# CS 6210 Project 3 – RPC-based Proxy Server
**Name: Wan-Chen Yeh, Huangwei Fang**

## I. Introduction

This project is to implement the remote procedure call (RPC) based proxy server using Apache Thrift. The proxy server provides multiple services based on the different cache replacement policies.

In this project, we implement four replacement policies (Random, FIFO, LRU, and MAXS) in the proxy server. To evaluate the performance, we create workloads with uniform and normal distribution. In normal distribution, the url for larger size web page has lower visit probability. In other words, we assume that client access the large size website infrequently.

We evaluate the performance by two metrics: elapsed time and cache hit rate. The elapsed time of accessing all the websites from the url list is stored on the client side, and the cache hit rate is saved on the server side.

The rest of the report is organized as follows: the cache replacement policies we have implemented are described in Section II. In Section III, we will introduce the metrics for evaluation and how we implement them in our code. The workload generation is illustrated in Section IV. The experiment results are demonstrated in Section V, and the analysis for the results are discussed in Section VI. Finally we conclude this report in Section VII.

## II. Cache design and policies

In our cache design, we use the map function provided by C++ as the data container to save the url as the key and the corresponding web page content as value. The elements in map are associative and sorted. It can be easily access by the key or iterator to obtain the value. When a new url and its content are updated to cache, we simply insert the pair <url, body> to the map. When a <url, body> pair is needed to be evicted, we remove it from the map by specifying the key (url). This method can simplify the implementation of searching and eliminate the computation complexity.

In this project, we implement four cache replacement policies: random, First In First Out (FIFO), Least Recently Used (LRU), and Largest Size First (MAXS). The details of each design are discussed in the following paragraphs:

In random policy, it simply evicts a web page from cache with random selection. Since the eviction does not need any extra information about the web page stored in cache, we just generate a random number as an iterator in the map to specify the element to be deleted. Although random policy is easy to implement, it cannot provide good performance due to eviction is not with any preference.

In FIFO policy, we create a queue to store all the visited urls sequentially. When a new web page is saved in the cache, we insert the corresponding url to the queue. When eviction occurs, we pop out the url which is the earliest stored in the queue. The FIFO can be implemented easily as well. However, it does not perform well if the visited url is requested repeatedly in a short period. In

other words, the eviction rule is only based on the time of the creation instead of the last access time. The later policy is actually LRU policy, we will have more details in the next paragraph. Nevertheless, if the cache size is large enough to hold more urls, then FIFO will have better performance.

In LRU policy, we create one more map to save <url, count> pair, where the count is for recording the access history of the url. All the urls present in the cache are kept in this map. When the client requests a url that already in the map, we update the associated count to 0. Otherwise, we insert the url to the map and set its count to be 0 which means that the page is most recently used. At the same time, we increase others' counts by 1. In other words, the url which has the largest count is the oldest (least recently used). Therefore when eviction happens, we remove the web page which is least recently used. The LRU should have a good performance if we assume that clients will revisit those pages that they access most recently. However, it does not perform well all the time if the cache size is relatively small. We will explain the reason in detail later in this report, based on the experimental results.

In MAXS policy, the eviction is determined by the size of the web page. As we mentioned in the beginning of this section, we use map to save the url and web page. In the map, the web page is saved as a string, so we can easily get the size of the string by the API provided by C++. Since we can get the size of web page, we can delete the web page with largest size in the cache, if eviction happens. The MAXS policy works perfectly if we do not access the large size web page frequently. Moreover, for small cache size, the MAXS policy also works pretty good. However, the performance improvement is not notable when the cache size is relative large and when the network connection is very bad.

## III. Metrics for evaluation

We are using two metrics for evaluation. The first one is the elapsed time for getting the contents of the webpages back to the client. The elapsed time includes reading the urls from the text file (url_list.txt), sending request to server, and receiving the response from server. In this project, we calculate the elapsed time for obtaining 200 urls for each configuration (cache policy and cache size). Besides, we use the function gettimeofday() to take timing measurements. The function returns the system time in the order of seconds and microseconds. Another metric is the numbers of cache attempts, hit, and miss. For each configuration, the server stores the status of cache access. If the content of a url can be found in the cache, then the hit count is increased by 1, otherwise the miss count is increased by 1.

