The Multikey Web Cache Simulator: a Platform for Designing Proxy Cache Management Techniques

L.G. Cárdenas, J. Sahuquillo, A. Pont, and J.A. Gil[1]
Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
Cno. de Vera s/n 46071, Valencia (Spain)
luicaral@doctor.upv.es {jsahuqui, apont, jagil}@disca.upv.es

Abstract

Proxy caches have become an important mechanism to reduce latencies. Efficient management techniques for proxy caches which exploits web-objects inherent characteristics are an essential key to reach good performance. One important segment of the replacement algorithms being applied today are the multikey algorithms that use several key or object characteristics to decide which object or objects must be replaced. This feature is not considered in most of the current simulators. In this paper we propose a proxy-cache platform to check the performance of web object based on multikey management techniques and algorithms. The proposed platform is coded in a modular way, which allows the implementation of new algorithms or policies proposals in an easy and robust manner. In addition to the classical performance metrics like the hit ratio and the byte hit ratio, the proposed framework also offers the response time perceived by users.

1. Introduction

Web traffic is constantly increasing due to the increasing number of users connected to Internet. Traffic doubles every 6 months [8] [17] [18], affecting adversely user perceived latencies. As a consequence, network infrastructure needs to be constantly improving to satisfy QoS users demand, including both technology aspects (e.g. fastest links, proxies and servers) and related software.

Caching and prefetching are the common adopted solutions in the Web architecture to improve performance. Caching involves storing the most demanded [1] [5] (popular) web objects closer to the user. Proceeding on this way, users can fetch objects from its browser cache or an intermediate cache (proxy-cache) instead of the original web server, so reducing user perceived latencies. On the other hand, prefetching initiates the fetch of the object before the user ask for it. The goal of this technique is to have the object in a cache when required later. In general, the prefetching process uses idle times (in the browser or in the proxy) to prefetch objects. Note that prefetching and caching are orthogonal techniques so they can be applied together.

Caching can be implemented at different stages in the path of a HTTP request: i) in the browser of the client, ii) at some middle point, between the client and the server (proxy web server), and iii) close to the original web server (inverse proxy or surrogate). Browser caches store objects requested by a single user while proxy caches store objects for many users, usually in a homogeneous segment of Internet users; e.g., a university, a private or public corporation, or an ISP. Due to the larger number of users connected to the same proxy server, object characteristics (popularity, spatial and temporal locality) can be better exploited increasing the cache hit ratio and improving web performance. Our work focuses on proxy caching because this cache offers more benefits in the Web architecture and more clients can benefit from it.

Cache performances are strongly related with the performance of their replacement algorithms. These algorithms use an individual or multiple key to decide which object or objects must be replaced. An individual key can be well a single object characteristic (like object size, frequency, etc) or well a value obtained from the application of a mathematical formula which uses different object characteristics as inputs. Example of algorithms using only one characteristic are LRU, LFU, SIZE [15]; and using a formula are GDS [6], GDSP [9], PSS [2], LRV[12] LRFU [11] and LUV [3]. Algorithms using multiple keys refer to those algorithms which apply sequentially several criteria like LOG2-SIZE [2], HYPER-G [2]. Among algorithms, it has be shown [4] [6] [10] [11]

[1] This work has been partially supported by the:
Spanish Government Grant CYCYT TIC2001-1374-C03-02,
Generalitat Valenciana grants CTIDIB/2002/32 and CTIDIB/2002/114,
and Mexican Government Grant CONACYT 177846/191901.

that those using recency and frequency are the ones which achieve the best performance.

In this paper we present the Multikey web-cache simulator environment capable to model in an easy way all the mentioned above algorithms. The remainder of this paper is organized as follows. Section 2 discusses some related work. Section 3 details the Multikey simulation environment. Section 4 illustrates how easy can be a replacement algorithm implementation under the Multikey environment. Section 5 shows some performance results using real traces, and finally 6 presents some concluding remarks.

