Project 3: RPC-Based Proxy Server

Sandeep Manchem *[902754732]*
Sunayana.G *GTID: [902976672]*
GitHub Repo: https://github.com/smanchem/RPC-ProxyServer

I. INTRODUCTION

Every time the content of a website is requested by a machine using a URL, the data is fetched from the web. In this project, we built a proxy server that caches data obtained from fetching it from the web and is sent to the client if the same URL is accessed again, so that access time is reduced. Apache Thrift and libcurl were used to implement the proxy server and client. In addition to this, the caching mechanisms Random, First in First out (FIFO) and Largest Size first (MAXS) have been implemented. The performance of the client-server model with cache and without cache has been evaluated. This is done by measuring the two metrics Access time, Hit rate and Byte Hit Rate. Also, three different workloads of 18 distinct URLs that follow Random, Uniform and Normal distribution that together generate 100 requests have been used. The experiments have been performed for varying cache sizes.

II. CACHE DESIGN DESCRIPTION

The Client send the URL as a String and the Server send the data corresponding to that URL in the form of a String. So the Thrift code is very straight forward as follows:

```
service ProxyServer {
      i32 ping(),
      string echo(1:string str)
}
```

Data Structures

The data corresponding to each web page (URL) is maintained in a **Page Object** with the following information:

```
string data;
string url;
int size_of_page;
```

A **Vector of Pages** contains the **Cached Data**. We used to a Vector as it is a dynamic array and each individual page can be retrieved using its index.

A **Map** of Index and URL is maintained.

A **Queue** of Indices of the Vector (integer) is maintained and used in the FIFO Caching Scheme.

A **Priority Queue** of a Struct called Size_Index is maintained and the comparison is based on the size of the page, so that the page with the largest size is at the top and can be replaced for the MAXS Caching Scheme.

```
typedef struct page_size_index {
    int page_size;
    int index;  // Index in the Vector of Pages
} size_index;
```

Algorithm

1. Search the Map using URL to find appropriate Index in the Vector
    a. If found (Cache Hit), retrieve the Data of the Page stored at that location in the Vector.
    b. If not found (Cache Miss), fetch the data from web using the libcurl, save the data as Page and add it to the Vector of Pages if there is space.
        i. If enough space is not there, find which pages to remove from Cache using the page replacement policy being used.
        ii. If RANDOM, then randomly pick and index and remove corresponding page from the Vector. If enough space is not created by that, repeat process till enough space exists for the new page.
        iii. If FIFO, remove the top most entry in Queue and using that index remove corresponding page from the Vector. If enough space is not created by that, repeat process till enough space exists for the new page.
        iv. If MAXS, remove the top most entry in Priority Queue and using that index remove corresponding page from the Vector. If enough space is not created by that, repeat process till enough space exists for the new page.
    c. After adding new Page to the Cache, return data to client.

III. CACHING POLICIES DESCRIPTION

A cache is used to store some of the data returned by URLs in a proxy server so that when the client sends a request, the data can be obtained from the cache (if present) instead of fetching from the web-server. The caching policies decide which URLs' data will be stored in a cache. When a cache reaches its maximum capacity, some of the data has to be evicted to accommodate data from (relatively) newer requests. For every new insertion, it is checked if the remaining space in cache, if any, is sufficient for the new incoming block. If not, one or more of the blocks are evicted till the new block can be accommodated (provided the fetched content is not larger than the cache). The three policies implemented in this project are:

A) Random Replacement Policy

In this policy, when a new element does not fit into cache, a random element is chosen and evicted. If the space created in cache is enough for the new block, it is inserted. If not, random eviction is continued till there is enough space in the cache. The random function should be seeded with time so that the URLs requested by the client are as random as possible.

The advantage of this policy is that it does not require any metadata and may perform better than other policies for random URL requests. This policy however does not take into account any other properties of elements like temporal locality, frequency of accesses while eviction. For example, it might evict an element that is most frequently accessed or one that is likely to be accessed again in the near future. Since URL requests in real life are very rarely random, such a policy would not be of great benefit to the cache.

B) First in First Out (FIFO)

In the First in First Out policy, the cache is analogous to a queue in which a new element is inserted at the end of the queue and deletion is always done from the head of the queue. This policy assumes that an element that was most recently cached is more likely to be reused in the near future than an element that was cached before it.

When the client sends a request and if it is present in the cache, the content is sent right away. If however, it is not present in cache, the content is fetched from the web server and sent to the client. It should also be stored in the cache provided the content is smaller than or equal to the cache capacity. If the cache has enough space to store the fetched content (i.e. evictions are not required), it is added to the end of the queue. However, if evictions are required, they are made from the head of the queue until there is enough space for the new content to be stored.

The FIFO caching mechanism is simple to implement and is also faster. Adding an element to the cache and eviction each happen as O(1) operations. In general, a user who accesses a website is likely to access it again in the near future. A proxy cache that uses FIFO mechanism will try to preserve the recently cached elements and will thus help increase the hit rate.

FIFO does not keep track of the number of times a given element is accessed. An element that is used very frequently will get evicted if it is at the head of the queue and may result in a drop in hit rate. This however, happens only once every 'n' times, n being the number of URLs cached. If the number of elements in the cache is very less either due to a small cache or large chunks of content fetched for each request, it may lead to thrashing and severely affect the performance.