## IV. Workloads

In our project, we create two workloads to test the performance of different policies. One is the normal distribution and the other one is uniform distribution using rand() function provided by C++. In order to create a workload with the statistical distribution, we put 10 url candidates in an array then we use statistical methods to randomly generate a number in the interval [0, 9] to specify the index of the array. The array is ordered based on the body size: the large-sized urls are put on the endpoints, and the small-sized urls are put in the middle of the array. In the project, we generate 200 urls for each experiment. In uniform distribution, since we use rand() function, it is approximately uniform distribution because the smaller numbers may have higher probability. In the normal distribution, the web pages with larger sizes have the lower access probabilities, and

the smaller-sized web pages have higher probabilities. The distribution of the url lists are illustrated in Figures 1 and 2, which are obtained from the url list generated by norm_dist.cpp.

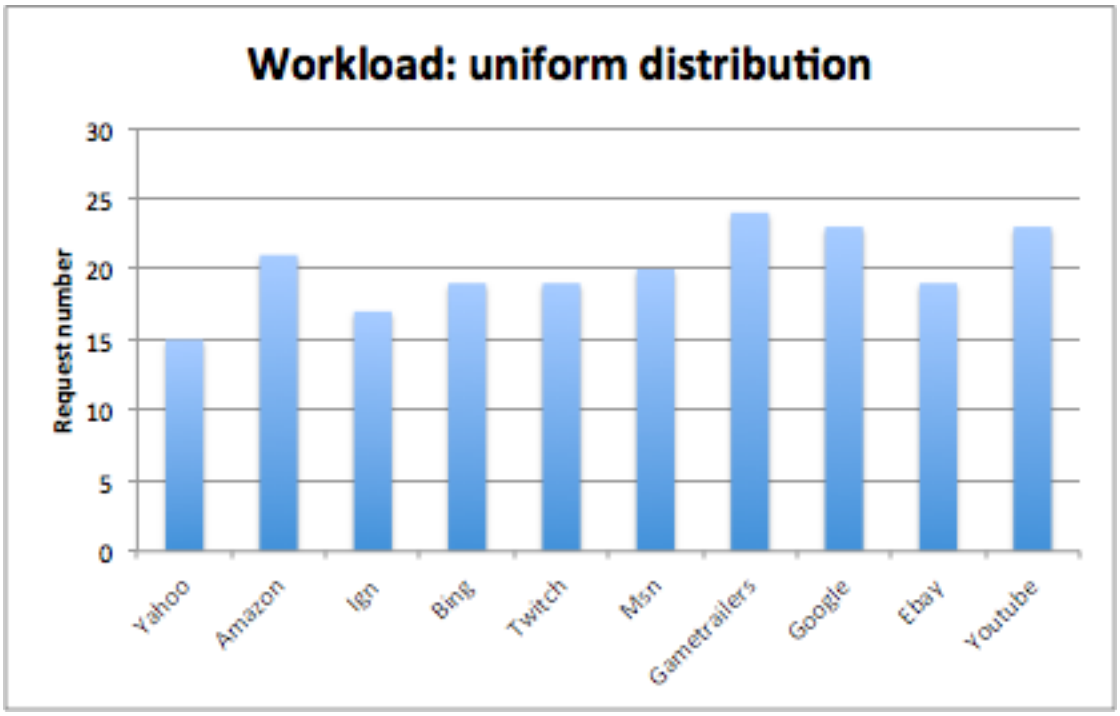

Figure 1: Uniform distribution for workload 1

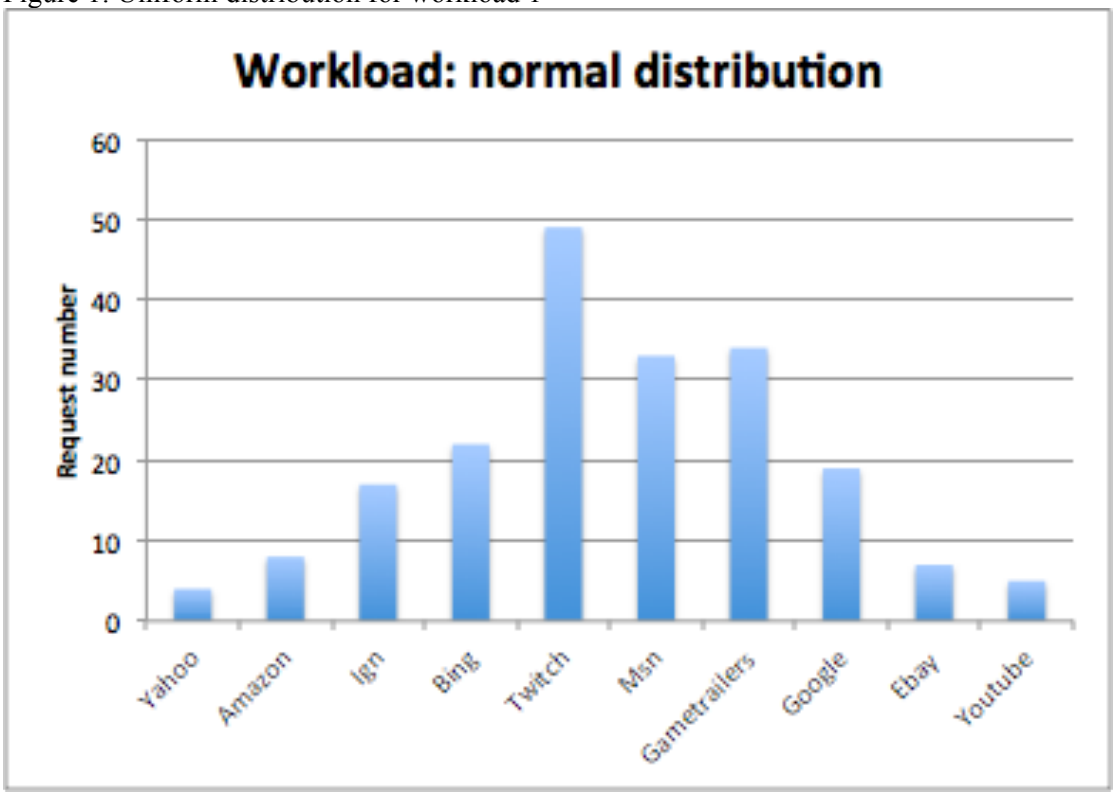

Figure 2: Normal distribution for workload 2

## V. Experiment Results

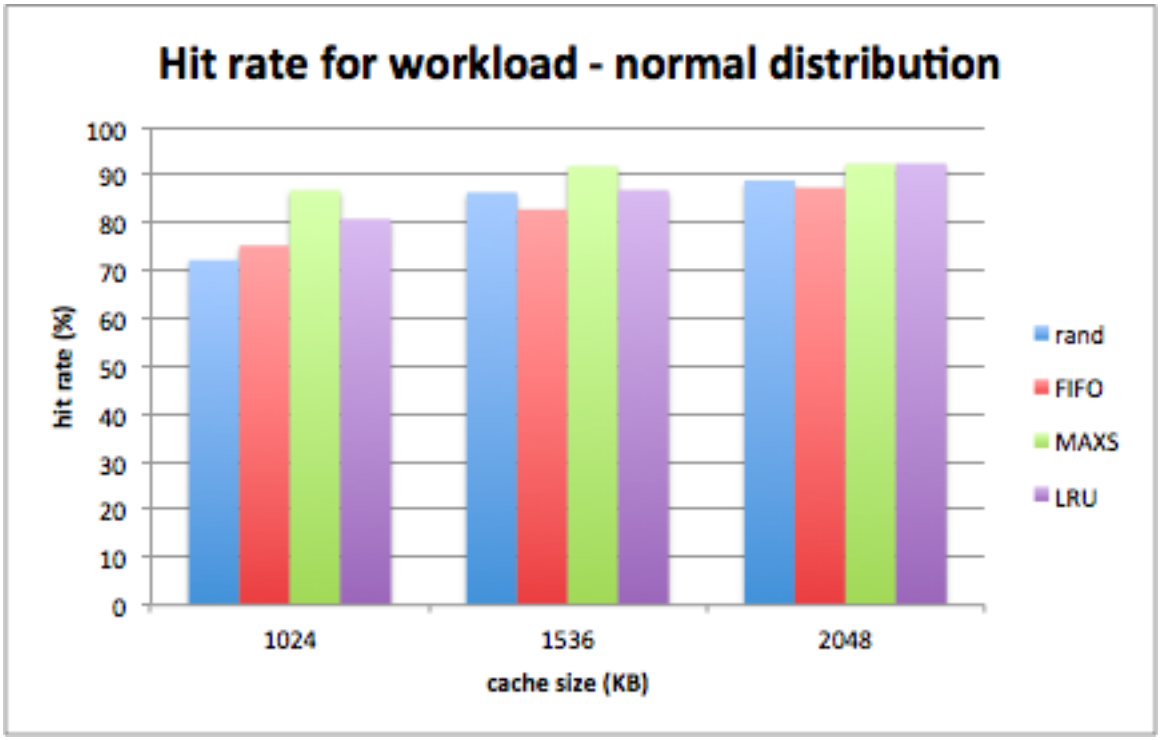


Figure 3: cache hit rate for different cache sizes (Workload 1: rand())

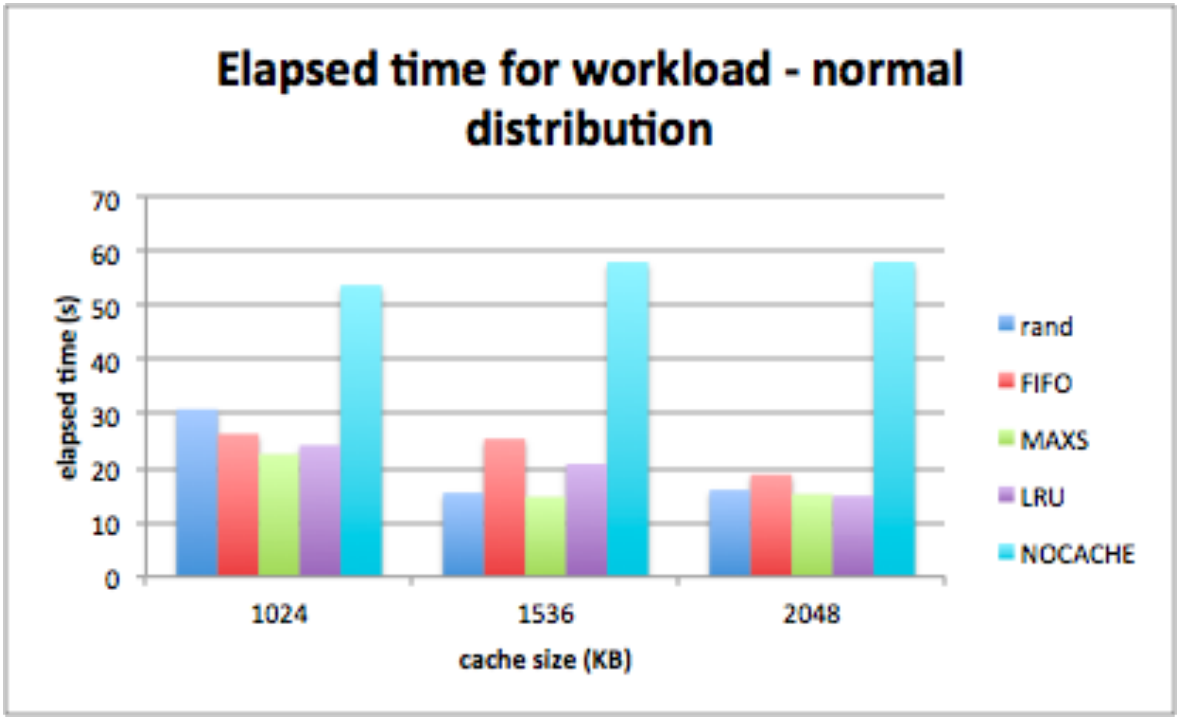


Figure 4: elapsed time for different cache sizes (Workload 1: rand())

