# Autonomy Oriented Load Balancing in Proxy Cache Servers

Kwok Ching Tsui, Jiming Liu, and Hiu Lo Liu

Department of Computer Science, Hong Kong Baptist University
Kowloon Tong, Kowloon, Hong Kong
{tsuikc,jiming,c8002287}@comp.hkbu.edu.hk

**Abstract** Proxy cache servers are used to deal with the increasing demand for information on the Internet by caching the frequently referenced web objects. It is common to have more than one proxy cache servers being installed in one local network. The problem of load balancing then arises as organizations want to utilise the resources in the best way. This article proposes two methods to tackle the load balancing problem. The two methods are based on the notion of autonomy oriented computation where entities in the model are allowed to make local decisions and they only need to interact with local neighbors.

## 1 Introduction

It is well known that the size of the Internet is increasing everyday. This can be measured both in terms of the number of web pages being put online, the number of people 'connected' to the global network, or the number of web servers being put online. All of these factors together contributed to one single phenomenon - traffic on the Internet is getting heavier and heavier. This has already put great stress on the networks that support all these requests for web objects.

One of the solutions to alleviate this problem is to store some frequently referenced web objects in a local machine so that the need to retrieve the same object from its host web site is reduced [3]. The Internet traffic is expected to be reduced and the response to user request is expected to be improved. The local machine is commonly known as the proxy cache server. It has a cache of considerable size for storing frequently used web objects. With the increasing demand for information from the Internet, multiple proxy cache servers in one site is commonly seen. The question now is what is the best strategy for assigning requests to these proxy cache servers. This is commonly known as the load balancing problem.

This article proposes two methods to tackle this load balancing problem. These methods are based on the notion of *autonomy oriented computation* (AOC). The basic idea is to allow the elements of a system to make decisions based on some simple local behavoir model that only need limited information about the system. A central control unit is definitely absent from algorithms following the AOC approach. This article will first describe some work related to the proxy cache server load balancing problem. A brief introduction to autonomy oriented computation follows, which will form the background to understand the two proposed methods. The next section is then devoted to describe the methods together with some experimental results. The article concludes with discussions on some interesting observations and future research directions.

## 2 Related Work

Proxy cache servers help to lower the demand on bandwidth and improve request turnaround time by storing up frequently referenced web objects in their local cache. However, the cache still has a limit to its physical capacity and objects in the cache need to be replaced so as to make room for new objects that need to be stored in the cache. A commonly used strategy is least recently used (LRU) where the oldest object in the cache is replaced first. There are a lot of work on improving this base strategy.

Another approach to alleviate the bandwidth load problem is by prefetching. As web usage patterns have peaks and troughs, this approach tries to spread the workload across the usage cycle. Hence, the average turnaround time per user request is lower. This approach relies heavily on reguarities in web usage pattern but cannot adapt well to daily variations as well as sudden changes in the network environment. See [9] for a more deailed discussion on the limitations of this approach.

Others have worked on structuring the proxy cache servers so as to improve the chance of locating the required object in one of the proxy cache servers. A common technique is to arrange a host of proxy cache servers in a hierarchical manner so that some proxy cache server, not necessarily on the same local area network, is the proxy of many other proxy cache servers while serving as the proxy of some local users [12]. This approach shortens the distance between the web server and the user who requests the object. However, some work has to be done on the design of the proxy hierarchy.

Besides caching web objects in local servers, they can be cached on the other side of the network closer to the web server [2]. In relation to this, adaptive web caching suggests arranging some proxy cache servers into overlapping clusters [15,16]. Note the need of a self-configuring cluster grouping and some algorithms has been proposed. Adaptive web cache concerns the restructuing of the Internet for sharing caching capability while we concerns the self-organisation of proxy cache servers within an organization, which might span over more than one physical location, where cooperation is a matter of strategy rather than relying on global concensus.

This article proposes two approaches based on the idea of self-organization of autonomous entities, or autonomy oriented computation. The main different between the AOC methods and the above approaches is that the architecture of the proxy cache servers is not predefined. The algorithms help the proxy cache servers to find the best distribution of load between them. Therefore, minimal human intervention is required and the system adapts well to rapid changes in usage requirement and network conditions.

