CS 6210-A - Project 3
RPC Based Proxy Server

Team Members :

Archana Venkatesh (avenkatesh8) - GT id: 902952246
Shruti Padamata (spadamata3) - GT id: 902915013

Introduction

A proxy server is a server that acts as an intermediary for requests from clients seeking resources from other servers. Today most proxy servers are web proxies which facilitate access to content on the web and preserve anonymity. A simple method to implement a proxy server is by using the functionality provided by remote procedure calls. The client makes a remote procedure call to the proxy server. The proxy server on behalf of the client performs the HTTP request and other network handling. Often, the proxy server maintains a cached copy of the frequently requested web pages in order to reduce the response time and waiting time of the client. The goal of this project is to understand the working of a proxy server with focus on the caching policy. We have implemented three different cache replacement policies and tested the system with two different workloads.

Proxy server Design

The proxy server is implemented as an RPC server. It exports two functions :

- getURL(string url)
- getCacheStats()

These two functions are available to the client. The client requests the contents of a URL by calling the getURL function. If caching is enabled on the proxy server, it first does a local cache lookup to determine if the requested url is available in the cache. If it is, the server returns a cached copy of the webpage. If not (or if the caching is disabled), it performs a HTTP GET using the CURL library. General exception handling is provided by the CURL API. In addition to the HTTP error responses returned by the CURL call, the server implements a timeout functionality i.e it stops trying to fetch the web page after a certain time interval. This simple model was implemented over APACHE THRIFT which provides the basic primitives for implementing an RPC based client and server.

Cache Design Description

We have implemented three caching policies for the web cache:

- Random
- FIFO
- LRU-Min

All three policies require the contents of the requested url to be stored in a queue-like structure and a data structure to enable fast lookups on the queue. Thus, our implementation mainly uses two data structures:

- a doubly linked list with each node containing the url and the url contents.
- a hash table indexed by the url string and containing a pointer to the corresponding node in the linked list.

The list serves as a queue storing all the URLs that are present in the cache along with the page contents for those URLs. The list is accessed differently for each of the caching policies namely FIFO and LRU-min. The hash table is mainly used for improving the lookup time in the cache since searching in the list would take linear time. The hash table is implemented using the C++ map STL. The map enables us to achieve lookups with a time complexity of O(logN) in the number of cache entries or URLs, as the underlying data structure is an R-B Tree. The hash table stores the mapping of URLs to the nodes in the linked list.

On a URL request, the proxy first tries to get the contents from the cache in case they are already present. There is a hash table lookup to determine if the URL is present in the web cache. If it is, the contents can be retrieved from the corresponding node whose address is obtained from the hash table.

When a new url needs to be added into the web cache, a replacement policy is applied which evicts one or more urls (depending on the memory requirements of the incoming url) followed by creating a new node in the list with the incoming url and its contents and an entry into the hash table. All insertions to the list are done at the head of the list, while the removal for eviction is done from the tail, as we require queue behaviour for both FIFO and LRU-Min policies.

Caching Policies

Random

The random replacement policy is a simple replacement policy in which a url is selected at random from the entries. This web page is removed from the cache to make room for the new entry. In the case when evicting one web page is not sufficient to fit the incoming page, the above is repeated i.e another url is selected at random and evicted until there is enough memory for the incoming page. The advantages of random replacement policy is that it is not partial towards any kind of workload or access pattern. While it is not the best algorithm for replacement it is bound

to perform reasonably well for all kinds of workload.

FIFO

The FIFO (First in First Out) replacement policy considers the time the web page has been in the cache as a factor for eviction. The page which has been in the cache for the longest is chosen and evicted. As with the random replacement, if evicting one page from the cache does not free up enough memory to host the incoming page, the next in queue is evicted. This is continued until there is enough memory to store the incoming page. The main idea behind this algorithm is that old pages should not pollute the cache. But this algorithm does not consider the usage of the page. It is poor replacement policy. The advantage of FIFO is that the algorithm requires almost no bookkeeping. The deletions are performed from the tail of the list and insertions are done at the head of the list. It is an intuitive replacement algorithm but may not perform well in most practical situations.

LRU - MIN

A general LRU replacement policy is based on the idea that the most recently used pages are likely to be used again. So the access time of the pages is used as an indication of the future access times. Thus, when eviction is required, it evicts the least recently used page from the cache. But replacing one page under this policy may not free up enough space in the cache and we may need to replace multiple pages, as done in the previous policies. LRU Min policy tries to optimize the replacement by evicting larger pages first, thus freeing up more memory. It takes a parameter S and the eviction first replaces all the LRU pages that have a size greater than S, followed by the LRU pages with size greater than S/2, and progresses by halving the size so on until it frees up enough space. Thus it is likely to evict lesser number of pages to make room for the newly arrived page.

