Self-Organized Load Balancing in Proxy Servers: Algorithms and Performance

Kwok Ching Tsui (tsuikc@comp.hkbu.edu.hk)
Department of Computer Science, Hong Kong Baptist University

Jiming Liu (jiming@comp.hkbu.edu.hk)
Department of Computer Science, Hong Kong Baptist University

Markus J. Kaiser (mjk@gmx.it)
Department of Computer Science, Hong Kong Baptist University

Abstract. Proxy servers are common solutions to relieve organizational networks from heavy traffic by storing the most frequently referenced web objects in their local cache. These proxies are commonly known as cooperative proxy systems and are usually organized in such a way as to optimize the utilization of their storage capacity. However, the design of the organizational structure of such proxy system depends heavily on the designer's knowledge of the network's performance. This article describes three methods to tackle this load balancing problem. They allow the self-organization of proxy servers by modeling each server as an autonomous entity that can make local decisions based on the traffic pattern it has served.

Keywords: autonomy oriented computation, self-organization, load balancing, adaptive proxy server

1. Introduction

It is well known that the size of the Internet is increasing everyday. This can be measured in terms of the number of web pages being added, the number of people 'connected', and the number of web servers being put online. All 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 (Barish and Obraczka, 2000; Caceres et al., 1998). The Internet traffic is expected to reduce and the response to user request is expected to improve (Abdulla, 1998; Duska et al., 1997). The local machine is commonly known as the proxy server. It has a storage capacity of considerable size for storing frequently used web objects. With the increasing demand for information from the Internet, multiple proxy servers in one site is common. These cooperative proxies usually share the knowledge about their cached data and allow faster document fetching through request forwarding (Wang, 1999), but maintaining a stable tradeoff for ideal performance remains tricky.

Another way to relieve the network congestion problem is to install a 'reverse proxy' at the origin server end (Bunt et al., 1999; Cardellini et al., 1999). This will speed up server response while balancing the load between multiple servers that might be mirrors of the main server. Various approaches have been suggested. Refer to (Bryhni et al., 2000) for detailed comparison. The advantage of this approach is transparency to user, no matter the server is located locally or distributed across the Internet. However, this solution does not bring the web objects closer to the clients.

The load balancing problem can be tackled in many different ways. A knowledge-intensive approach relies heavily on the experience of network designers who 'understand' the network traffic behavior and try to configure the proxy servers appropriately. This is an unreliable approach as the network traffic can be unpredictable. A slightly automated approach is to define certain heuristics in, say, an expert system that can react to the changes. This approach takes the human out of the loop but still suffers the same adaptability problem. A highly adaptive solution is needed that can react quickly to the change in traffic behavior, such as sudden burst of requests or break down of certain part of the Internet. It should also be able to learn the different modes of operation online so that the system does not need to be taken offline.

The successful discovery of such an adaptive system will allow organizations to deliver better quality of service to local users. The same benefit also applies to subscribers of Internet service providers if such an adaptive load balancing strategy is adopted. This article describes three methods to tackle this load balancing problem. They allow the self-organization of proxy servers by modeling each server as an autonomous entity that can make local decisions based on the traffic pattern it has served.

The three proposed methods can broadly be divided into centralized and decentralized approaches. The *centralized history (CH) method* makes use of the transfer rate of each request to decide which proxy can provide the fastest turnaround time for the next job. The *route transfer pattern (RTP) method* learns from the past history to build a virtual map of traffic flow conditions of the major routes on the Internet at different times of the day. The map information is then used to predict the best path for a request at a particular time of the day. The two methods require a central executive to collate information and route requests to proxies. Experimental results show that self-organization can be achieved (Tsui et al., 2001). The drawback of the centralized approach is that a bottleneck and a single point of failure is created by the central executive. The decentralized approach - the *decentralized history (DH) method* - attempts to overcome this problem by removing the central executive and put a decision maker in every proxy (Kaiser et al., 2002b) regarding whether it should fetch a requested object or forward the request to another proxy.

This article will first describe some work related to the proxy server load balancing problem. The details of the three methods will then be described together with experimental results on their performance. The article concludes with discussions on some interesting observations and future research directions.