C) Largest Size First (MAXS)

The MAXS algorithm evicts the largest block that is currently cached. This assumes that URLs that pull up huge chunks of data are less likely to be accessed by users than others.

When the client sends a new request, if the request can be serviced with data cached in proxy server, the webserver need not be contacted. If not, the request is forwarded to the web server and data is sent to the client. If the content is large enough to fit into cache and if there is enough free space, it inserted into the cache. If the cache is full, the block with largest data size is evicted from cache. These evictions are repeated till enough room can be made in the cache for the new data and it is then inserted.

Since the MAXS algorithm evicts the largest block, the total number of evictions decreases and hence the contents of the cache are preserved. This increases the hit rate. MAXS is simple to implement and does not require any extra storage except a pointer to the largest block in cache.

This algorithm will perform poorly if URLs that fetch large amounts of data are accessed frequently. When the largest block is evicted and accessed again shortly after the eviction, a larger number of smaller sized blocks will be evicted to make room for the large block. This destroys temporal locality, is unfair to the large block and may also cause thrashing.

Also, before every eviction, the largest block should be found which has a worst case complexity of O(n). This can be avoided by simply checking if the incoming block is larger than the current maximum and updating the pointer if required.

Even if the assumption that requests for URLs that fetch large chunks of data are sent rarely is true, MAXS may work in a manner similar to the random eviction policy. When there are large gaps between accesses to such URLs, the cache will contain data from URLs of similar sizes. MAXS when used will simply evict the largest block even though its size may be comparable to the average size of all the blocks in the cache. Depending on the kind of workload on a client, the MAXS algorithm will behave either like a random replacement policy or result in unfair eviction.

The MAXS algorithm should be avoided for small caches.

IV. METRICS FOR EVALUATION

Three metrics have been chosen for evaluation with varying cache sizes. The three caching mechanisms have been evaluated based on the measurements of hit rate, byte hit rate and access times for different workloads.

A) Hit rate

When the client sends a request, if it was serviced by the proxy server by fetching data from its cache, it is considered a "cache hit". If the contents were absent in cache and the webserver had to be contacted, it is called a "cache miss". The total number of requests sent by the client will be the sum of these two parameters.

To calculate the hit rate, the number of hits and the total number of requests sent by the client in a given workload have to be recorded. The hit rate is simply the fraction of requests sent by the client that were hits in the proxy server.

A high hit rate can either mean that the cache is large enough to store all of the requests or that the eviction policy is highly accurate and evicts only those blocks that are not used in the future. Experiments are therefore performed for varying size of cache and the hit rate for all the three policies have has been recorded so that and compared.

B) Byte hit rate

The number of cache hits and misses needs to be calculated in order to calculate the total byte hit rate. In addition to these, the number of bytes transferred in each transaction also has to be recorded.

Byte hit rate is a ratio of the total volume of hits to the total volume of all transactions. Volume here refers to the number of bits transferred in each transaction. The byte hit rate is a measure of the bandwidth saved by the cache. An increase in hit rate may not always translate to an increase in the byte hit rate.

While the hit rate is an indication of the accuracy of each caching mechanism, the byte hit rate is a measure of the bandwidth saved. Optimization of both of these parameters is required in order to improve performance. A caching algorithm that considers the size of a block before evicting it will optimize the byte hit rate. MAXS is one such algorithm that tries to optimize both the number of hits and byte hits.

C) Average access time

The main aim of using a caching policy is to reduce the average access time of the content on a cache. The thread that made the request in a client is idle till the content is provided to it by the proxy server either from its cache or from the web server. The access time is the time for which the client is idle while it waits for its requested content.

The access time increases if the content has to be fetched from the server as the RTT increases. The proxy cache by implementing a caching mechanism aims to reduces this average access time by increasing the hit rate. Since the path traveled in the network between the client and the proxy server is shorter than the path to the web server, the access time decreases with increase in hit rate and byte hit rate.

V. WORKLOAD DESCRIPTION

While conducting experiments for the three cache policies, we used three different workloads. Three of these workloads have the same set of URLs but follow different distributions (Random, Normal and Uniform). The content fetched by each of the URLs was recorded. A balanced

workload was generated such that there are URLs that fetch large, medium and small amounts of data (relative to each other). Each of those workloads are described below:

A) Workload_Normal

This is a workload containing 100 entries of 18 different URLs. Each entry is fed one by one to the client which then makes the request. A normal distribution is generated with a mean of 10.0 and a standard deviation of 6.0. Each of the URLs are accessed in random. However, the frequency of access of each of the URLs follows a normal distribution.

A random number generator of type "normal_distribution" is created and a normal distribution is generated in C++. 100 numbers are generated by this generator in the range [0,17] and each of them mapped to a unique URL. In this way, a workload that follows Normal distribution containing 100 entries composed of 18 unique URLs is created.

B) Workload_Random

This workload is similar to Workload_Normal except that the URLs follow the random distribution. A random number generator that follows random distribution is used in a similar C++ code that generates numbers in the range [0,17]. A URL is mapped to each of these numbers and a workload that follows Random distribution is generated.