2. Related work

Researchers check their ideas by using different alternatives: real systems, trace-driven simulators, or any other kind of simulators before implementing them in commercial software products. Although there are a certain number of web cache simulators, very few of them are open source available on the web. This section only focuses on some of them offering open source distribution.

Pei Cao proposed the Web Cache simulator [16] which is a trace-driven simulator that can simulate replacement algorithms using a single key (object characteristic or formula). Written in C++, includes code for LRU, SIZE, LRV and HYBRID replacement algorithms. The simulator employs a priority queue to keep objects ordered by a priority field defined in the algorithm (size, frequency, last access time) and determine the object to evict from the cache, when new space is required. The Pei Cao Web Cache Simulator requires a special trace format. The main characteristic of this trace is that the URI field is composed by two identifiers, one referring to the web server and the other referring to the corresponding URI. The remaining fields are easily obtained from the original proxy logs. In this simulator, the response time of a miss is taken directly from the original proxy log, instead of being estimated.

Network Simulator (NS) [19] is one of the best network simulators available. It uses detailed network protocols model to estimate performance metrics. It supports TCP emulation, routing, multicast protocols both, in common and wireless networks. The NS is composed by an event scheduler and a set of libraries of network components written in C++. To be able to perform a NS simulation, the following steps should be taken: i) write a tcl script to initiate the event scheduler, ii) select network topology by using the implemented network objects, and iii) configure the traffic through the event scheduler (when should start and stop transmitting network packets). NS generates several files with detailed data of the simulation. This data can be used to perform an analysis or as an input for NAM (network animator), a graphic animated visualization tool included in this simulator. Caching is not a strong feature in NS. It only includes an HTTP class to model a simple infinite size cache. It does not implement replacement algorithms, neither measures caching performance.

Brian Davison developed another trace-driven network and cache simulator (NCS) [7]. This is a trace-based simulator, as NS, focuses more on network issues rather than web-cache management techniques. This simulator does not consider web object characteristic but it models very carefully other network aspects such as DNS effects, idle times, connection times, and waiting times. It also includes some level of prefetching.

The Proxim simulator [20], as NCS does, emulates proxy caches using traces. The simulator considers interaction between HTTP and TCP messages and the network environment. It takes into account several factors that could affect object transferences, such as storage restrictions and data in transit during connection aborts. To feed Proxim traces must be filtered to include information related with TCP events, HTTP events, HTTP headers, byte counts, etc. Simulations can be run considering or avoiding the proxy cache.

3. The Multikey Simulation Environment

The tool presented in this paper is a trace-driven simulator, which is modularly coded using classes in the C++ programming language. The source code of the simulator is available from [21]. The tool is divided in two modules: i) a filter program that prepares traces to feed the simulator and calculates estimated transmission times, ii) a cache simulator that models web-proxy caches. Figure 1 shows the interrelation between both modules.

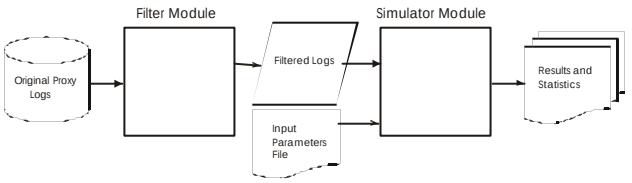

Figure 1. Modules and interrelation between modules in the Multikey Simulation Environment.

3.1. The Filter Module

As some trace-driven simulators [16] [19] [20] [7] a filter program is needed to prepare original proxy logs in a readable format suitable for the simulator. Figure 2 shows the block diagram of the filter module. In the current version, the filter only accepts squid-proxy logs, but it can be easily modified to incorporate more proxy logs types such as Harvest, DEC, UCB, CRISP, etc. Like in the Squid proxy cache [14] our simulator uses the Message Digest Algorithm (MD5) [13] to codify each URI. This algorithm guarantees a unique 128 bits footprint from any arbitrary URI (or any other input). Due to the relatively small size of the footprints, the object search time in the

simulator is quicker than using the full URI as object label. Another optional feature offered by the filter module is that log requests can be filtered by web object type; e.g., gif, jpg, html, dynamic, or by HTTP result code.