2. Related Work

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

Existing cooperative proxy systems can be organized in hierarchical and distributed manners (Dykes et al., 1999; Raunak, 1999). The hierarchical approach is based on the *Internet Caching Protocol* (ICP) with a fixed hierarchy. A page not in the local cache of a proxy server is first requested from neighboring proxies on the same hierarchy level. Root proxy in the hierarchy will be queried if requests are not resolved locally and they continue to climb the hierarchy until the request objects are found. This often lead to a bottleneck situation at the main root server.

The distributed approach is usually based on a hashing algorithm like the *Cache Array Routing Protocol* (CARP) (Cohen et al., 1997). Each requested page is mapped to exactly one proxy in the proxy array in a hashing system and will either be resolved by the local cache or requested from the origin server. Hashing-based allocations can be widely seen as the ideal way to find cached web pages, due to the fact that their location is pre-defined. Their major drawback is inflexibility and poor adaptability.

Adaptive Web Caching (Zhang et al., 1998) and *CacheMesh* (Wang and Crowcroft, 1997) try to overcome specific performance bottlenecks. For example, Adaptive Web Caching dynamically creates proxy groups combined with data multicasting, while CacheMesh computes the routing protocol based on exchanged routing information. Both approaches can still be considered experimental. Yet other approaches like pre-fetching, reverse proxies and active proxies can usually be seen as further improvements to speed up the performance of a general hierarchical or distributed infrastructure and go hand in hand with our proposed self-organizing approach. See (Jacobson and Cao, 1998) for a more detailed discussion on the limitations of the pre-fetching approach.

Others have work on structuring the proxy servers so as to improve the chance of locating the required object in one of the proxy servers. Common technique is to arrange a host of proxy servers in a hierarchical manner so that some proxy server, not necessarily on the same local area network, is the proxy of many other proxy servers while serving as the proxy of some local users (Rodriguez et al., 1999). 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.

Wu and Yu (1999) have done some work on load balancing at the client side using proxy servers. They emphasized on tuning the commonly used hashing algorithm for load distribution. Researchers from MIT, on the other hand, have proposed a new hashing algorithm called consistent hashing to improve caching performance when resources are added from time to time (Karger et al., 1999).

The three proposed methods emphasize the self-organization of autonomous proxy servers. The basic idea is to allow the elements of a system to make decisions based on some simple local behavior model that only need limited information about the system – a notion central to the computational paradigm known as *Autonomy Oriented Computation (AOC)* (Liu and Tsui, 2001). Unlike common hashing algorithms, routing of requests are not pre-defined. In this sense, central control unit is absent in the proposed methods. Moreover, proxy systems using the proposed methods do not require any prior knowledge about any hardware differences between the proxy server, nor do they need to know in advance the client traffic pattern.

3. Centralized History (CH) Method

The performance of a proxy 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.

self-organize-by-centralized-history (CH):
begin
  for each proxy server i **do** *[initialize]*
    $score_i \leftarrow 100$
  enddo

  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**
      $score_s \leftarrow 50$
    else if $score_s > 1000$ **then**
      $score_s \leftarrow 1000$
    endif
    endif
  enddo
end

Figure 1. The CH algorithm for load balancing based on history.

In this self organized approach to proxy servers selection, a centralized proxy virtual manager (VM) sits between users on the local network and the proxy servers. It is the proxy server known to the users. However, its real job is to route the user requests to the appropriate proxy server based on some knowledge learned from the proxy server.

The main idea of this approach 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. Reinforcement learning algorithms (Kaelbling et al., 1996; Sutton and Barto, 1998) are well-known machine learning techniques based on feedback from the environment in which the system is operating. They have been applied to a wide variety of applications. Figure 1 presents an overview of the proposed algorithm based on server performance history.

Each proxy server in this setup is initialized with a performance score, say 100, and is bounded between 50 to 1000. This score is adjusted to reflect the current performance of the proxy servers. Formally, let $S_i(y_i)$ be the performance score of proxy 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 starts to function, jobs are assigned to the proxy servers randomly until all of them have been used at least once. The proxy VM then starts to use the performance score to assign requests. Specifically, the probability ($P(i)$) of a proxy server i being selected is:

$$P(i) = \frac{S_i^{'}}{\sum_{j=1}^{n} S_j^{'}}, S_i^{'} = \begin{cases} 1000 & \text{if } S_i + \alpha > 1000, \\ S_i + \alpha & \text{otherwise} \end{cases} \tag{1}$$

where n is the number of proxy servers available and $\alpha \leq \pm 0.05$ is a random number added to avoid the system from being stuck in a suboptimal condition, or 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 server transfer rate $T_i(y_i)$, where x is the total number of requests served by all proxy servers all together and y_i is the total number of requests served by proxy server i.