For implementing this policy, we have divided the cache (linked list) into four parts, each of which maintains the LRU order. The parameter S is taken as 250KB, slightly less than the average max page size observed, so that the intervals will have approximately equal number of pages. The four lists use the same generic list implementation that we have used in the above two policies, with one additional method to implement the movement of pages to the head on access, in order to preserve the LRU ordering. The lookup for the pages still goes through the hash-table. A list id field is added into the list entry to identify the list on a cache hit. Using this, the list entry is moved to the corresponding list head. On seeing a miss, if the free space in the cache is not sufficient to hold the incoming page, the replacement function would progressively evict the LRU pages from each list until enough space is available. For inserting a new page, the list to which it should belong is determined based on its size and it is queued into the corresponding list, along-with the addition into the hashtable.

In the research paper that first proposed this policy, it was observed to perform better than the plain LRU policy. But we could not verify this since we had not implemented LRU policy.

Experiments

Performance Metrics

The two metrics chose for the evaluation are :

- Cache hit ratio
- Average page access time as seen by the proxy client

The cache hit ratio as the name suggests is the ratio of the number of url requests serviced by retrieving the contents from the web cache to the total number of url requests. Higher the cache hit ratio better the performance. A high cache hit ratio suggests more effective caching for the workload. This ratio will differ depending on the workload access pattern and the replacement policy used.

The average access time is the average time to retrieve a given url. The request may be serviced from the local web cache or the fetched using HTTP GET. An average access time gives a good indication of the proxy server performance . This time also includes the network latency.

Workloads

A list of common websites (in the file ‘urls.txt) was taken from a public repository. Though it contains a million urls, we have used only 125 urls for the testing, in order to keep it comparable to the cache sizes used. The number of proxy requests was kept 5 times the total number of urls, i.e., 625 to allow for multiple requests for the same pages. Two kinds of distributions of the input URLs were used to evaluate the the performance:

- uniform distribution
- exponential distribution

Uniform distribution is a fair distribution where in every url is requested equal number of times. In our workload, each ulr out of the 125 urls are requested five times each. The order of requests is random and unknown. Thus the only guarantee obtained by this distribution is that each url is requested equal number of times. The access pattern will depend on the random ordering that the workload is generated in.

In case of exponential distribution, there are small number of urls which are requested more number of times than the others. It is not a fair distribution. As with uniform distribution the access pattern is unknown as the ordering of requests is random.

The distributions were obtained using the boost library functions.

Experiment Description and Expected Outcome

The experiments were performed on Georgia Tech network, with the client running on Virtual Box on one machine and the proxy server running on another machine, both using Ubuntu. The performance metrics were measured using counters and timing functions added in the code. The cache size is varied between 1MB to 2.5MB in 0.5MB intervals. Based on the observed average page size of approximately 100KB, we expected around 20-30 cache entries for a cache size of 2.5MB and thus restricted the total number of urls to 125, to get a load of around 10 times the max size of the cache.

With the uniform distribution, all three replacement policies might show approximately equal performance since the distribution does not favor any of the policies. We expect the random replacement policy to perform slightly better than the FIFO and LRU policies since the total number of pages requested is much higher than those that can fit in the cache and the queue based policies would evict the pages in a particular order while the accesses are made randomly over a fixed interval.

For the workload with exponential distribution, the LRU Min policy is expected to perform very well, followed by the FIFO policy and random policy. This is because the exponential distribution would result in a small number of urls being requested multiple times. So, the most requested urls are expected to stay in the cache longer with LRU Min. The LRU Min policy with this workload is also expected to give a higher hit rate as compared to the previous workload.

Experiment Results and Analysis

Given below are the charts showing the results of the experiments with varying cache sizes for different replacement policies.

