cache entry must be updated.
- To do this, we query our map data structure (logarithmic time complexity) using the URL and obtain the pointer to the corresponding cache entry. We dereference it to obtain the unique identifier corresponding to this cache entry.

- Now the eviction queue must be updated to reflect the new timestamp. So we index into it using the identifier and update the corresponding position in the keys array. This entry would sink down in the priority queue keeping the element that was accessed the earliest (lowest timestamp/key value) at the top of the priority queue.
- Lastly, the contents of the webpage requested are returned to the get_data() function which in turn returns it to the client.

**2] On a cache miss:**

- If the requested URL was not found in the cache (map data structure), cURL performs an HTTP GET request to fetch the webpage.
- Now this entry must be stored in the cache. First we check whether the cache has enough space to store the new webpage. If it does not, the least recently accessed webpages are deleted to make room.
- Evicting the LRU candidate from the eviction queue has a logarithmic time complexity. The root of the binary heap/indexed priority queue is the element with the lowest timestamp. To avoid creating an empty spot, we bring the element at the last index to the top (a simple swap) and since it would most probably be larger than its children, we allow it sink to a suitable position such that the element with the smallest timestamp stays at the root position. We ensure that the delete function returns the identifier of the element removed, using this identifier we index into the dynamic array, obtain the pointer to the corresponding cache entry, dereference it to obtain the key/URL and index into the hash map using the key to delete the value/webpage. Since a cache entry has been evicted, the corresponding identifier is available for reuse and is pushed to the front of the "id_queue" deque.
- Now that there is sufficient room in the cache, we can add a new entry. First we pop the front of the "id_queue" deque to obtain an available identifier. We index into the dynamic array using this identifier and store all the parameters (URL, data and value_size). Next the entry is added to hash map. Lastly, the timestamp corresponding to this cache entry is inserted into the priority queue at an index equal to the identifier. We allow the heap to rebalance itself by inserting the identifier at the end of pq array and thereafter invoking the swim method to ensure this element is moved to an appropriate position in the queue depending on its timestamp.
- Lastly, the contents of the webpage requested are returned to the get_data() function which in turn returns it to the client.

<u>Pros</u>:

- It will perform well when a set of URLs are accessed frequently. The idea is that items requested recently are more likely to be requested in the near future.
- It keeps a record of cache accesses using timestamps and adapts to the data access pattern/workload.

<u>Cons</u>:

- It performs poorly if a set of pages are accessed sequentially with the period of page recurrence being greater than the maximum capacity of the cache.

**MAXS:**

MAXS evicts the largest page from the cache to make room for a new entry. For our MAXS implementation, we retained the indexed minimum priority queue as our eviction queue. However, instead of storing timestamps in the keys array, we calculated the difference between the cache_capacity and the size of each page (the variable was named 'val_size_complement') and stored the same in the priority queue. The page with the smallest 'val_size_complement' would be the largest page and would also be the root of the binary heap/priority queue. Another important difference was that there was no need to update the priority queue whenever there was a hit in the cache.

Pros:

- It ensures that a larger number of webpages can be accommodated in the cache.
- Users are less likely to access larger webpages owing to the latency incurred in fetching them.

Cons:

- Performs poorly is if the user accesses larger pages frequently.

**RANDOM:**

In our eviction queue implementation using a vector, on creating a new cache entry, we merely push the corresponding identifier to back of the vector. For evicting an entry, we use the rand() function to obtain a random number between zero and the size of the vector and "erase" the element at that position. However, on further analysis we realized that the time needed for removing an element from an arbitrary position in a vector is linear in terms of the number of elements following that element. In our second implementation, we used an indexed random queue. The time required to evict an arbitrary element from this data structure is an amortized constant. The element at the random index (obtained using the rand() function) in the pi array is merely swapped with the last element of the array and the corresponding entry in the ip array is set to -1 indicating that the element is no longer valid/has been removed.

Pros:

- It is easy to implement since there is no need to maintain a record of the previous cache accesses.
- The random cache eviction scheme would perform well when the user is accessing URLs arbitrarily and not accessing a set of URLs multiple times.
- The idea is that, even if it does not perform maximally well, it will not perform absolutely abysmal under any load.

Cons:

- Random might perform poorly when the user is frequently visiting a set of URLs.