C) Workload_Uniform

This workload follows a Uniform distribution. A random number generator of type "uniform_distribution" is used and a sequence of numbers with upper bound as 17 and lower bound as 0 are generated. A URL is mapped to each of these numbers and a workload that follows Uniform distribution is generated.

VI. EXPERIMENTAL DESCRIPTION

Experiments were conducted on the above workloads for varying cache size. Cache sizes were varied from 250KB to 4MB. The experiments were run using two different machines both running Linux (Ubuntu 13.*), one running the Proxy Server and the other running the Client that sends URLs to the server and receives the content hosted at that web address.

Network

The network used to run the experiments was **GaTech WLAN** connected through GTWifi. The experiments were run in the Library during the day, so as to be a good representative of normal network traffic conditions.

Measurements

The Client side code was modified to measure the time between the URL between sent and Data being received, to give the Access Time.

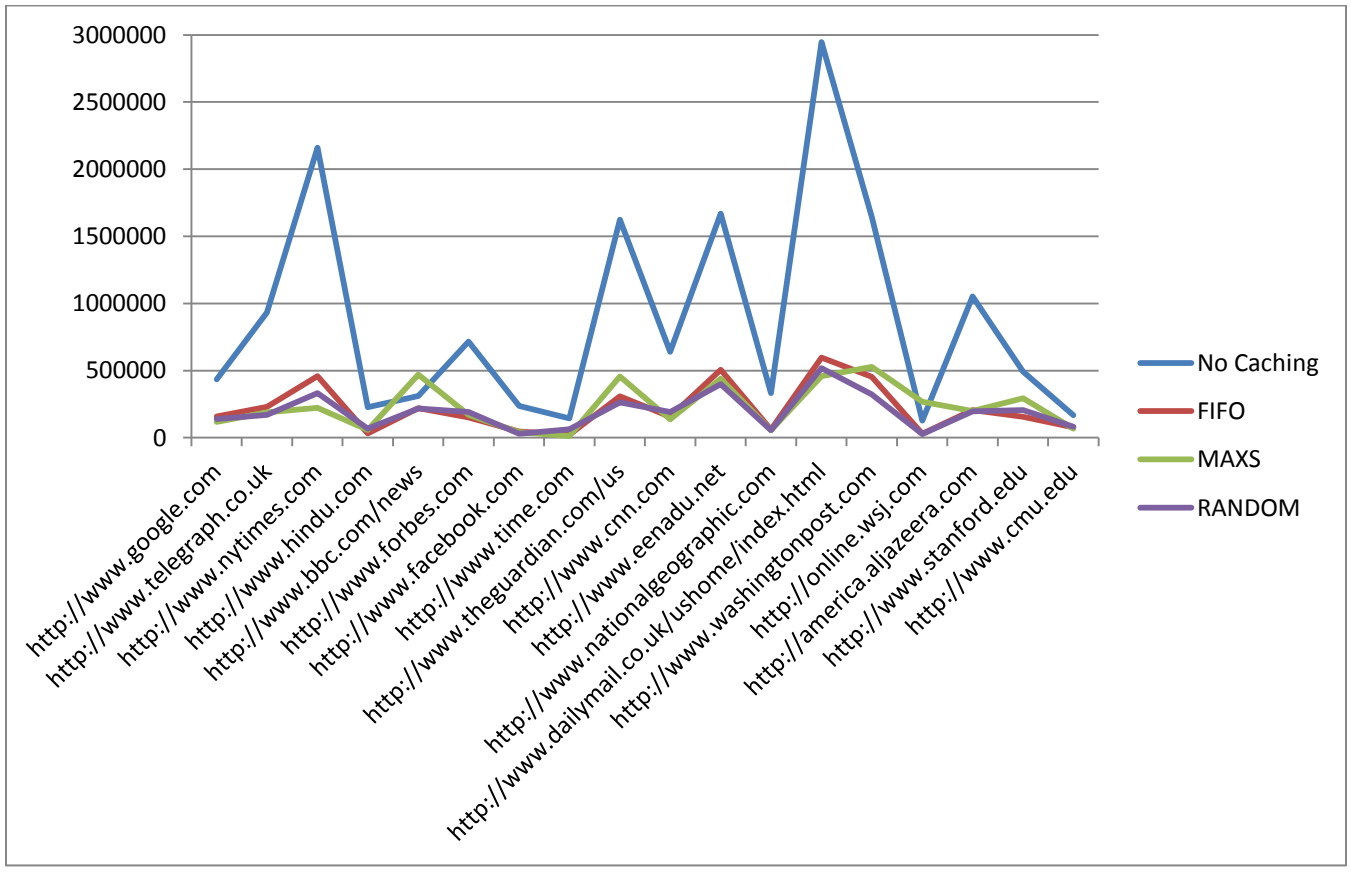
The Server side code was modified to measure "Cache Hits and Misses" and "Total Bytes retrieved and sent using the Cache" and "Total Bytes retrieved and sent by fetching from the Web"