## 3 Autonomy Oriented Computation

Autonomy oriented computation (AOC) [10] refers to computation algorithms that employ autonomy as the core model of any complex system. They aim at modeling, explaining and predicting the behavior of such systems, which are hard to model using top down approaches. Local interaction between the autonomous entities is the primary driving force of AOC. An abstracted version of some natural phenomenon is the starting point of AOC so that the problem at hand can be recasted. There are three different approaches to AOc and are applicable in different circumstances.

*AOC-by-fabrication* aims at replicating certain self-organized collective behavior observable in the real world. The operating mechanism is more or less known and may be simplified during the modeling process. Research in the Artificial Life community falls well into this approach. The natural application of this type of AOC algorithms is as a problem solving technique, such as optimization algorithm. Nature-inspired techniques such as Genetic Algorithm [8] and Ant system [4] are typical examples of such extension.

*AOC-by-prototyping* aims at understanding the operating mechanism underlying the system to be modeled by simulating the observed phenomenon, through modeling a society of autonomous entities. Examples include the study of traffic jams [7], crowd control [6,13] and web log analysis [11]. This approach to AOC relates to multiagent approach to complex system in Distributed Artificial Intelligence.

*AOC-by-self-discovery* aims at the discovery of a solution by employing autonomy. The trial-and-error process of the AOC-by-prototyping AOC is replaced by autonomy in the system. In other words, the distance measure between the desired emergent behavior and the current emergent behavior of the system in question becomes part of the environmental information used by the autonomy. Some evolutionary algorithms that exhibit self-adaptive capability are examples of this approach [1,5].

## 4 Self-Organized Approaches

The performance of a proxy cache server depends on the configuration of the machine, especially the cache size, the bandwidth of the connection dedicated to it, and its current workload, among others.

In the two self organized approaches to proxy cache servers selection, a centralized proxy virtual manager (VM) sits between users on the local network and the proxy cache servers. It is the proxy cache server known to the users. However, its real job is to route the user requests to the appropriate proxy cache server based on some knowledge learned from the proxy cache server. The following is a brief summary of the two approaches and details of them are given in the following two sections.

The first approach to load balancing is based on the likelihood that a particular server delivering an above average service. The likelihood is estimated from past job history and the average system performance. A reinforcement learning process ensures that good performers are credited while poor performers are suppressed. Note that one proxy cache server's chance of being chosen is a relative quality, depending on the actual performance of the other proxy cache servers.

The second approach to load balancing is learned from each user request. Specially, the system learns the characteristics of traffic pattern between the local network and the destination in relation to time. With this information, the proxy VM will try to find the best proxy cache server that can best serve the request on hand. A lattice of vetcors similar to a self-organized map is used to encode the traffic patterns and used to select an appropriate server.

### 4.1 Server Performance History Method

Each proxy cache server in this setup are initialized with a performance score, say 100. The objective of this approach is to adjust this score to reflect the current performance of the proxy cache servers. This score is bounded between 50 to 1000. The discussion of the score update rules will justify the need to have these boundaries. Formally, let $S_i(y_i)$ be the performance score of proxy cache server $i$ after $y_i$ requests have been completed, then $S_i(0) = 100$, and $50 \leq S_i(y_i) \leq 1000$.

When the system first start to function, jobs are assigned to the proxy cache servers randomly until all proxy cache servers have been used at least oncce. The proxy VM then starts to use the performance score to assign requests. Specifically, the probability $(P(i))$ of a proxy cache server being selected is:

$$P(i) = \frac{S_i^{'}}{\sum_{j=1}^{n} S_j^{'}}, S_i^{'} = S_i + \delta \tag{1}$$

where $n$ is the number of proxy cache servers available and $\delta \leq \pm 0.05$ is a random number added to avoid the system from being stuck in a sub-optimal condition. In other words, some random noise is added to the selection process to avoid the dominance of a super-fit server.

When a request is completed, the proxy VM updates the system average transfer rate $T(x)$ and proxy cache server transfer rate $T_i(y_i)$, where $x$ is the total number of requests served by all proxy cache servers alltogether and $y_i$ is the total number of requests served by proxy cache server $i$. All transfer rates are measured in bytes per second.