In addition to the features discussed above, the filter program also estimates some latency related parameters that will be discussed below.

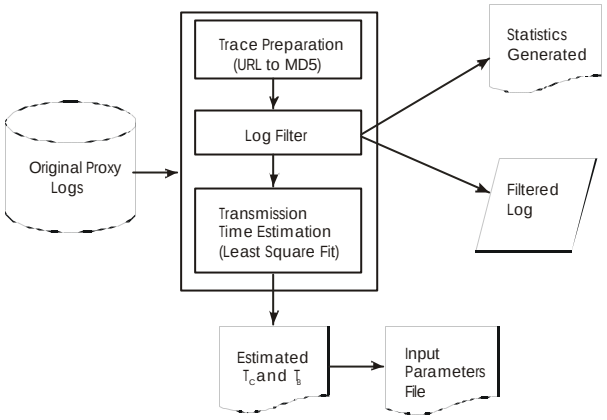

Figure 2. Filter module of the MSE environment.

3.1.1. Obtaining time related parameters. In classical processor caches, the performance of the replacement algorithm depends directly on the hit ratio obtained, because the penalty time in a miss is assumed to be constant. In other words, when a miss occurs, the block (which has a constant number of bytes), is fetched from main memory and this time is fixed by the technology and the memory organization. This feature cannot be straightforward applied to web objects because their size widely differs among them, and also the transmission time due to both Internet traffic and the huge variability of the Web Servers workload throughout the day. Consequently the fetch and the penalty times cannot be assumed constant. This reasoning implies that simulation environments should take into account the penalty time, in order to enable precise replacements algorithms for comparison studies. In other words, it is possible that the algorithm offering the higher hit ratio (or byte hit ratio) is not the one achieving the best performance (average response time per request). This penalty time (T_P) is composed by a connection time (T_C) plus the object size (in bytes) per the mean time needed to transfer a byte (T_B), i.e., $T_P = T_C + T_B * size$, where *size* refers to the size of the web-object, and this time can be closely modeled by the least square fit method already employed in [9].

The Filter program estimates T_C and T_B from the original traces. These parameters are used to estimate the penalty time for each cache miss incurred during the simulation. The simulator uses this time to estimate the average response time perceived by the users, after serving all the requests. Thus, offering the possibility to compare algorithms on a time latency basis.

3.2. Web Cache Simulator Module

Figure 3 shows the block diagram of the cache simulator module. As mentioned above its inputs are: i) a set of logs prepared by the filter program and ii) the input parameters file with the corresponding cache information. The cache module simulates a web cache having as inputs the requests included in the log and it generates statistics such as Hit Ratio, Byte Hit Ratio and Average Service Time into a simulation statistics file, and an estimated response time for each request in a separated file. Some parameters needs to be specified in order to run simulations, including name and log path, number of requests to be simulated, T_C, T_B, and the cache size. All these parameters are taken from an input parameters file.

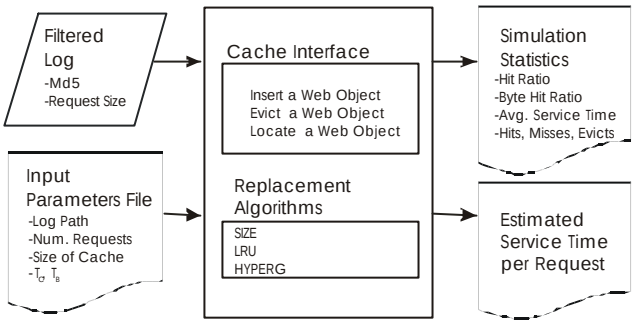

Figure 3. Cache simulator module.

