A Windows Based Web Cache Simulator Tool

F.J. González-Cañete E. Casilari, A. Triviño-Cabrera
University of Málaga
ETSI Telecomunicación
29071- Málaga (Spain)
+34 952 13 27 55
{fgc,ecasilari,atc}@uma.es

ABSTRACT
In this paper, we describe a Windows based Web cache simulator tool. This tool is able to process the IRCache based traces files and reproduce the behavior of a Web Proxy Cache. It can be configured to simulate fifteen replacement policies and two admission control policies. The cache size, the percentage of warm-up and the cost function of some replacement policies can also be configured. The simulations can be performed in a batch process and the results are stored in text format files that can be automatically analyzed using other tools like Matlab to obtain performance graphs.

Categories and Subject Descriptors
J.2. [**Computer Applications**]: Physical Sciences and Engineering, Telematics

General Terms
Measurement, Performance.

Keywords
Web cache, replacement policies, simulator, IRCache.

1. INTRODUCTION
When a new system is developed there are two ways of checking if its functionalities are fully and correctly implemented. The first of them is to build the system and start working with it and the second is to simulate the system behavior using a simulator. In some situations the first way is absolutely inadequate because it can be costly to develop the real system and if problems are detected in the production phase the system have to be redesigned or refined. On the other hand the simulation alternative gives a choice to reproduce the behavior of the system and it can even reproduce situations that are difficult, costly or even impossible to perform using the real system.

SIMUTOOLS 2008, March 03-07, Marseille, France

DOI 10.4108/ICST.SIMUTOOLS2008.2933

One of the environments where the simulations are the main tool to measure the performance of a system is the Internet. In that way, an entire network can be simulated using different types of traffic, interruption, connectivity or even mobility models as can be performed with network simulators such as ns-2 [1]. Although the ns-2 simulator is one of the most popular network simulators [2] it only simulates the behavior of a proxy cache in a basic form as it simply implements an infinite cache without any replacement policy or admission control features.

Due to the lack of an open source proxy cache simulator, some researchers have developed their own simulators in order to test the replacement policies or admission control policies they propose for caching. DavisSim [3] is a simulator implemented using C++ and based on the Winsconsin Web Cache Simulator [4]. Both of them are UNIX/Linux based simulators. They utilizes a pro-processed Web trace that contains the server identification, the page identification, the size of the document requested, the time to serve the document, the time of the last modification, time of access and the user id. This simulator takes input files for the parameters of the simulations and output the cache performance parameters such as the cache hits and misses in a file. Another simulator is the Multikey Web Cache Simulator [5] which simulates the behavior of some key-based replacement policies in a Web cache. This is also a Linux based simulator implemented in C++ using a modular method in order to allow an easy way to expand the capabilities of the simulator. The simulator accepts Squid proxy logs [6] as traces to simulate.

All the previous works are meant for Linux systems. However, there are many users of Windows operating systems who may benefit from the utilization of a Web cache simulator. This paper presents a Windows-based Web cache simulator that accepts Squid proxy logs as input traces and also implements a great variety of replacement policies and some admission control policies. The parameters of the simulations can be introduced in a visual way using a user-friendly interface. It also allows creating batch files for simulations using a visual method.

The rest of the paper is organized as follows. Section 2 summarizes the cache simulator architecture and the modules it contains, defining the behavior of the Filter Module and the Cache Module. Section 3 comments the Cache Module as well as the implemented replacements and admission control policies. It also explains the metrics and statistics obtained from the simulations.

2. Cache simulator architecture

The cache simulator is divided into two modules. The first module corresponds to the Filter module and the second module is the cache simulator itself. The purpose of the Filter module is to process the Squid trace files and adapt them to a more accurate format for the simulator. The filtering architecture is shown in Figure 1.

The Filter Module takes the Squid trace files as input and performs a first filtering process (Filter 1) purging those requests that have been generated dynamically by CGI (Common Gateway Interface) because the documents returned by these kind of requests are unique for each request and therefore they should not be cached [7]. Because of this fact, the requests that contain the strings 'cgi', 'cgi-bin' or '?' have been discarded. Those requests that contain the string ':3128' have also been filtered as this is the port that Squid utilises to interchange information between collaborating caches. Finally only those requests with a cacheable response code have been considered, that is, 200 (OK), 203 (Partial), 206 (Partial Content), 300 (Multiple Choices), 301 (Moved) and 302 (Redirects) and 304 (Not Modified). The second filtering process (Filter 2) discards some of the parameters of the requests included in the original Squid traces and generates an output file with the access time, the latency of the transfer, the size of the document, the identification of the document and its content-type. The identification of the documents and the content-types are numerically coded in order to achieve a faster operation of the simulator.