$$T(x) = \frac{T(x-1) \times (x-1) + \text{transfer rate for } x\text{th job}}{x} \tag{2}$$

$$T_i(y_i) = \frac{\text{size of the file received}}{\text{time used}} \tag{3}$$

***self-organize-by-history:***
**begin**
*initialize:* **for** each proxy cache server $i$ **do** (*initialize*)
$score_i \leftarrow 100$
**enddo**
*main-loop* **while** there is user request **do** (*main-loop*)
**if** not all servers are assigned a job once **then**
randomly select a server
**else**
select server $s$ based on $\frac{score_s}{\sum_j score_j}$
**endif**
assign job to server $s$
calculate server transfer rate $T_s$
calculate system transfer rate $T$
$score_s \leftarrow score_s + (score_{max} - score_s) \times (\frac{T_s}{T-1})$
**if** $score_s < 50$ **then**
$score_s \leftarrow 50$
**endif**
**enddo**
**end**

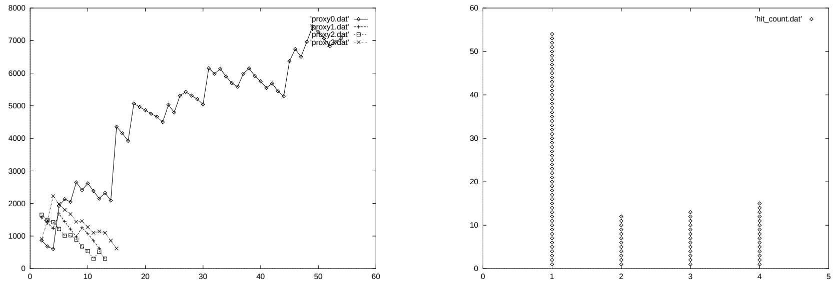
**Figure1.** The algorithm for loan balancing based on history

The above transfer rate are used to update the performanace score of the proxy cache servers as follows:

$$S_i(y_i) = S_i(y_i - 1) + (S_{max} - S_i(y_i - 1)) \times (\frac{T_i(y_i)}{T(x)} - 1) \quad (4)$$

The general idea of the above upate rule is that if the transfer rate achieved by the proxy cache server processing the current request is higher than the system average transfer rate, then its score should be increased. The increment is proportional to the distance between the proxy cache server's score and the maximum allowable score. Otherwise, it should be decreased to lower its likelihood of being chosen. Note that the transfer rate is an aggregated represention of a proxy cache server's overall performance and workload. As a busy proxy cache server will have a corresponding lower transfer rate. A lower bound is needed in this update rule in order to prevent the performance score from falling below zero, which will make the selection probability (equation 1) becomes negative. The upper bound is there to insure that a 'super-fit' proxy cache server will not have its score increased indefinitely and dwarfing the rest proxy cache servers. The update rule is used every time a request is completed and will be used for the whole lifetime of the system. Figure 1 presents an overview of the algorithm.

**Experimental Results** Some artificially generated data has been used which will generate a maximum of 2 requests per second to a randomly selected destination. A total of 21 days of traffic and four proxy cache servers are simulated




(a) performance score (b) number of jobs assigned

**Figure2.** Changes in proxy cache server performance score and number of job assignments. proxy cache server 1 has the best configuration and hence has highest score and have been assigned the largest number of requests

in the experiments. The difference in perfromance is simulated by adding delays to the roundtrip time for each request. Figure 2a shows that strength of the four proxy cache servers over time. It shows that the performance strength reinforcement algorithm is able to help the proxy VM to differentiate between fast and slow servers. In figure 2b, it can be observed that proxy cache server with highest specification (1 in this case) is given the most number of job.

### 4.2 Route Transfer Pattern Method

It is well known that traffic on the Internet has peak access times within a day and within a month. Therefore, apart from the hardware limitations of the proxy cache server, the traffic on the Internet, particularly the traffic on a certain zone from which a local user wants to get information, will greatly affect the overall transfer rate achievable by a proxy cache server. The load balancing method based on route transfer rate attempts to address this issue.

Assuming there are $N$ major nodes the Internet. The proxy VM maintains an array of size $N$, one entry for each node, which contains a pointer to the a 10 by 10 map of vectors (MOV). They are used to learn the traffic pattern of each route between the local network and the major nodes. Each vector contains the following information:

- time of the day
- an array of length equal to the number of proxy cache servers, each containing the latest transfer rate of the corresponding proxy
- the best proxy cache server
- a 'ready' flag, which is set to false initially

All vectors in all the maps are initialized with randomized data and all *ready* flags are set to false, signifying that the vector should not be used for decsision making. Therefore, the proxy VM will have to make some random decision while building up the MOV. When a request is received by the proxy VM, it locates the MOV using the requested URL. The *time* entry in all vectors in the chosen MOV are then searched to find the *closest* one (the winner) that match the time of the request. If the winner is ready to be used, the proxy cache server number contained in the best *proxy cache server slot* will be assigned the job. The transfer rate of the chosen proxy cache server is then updated when the completed. The time is also set as follows:

$$time_w = time_w + \alpha \times (time_{req} - time_w) \tag{5}$$

where the subscripts $w$ and $req$ represents the winner and request respectively and $0 < \alpha < 1$. This time update rule modifies the chosen vector to the time of the request with some noise. When all the transfer rates of the proxy cache servers are available, the ready flag is set to true and the best proxy cache server slot is updated.