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

The above transfer rates are used to update the performance score of the proxy servers as follows:

$$S_i(y_i) = S_i(y_i - 1) + (S_{max} - S_i(y_i - 1)) \times (\frac{T_i(y_i)}{T(x)} - 1) \tag{3}$$

The general idea of the above update rule is that if the transfer rate achieved by the proxy 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 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 representation of a proxy server's overall performance and workload. As a busy proxy 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 server will not have its score increased indefinitely and dwarfing the rest proxy servers.

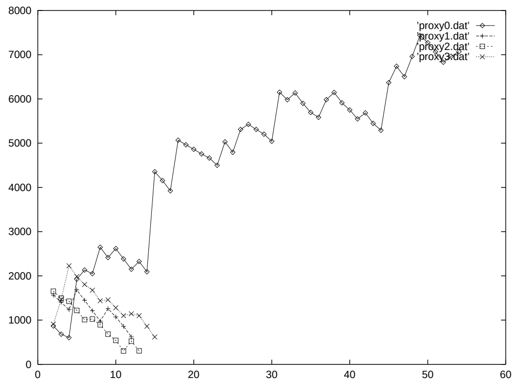

Figure 2. Changes in proxy server performance score. Proxy server 1 has the best configuration and hence has highest score.

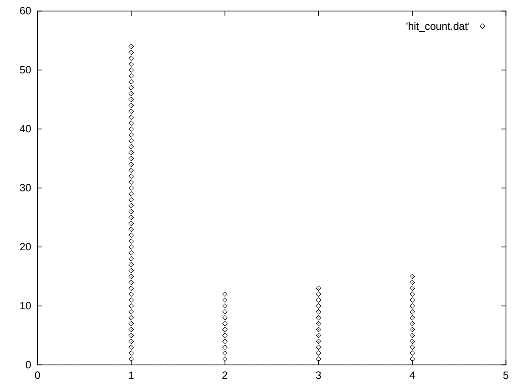

Figure 3. Number of job assignments in the route transfer pattern method simulation. Proxy server 1 has been assigned the largest number of requests because it has the best configuration.

3.1. Experimental Results

Some artificially generated data has been used which will generate a maximum of 2 requests per second to a randomly selected destination. Four proxy servers are simulated in the experiments. The difference in performance is simulated by adding delays to the round trip time for each request. Figure 2 shows that score of the four proxy 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 3, it can be observed that proxy server with the highest specification (server 1 in this case) is given the most number of jobs.

4. Route Transfer Pattern (RTP) Method

We has shown in the previous section that the virtual manager is able to make load distribution decisions based on the performance of the proxy servers. The transfer rate information that the virtual manager uses reflects, to certain extent, the required performance assessment, i.e. the workload of each server and the rate data is transferred from the origin server. Another kind of information that is useful to the virtual manager is the condition with the various routes between the proxy and the origin server.

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 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 server. The load balancing method based on route transfer rate attempts to address this issue. Figure 4 presents the overall algorithm of this method.

Specifically, 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 server that can best serve the request on hand. A lattice of vector similar to a self-organized map (SOM) (Kohonen, 2001) is used to encode the traffic patterns and used to select an appropriate server. SOM is an unsupervised clustering algorithm and has been applied to numerous problems. The most noteworthy of all is the work on knowledge discovery by clustering over one million Usenet newsgroup articles (Kohonen et al., 2000).