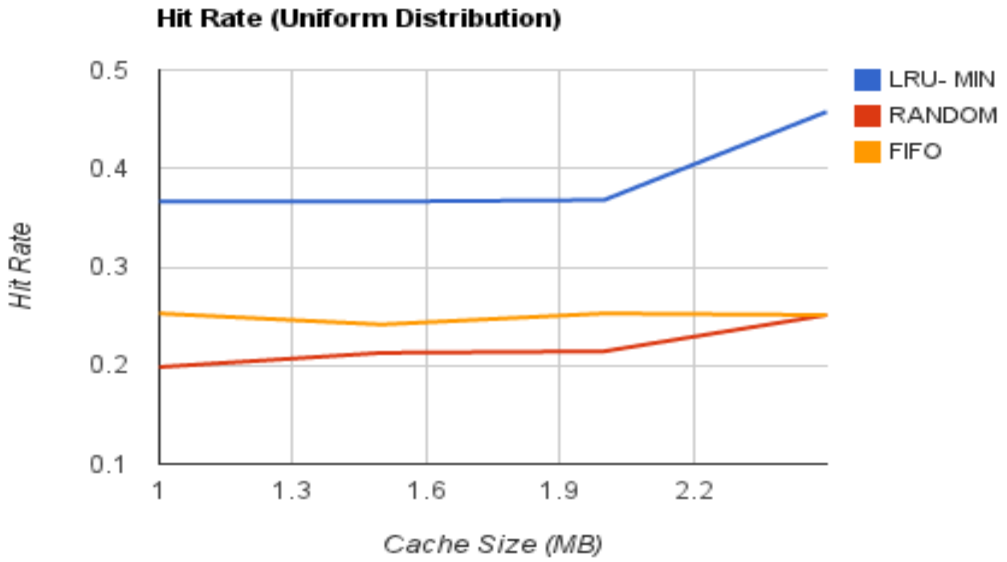

Figure 1: Hit Rate (Uniform Distribution)

Figure 1 shows the hit rate results with uniform distribution workload, for the various policies. Unlike what we expected, the hit rate with Random and FIFO policies is much lesser than that with LRU-Min. This is probably due to the eviction of larger pages by LRU Min, which allows more number of pages to fit into the cache. Our expectation was based only on the access pattern and not the size of the files. This shows that the eviction of larger pages is beneficial even for the accesses that do not show much temporal locality. Also, the hit rate for LRU Min is almost constant for small cache sizes and increases for the largest cache size we have used. This indicates that for the LRU Min case, there might be a threshold cache size at around 2MB, above which the increase is cache size shows gradual improvement in the hit rate. But this is just a speculation, since we did not perform the measurements with higher cache sizes.

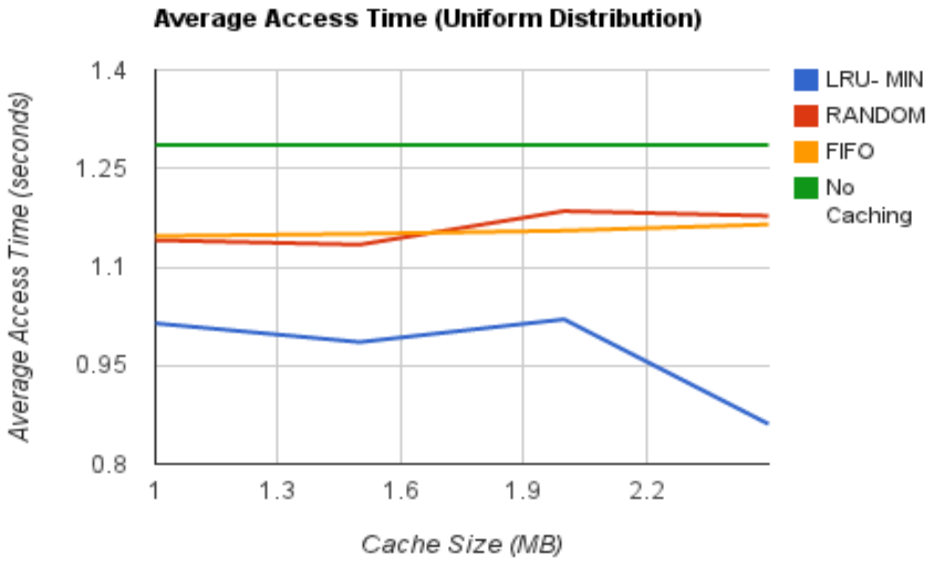

Figure 2 : Average Access Time (Uniform Distribution)

The timing measurements for the same workload are shown in Figure 2. They do not show consistent trends with the variation in cache size, probably because the experiments were performed at different time periods and the network speeds might have been different. We see that the average time taken for a page request without the cache is higher than that with any of the caching policies. This shows that the use of cache is indeed reducing the wait time at the client. Between the different replacement policies, the timing variation is in accordance with the hit rate trends seen above. LRU Min policy with the highest hit rate gives the least average access time due to more pages being served from the cache. FIFO and random policies have almost equal average access times, again due to similar hit ratios.