The next round of update concerns the neighbors of the vector. Vectors that are one and two positions away from the current vector in the 10 by 10 grid are considered as neighbours. Their time and transfer rate are then updated according to the following rule:

$$time_n = time_n + K^{distance_n} \times \alpha \times (time_{req} - time_{win}) \tag{6}$$

where the subscript $n$ represents neighbor, $K < 1$ is a contant and $distance_n$ is the distance of the neighbor from the winner. The transfer rate, best proxy cache server and ready flag are updated in the same manner as the winner. Note that the neighbors recieve a discounted time change proportional to its distance from the winner. This effectively help to build up the MOV with localized time information. This update process runs continuously so that the MOV can adapt to the traffic pattern quickly. Figure 3 presents the overall algorithm of this method.

**Experimental Results** An artficial world with 100 randomly located nodes is set up to test the effectiveness of the route traffic pattern method. The existence of a link between any two nodes $u$ and $v$ is determined by the following probability [14]:

$$P(u, v) = \alpha e^{\frac{-d}{\beta L}} \tag{7}$$

where $d$ is the Euclidean distance between nodes $u$ and $v$, $L$ is the maximun distance between any two nodes, and $0 \leq \alpha, \beta \leq 1$. In addition, traffic of each link is characterised by one of the following functions are $y = ((sin(x)/x) + 0.3)/1.3$, $y = (x * sin(x) + 5)/7$, $y = (x * sin(x) + 3.5)/7$ and $y = (sin(x) + 1.1)/2.1$. Furthermore, the discount factor $\alpha$ in equation 5 is set at 0.95. User requests are generated in the similar way as in the previous experiment.

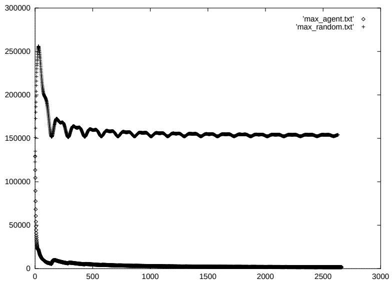
In order to verify the success rate of the proposed method, the experiment is run 6 times, one with the proxy VM equiped with the propsoed method and five

```
self-organize-by-route-traffic-pattern:
begin
initialize:        for each 10 by 10 map in the MOV do
                       for each vector in the map do
                           time ← random value between 0000 and 2400
                           ready ← false
                       enddo
                   enddo
main-loop:         while there is user request do
                           locate the MOV corresponding to the requested URL
                           if ready flag = false then
                              randomly select a server s
                           else
                              select server s from best_server slot
                           endif
                           assign job to server s
                           calculate server transfer rate T_s
                           update-server
                           update-neighbor
                   enddo
update-server:     update corresponding server_transfer_rate slot
                   if all server_transfer_rate slots are filled then
                      find the best transfer rate
                      update best_server slot
                   endif
                   end
update-neighbor:   while no all neighbor are updated do
                           time ← time + distance ×α× (request_time - time)
                   update-server
                   enddo
                   end
end
```

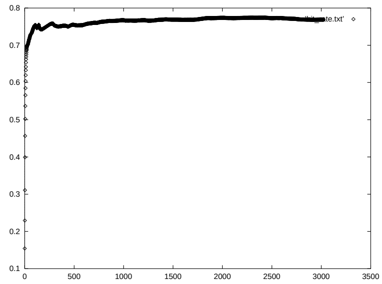
**Figure3.** The algorithm for the route traffic pattern method

others are run with only one of the five proxy cache servers present. The same set of test data is used throughout. The controlled experiments allow us to find the best proxy cache server, i.e. with the fastest transfer rate, to be identified. A baseline is established by assigning job randomly to one of the five proxy cache servers. These results are shown in figure 4.