## METRICS FOR EVALUATION

### 1] Hit Ratio:

It is basically the ratio of the total number of hits in the cache to the total number of webpage accesses (number of entries in the workload). It is a good measure of the efficiency of the cache eviction policy employed since a higher hit ratio implies that a large number of webpages requested were found in the cache thereby eliminating the need of the proxy server to perform an HTTP GET request for that webpage.

**2] Average Time Taken:**
This represents the average time taken by the proxy server to service a webpage request. We use the time() function which gets the current calendar time as a value of type time_t. The begin time is calculated before we query our cache and the end time is calculated before we return the webpage content to the client. The difference between the two is ascertained for each webpage request. We also find the sum of individual access times and divide it by the total number of accesses to obtain the average time taken for a single access.

It is possible that a cache might have a high hit ratio but might employ a suboptimal lookup algorithm. This is where the average time taken assumes importance. Hence, the two metrics together give us a better idea about the efficiency of an eviction policy.

## DESCRIPTION OF WORKLOADS

1] Random replication followed by random shuffle: We wrote a batch_generator program in C++ to generate this workload. We populated a file "urls.txt" with a set of unique URLs without any repetitions. The program reads URLs from this file one line at a time, picks a random number between 7 and 16, replicates each URL those many times and pushes all the entries into a vector. Thereafter, we perform a random_shuffle on this vector before writing its contents to a file "Workload1.txt".

2] Statistical distribution: This workload was designed keeping in mind the actual web browsing experience. First a few small webpages are requested, followed by a couple of larger webpages and finally the largest webpage is requested. For example, if the user is searching for something on http://www.google.com/. Initially he navigates through the search results presented by Google (small pages) before clicking on a particular page (large page). He might not find this satisfactory, he would then go back and continue searching. Eventually he finds what he is interested in and downloads the web object (largest page). This workload is mentioned in the file Workload2.txt.

## EXPERIMENT DESCRIPTION

We ran the client and server on two separate machines running Ubuntu 14.04.1 LTS. Both the machines were in in the same home network. The router was NAT enabled and the IP addresses of the machines were in the private IP subnet 10.0.0.0/24. The server was run with the desired cache eviction policy and total cache capacity as command line arguments. The client first creates a TCP socket with the IP address and port number of the server as parameters and then opens the transport tunnel for reading or writing. The URLs are read line by line from the workload files and for each URL, the client invokes the RPC method get_data() on the proxy server passing the URL as a parameter. On receiving the client request, the proxy server obtains this webpage (from its cache or by making an HTTP get request), stores it in a string and returns it to the calling procedure on the client.

The statistics measured for a workload are the hit ratio and the average time taken. In order to return these statistics to the client we wrote the method print_statistics() and invoked the same from the client using an RPC call.

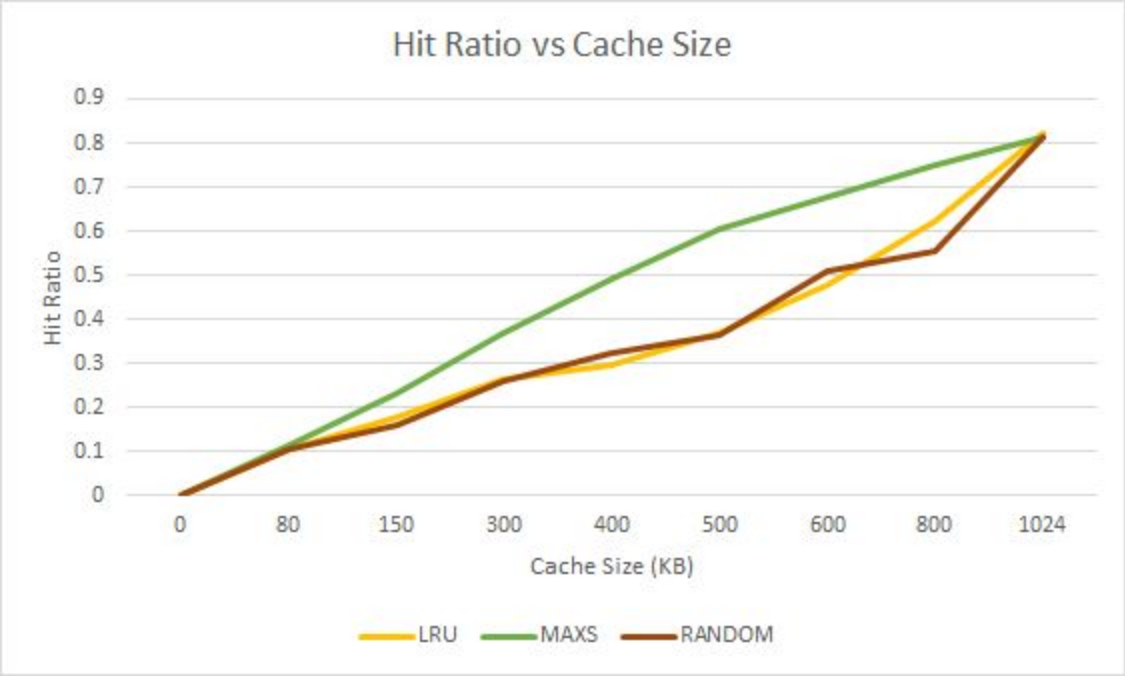
Hypothesis:

1] Random replication followed by random shuffle: There was no mathematical relation/statistical distribution used to generate this workload, it was completely random as the name suggests. Hence, we assumed that the random cache eviction policy would perform the best.

2] Statistical distribution: For this workload, we assumed that the LRU cache eviction policy would perform better than MAXS. This is because in this workload, most webpages are accessed multiple times. MAXS would end up evicting the largest webpage which would be accessed again in the near future. Since LRU exploits temporal correlation to a greater degree, we assumed that it would perform better than random and MAXS.

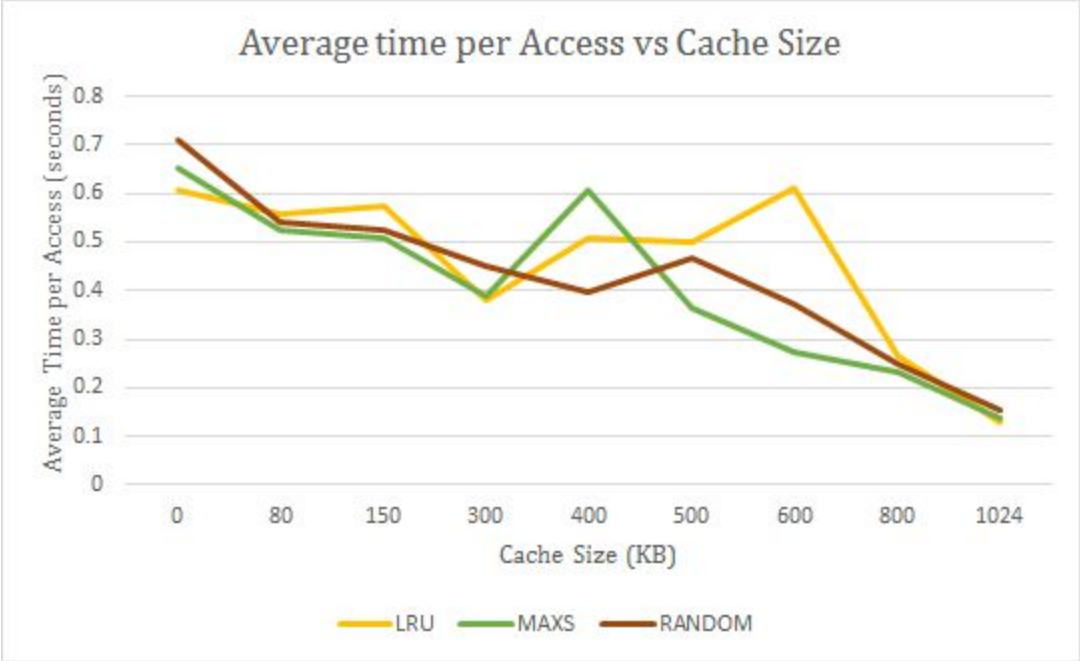
## EXPERIMENTAL RESULTS AND ANALYSIS

Random replication followed by random shuffle:




This result was fairly counterintuitive since we had expected the random cache eviction policy to perform the best. To understand the reason for this behavior, we ascertained the size of all the webpages used in this workload. We observed that were a couple of really large webpages and a greater number of smaller webpages. MAXS evicts the largest webpage each time thereby freeing up memory in the cache which would be sufficient for several smaller webpages. Due to this, the cache accommodates a larger set of webpages thereby improving the possibility of a hit.