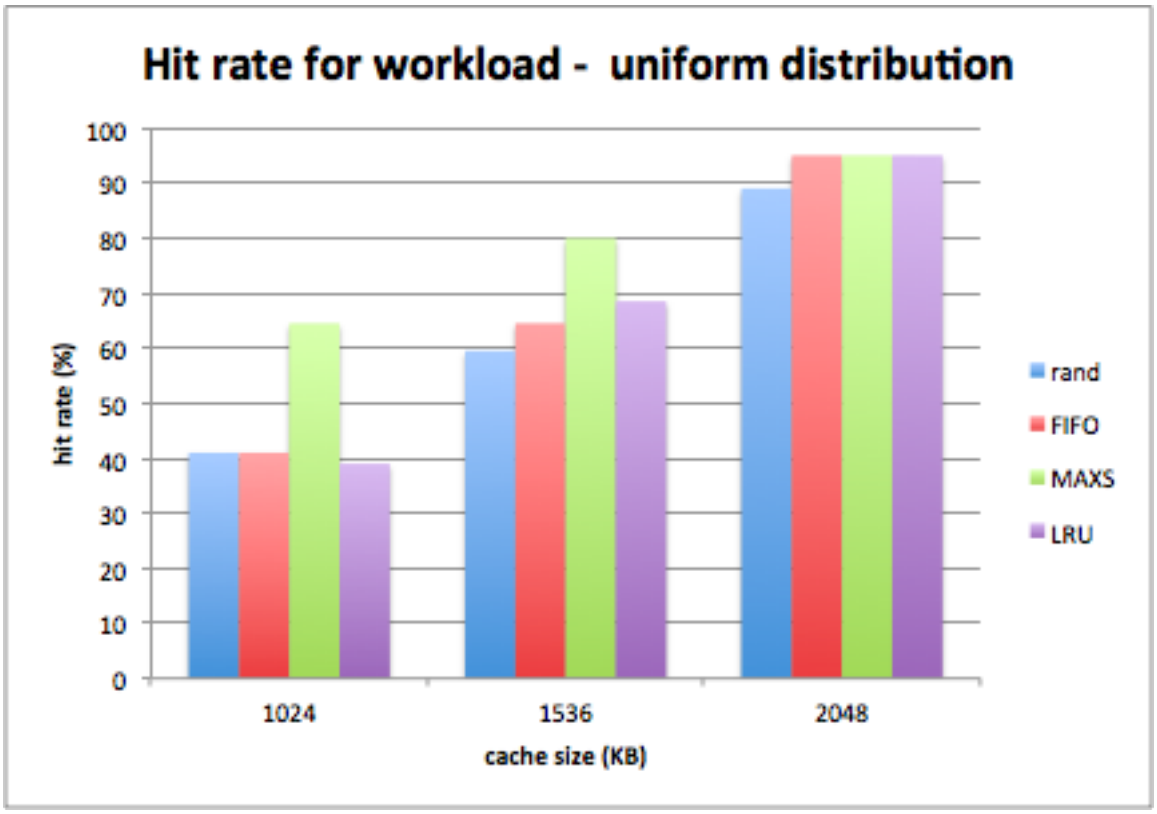

Figure 5: cache hit rate for different cache sizes (Workload 2: normal)