Expectations

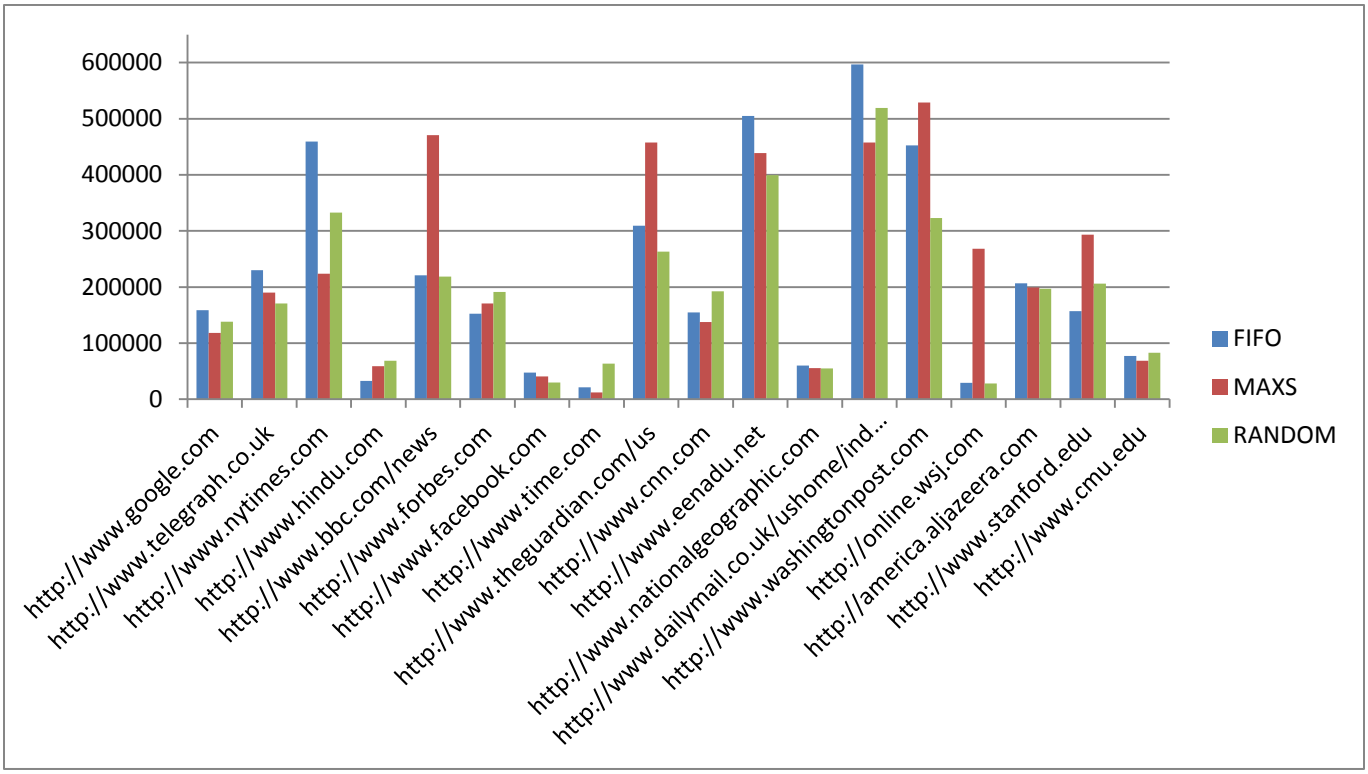
We expect that each Caching Scheme will provide different Hit Rates based on the distribution of the workload used. MAXS is expected to do well when there are fewer Large-Size requests (large webpage requests) and more of Small-Size requests. FIFO is expected to do well when recent URLs are requested more frequently than the older URLs. Random Page Replacement Policy is expected to give an average performance irrespective of the distribution.

VII. EXPERIMENTAL RESULTS

The following graph (1) shows the Access Times in micro-seconds for different Caching Schemes using Normal Distribution with 750KB Cache Size for each URL used.