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 servers, each containing the latest transfer rate of the corresponding proxy
- the best proxy 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

```
self-organize-by-route-traffic-pattern (RTP):
begin
    for each 10 by 10 map in the MOV do [initialize]
        for each vector in the map do
            time ← random value between 0000 and 2400
            ready ← false
        enddo
    enddo

    while there is user request do [main-loop]
        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

    if all server_transfer_rate slots are filled then [update-server]
        find the best transfer rate
        update best_server slot
    endif

    while no all neighbor are updated do [update-neighbor]
        time ← time + distance ×β× (request_time - time)
        update-server
    enddo
end
```

Figure 4. The RTP algorithm for the route traffic pattern method.

not be used for decision 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 server number contained in the best *proxy server slot* will be assigned the job. The transfer rate of the chosen proxy server is then updated when the job is completed. The time is also set as follows:

$$time_w = time_w + \beta \times (time_{req} - time_w) \qquad (4)$$

where the subscripts w and req represents the winner and request respectively and $0 < \beta < 1$. 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 servers are available, the ready flag is set to true and the best proxy 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 neighbors. Their time and transfer rate are then updated according to the following rule:

$$time_n = time_n + K^{distance_n} \times \beta \times (time_{req} - time_{win}) \qquad (5)$$

where the subscript n represents neighbor, $K < 1$ is a constant and $distance_n$ is the distance of the neighbor from the winner. The transfer rate, best proxy server and ready flag are updated in the same manner as the winner. Note that the neighbors receive a discounted time change proportional to its distance from the winner. This effectively helps 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.

4.1. Experimental Results

An artificial 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 (Zegura et al., 1996):

$$P(u, v) = \gamma e^{\frac{-d}{\delta L}} \qquad (6)$$

where d is the Euclidean distance between nodes u and v, L is the maximum distance between any two nodes, and $0 \leq \gamma, \delta \leq 1$. In addition, traffic of each link is characterized by one of the following functions: $y = ((sin(x)/x)+0.3)/1.3$, $y = (x*sin(x)+5)/7$, $y = (x*sin(x)+3.5)/7$ and $y = (sin(x) + 1.1)/2.1$. Furthermore, the discount factor β in Equation 4 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 equipped with the proposed method and five others are run with only one of the five proxy servers present. The same set of test data is used throughout.

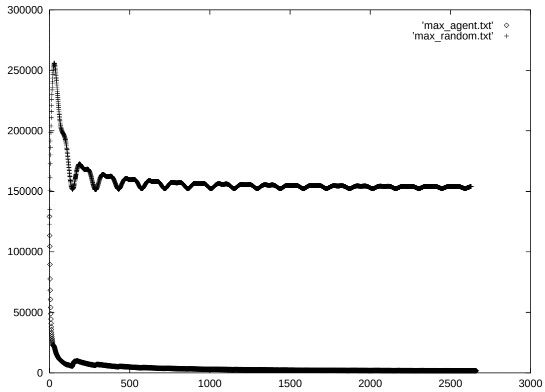

Figure 5. Difference in transfer rate between the proposed method and the ideal case, and between the random choice and the ideal case.

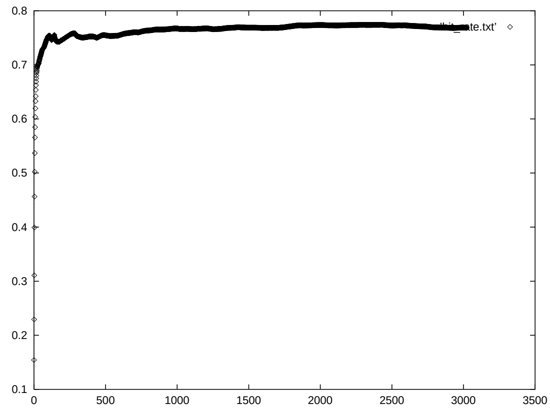

Figure 6. The percentage of times when the proposed method has chosen the best proxy server as determined by the control experiments.

The controlled experiments allow us to find the best proxy server, i.e. with the fastest transfer rate, to be identified. A baseline is established by assigning jobs randomly to one of the five proxy servers.

Figure 5 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 6 is another view of how successful the proposed method is. This figure shows a hit rate achieved by the proposed method. A 'hit' is counted every time the proposed method selects the same proxy server 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. Distributed History (DH) Method

The previous two sections described two centralized load balancing methods where a virtual manager sits between the clients and the proxy servers. While the virtual manager has been shown to be making appropriate decision based on either the transfer rate history or the route transfer pattern, it represents a potential fatal limitation of the methods. To put it simply, the virtual manager becomes the bottleneck of all web requests and it is a potential single point of failure. This section presents a decentralized approach that models after the centralized history method.