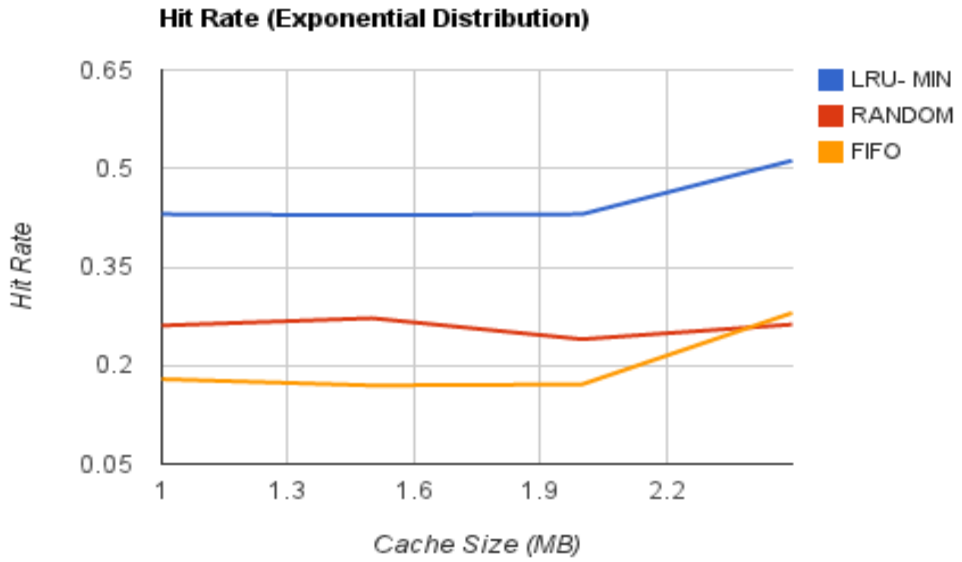

Figure 3 : Hit Rate (Exponential Distribution)

Figure 3 shows the hit rate for the exponential distribution workload. As expected LRU-MIN outperforms both the Random replacement and the FIFO replacement policy. This is because in an exponential distribution, a small number of urls are accessed more number of times than the other urls. Therefore there are higher chances of finding those pages in cache using a LRU-MIN replacement policy as the recently used pages are not evicted. For the same workload Random replacement mostly performs better than the FIFO replacement. In case of FIFO replacement policy only the time is considered for eviction. If the page has been in cache for the longest, it is evicted ; even if it has been accessed more than once. Therefore the chances of a miss are higher with this replacement policy.

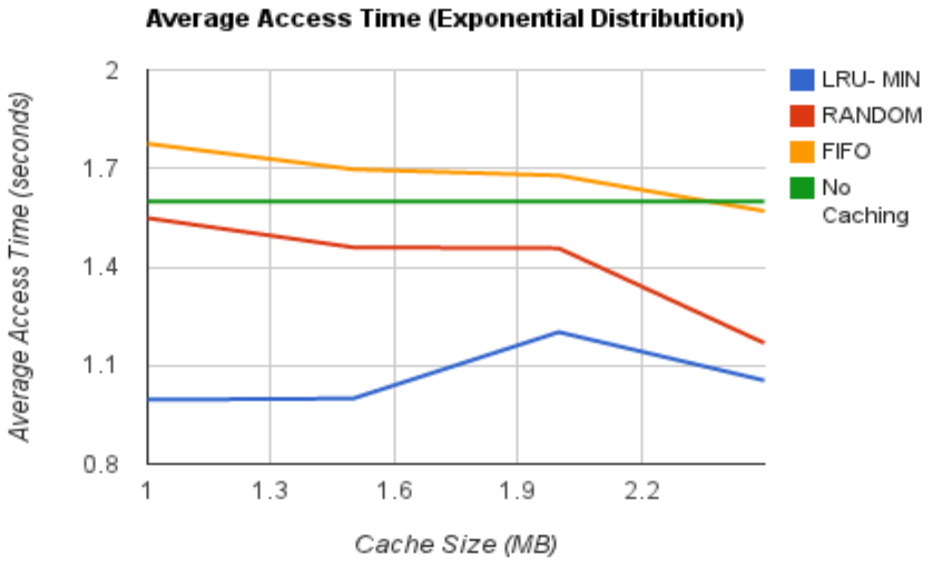

Figure 4 : Average Access Time (Exponential Distribution)

Figure 4 represents the average access time for the different replacement policies for the exponential workload. Since the LRU-MIN has the maximum hit rate, it is seen that it has the least average access time. Surprisingly the FIFO replacement has a higher access time than the access time without caching. A possible explanation for this anomaly could be the overhead introduced by the FIFO cache replacement. The miss rate of FIFO is high. Therefore most of the requests turn out to be HTTP requests. Once a page is fetched from the web, it is then added to the cache. The FIFO replacement requires eviction from the tail of the list until enough memory is freed for the incoming page. It is possible that for lower cache sizes, the cache is always full and a miss results in eviction of more than one page for replacement. If this continues to happen very often, the average access time could be significantly higher. It can be seen the time decreases for higher cache sizes. We can only speculate that the trend continues and the average access time reduces as the cache size grows.