The architecture of the simulator is shown in Figure 2. The processed traces first enter the Admission Control process that decides if the document passes the admission control policy selected (if there is one selected) and hence the document enters the Cache Process. The Cache Process executes the replacement policy selected deciding which documents to evict from the cache to make room for the new one if necessary. Finally, the Measurement and Statistics Module collects information of the above mentioned processes such as the number of requests, documents accepted or rejected, documents evicted or hits in the cache in order to calculate statistics about hit ratios, byte hit ratios or other metrics which are employed to obtain performance comparisons. Once the simulations are finished, the Measurement and Statistics Module stores the results in a text file that can be processed automatically.

3. The Cache Module

3.1 Replacement Policies implemented

The purpose of a replacement policy is to evict the documents with the lowest probability of being referenced in the near future to make room for the new ones. The Cache Module implements thirteen replacement policies as well as two replacement policies that divide the storage space depending on the size and the content-type of the documents respectively. In this section we

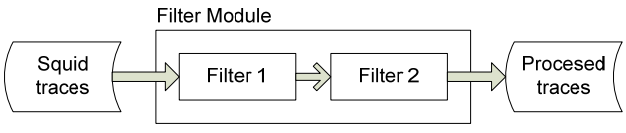

Figure 1. Filter Module architecture

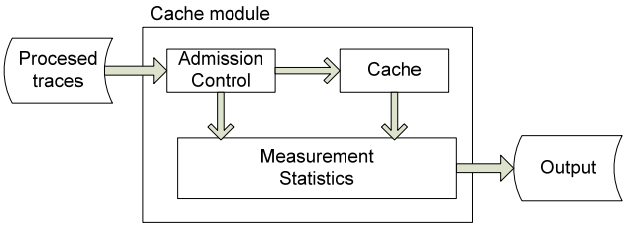

Figure 2. Cache Module Architecture

give details of the replacements policies actually implemented by the simulator:

- FIFO: It is the simplest replacement policy. The first document that enters the cache is the first to be evicted when storage space is needed.
- LRU (Least Recently Used) replaces the document that was referenced longer ago.
- LFU (Least Frequently Used): This algorithm evicts the document that has been least referenced. If there are some documents with the same reference count, LRU is used.
- LFF (Largest File First) evicts the documents with the largest size.
- LFU-DA (LFU-Dynamic Aging) [8]: This algorithm uses a variable that contains the "age" of the cache, i.e. the number of references to the least frequently used document. This aging method is used to avoid that documents that have been referenced very often in the past but are not popular any more will be maintained in cache and hence will not be evicted. When a new document is inserted or the referenced document is already in cache, the reference count of the document is added to the aging variable.
- GD-SIZE (Greedy Dual-Size) [9]: It utilizes a cost function to evaluate the documents. The value of a document is calculated as shown in Eq.1 where c_d is the transmission cost of document d and s_d is the size of d in bytes. The document evicted will be the one with the lowest evaluation function value. Two cost functions can be selected, the constant cost function ($c_d=1$) and the number of packets to transmit d.

$$V(d) = \frac{c_d}{s_d} \qquad \text{(Eq. 1)}$$

- GDSF (Greedy-Dual Size with Frequency) [10]: It is a modification of the Greedy-Dual-Size algorithm which also considers the reference count of documents. The function which weights each document is shown in Eq. 2, where f_d is the number of references to the document d.

$$V(d) = f_d \frac{c_d}{s_d} \qquad \text{(Eq. 2)}$$

- GD* (Greedy-Dual*) [11]: This algorithm is a modification of the GDSF algorithm taking into account the temporal correlation of references using a parameter β in the value function (Eq. 3). This parameter is a number between zero and

one that models the temporal correlation between two successive accesses to the same document.

$$V(d) = \left(f_d \frac{c_d}{s_d} \right)^{\frac{1}{\beta}} \quad \text{(Eq. 3)}$$

- RANDOM: The documents to be evicted are randomly selected using a uniform distribution, that is, all documents have the same probability of being evicted.
- CLIMB: In this replacement a queue such as LRU is maintained, but when there is a cache hit the document climbs a position in the queue instead of being moved to the end of the queue as occurs with LRU.
- CLIMB-C [12] is a randomized version of CLIMB. In this algorithm, when a document *d* that is in the cache is requested again, the probability of being climbed is presented in Eq. 4, where the denominator represents the maximum cost of the *N* documents that the cache contains.