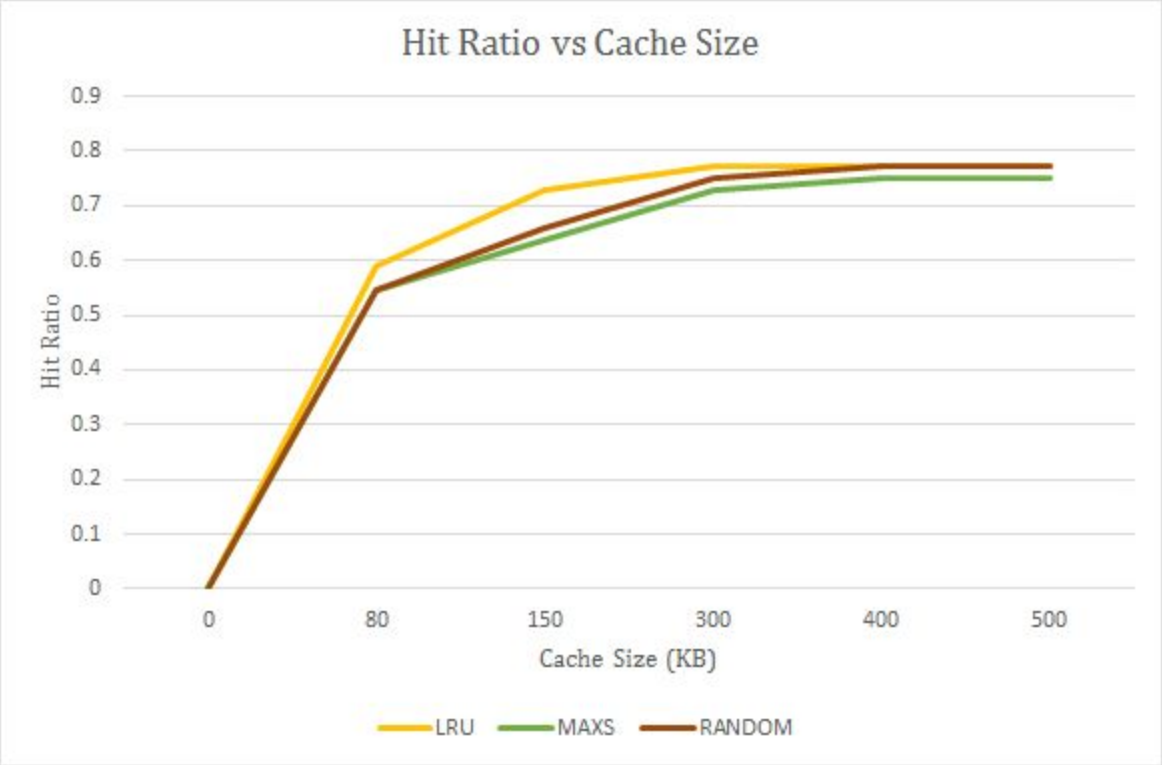
Also, the hit ratio increases with increase in cache size. This is what we expected as larger the cache, the probability of a webpage being found in the cache is higher and hence the hit ratio would be higher.