In real life a self-organizing proxy architecture based on autonomous proxies can be compared to a simple market buyer-seller environment where the buyer (client) acts as a customer who always choose the same shop for all his/her requests (similar to pre-defined proxies in web browsers). The maximization of the market depends completely on the sellers (autonomous proxies). Each shop has a limited local stock (cache) and the goal is to maximize the customer satisfaction. There are two ways in supplying good service, either by having the requested item in the local stock or by knowing the most suitable way to supply the item. Additionally each proxy tries to attract more requests (not customer but other proxies) by specializing on a certain category of items (clustering). This decision is usually made based on the current specialization of the shop and the incoming request pattern.

The above seller-buyer scenario in a dynamic market is a very fitting analogy for the goal to maximize the hit rate for proxy requests in a distributed autonomous proxy environment.We assume there is an effective way to compare and categorize incoming requests. In the context of hashing algorithms, a simple modulo function, for example, can define similarity, over the requested URL. But such a simple categorization is insufficient.

Figure 7 shows the core steps in a simplified version of the distributed history algorithm without emphasize on the evaluation subroutines. This method models after the centralized history method by taking server performance history as the basis of decision making. In addition, each proxy makes its own decision regarding request handling and caching.

5.1. Components of the System

In this section we will describe in detail the components of the self-organizing autonomous proxies using the distributed history method.

```
self-organize-by-distributed-history (DH):
begin
  while (there is client request) do
    if (page exists)
      update cache using LRU
      send data to client
      randomly select a requester
    else if (hops > max_hops)
        forward request to web server
      else
        forward function
      endif
      feedback function
      selective caching
      send data to requester
    endif
  enddo
end
```

Figure 7. The DH algorithm of the distributed history method.

All proxies have the same amount of cache and use LRU as their cache replacement algorithm.

5.1.1. *Routing Table*

The routing table stores numerical values to be used by the forwarding function to select an ideal forwarding path of an unresolved request. The table contains one row for each known category and the columns represent the list of all known proxies. In other words, the proxy has multiple paths to choose from to fetch the requested object for each category. Each table entry represents the accumulated average value for the last n requests forwarded through this path and is calculated by the request feedback function.

5.1.2. *Forwarding Function*

The forwarding component is triggered when a request could not be resolved by the locally cached data and the routing table is consulted to find the most suitable path for the unresolved request. After identifying the main domain of the request, it looks up the specific row and calculates a weight for each entry with respect to the direct proxy/server communication. The weight for a direct proxy/server communication always receives the value 1.0 and represents a reference value for comparison. Paths that are usually two times faster than a direct communication will receive the weight 2, and a path that is half as fast as the direct communication will receive the weight 0.5.

$$W_{c,i} = \begin{cases} 1.0 & \text{if } R_{c,i} = R_{c,direct}, \\ \frac{R_{c,i} - \max_{z \in \mathcal{C}}\{R_{z,i}\}}{R_{c,direct} - \max_{z \in \mathcal{C}}\{R_{z,i}\}} & \text{if } R_{c,i} > R_{c,direct}, \\ \left(\frac{R_{c,i} - R_{c,direct}}{\min_{z \in \mathcal{C}}\{R_{z,i}\} - R_{c,direct}} \times \epsilon\right)^{\zeta} & \text{otherwise.} \end{cases} \quad (7)$$

where $R_{c,i}$ and $W_{c,i}$ are the route table and weight table entry for proxy i for category c, respectively, $\mathcal{C}$ is the set of all categories, $\epsilon > 10$ and $\zeta > 50$ are scaling factors.

Note that a high value in the routing table always leads to a low value in the calculated weight table to avoid this path for future selections. Similar to equation 1, a small random noise value is added to avoid the system to become stuck in local minima. All weights are then normalized and used as a probability for path selection.