$$P(d) \propto \frac{c_d}{\max\{c_1, c_2, ..., c_N\}} \quad 1 \le i \le N \quad \text{(Eq. 4)}$$

- CLIMB-S [12] works like CLIMB-C but using the Eq. 5 as the probability to climb a position in the queue when there is a cache hit.

$$P(d) \propto \frac{\min\{s_1, s_2, ..., s_N\}}{s_d} \quad 1 \le i \le N \quad \text{(Eq. 5)}$$

- C-LRU [13] and PART [14] are two replacement policies that classify the documents according to its size in various groups. Each group of documents is managed with a LRU queue. This simulator allows defining the groups of sizes and also allows applying any of the abovementioned replacement policies and not only LRU.
- As well as the previous C-LRU and PART, this simulator implements a schema that classifies the documents according to its content-type and it also allows applying any replacement policy [15].

3.2 Admission Control Policies implemented

The function of an admission control policy is to decide if a document has to be stored in the cache or if it is not worth storing because, for example, it is probable that it will not be referenced again or the size of the document will cause a great number of evictions in the cache. The cache module implements only two admission control policies that can be selected simultaneously:

- Threshold size: It allows defining a minimum and the maximum size of the documents to enter the cache.
- Minimum number of references: This policy defines the minimum number of times that a document has to be referenced before it could be cached.

3.3 The Measurements and Statistics module

This module is continuously monitoring the Admission Control and the Cache Module in order to obtain some useful data for statistics. This data can also be shown on screen in simulation time to visualize the evolution of the cache.

The Measurements and Statistics module checks the amount of storage space that is occupied, the number of documents stored in the cache, the number of evicted, modified and discarded documents, considering that a document is discarded when it has not even been stored in the cache because it is bigger than the cache and the amount of documents modified. To distinguish the modification of a document from the interruption of a transfer we compare the difference between sizes of successive requests to the same document. If the difference is less than 5% of the document size, we consider that the document has been modified and it has to be treated as a new document; otherwise a cancel is considered [16].

This module also monitories the amount of requests and bytes processed and served from the cache (cache hits) as well as the total number of documents rejected by the Admission Control module.

Based on the data collected, this module calculates the classical metrics to measure the cache performance HR and BHR:

- HR (Hit Ratio): It is defined as the total number of requests that cause a hit in the cache divided by the total number of requests processed by the cache.
- BHR (Byte Hit Ratio): It is defined as the summation of the document sizes that cause a hit in the cache divided by the size of the documents processed by the cache.

Furthermore, this simulator also calculates another metrics specifically designed to evaluate the performance of a cache with admission control policy. Eq. 6 and 7 define the NUHR (Not Unique Hit Ratio) and the NUBHR (Not Unique Byte Hit Ratio) [17] where *#Hits* is the total number of requests that cause a hit in the cache, *#Tot_req* is the number of requests that enter the cache and *#R_ok* is the number of request that were correctly discarded by the admission control policy, considering that a document has been correctly discarded when it is not requested again in the workload or it has been modified since the last time it was requested, i.e. the document reference is unique or it is modified before it is referenced again. Similarly, the terms of Eq. 7 are related to the size of the requested documents.

$$NUHR = \frac{\#Hits}{\#Tot\_req - \#R\_ok} \quad \text{(Eq. 6)}$$

$$NUBHR = \frac{\#S\_Hits}{\#S\_Tot\_req - \#S\_R\_ok} \quad \text{(Eq. 7)}$$

To measure the performance of the admission control policy the simulator calculates the ACHR (Access Control Hit Ratio) and the ACBHR (Access Control Byte Hit Ratio) defined in Eq. 8 and Eq. 9 respectively [17].

$$ACHR = \sqrt{\frac{\#R\_ok}{\#R\_Tot} \cdot \frac{\#Ac\_ok}{\#Ac\_Tot}} \quad \text{(Eq. 8)}$$

$$ACBHR = \sqrt{\frac{\#S\_R\_ok}{\#S\_R\_Tot} \cdot \frac{\#S\_Ac\_ok}{\#S\_Ac\_Tot}} \quad \text{(Eq. 9)}$$

Where *#R_ok* is the number of requests that were correctly rejected, *#R_Tot* is the number of rejections, *#Ac_ok* is the number of requests accepted that caused a hit in the cache and *#Ac_Tot* is the number of requests accepted by the access control. Similarly, the terms of Eq. 9 are related to the size of the requests.