In order to model a web-cache environment the simulator defines two main classes: a Cache class and a Web-Object abstract class.

3.2.1. The Cache class. The Cache class emulates the behavior of the cache of a proxy server. To model its behavior the Cache class includes three main methods: web object search, web object insertion and web object eviction. Table 1 shows a brief description of the public methods provided in the Cache-class (defined in the cache.h file). These methods are the interface of the Cache class with the Web-Object module to support the implementation of management algorithms.

To implement these features the Cache class defines two main object structures: an object localization structure (OLS) and a priority ordered ternary tree (PTT).

Table 1. Basic support routines of the Cache-class.

Routine	Description
boolean isEmpty()	It verifies if the Cache structure is empty
boolean enoughSpaceFor(Size)	It verifies if there is enough space to fit in an object of this Size in the Cache structure.
void insertsWebObject(Object)	It places an object in the Cache structure
void reinsertWebObject(Object)	It updates the position of the web-object in the Cache structure and its counter value.
Object evictWebObject()	It removes an object from Cache structure and returns the evicted object.
Object locateWebObject(URI)	It searches the Object in the Cache structure. It returns an object or NULL.

The OLS is used to enable a fast access to the web-objects contained in the PTT structure when searching by an object URI. The OLS is composed by two data structures: an array of pointers, and binary trees. The array has 2^n elements indexed with the n less significant bits (n is set to 16 by default) of the 128 MD5 web-object URI identifier (coded for each web-object URI in the filter module). Each element in the array points to a binary tree whose nodes point to web-objects in the PTT structure. Each node in the OLS structure points also to the corresponding object in the PTT structure. Figure 4 shows the block diagram of the object localization structure (OLS). The OLS works as follows when the locateWebObject method is invoked: i) first, it uses the n less significant bits of the MD5 web-object URI to determine the tree that could contain the object, ii) then, the remaining MD5 web-object URI is used to search such URI object in the selected binary tree.

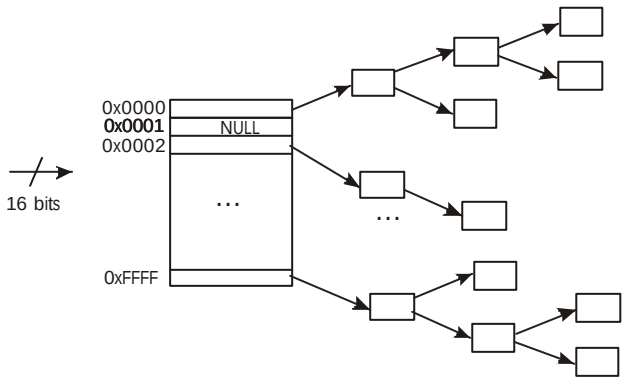

Figure 4. OLS structure block diagram: arrays and binary trees.

On success, the function returns a pointer to the web-object in the PTT; otherwise, returns a null pointer. Using this method the computation time required to do a search is less than $O(log\ N)$, where N is the total number of objects in the corresponding binary tree. The maximum number of objects supported by our OLS structure is 2^{128} (number of possible codifications with the MD5), which are spread among the 2^n trees.

The other data structure used to manage web-objects is a priority ordered ternary tree (PTT). Figure 5 shows the block diagram of the priority ternary tree (PTT). The purpose of this structure is to keep the objects ordered by their priority, which is defined in the Web-Object derived class discussed below. The meaning of the priority of a web object depends on the replacement policy implemented (size, frequency of use, or a combination). One important feature of the PTT is the quick access to the least priority object and more priority object, which eases the evictWebObject method to implement fast object evictions. Note that each time the priority of an object changes (due to the change of any object characteristic), the object must be reinserted in the PTT accordingly to its new priority.

