CS 6210 Advanced Operating Systems

Project 3: Proxy Server

Qiuxiang Jin

1 Introduction

In this project, we build a proxy server with cache, and use Apache Thrift to establish the communication between the proxy server and clients. Using this proxy server, we compare the response time and hit rate of different caching policies on different workloads. We also analyze the influence of cache size on the proxy server performance. We finds that in general MAXS give us the best performance among the three caching policies we test, but this advantage is not significant when the cache size is large enough or the workloads exhibit enough temporal locality.

2 Cache Design Description

The cache in the proxy server is used to store the html documents that have been recently requested by the client. The basic assumption behind the cache is that an item requested by the user is likely to be requested again in the near future. So the cache will store item that is requested. When the item is requested again and the item is still in the cache, the proxy server can directly send the cached item to the requestor instead of going to the remote website and fetching it again, which will help reduce the response time.

2.1 Caching Policies Description

When the cache runs out of space and a new request for document not in the cache comes, the cache must evict some old documents and make room for the new document. Caching policies are about how to choose the next item to evict. There are many caching policies available. For this project, we use and compare three kinds of caching policies, namely Random policy, First In First Out (FIFO) policy, and Largest Size First (MAXS) policy.

2.1.1 Random

The Random policy evicts a document at random. This policy is easy to implement because it doesn't need to record any state information about the content in the cache (e.g. time stamp, size). However, since it doesn't utilize any information from requests, it might not fully exploit cache, especially when there is some apparent pattern in the requests.

2.1.2 First In First Out (FIFO)

The First In First Out policy evicts the document that has been in the cache for the longest time. The assumption of behind this policy is that the earlier an item enters the cache, the less likely it will be requested in the future. However, this assumption normally doesn't hold in reality, especially when an early-coming item is requested over and over again.

2.1.3 Largest Size First (MAXS)

The Largest Size First policy will evict the document that has the largest size. This policy is unique to website content caching when compares to CPU cache, because different items can have different size for content caching, while for CPU cache the size of basic item (cache block) is fixed. By using this policy, the proxy can minimize the number of old items to evict for a newly coming item, especially when the new item is very large.

2.2 Implementation Details

We use a binary search tree as the main data structure for the cache of the proxy server. Each node in the tree represent an item in the cache, which contains a key-value pair, where the key is the URL and the value is the content of the URL. An item is inserted into the binary search tree according to the string order of the URL. The binary search tree is always rebalancing itself, so the binary search tree is always balanced, and the insertion complexity is O (N*log N). When a new URL request comes, the proxy server will search for the URL in the binary search tree. The time complexity of this search is also O (N*log N). We use the Map data structure in C++ to implement the balanced binary search tree.

For FIFO policy, an additional queue is used to keep track of the relative order of eviction of each item. When item is added into the cache, its key (URL) will also be put into the tail of queue. Upon eviction, the proxy server will choose the URL on the head of the queue and remove it from the cache as well as the queue.

For MAXS policy, an additional binary search tree is used to keep track of the size information of each item. Each node of the binary search tree is a key-value pair, where the key is size of a particular document and the value is the URL of that document. When a new item is put into the cache, its size-URL pair will be inserted into this binary search tree according to the size. Upon eviction, the URL on the rightmost of the binary search tree (i.e. the one with the largest size) will be removed from the cache as well as this binary search tree.

For simplicity, we don't consider the timeout or canonicalization of URLs.

3 Evaluation method

3.1 Metrics for Evaluation

We use two metrics, namely average response time and cache hit rate to evaluate the performance of the proxy server with different caching policies.

3.1.1 Average response time

Average response time is the average elapsed time between the end of client request and arrival of the response. It's a user-centric metric, and can comprehensively reflect the performance of a proxy server. In addition to the caching policies, many other factors can influence the average response time, such as the network condition between the client and the proxy server, the network condition between the proxy server and the website server, the size of the request content.

3.1.2 Cache hit rate

When an item request by a client is already in the cache, we call it a cache hit. Cache hit rate is the percentage of cache hits to the total number of requests. Cache hit accurately reflects the performance of caching policies since other factors such as network conditions will not influence this metric. Higher cache hit rate indicates a better caching policy.

3.2 Workloads Description

Two kinds of synthetic workloads, random workload and repeated workload are used to evaluate the performance.

3.2.1 Random workload

Random workload generator randomly selects an item from a list of URLs each time, and makes the URL request to the proxy server. This random process is repeated many times, generating a sequence of random URL requests.

3.2.2 Repeated workload

Repeated workload assumes that a user will only request for a small number websites in a certain period. This workload divides the whole URL list into many small sublists with the same number of URLs. Instead of randomly selecting a URL from the entire URL list, the workload generator will pick a sublist, and make the requests randomly in this sub-list for a certain number of times. After that, another sublist is picked at random and so on. The number of requests made in each sublist is several times larger than the number of URLs in each sublist.