Figure 4a shows the difference in transfer rate between the best case (found by the control experiments) and that achieved by the proposed method (lower line), and between the best case and the baseline (upper line). It can be noted that the proposed method tracks the best case performance closely after only a brief learning period. Figure 4b is another view of how successful the proposed method. This figure shows a 'hit rate' achieved by the proposed method. A 'hit' is counted everytime the proposed method selects the same proxy cache server




(a) difference in transfer rates



(b) percentage of marking correct choices

**Figure4.** Peformance of the route traffic pattern method. (a) shows the difference in transfer rate between the proposed method and hte ideal case, and between the random choice and the ideal case. (b) shows the percentage of times when the proposed method has chosen the best proxy cache server as determined by the control experiments.

as the best case. The proposed method managed to make the right choice more than 75% of the time. We can confidently conclude that the proposed method can quickly learn the characteristics of the traffic on the Internet.

## 5 Discussion and Future Research

Two autonomy oriented load balancing methods have been proposed and described in this article. Experimental results on some artificially generated traffic and network setup are encouraging.

The first load balancing method based on server performance history makes decision is able to distinguish between the good and bad performer and allocate jobs accordingly. However, in the event of a proxy cache server failure of link problem, the proxy VM will not be able to stop sending job to the faulty server unless it is aware of the situation. One possible action is to stop sending jobs to this server suspected to be faulty. An alternative is to add a time-out condition to the transfer rate update mechanism. When a time-out is triggered, the proxy VM will just assume a very low transfer rate and update the statistics. If the problem persists, the server selection rule (equation 1) can provide a means to lower the priority of the faulty servers after a small number of job assignments. This would allow the system to respond to the disaster gracefully.

For the load balancing method based on learning the Internet traffic pattern, the algorithm can successfully learning the traffic characteristics by updating the information on the map of vectors (MOV). The neighborhood update mechanism allows the system to learn from one data point and extrapolate into areas in close proximity. The side-effect is that every vector will be modified and some previously learned information will be 'un-learned' in some sense. This can be observed from the transfer rate graph (figure 4.2) where cyclical fluctuation with decreasing amplitude is found. Similar fluctuation can also be observed in the baseline case but with much bigger amplitude.

The next step is to test these two methods using some real data to verify their performance.

## References


1. T. Bäck. Self-adaptation. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C7.1:1–15. Institute of Physics Publishing and Oxford University Press, 1997.
2. R. B. Bunt, D. L. Eager, G. M. Oster, and C. L. Williamson. Achieving load balance and effective caching in clustered web servers. In *Proceedings of the Fourth International WWW Caching Workshop*, 1999.
3. R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: The devil is in the details. *Performance Evaluation Review*, 26(3):11–15, December 1998.
4. M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperative agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 26(1):1–13, 1996.
5. B. Freisleben. Metaevolutionary approaches. In T. Bäck, D. B. Fogel, and Z. Michalewicz, editors, *Handbook of Evolutionary Computation*, pages C7.2:1–8. Institute of Physics Publishing and Oxford University Press, 1997.
6. D. Helbing, I. Farkas, and T. Vicsek. Simulating dynamic features of escape panic. *Nature*, 407:487–490, 28 September 2000.
7. D. Helbing and B. A. Huberman. Coherent movng states in highway traffic. *Nature*, 396:738–740, 24/31 December 1998.
8. J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, 1992.
9. Q. Jacobson and P. Cao. Potential and limits of web prefetching between low-bandwidth clients and proxies. In *Proceedings of Third International WWW Caching Workshop*, 1998.
10. J. Liu and K. C. Tsui. Introduction to autonomy oriented computation. In *Proceedings of 1st International Workshop on Autonomy Oriented Computation*, pages 1–11, 2001.
11. J. Liu and S. Zhang. Unveiling the origin of web surfing regularities. In *Proceedings of the INET 2001 Conference*, June 5-8 2001.
12. P. Rodriguez, C. Spanner, and E. W. Biersack. Web caching architectures: Hierarchical and distributed caching. In *Proceedings of the Fourth International WWW Caching Workshop*, 1999.
13. G. K. Still. *Crowd Dynamics*. PhD thesis, Mathematics Department, Warwick University, August 2000.
14. E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of INFOCOM 96*, 1996.
15. L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the Second International WWW Caching Workshop*, 1997.
16. L. Zhang, S. Michel, K. Nguyen, and A. Rosenstein. Adaptive web caching: Towards a new global caching architecture. In *Proceedings of the Third International WWW Caching Workshop*, 1998.