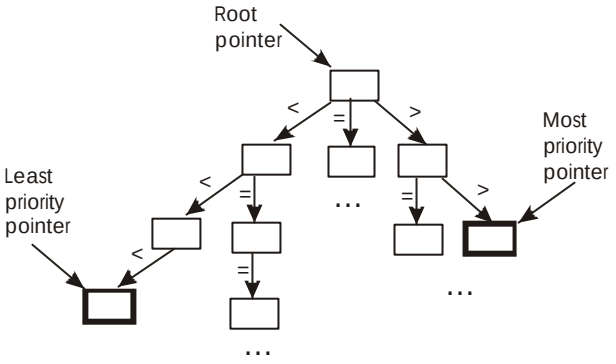

Figure 5. PTT block diagram.

3.2.2. The WebObject class. The Web-Object abstract class defines the main characteristics of the web-objects contained in the structures of the Cache class. This includes typical properties such as size, popularity, and the URI for each web-object. This class also defines three priority methods: greater, less and equal, which are used for comparing priorities when inserting a web-object in the PTT structure. These methods permit to traverse the tree in order to accommodate web-objects in the PTT according to their priority. These priority methods are used inside insertWebObject() and reinsertWebObject() methods defined in the Cache class.

To simulate each web-cache replacement algorithm, the Web-Object abstract class must be derived in a specific Web-Object class. Proceeding in this way, it is possible to add to the Web-Object derived class the most relevant properties to implement a replacement algorithm and to

define the priority scheme by overriding the greater, less and equal methods. Note that in the priority methods we can test as many properties as we want in order to implement multiple key replacement algorithms.

4. Implementing Replacement Algorithms

In order to model a cache replacement policy using the Multikey simulation environment we need: i) to implement the replacement algorithm code using the Cache interface functions, ii) to obtain a derived class from the Web-Object abstract class for that particular replacement algorithm. Figure 6 shows the relation between the classes and the corresponding files. This section shows some examples to illustrate both single key and multiple key implementations.

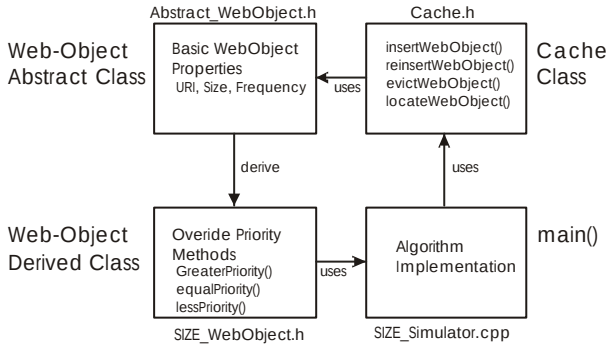

Figure 6. Relation between classes and the corresponding files.

Figure 7 illustrates the implementation of the Size replacement algorithm as example of a single key algorithm. Figure 7 also shows the section where the routines of cache.h are called to simulate the Size replacement algorithm, while Figure 8 shows the priority method in the Size Web-Object derived class. As one can see the implementation is quite straightforward by using the cache interface offered by the simulation environment. Notice that in the case of single key, where only one parameter is involved, the priority order method is reduced to just one line of code; although the key could be a complex mathematical formula.

```
class SIZE_WebObject : public Abstract_WebObject {

public:

      SIZE_WebObject(char *pUri, const long pSize, const long pTime);

      bool greaterPriority(Abstract_WebObject *compWOPtr)
      bool lessPriority(Abstract_WebObject *compWOPtr);
      bool greaterPriority(Abstract_WebObject *compWOPtr);
};
// Use Abstract_WebObject constructor
SIZE_WebObject::SIZE_WebObject(char *pUri, const long pSize, const
                        long pTime):Abstract_WebObject(pUri,pSize,pTime) {}

// Redefine priority methods in order to suit SIZE implementation
bool SIZE_WebObject:greaterPriority(Abstract_WebObject *compWOPtr) {
      return ( returnSize() < compWOPtr->returnSize()) }

bool SIZE_WebObject:lessPriority(Abstract_WebObject *compWOPtr) {
      return ( returnSize() > compWOPtr->returnSize()) }

bool SIZE_WebObject:equalPriority(Abstract_WebObject *compWOPtr) {
      return ( returnSize() == compWOPtr->returnSize()) }
```