When a simulation ends all the previous data and statistics are saved in a text file for further processing although this information can also be saved on each step of simulation to study the evolution of the cache in time.

4. The simulator interface

The simulator interface is organized in five tabs: General Configuration, Standard Configuration, Content-type Configuration, Size-based Configuration and Simulation Statistics. Some snapshots of the application can be seen at [18].

The General Configuration tab allows configuring the general parameters not specific for any replacement or admission control policy. This tab is utilized to select the trace files for filtering or simulating as well as the batch files that contain the parameters for the simulations. The percentage of the trace file employed to 'warm-up' the cache can also be selected.

The Standard Configuration tab is designed to configure the general parameters of the simulations when a replacement policy that utilizes only one queue is selected, i.e. not content-type or size based replacement policies. The size of the cache, the replacement policy and the admission policy as well as their corresponding parameters can be selected. This tab is also useful to create batch files simply pressing a button that stores the current configuration into a file.

In the Content-Type Configuration tab the parameters such as the content-types to consider and the size of each queue can be selected. Furthermore the replacement policy and the admission control for each queue can be assigned. Batch files can also be created.

The Size-based Configuration tab manages the ranges of sizes assigned to each queue, the size of the queues and the replacement policy applied to each of them.

Finally, the Simulation Statistics tab shows in simulation time all the information about the process as well as the statistics.

5. Implementation details

The simulator has been implemented using the Borland C++ Builder 2006 IDE and hence it widely utilizes the VCL (Visual Common Library) library for the interface and the classes to manage the queues.

The simplified class diagram of the application is shown in Fig. 3. The *Cache* class contains one object that implements one of the types of replacement policies, that is, *SimpleCache* that represents

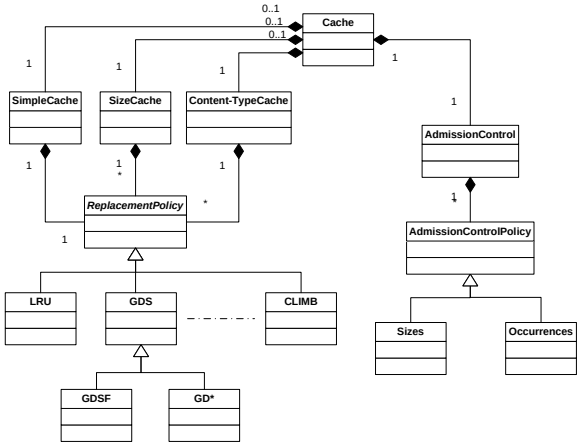

Figure 3. Simplified class diagram of the simulator

those replacement policies that utilizes an unique queue to store the documents, the *SizeCache* class that represents the replacement policy that divides the documents and stores them in different queues according to their size and the *Content-TypeCache* class that represents the replacement policy that stores the documents in a different queue according to its content-type. The *SimpleCache* class utilizes a *ReplacementPolicy* object to implement the replacement policy selected. On the contrary the *SizeCache* and *Content-TypeCache* classes make use of more than one *ReplacmentPolicy* object because each queue can operate with a different replacement policy.

The *ReplacementPolicy* class is an abstract class that implements most of the methods necessary to manage the data structures of the replacement policies. Table 1 summarizes the virtual methods that have to be or can be overridden for the subclasses of the *ReplacementPolicy* class.

The *ReplacementPolicy* class is a super class for the classes that implement the replacement policies such as the *LRU*, *GDS* or *CLIMB* classes. There is a class for each replacement policy implemented although the *GDS* class is an abstract class that implements a generalization of the GDS family (GDSize, GDSF and GD*).

The *Cache* class also contains an *AdmissionControl* object that manages the admission control policies assigned to the current simulation. In that way more than one *AdmissionControlPolicy* object can be assigned to the admission control. The *AdmissionControlPolicy* class is an abstract class and is a super class for the classes that implement each admission control policies. At the moment the *Sizes* and *Occurrences* classes implement the size based and number of occurrences based admission control policies respectively. These subclasses have to implement a method that defines if the document that has just reached the cache passes the admission control policy or not.

As can be observed from the previous descriptions the modular architecture of the application allows adding new functionalities such as new replacement policies or new admission control policies by just deriving from the base class and overriding the necessary methods.