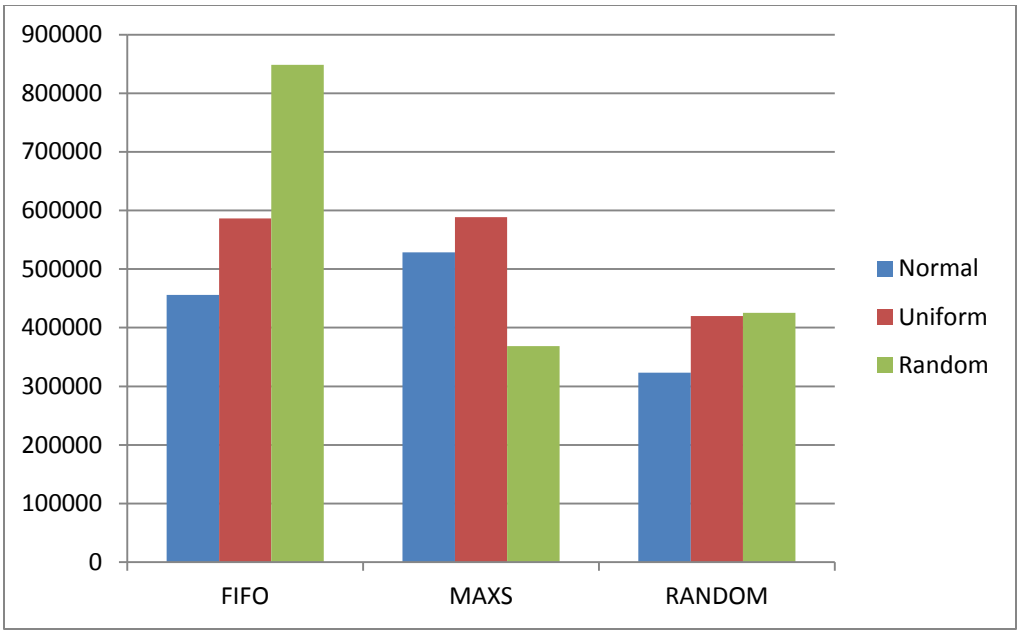


1. Access Times in micro-seconds with Normal Distribution and 750KB Cache Size

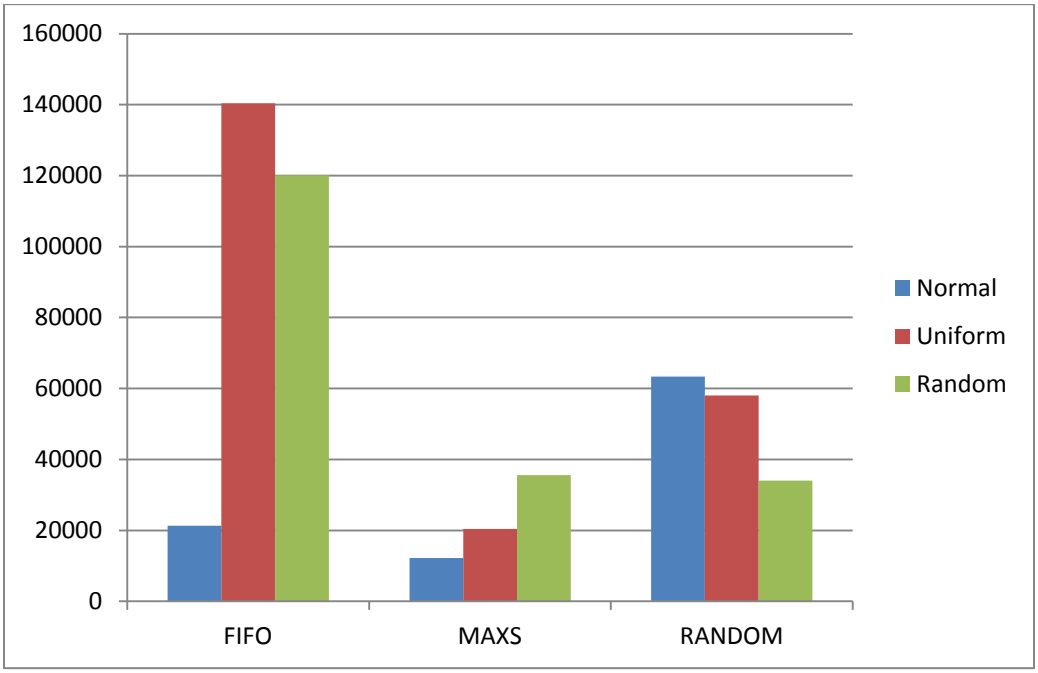


2. Access Times in micro-seconds with Normal Distribution and 750KB Cache Size

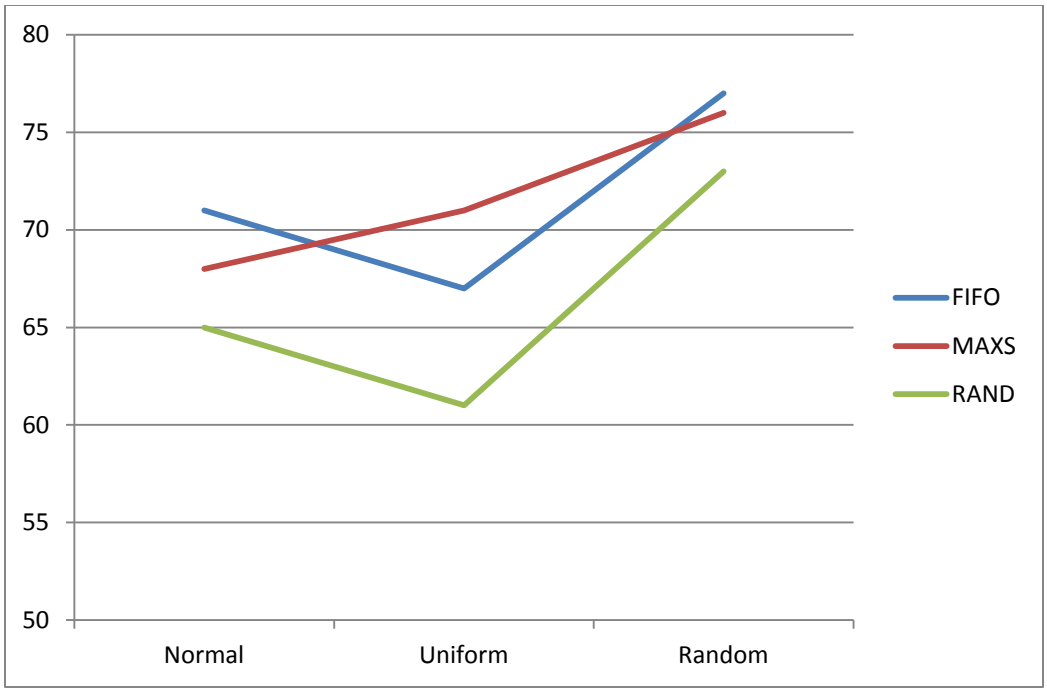
The below graphs show the difference in Access Times (in micro-seconds) for different workload distributions for each Cache Replacement Policy and for two different web pages, one large and one small with 750KB Cache Size.