Figure 8. Specification of the priority methods (in the derived class) for the Size replacement algorithm.

```
Cache_SIZE Cache(simCacheSize,simTc,simTb);// Instance a Cache with : Cache Size = simCacheSize, Tc = simTc, Tb = simTb read from the Input Parameters File

// At this point, a log line has been read into variables req_Uri, req_Size and current time is set to now.

if (req_size < Cache_SIZE.cacheSpace() ) {                                  // Cannot place web-objects larger that  the space occupied by the Cache.
     SIZE_WebObject *new_RequestPtr = new SIZE_WebObject(req_Uri,req_Size,now);   // Instance a new web-object from request data.
     Cache_SIZE.updateCacheRequests(new_RequestPtr);                        // Update Requests statistics.

     SIZE_WebObject *search_ObjPtr = (SIZE_WebObject *) Cache_SIZE.locateWebObject(req_Uri); // Look for the web-object in the Cache structure.
          if (search_ObjPtr != NULL) {                                      // there is a Hit.
               search_ObjPtr->verifiySizeChanged(req_Size);
               search_ObjPtr->setTime(now);
               Cache_SIZE.updateCacheHits(search_ObjPtr);                   // Update Hits statistics.
               Cache_SIZE.relocateWebObject(search_ObjPtr);                 // Relocate the web-object in the Cache structure.
          } else {                                                          // there is a Miss.
               Cache_SIZE.updateCacheMisses(new_RequestPtr);                // Update Misses statistics.
               while (!Cache_SIZE.enoughSpaceFor(req_Size)) {               // Evict objects until the new object fits in the Cache structure.
                    SIZE_WebObject *evicted_ObjPtr = (SIZE_WebObject *) Cache_SIZE.evictWebObject();
                    Cache_SIZE.updateCacheEvictions(evicted_ObjPtr);        // Update Evictions statistics.
               }
          Cache_SIZE.insertWebObject(new_RequestPtr);                       // Place the new web-object in the Cache structure.
          }
}
```

Figure 7. Fragment code of the size replacement algorithm.

For a multiple key replacement algorithm the derived Web-Object class code is not too much complex. As an example, Figure 9 illustrates the greater priority method of the derived class corresponding to the Hyper-G replacement algorithm, whenever the insertWebObject or relocateWebObject of the Cache class calls for the greaterPriority method. In order to place the web-object accordingly to its priority, first it compares the frequency of reference of the object to be placed (object A) with the frequency of a given object in the PTT (object B). This comparison is done with the root element of the PTT. If the frequency of the object A is greater, the comparison of the first key ends successfully returning true, and descending to the right side of the subtree of the PTT. Otherwise, if both frequencies are equal, then proceeds the second key comparison, and so on until the third key. Note that this comparison must be performed by the inserting routines of the Cache class in order to place correctly the web-object in the PTT structure. The comparisons are supported by the priority methods defined in each derived Web-Object class.

```
bool HYPERG_WebObject::greaterPriority(Abstract_WebObject *compPtr){

    if ( returnFrequency() > compPtr->returnFrequency())
        return true;
    else
        if (returnFrequency() == compPtr->returnFrequency())
            if (returnTime() > compPtr->returnTime())
                return true;
            else
                if (returnTime() == compPtr->returnTime())
                    if (returnSize() < compPtr->returnSize())
                        return true;
                    else return false;
                else return false;
        else return false;
}
```

Figure 9. Example of greater priority method for a multikey Hyper-G Web-Object.

5. Obtaining Performance Results

The Multikey environment implements some statistical methods to obtain performance results and other useful metrics for evaluation studies, like the number of requests, number of evictions, object hit ratio, average-response time, etc.