5.1.3. *Feedback Function*

The feedback function is executed after a forwarding proxy received the returned data object. It will use the received latency value to update the appropriate cell in the routing table. Basically, each value in the routing table is the average latency of the last n requests for this path. The formula is based on the simple weighted average calculation for the tracking of a non-stationary problem.

$$R_{c,i} = \frac{R_{c,i} \times \eta + L_x}{\eta + 1} \quad (8)$$

where L_x is the latency for request x and $\eta < n$ is the number of requests prior to x.

5.1.4. *Selective Caching*

The selective caching component decides if the data from a resolved request will be added to the local cache or discarded. It is introduced to promote efficient clustering based on the overall traffic pattern and not just on the recently observed requests. Selective caching uses the current cache status for the requested category and calculates a ratio with respect to the category with the highest page assignments. The ratio is adapted with a small random noise, to allow categories with a low ratio to leave their minimum in case of a sudden change in focus on a certain category.

$$P_{cache}(y) = \begin{cases} rand() & \text{if } Q_c < Q_{min}, \\ \frac{Q_c - \min\limits_{z \in \mathcal{C}}\{Q_z\}}{\max\limits_{z \in \mathcal{C}}\{Q_z\} - \min\limits_{z \in \mathcal{C}}\{Q_z\}} + rand() & \text{otherwise.} \end{cases} \tag{9}$$

where $P_{cache}(y)$ is the probability to cache an object y, Q_c is the number of cached objects for category c, Q_{min} is a system parameter setting the lower bound for the number of objects to be cached for any category.

5.2. Experimental Results

The simulated proxy system has 10 proxies connecting to 20 remote servers. We tested the system's ability to balance its load and cluster using two types of synthetic data. We assume that:

- going to any proxy within the array of proxies, even by doing a maximum number of hops, is always faster than requesting the data from the origin server;
- full knowledge and full connectivity within the proxy system, so that each autonomous proxy knows about every other autonomous proxy and they are capable of connecting to each other without network restrictions;
- each proxy is equally able to connect to the 20 servers with the same latency value.

Resent research has shown that a Zipf distribution is very suitable to describe the request pattern on a proxy server (Breslau et al., 1998). In our simulation, clients inject requests for all existing servers based on a Zipf distribution. We then artificially create hot spot situations. A hot spot is defined in such a way that the algorithm first chooses a subset of origin servers, which shall be part of the hot spot, and the Zipf distribution will be placed upon this subset in such a way that chosen servers will be the most requested ones, and the remaining set of servers receive almost no requests.

The baseline for comparison are two hashing algorithms. The *Hashing and Caching* (H&C) algorithm is similar to the CARP algorithm where all unresolved requests are forwarded to the location as defined by the hashing algorithm. After the requested web object is returned, the forwarding proxy will *always cache* it. The *Hashing with Selective Caching* (HSC) algorithm will still forward unsolved requests to other proxies. However, the returned object will only enter its cache if the object was originally assigned to it by the hashing algorithm.

5.2.1. *Evaluation Metrics*

Three criteria for evaluating the performance of the distributed history method and the hashing algorithms are as follows:

Average Hit Rate The number of requests that were resolved locally by the cooperative proxy system. A high average hit rate means the system is capable of 'moving' the most needed web objects closer to the clients.

Average Hops Rate The number hops that are needed to resolve a request. The action of forwarding a request between client/proxy, proxy/proxy or proxy/server constitutes a hop. The lower the average hops rate, the lesser the time it takes to resolve the requested object.

Average Load Variance The variance in the number of requests a proxy experienced throughout the simulation. A very small variance value represents an almost equal load for all proxies.

5.2.2. *Normal Zipf Requests*

Zipf-distributed requests mean that the probability of any web object being requested, when arranged in descending order, follows the power law. figure 8 shows the average hit rate achieved by H&C, HSC and the distributed history (DH) method. The input traffic is generated over a 10,000 object space using a Zipf distribution with power -1.0995. The two hashing algorithms have achieved very quickly a good hit rate. This is somewhat expected as the hashing algorithms are designed for such purpose. However, it should be pointed out that the HSC algorithm actually performs better than H&C. This proves selective caching is a good strategy. The proposed method perform slightly better than H&C but takes a bit longer to achieve that result.

The performance of the two hashing algorithms on the average hop rate follow a trend similar to the above (Figure 9). The distributed history method has surely adapts to the environment and shortened the average hop count from the initial value of 6.5 to slightly below 5. However, it lags behind the two hashing algorithms by 1 or 2 hop counts.