4 Experiment and analysis

4.1 Experiment platform description

We develop the proxy server and client program in C++, and use Apache Thrift to establish the communication between the client and server.

We deploy both the server and client on two different Amazon AWS EC2 t2.small instances, both of which are located in Amazon's data center in Virginia. Because the server and client are in the same data center, it's much faster for client to get the cached result from the proxy server than go to the original web server for the content. The operating system version of both server and client is Ubuntu Server 14.04 LTS.

4.2 Experiment method description

On the client side, we use the client to generate two kinds of workloads discussed in the section 3.2. We choose the main pages of Alexa top 50 websites as our URL list. The random workload generator will randomly select one URL from the list each time, and generate 450 requests in total. For repeated workload generator, it will divide the list in to 10 sublists, and each sublist contains 5 URLs. The repeated workload generator randomly chooses a sublist each time, and keeps making requests in that sublist for 15 times at random. And repeated generator will repeat the sublist selection process for 30 times, so it will also generate 450 requests in total (15×30). A new request will be made only after the client has received the response of the previous request.

One the server side, we use four kinds of caching policies (Random, FIFO, MAXS, no cache). And for policies with cache, we also vary the cache size from 1024 KB to 8192 KB to observe the influence of cache size on the proxy server performance. We assume the MAXS will have better performance than FIFO and Random with random workload when the cache size small, since it can minimize the number of evictions and thus maximize the number of URLs in the cache when the cache size is limited.

We measure the total time it takes to receive the response of the 450 requests on the client side and calculate the average as the average response time. We also count the number of cache hit of the 450 requests on the server side and use it to calculate the hit rate.

4.3 Experimental Results and Analysis

Figure 1 and Figure 2 show the average response time of different caching policies with random workload and repeated workload respectively. And Figure 3 and Figure 4 show the hit rates of different caching policies with random workload and repeated workload respectively. The average response time for proxy server with no cache is also plotted on Figure 1 as reference.

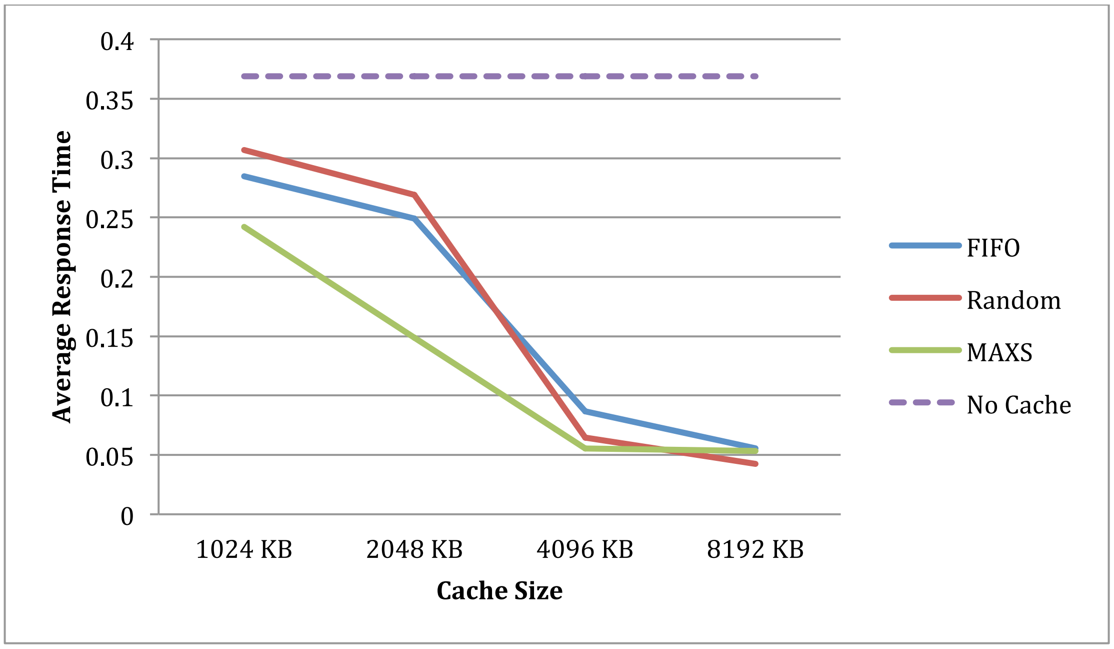

Figure 1: Average response time of different caching policies with random workload