This section presents some simulations results obtained using a trace collected from the proxy servers of the Universitat Politècnica de València (Spain). The trace contains about 5M requests collected during 14 days. The original trace was prepared using the filter-module and a filtered trace contains 3,145,929 requests that suppose a total space of 18.64 GB required. Only 867,467 were unique requests (meaning that there are totally this quantity of unique web-objects) needing 10.45 GB of data storage. The Infinite Hit Ratio and Infinite Byte Hit Ratio are 72.43 and 0.5 respectively (the maximum Hit Ratio and Byte Hit Ratio that could be obtained with this trace). Table 2 summarizes these results.

Table 2. Characteristics of the trace used in the simulation.

# Requests	3,145,929
# Unique Requests	867,467
Average Request Size	6.21 KB
# Requested Bytes	18.64 GB
# Unique Bytes	10.45 GB
Infinite Hit Ratio	72.43
Infinite Byte Hit Ratio	0.5

With this trace, T_C and T_B values were obtained using the filter module as explained in section 3.1. Only requests meaning a miss in the squid logs were considered (e.g. TCP_REFRESH_MISS, TCP_CLIENT REFRESH_MISS and TCP_MISS); therefore, the times considered to calculate T_C and T_B correspond to objects fetched by the proxy server. Notice that as some original servers use the HTTP/1.1 protocol which implements persistent connections, the obtained connection time estimates the connection time per object instead of the real time needed to establish a connection. Table 3 shows the estimated values.

To guess the average response time per request, a simulation was performed using the filtered trace, employing the T_C and T_B obtained in the filter modules. The average response time per request obtained was 648.93 ms; this time was calculated as the sum of the penalty time for all the misses occurred during the simulation and divided by the total number of requests received by the cache. This penalty time is automatically calculated inside the statistics routine updateCacheMisses.

Table 3. T_C, T_B obtained in the filter module and average response times calculated in the simulator module. Legend: i) Cache size has been assumed 5 times less than the total objects size; ii) The Size algorithm has been used and iii) The hit ratio is 70% so that only 30% of the requests require extra connections.

Filter Module	Connection Time (T_C)	627.60 ms
	Transmission Time per Byte (T_B)	0.10 ms
Cache Module	Average Response Time perceived by users	648.93.ms

Figure 10 shows the original response time per object size obtained from the original log file and the simulated response time for each miss incurred throughout simulation. In the simulation, the penalty time for each miss was obtained using the formula presented before. As one can see, responses times show a high variability because all external connections were considered. In order to simplify the approach, we consider the 24 hours of every day as well all the accessed servers.

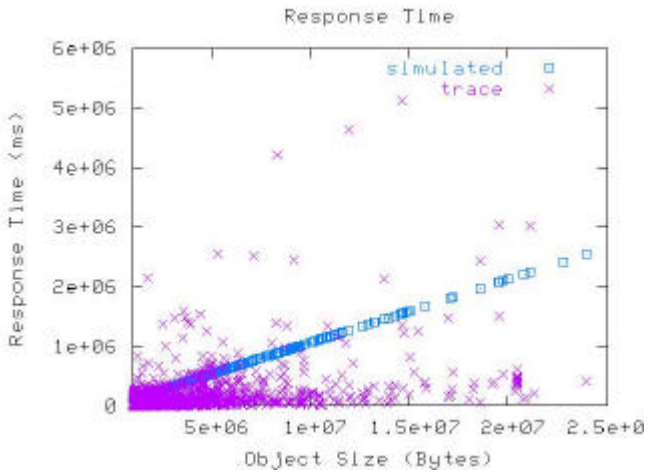

Figure 10. Original and simulated response time per object size.

6. Conclusions

Efficient proxy management algorithms are required to achieve good performance. Researchers check their ideas by using different alternatives: real systems, trace-driven simulators, or any other kind of simulators before implementing them in commercial software products.