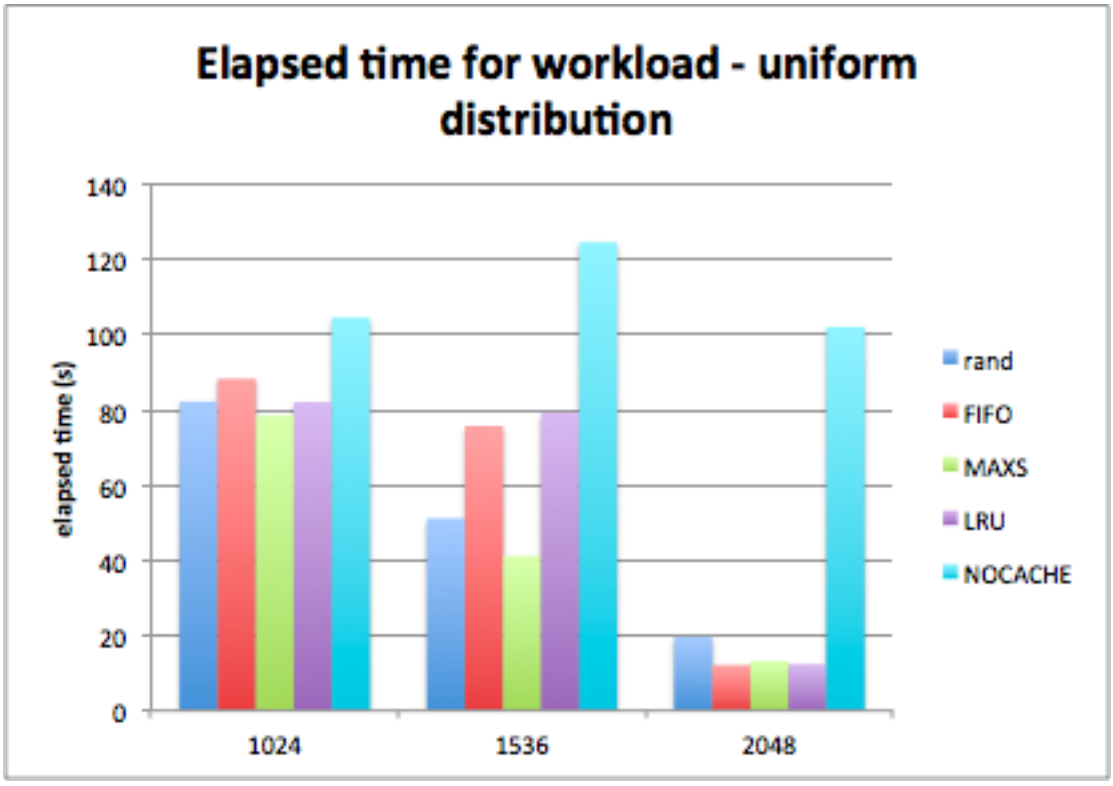

Figure 6: elapsed time for different cache sizes (Workload 2: normal)

# VI. Experimental Analysis

## Workload 1:

Workload 1 is uniformly distributed, which means that each url has identical probability to be selected. For this workload, we can find that for larger cache size (2048 KB), the hit rates are almost the same among all the cache policies (95%) except for the random policy (89%). This is because the eviction for random policy is just randomly selecting an url to evict, rather than selecting a url with preference. For the other policies, since the cache size is large enough, the cache miss only occurs when the url is requested at the first time. The observation is consistent with the results that there are 190 cache hits out of 200 cache requests for cache size is 2048 KB. Similar observation can be found in the elapsed time. The slight difference for FIFO/LRU/MAXS might be due to the network status.

On the other hand, for smaller cache size (1024 KB), it is obvious that the MAXS policy outperforms others by 20%. Note that the webpage body sizes are from 150 bytes (http://www.twitch.tv) to 750K (https://www.yahoo.com) bytes, which is a fairly large range. Therefore, for the smaller cache size, the body size of Yahoo is relatively huge. In other words, when the cache eviction happens, the MAXS policy always evicts the largest cache, which benefits the cache utilization. For the other policies, since the eviction size is not maximized, the number of cached web pages is not maximized as well, which might reduce the hit rate. Moreover, when the eviction occurs, the evict size may not be sufficient for the incoming data so that the eviction is need to be performed more than once, which leads to more cache miss. However, there is an interesting observation in the time evaluation for the small-sized cache. Although MAXS policy outperforms others, the elapsed time is not the shortest though. It is because that evicting the large-sized webpage increases the possibility of cache miss for those web pages. Moreover, the time to retrieve large-sized web pages is longer than accessing others. Therefore, the time performance is not notable for MAXS policy presented in cache and thus increase the miss rate.