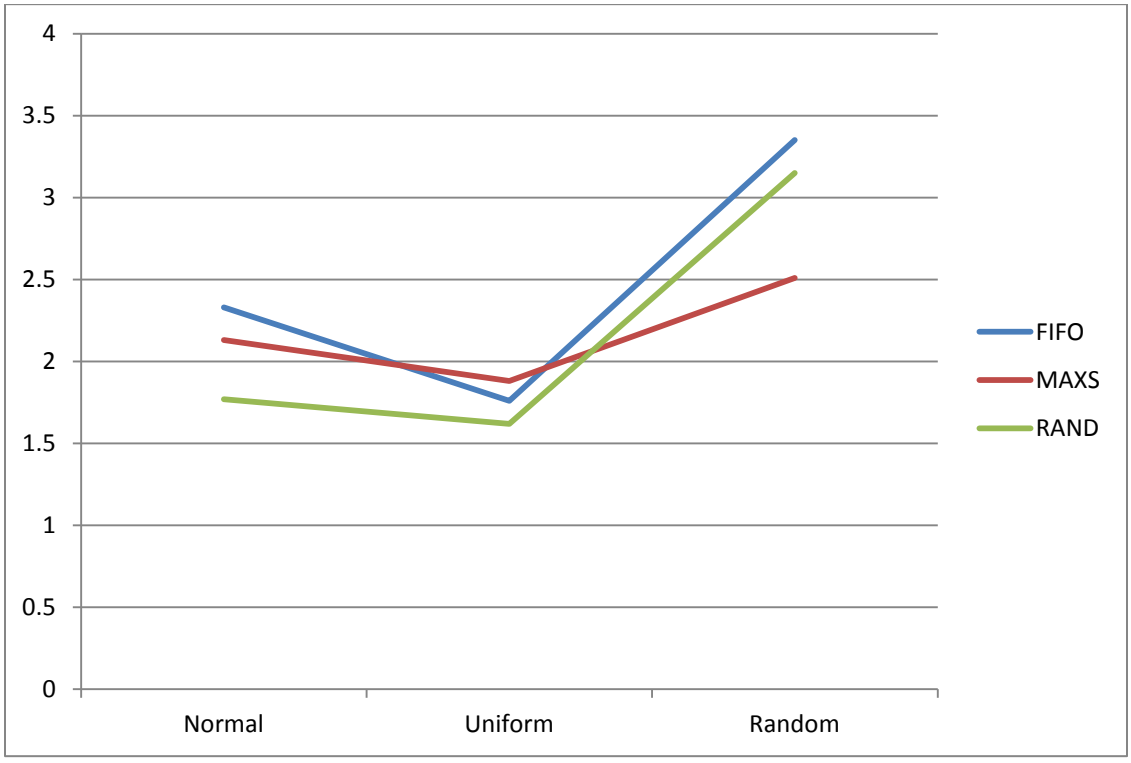
3. Access Time in micro-seconds for WashingtonPost.com



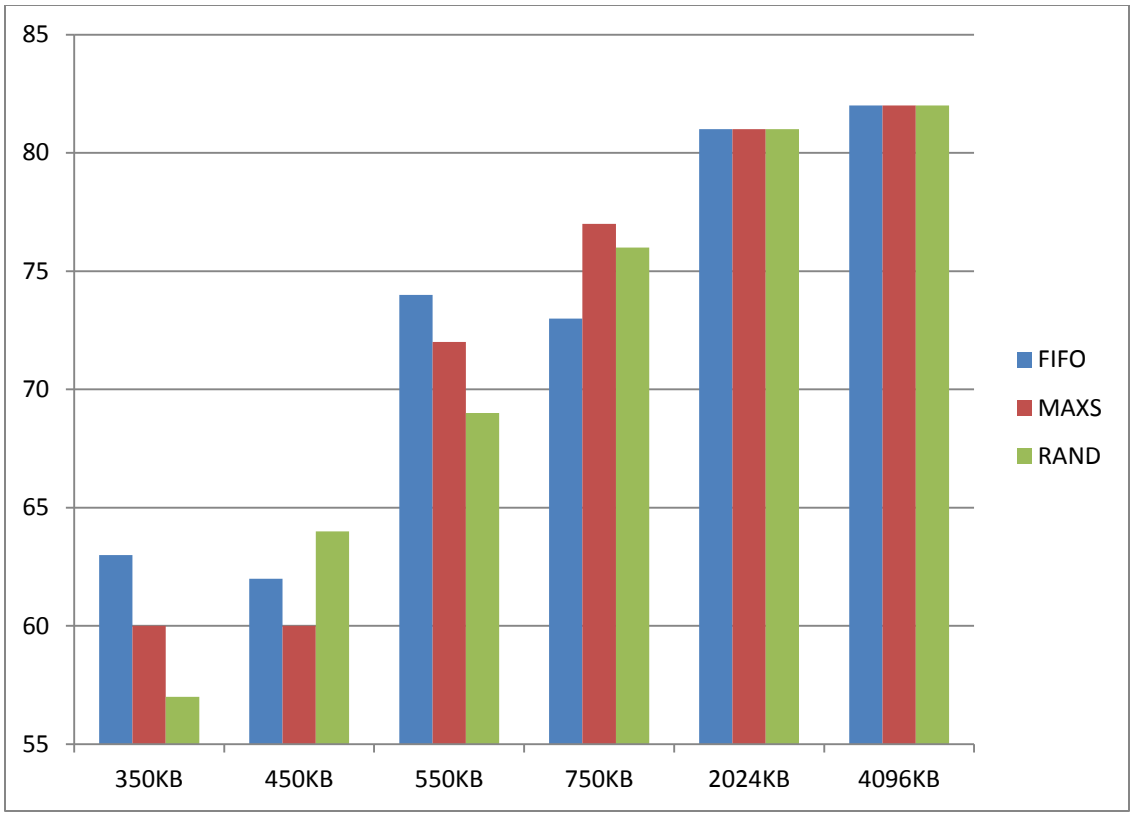
4. Access Time in micro-seconds Time.com