In this paper we propose a new software environment to explore new proxy algorithms proposals. The main contributions of our open source environment are i) it provides cache interface functions which allows a quick proxy cache replacement algorithm implementation, ii) its interface permits multiple key replacement implementations, which are an important segment of those that can be found in the open literature, iii) the data structures implemented permit short simulation times, and iv) its filter module offers time related parameters which allows the simulator to estimate average response times perceived by users, assigning a penalty time for each miss incurred. The last contribution is very important in order to perform fair performance comparison studies.

7. References

[1] G. Abdulla, E. Fox, and M. Abrams, "Shared used behavior on the World Wide Web," *Proceedings of the WebNet97*, Toronto, Canada, 1997.

[2] C. Aggarwal, J. L. Wolf, and P. S. Yu, "Caching on the World Wide Web," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 94-107, 1999.

[3] H. Bahn, S. K. Koh, S. L. Min, and S.H. Noh, "Efficient Replacement of Nonuniform Objects in Web Caches," *IEEE Computer*, vol. 35, no. 6, pp. 65-73, 2002.

[4] L. Breslau, P. Cao, L.Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *Proceedings of the IEEE Conference on Computer Communications (INFOCOM99)*, New York, U.S.A., pp. 126-134, 1999.

[5] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of WWW Client-based Traces," *Technical Report BU-CS-95-010*, Computer Science Dept., Boston University, U.S.A., July, 1995.

[6] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algoritms," *Proceedings of the 1st Symposium on Internet Technology and Systems (USITS97)*, Monterey, U.S.A., pp. 193-206, 1997.

[7] B. D. Davison, "NCS: Network and Cache Simulator -- An Introduction," *Technical Report DCS-TR-444*, Department of Computer Science, Rutgers University, U.S.A., 2001.

[8] J. Gwertzman and M. Seltzer, "World-Wide Web Cache Consistency," *Proceedings of the 1999 USENIX Technical Conference*, pp. 141-152, San Diego, U.S.A., 1996.

[9] S. Jin and A. Bestavros, "Popularity-aware GreedyDual-Size Web Proxy Caching Algorithms," *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS2000)*, pp. 254-261, Taipei, Taiwan, 2000.

[10] T. P. Kelly, Y. M. Chan, S. Jamin, and J. K. MacKie-Mason, "Biased Replacement Policies for Web Caches: Differential Quality of Service and Aggregate User Value," *Proceedings of the 4th International Web Caching Workshop*, San Diego, U.S.A., 1999.

[11] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. Kim, "On the Existence of a Spectrum of Policies that subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pp.134-143, Atlanta, U.S.A., 1999.

[12] P. Lorenzetti, L. Rizzo, L.Vicisano, "Replacement policies for a Proxy Cache," *Technical Report LR-960731*, Dipartimento di Ingegneria dell' Informazione, Università di Pisa, Italy, 1996.

[13] R. Rivest, "The MD5 Message Digest Algorithm," *RFC 1321*, http://www.faqs.org/rfcs/rfc1321.html, 1992.

[14] Squid Web Proxy Server Documentation, http://www.squid-cache.org

[15] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla and E.A. Fox, "Removal Policies in Network Caches for World-Wide Web Documents," *Proceedings of the Conference on Applications, Technologies, Architectures and Protocols for Computer Communications SIGCOMM'96 Conference*, pp. 293-305, Stanford, U.S.A., 1996.

[16] P. Cao, Wisconsin Web Cache Simulator,

http://www.cs.wisc.edu/~cao/
[17] M. Gray, Web Growth Summary, http://www.mit.edu/ /people/mkgray/net/web-growth-summary.html, 1996.
[18] H. Schulzrinne, Long-Term traffic statistics, http://www.cs.columbia.edu/~hgs/internet/traffic.html
[19] The Network Simulator, http://www.isi.edu/nsnam/ns/
[20] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich, "Web Proxy caching: The Devil is in the Details," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, 1998.
[21] The MultiKey Simulation Environment, http://remo.disca.upv.es/ MSE/mse.html