On the average load variance (Figure 10), H&C achieved the best result. This means H&C managed to produce the best load distribution, a result significantly better than HSC. It is interesting to note that the distributed history method performs mid-way between the two hashing algorithms.

This experiment shows that the distributed history method is able to distribute requests quite evenly between the available proxies and

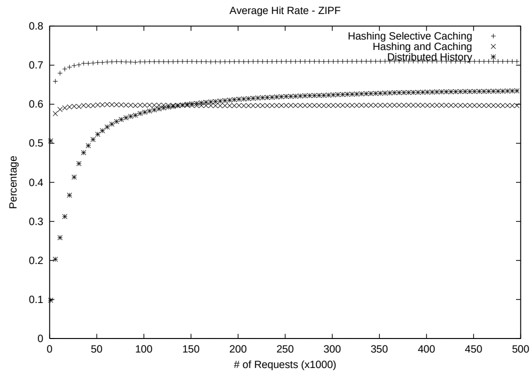

Figure 8. Average hit rates of HSC, H&C and DH algorithms under Zipf-distributed traffic.

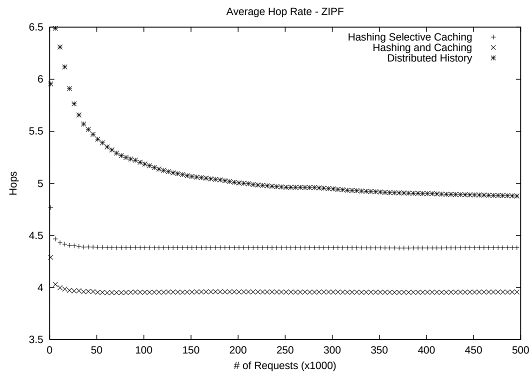

Figure 9. Average number of hops required by HSC, H&C and DH algorithms under Zipf-distributed traffic.

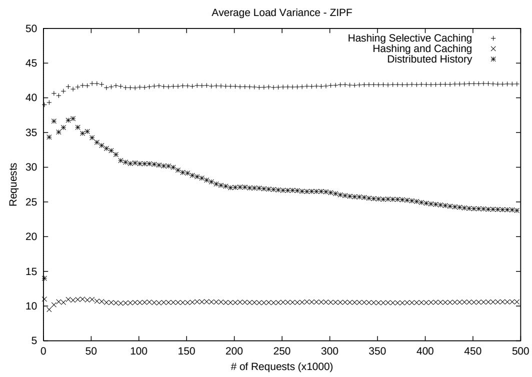

Figure 10. Average load variance of HSC, H&C and DH algorithms under Zipf-distributed traffic.

have managed to move most of the frequently requested web objects closer to the clients. The performance is comparable with the hashing algorithms. The tradeoff is that it needs a learning period to achieve that level of performance. The fact that it always requires more inter-proxy forwarding can be ignored as proxy/proxy connection is generally very fast.

5.2.3. *Hot Spot Situation*

Further to the Zipf distribution, we simulate the scenario where client requests are concentrated on certain group of objects at certain time. This is similar to the situation where all people starts to browse their favouriate newspaper sites mostly in the morning and move on to stock brokers' sites when trading begins. The hashing algorithms will not perform well under this situations as requests are directed to a limited number of proxy servers only. In addition, as the hot spot migrates to other categories or domains, objects residing in the cache of those proxy servers that were active before will quickly become obsolete and the whole proxy system will need to start from scratch to accumulate the 'useful' objects. On the other hand, the distributed history method should perform well because it does not have a fixed request assignment scheme. The input traffic is generated uniformly over 1,000 objects that have the same hash code.

In fact, the relative performance of two hashing algorithms in hot spot situation is very similar to that in normal Zipf traffic. The distributed history method, on the other hand, has made significant improvement. The only exception is that H&C achieved a higher average hit rate than HSC. This is due to the fact that only a few proxies have the required objects while reserving cache space for other objects, which will not come until a long while later when the hot spots have changed.