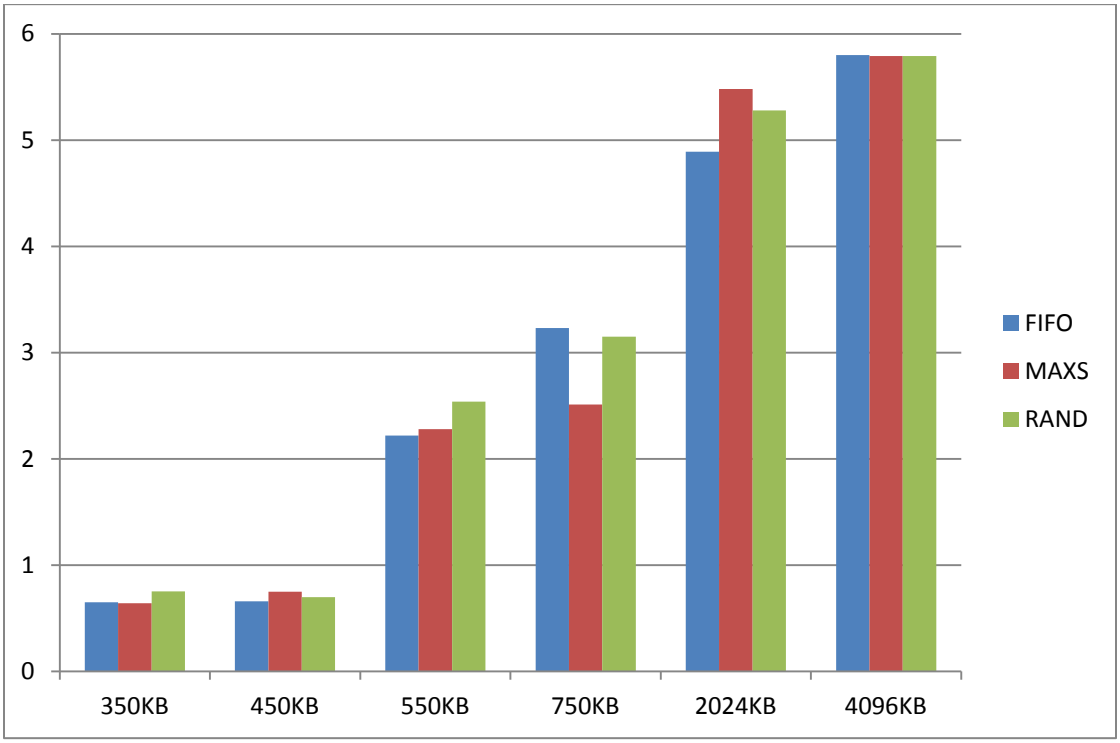
5. Cache Hit % for Cache Size of 750KB



6. Byte Hit Ratio ([Bytes from Cache]/[Bytes from Web]) for Cache Size of 750KB



7. Cache Hit Percentage with Increasing Cache Size and Random Workload



8. Byte Hit Ratio with Increasing Cache Size and Random Workload

VIII. ANALYSIS OF RESULTS

The graph 1, comparing the access times of No-Cache with Caching Proxy Servers, shows the obvious results. For every URL request in No-Caching method, the content has to be fetched from the Web and thereby is much longer. With Caching Schemes, most of the accesses are from Cache and hence lower access times.

Graph 2, compares the Access Times of different Caching Schemes with 750KB Cache Size. Different Access Times for different websites is because each of them sends out different sizes of data. The larger the page, longer it takes to access. For each individual URL, different Caching Schemes behave different based on the Size of the Page being fetched. This is more clearly shown by Graphs 3 and 4.