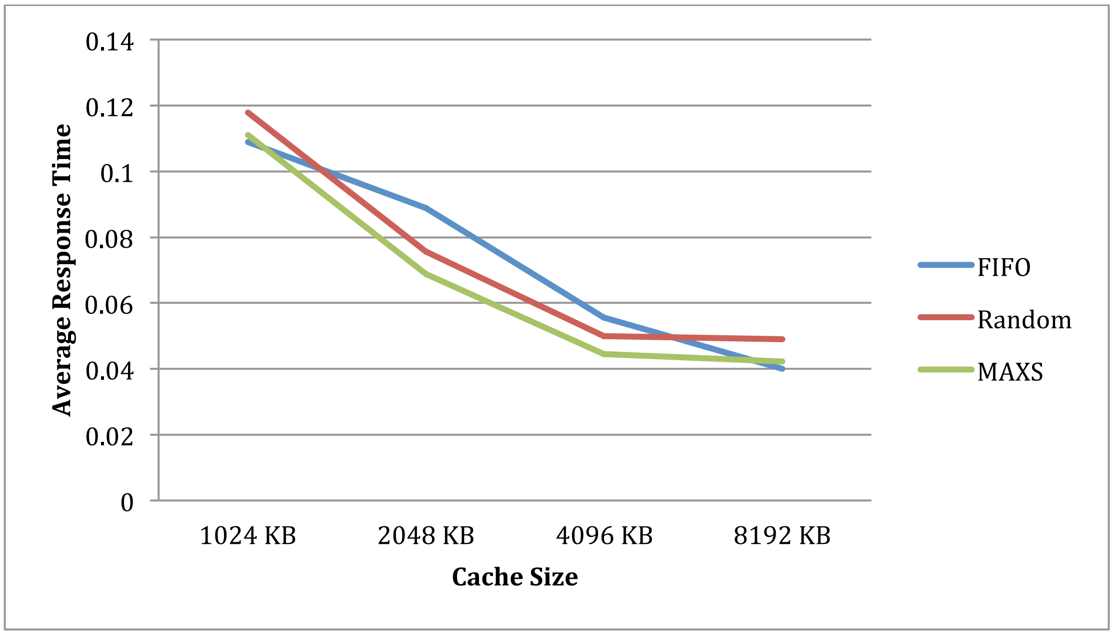

Figure 2: Average response time of different caching policies with repeated workload

As we can see from the result, the existence of cache significantly reduces the average response time. And the larger the cache size, the smaller is the average response time and the larger is the hit rate.

However, the gain obtained from larger cache decreases as the large size increases, especially for the MAXS policy in Figure 1 and Random policy in Figure 2, where the average response times remain nearly unchanged as the cache size goes from 4096 KB to 8192 KB, and for the Random and MAXS policies in Figure 4, where the hit rates stop increasing as the cache size goes from 4096 KB to 8192 KB. The reason for this trend is that the number of capacity misses decreases as the cache size goes up, while the number of compulsory misses remains unchanged. When the cache size is large enough, the capacity misses is nearly eliminated and compulsory misses will dominate the result. As shown in Figure 3 and Figure 4, the hit rates of three caching policies in both kinds of workloads reach to 0.889, which is highest hit rate they can achieve in theory.

Among the FIFO, Random and MAXS policies, the MAXS policy in general gives the best performance, especially when the cache is not big enough. That's mainly because the MAXS policy can minimize the number of items to be evicted from the cache, therefore more items will be able to stay in the cache. However, as the cache size increases, MAXS policy's advantage over FIFO and Random policies becomes insignificant due to the dominance of compulsory misses. For FIFO and Random policies, in general their performance is very similar. FIFO has slightly better performance when cache size is relatively small, while the Random policy outperform FIFO policy a little bit when the cache size is relatively large. That might be caused by the fact that the assumption behind FIFO only holds when the queue is short.

Moreover, all of three caching policies have much better performance on the repeated workload than on the random workload. The reason is that repeated workload has smaller request footprint for a certain period of time and is thus more close to the assumption behind cache. Furthermore, as we compare all the four figures, MAXS policy's performance advantage over FIFO and Random policies is much more significant on random workload than repeated workload, because even inferior caching policies will have relatively good performance when the workload is "cache friendly".