Table 1. Virtual methods of the *ReplacementPolicy* class

Method	Description
void reorderNodes(Node*, int)=0	It implements what to do when a cache hit occurs
bool isSortedPolicy()=0	It sets if the queue that manages the cache is sorted
Node *insertNode(Node*, int &);	It inserts a document in the cache
void makeRoom(Node*,int &, Node*);	It defines the operations to evicts the documents
void insertSorted(Node*, int &);	It inserts a document in a sorted way
void valueInitNode(Node *);	It assigns an initial value for the document

6. Conclusions

In this work we have presented the functionalities, architecture and implementation details of a Windows based Web cache simulator that takes Squid traces as an input for simulating a great variety of replacements policies proposed for the Web. The simulator also implements an admission control module for deciding if the documents must be stored in the cache or not. The simulator also monitors the statistics of the simulations and calculates some performance metrics.

The application has been designed to be user friendly. It presents a set of tabs that groups the parameters necessary to configure the simulations. Those parameters can even be saved in a file that will be utilized for batch simulations.

Due to the object oriented structure developed using C++ this simulator can be easily extended implementing new replacement and admission control policies.

7. ACKNOWLEDGMENTS

We would like to thank Adela Isabel Fernández Anta for revising the syntax and grammar of this paper.

This work was partially supported by the public Project TEC2006-12211-C02-01.

8. REFERENCES

[1] http://www.isi.edu/nsnam/ns/

[2] Kurkowski, S., Camp, T., Colagrosso, M. 2005. MANET Simulation Studies: The Incredibles. ACM's Mobile Computing and Communications Review, vol. 9, no. 4, pp. 50-61, 2005.

[3] http://pdclab.cs.ucdavis.edu/qosweb/DavisSim.html

[4] http://www.cs.wisc.edu/~cao/webcache-simulator.html

[5] Cardenas, L.G. Sahuquillo, J. Pont, A. Gil, J.A. 2005. The multikey Web cache simulator: a platform for designing proxy cache management techniques. Proceeding of the 12th Euromicro Conference on Parallel, Distributed and Network-based Processing (A Coruña, Spain, February 11-13, 2004), 390- 397.

[6] http://www.ircache.net

[7] Zhang, X. 2000. Cachability of Web Objects. Technical Report 2000-19

[8] Arlitt, M., Williamson, C. 1997. Internet Web Servers: Workload Characterization and Performance Implications, IEEE/ACM Transactions on Networking

[9] Cao, P. 1997. Cost-Aware WWW Proxy Caching Algorithms, Proceedings USENIX Symposium on Internet Technologies and Systems

[10] Cherkasova, L. 1998. Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy. Technical Report HP Labs HPL-98-69.

[11] Jin, S., Bestabros, A. 2001. GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams, International' Journal of Computer Communications, Vol. 24, No. 2, pp. 174-183, February, 2001.

[12] Starobinski, D., Tse, D. 2001. Probabilistic Methods for Web Caching, Performance Evaluation, vol. 46, no. 2-3, pp. 125-37, October 2001.

[13] Haverkort, B.R., Khayari, R.A., Sadre, R. 2003. A Class-Based Least-Recently Used Caching Algorithm for World-Wide Web Proxies. Proceedings Computer Performance Evaluation / TOOLS 2003, pp. 273-290, 2003.

[14] Murta, C.D., Almeida, V., and Meira, Jr. W. 1998. Analyzing Performance of Partitioned Caches for the WWW. Proceedings of the Third International WWW Caching Workshop (Manchester, Great Britain, June 1998).

[15] Khayari, R.A., Best M., Lehmann, A. 2005. Impact of Document Types on the Performance of Caching Algorithms in WWW Proxies: A Trace Driven Simulation Study. Proceedings IEEE 19th International Conference on Advanced Information Networking and Applications (Tamkang University, Taiwan, March 2005).

[16] Arlitt, M. et al. 1999. Workload Characterization of a Web Proxy in a Cable Modem Environment, Hewlet-Packard Laboratories. Technical Report HPL-1999-48.

[17] González-Cañete, F.J., Triviño-Cabrera, A., Casilari, E. 2006. Two New Metrics to Evaluate the Performance of a Web Cache with Admission Control. Proceedings 13th IEEE Mediterranean Electrotechnical Conference (Benalmádena, Spain, March 2006).

[18] http://pc23te.dte.uma.es/Simutools/index.html