Graph 3, shows the average access time taken for the Washington Post Page, which is of size 323KB(large), to be fetched when requested over 8 times out of a total of 100 requests, by each of the Caching Schemes under different Distribution of Workload. When the workload is of:

1. Normal Distribution: FIFO and MAXS take comparable time, whereas RAND takes much less times.
2. Uniform Distribution: Again FIFO and MAXS take similar time, while RAND is faster.
3. Random Distribution: FIFO takes much longer than the other two.

Graph 4, shows the average access time taken for the Time.com Page, which is of size 178B(small), to be fetched when requested over 8 times out of a total of 100 requests, by each of the Caching Schemes under different Distribution of Workload. When the workload is of:

1. Normal Distribution: FIFO and MAXS take comparable and much less time than RAND.
2. Uniform Distribution: FIFO takes way more time than MAXS and RAND. MAXS is the fastest for this scenario and that is because the small web page is rarely replaced from the Cache.
3. Random Distribution: FIFO takes way more time than MAXS and RAND.

Graphs 5, shows the Cache Hit percentage for different Workloads and Caching schemes with the same Cache Size of 750KB.

1. Normal and Random Distribution: FIFO is better than MAXS which is better than RAND.

2. Uniform Distribution: Performance of FIFO and RAND decreases while MAXS remains high. This is because Uniform Distribution makes uniform requests but MAXS replaces only the largest of the pages available, while FIFO and RAND remove pages more uniformly, thereby causing more cache misses.

Graph 6, shows the Byte Hit Ration for different Workloads and Caching schemes with the same Cache Size of 750KB and the results follow the same pattern as that of Graph 5 i.e., Cache Hit percentage.

Graphs 7 and 8 show how the performance of different schemes changes with increasing Cache Size with a Random Workload. As you can see beyond a certain point, any increase in Cache Size won't increase Cache Hit or Byte Hit Ratios and all schemes converge.

Cache Hits for RAND and MAXS increase monotonically while for FIFO it actually goes down a bit from 350KB to 450KB and then again from 550KB to 750KB. Though the drop is not much, it is not worthy and possible explanation is that small increases in size don't play significant role in improving Cache Hits with Random Workload, the size of Pages is not considered the likelihood of smaller but frequently requested pages is high and this is validated from the Byte Hite Ratio measurements.

It is interesting to note that the Byte Hit Ratio increases monotonically with increasing Cache Size and doesn't dip anywhere like Cache Hits for FIFO, because removing smaller size pages doesn't affect the Byte Hit Ratio to the same extent as it affect the Cache Hits.

In Summary:

1. FIFO does well with Normal Distribution and worst with Random Distribution while accessing larger web pages and it does poorer with Uniform Distribution for smaller web pages.
2. MAXS does well with Random Distribution for larger web pages and does best with FIFO for smaller web pages.
3. RAND does equally well with all three workloads for larger web pages and does best with Random Distribution for smaller web pages.
4. Overall RAND does well with large web pages and MAXS does well with smaller web pages.

IX. CONCLUSION

In this Project, we worked on implementing a Proxy Server that implements three different Caching Schemes and evaluate them using three different workload distributions. The Server and the corresponding RPC were implemented using LibCurl and Apache Thrift respectively. C++

was used to implement all logic. The Workload generation was automated and tested. Results show that each Caching Scheme has its advantages and disadvantages and must be used with knowledge of the kind of workload that is expected. If it is common to access large web pages, then RAND works best and if smaller web pages are accessed more frequently then MAXS is good. FIFO works best with Normal Distribution of workload.

X. REFERENCES

[1] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, Edward A.Fox, "Caching Proxies: Limitations and Potentials", Oct 7, 1995

[2] Chetan Kumar, John B Norris, "A new approach for a proxy-level web caching mechanism", Decision Support Systems, May 2008.

[3] "Web Caching - A cost effective approach for organizations to address
all types of bandwidth management challenges", Visolve Open Source Soluctions, March 2009

[4] MCS Lee, Aditya, Marc, "Thrift: Scalable Cross-Language Services Implementation".

XI. APPENDIX

All code and Data Measurements are uploaded in the GitHub repository: https://github.com/smanchem/RPC-ProxyServer

While creating workloads, we first measured the size of the data fetched by each URL in order to create a balanced workload. We did this without using any caching mechanism and measured the amount obtained on an average for each URL. The size of the content of the 18 URLs is mentioned in increasing order in the table below:

URL	Size in Bytes
http://www.facebook.com	6
http://www.time.com	178
http://www.bbc.com/news	238
http://online.wsj.com	303
http://www.cmu.edu	514
http://www.nationalgeographic.com	7530
http://www.hindu.com	16395
http://www.stanford.edu	28709
http://www.google.com	65546
http://www.forbes.com	101057
http://www.cnn.com	120568

http://america.aljazeera.com	157847
http://www.telegraph.co.uk	174648
http://www.eenadu.net	180297
http://www.nytimes.com	208789
http://www.theguardian.com/us	218278
http://www.washingtonpost.com	331155
http://www.dailymail.co.uk/ushome/index.html	563502

Project Work Distribution

Sandeep Manchem: Implemented Proxy Server and Caching Schemes. Co-wrote Report.

Sunayana Gali: Generated Workloads and Ran Tests. Co-wrote Report.