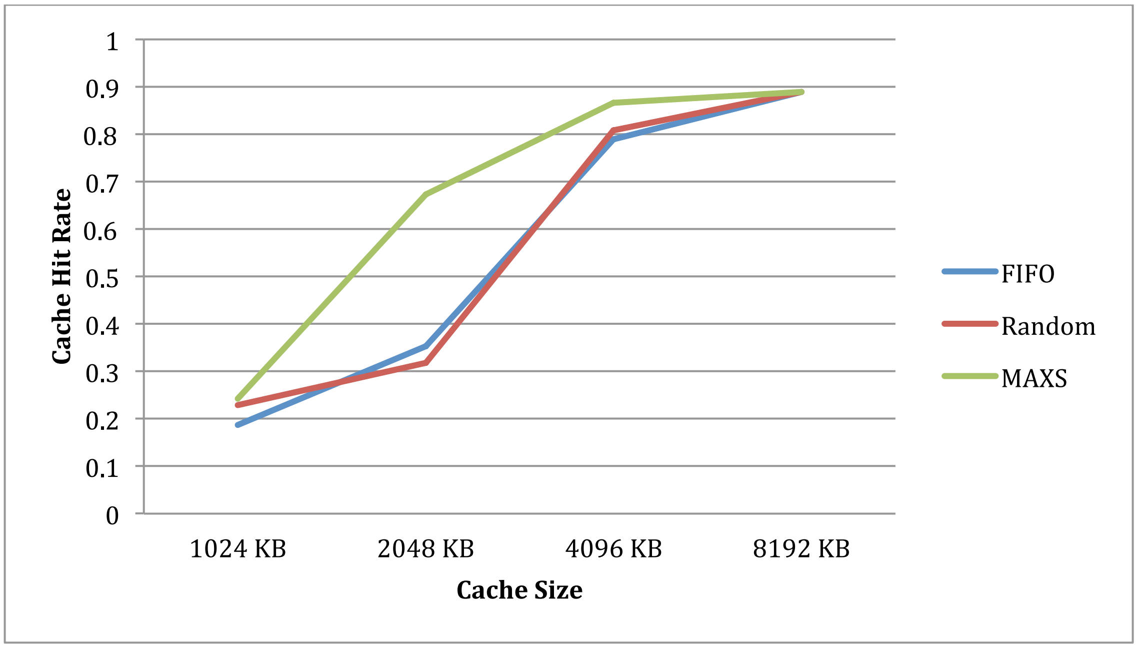

Figure 3: Hit Rate of different caching policies with random workload

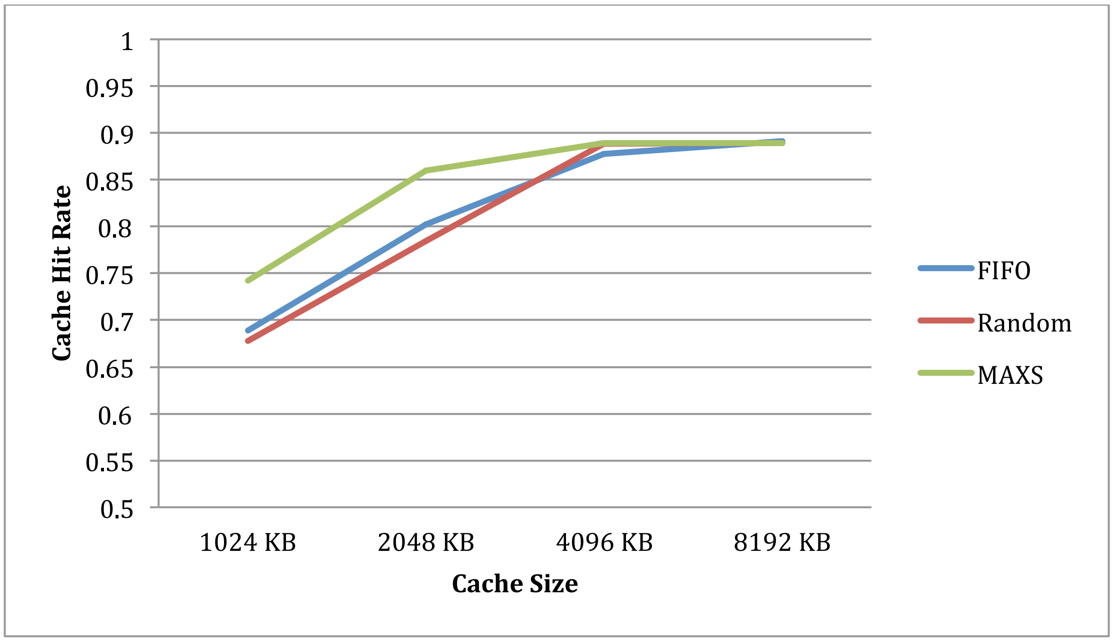

Figure 4: Hit Rate of different caching policies with repeated workload

5. Conclusion

In this project, we develop and deploy a proxy server and its clients. We compare the performance of different caching policies with different cache sizes using the proxy server and clients. The experiment results show that in general MAXS policy gives better performance than FIFO or Random polices. However, this advantage becomes insignificant when the cache size is relatively large or the workload has enough temporal locality, because in this situation the compulsory misses rather than the caching policies become the dominating factor in cache performance.