The FIFO eviction is based on the create time of a cache line, while LRU eviction is performed according to the last read time. In general, the LRU should outperform FIFO in most scenarios. The exception is that when the cache size is small, the eviction scenario of both policies are similar because a cache data (web page body) is likely to be accessed only one time when created, and evicted due to the insufficient cache size. In this case, the performance of both might be similar, or even unexpected. As we can see for the small-sized cache, the hit rates for FIFO and LRU are 41% and 39%, respectively.

For the medium-sized cache (1536 KB), the trend of performance is more understandable because the size is more suitable for this workload. As we expected, for the hit rate, the performances are ordered as MAXS > LRU > FIFO > random. On the other hand, we found that the time evaluation for cache size 1536 KB and 2048 KB have different trends. It might be due to the computation complexity in eviction is higher for LRU, which leads to higher elapsed time for LRU than for FIFO.

## Workload 2:

Workload 2 is designed as a normal distribution. As we mentioned in Section IV, the large-sized url has the lower probability to be accessed, and the probability for small-sized url is higher. In our experiments, we run each policy on the proxy server with three different cache size. As we can see in Figures 5 and 6, the general trends of performance of all the policies are the same. The

performance is increasing as the cache size becomes larger, which is a reasonable result. Because larger cache size makes proxy server store more web pages, which provides higher hit rate and reduces the possibility to fetch web page from the web server directly.

Besides that, we can see that the MAXS policy always has the best performance among those four policies in different cache sizes. The reason is that the large web pages are not frequently accessed for this workload. Also, the MAXS policy evicts the large size page as its first priority. Therefore, proxy server can delete the largest size page in its cache immediately and create more space for the coming web pages. And proxy server doesn't need to spend extra time to fetch those large-size web pages, because those pages may not be accessed again in the future.

Let's look at the random, FIFO and LRU in the Figures 5 and 6. In general, they also have relatively good performance. They have higher hit rate and lower latency. The LRU has very similar performance with MAXS. We may expect that the LRU should have the best performance among those policies. However, we may miss an important factor that the cache size in our experiment is very small. In our url list, the largest web page size is approximately 700 KB, which is larger than half of small cache size (1024 KB). In LRU policy, we always keep the most recently used page. Let's consider the case that if we just access a 700KB web page and want to access another 600 KB web page. We find that LRU does not provide any help for reducing the miss rate because the large-sized page will be evicted finally; instead, it increases the cache miss rate. Now, let's move to the hit rate when cache size is 2048 KB, which is relative larger than before, we found the performance for LRU has almost same performance as MAXS, which proves that the scenario that we described before does occur in the proxy server.

In Figure 6, we evaluate the performance in terms of latency of visiting all the urls in the url list. Basically, we get the same results as we get those from hit rate. The MAXS has the best performance. The random and FIFO has relatively bad performance. However, we can still find benefits of the proxy server with cache. For the experiment without applying cache mechanism, the latency for client to visit all urls one by one will take much more time comparing with the one that has proxy server.

## VII. Conclusion

According to our experiment results, we can get several conclusions as below.

Firstly, proxy server can reduce the latency for client to access the web for sure. From Figure 6, we can see that the latency with proxy server is still better than the one with non-cache proxy server in normal distribution workload. Therefore, proxy server certainly can provide some benefits.

Secondly, the design of proxy server should be based on the characteristics of client behavior. As we can see, the FIFO and random policies has the worst performance in the two workloads. Because they are designed without considering the workload characteristics. However, in LRU and MAXS, we can improve the performance a lot by the information of the workloads. In LRU, we assume that clients always visit the recently used web pages. In MAXS, we assume that clients does not visit large-sized web pages frequently.

Thirdly, a good replacement algorithm must consider the real scenarios happening in real world. For instance, the LRU algorithm cannot work perfectly in our workloads. The reason is that we does not consider the hardware specification. We did not aware that the cache size used in experiment is relatively small comparing to the web page size.

Lastly, one of the the easiest way to improve the proxy server performance is to increase the cache size so that proxy server can store more pages to avoid the miss and eviction. However, since the memory is a precious resource, using cache replacement policies is a better approach to improve the performance.