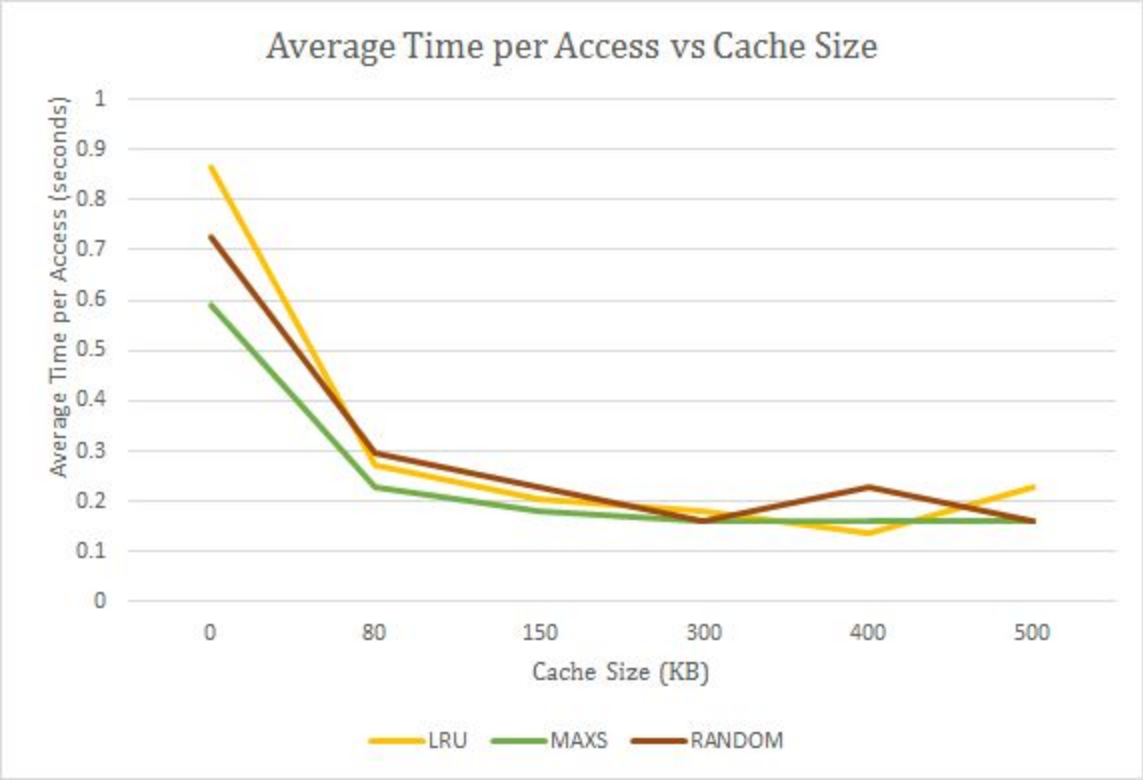

The reason for the behavior observed in this plot stems from the justification for the behavior observed in the previous plot. When MAXS evicts the largest page from the cache, multiple smaller pages can be brought into the cache. When these pages are accessed again in the future, they would be found in the cache and there would be no need for an HTTP GET request. Hence, this reduces access latency thereby improving the average time per access in the MAXS eviction policy.

Also, from the graph one can observe that as the cache size increases, the access time decreases. This too is justified since larger cache sizes would mean that the probability of a page being in the cache is more, thereby reducing the average time of accessing a web page since the page needs to be brought in from the cache which would be faster than actually fetching the page from the website.

Statistical Distribution:




The behavior observed in this plot was exactly as we had expected. In this workload, each webpage is accessed multiple times including the larger webpages. MAXS ends up evicting the largest page each time however, the page evicted is requested again in the near future. This leads to a cache miss and hence a poorer hit ratio. LRU on the other hand maintains an access history by means of timestamps and it evicts the least recently used webpage which is not necessarily the largest webpage. LRU clearly performs better than MAXS due to this reason. Also the hit ratio increases with increase in cache size since a larger cache can accommodate a larger number of webpages thereby increasing the probability of a cache hit.




It is interesting to observe that despite having performed worse than its peers in terms of hit ratio, MAXS actually has a lower average time per access and hence performs better than LRU and RANDOM for this workload. On careful analysis, we observe that the largest page in our workload is requested periodically. At times the webpage accesses are in increasing order of size. In this situation, the LRU candidate would be the smallest webpage, hence LRU would end up evicting multiple pages thereby nearly clearing the cache. All these webpages which were evicted are accessed in the near future leading to cache misses. MAXS on the other hand, starts evicting pages starting at the largest. Hence, the number of pages evicted by MAXS would be lesser than the number of pages evicted by LRU or RANDOM. Hence the possibility of finding a web page in the cache would be higher. As this data would be fetched from the cache, the latency of this operation would be much lesser.

As expected, we also observe that as the cache size increases, there is a decrease in the average time per access.

## CONCLUSION

In this project, we implemented a RPC based Proxy server using Apache Thrift. We also implemented 3 different caching mechanisms in the Proxy server and used 2 different workloads to understand and analyse as to which caching mechanism performed better for a particular workload.

Using Hit Ratio and the Average Access time together, we were able to have a better understanding of the performance of a caching mechanism. We observed that the MAXS eviction algorithm performed better in terms of Hit Ratio and Average Access time for random requests because eviction of a larger page would mean that a bunch of smaller pages could be brought in and since these web pages would be accessed again at some other time, the probability of finding the page in the cache would increase.

Also, for the second workload which took into account the usual behavior of a user who is accessing the internet, we observed that LRU performs better when we take into the hit ratio. But when we look at the graph of the access time, we observe quite an interesting trend where MAXS actually performs slightly better than its peers.

This project gave us a sound understanding of how one could design a proxy server and implement caching mechanisms based on a user's web page requests. While selecting a good cache eviction policy, the kind of webpages being requested, the size of the webpages, the browsing habits of a user, etc should also be considered.