Figure 11 shows that, with the proposed method, a proxy system is able to achieve an average hit rate around 15% higher than H&C (the better of the two hashing algorithms). It is expected that the distributed history method requires more inter-proxy communication (Figure 12). The most significant improvement lies with the load distribution where proxies using the distributed history method has the smallest load variance when compared with the hashing algorithms (Figure 13).

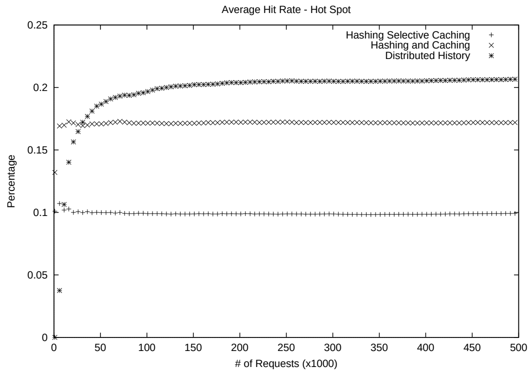

Figure 11. Average hit rates of HSC, H&C and DH algorithms in hot spot situations.

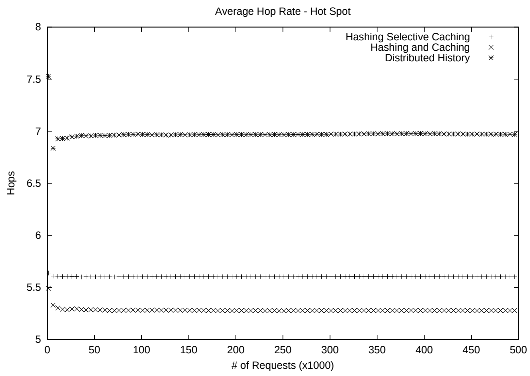

Figure 12. Average number of hops required by HSC, H&C and DH algorithms in hot spot situations.

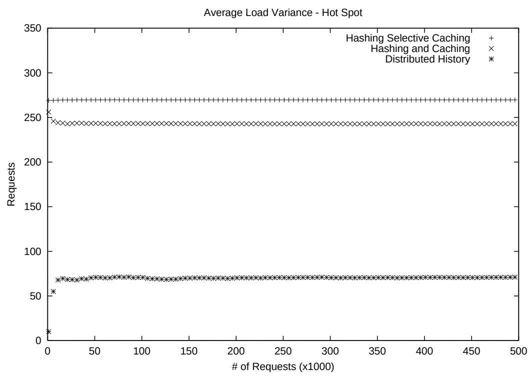

Figure 13. Average load variance of HSC, H&C and DH algorithms in hot spot situations.

6. Discussion and Conclusion

This article has described three methods for solving the load balancing problem in client-side cooperative proxy systems. The proposed methods modelled each proxy as an autonomous entity that makes request routing decisions based on its own local knowledge. Unlike the hashing algorithms, the proposed methods do not require any prior knowledge about client request patterns or hardware setup. In addition, there is no need for fixed mapping between clients and proxy servers. As a result, flexible assignment of clients requests to proxy servers can be achieved, and hence balancing the load between the proxy servers.

The centralized approaches, namely *Centralized History method* and *Route Transfer Pattern method*, make predictions based on traffic history either through the proxies or on the Internet. Experimental results show that the proxy system is able to self-organize itself into a state where job assignment is desirable. The only problem is the need for a virtual manager, which is a potential single point of failure.

Building on the experience of the centralized history method, the *Distributed History method* delagates the decision making responsibility to all proxies and take away the virtual manager. In addition, request forwarding and selective caching schemes are added. Each proxy maintains a routing table that states which proxy is responsible for the requests directed to certain web server. The routing table differs between proxies and will be updated continously. Proxies use the routing table information to decide whether to handle a request itself or forward it. Web objects are only selected to enter the cache of the proxy that has primary responsibility for it. Objects passing through a proxy not responsible for it will not be cached.

The strength of the distributed history method lies it its ability to adapt the routing table as the request pattern changes. It is particularly useful in situations where client requests are directed to few domains within a period of time and then move to other areas of interest. This is typical at certain time of day when everybody in an organization wishes to browse information from a handful of information providers. We call this a hot spot situation.

The decentralized method is tested against normal Zipf-distributed artificial data as well as an artificially created hot spot situation. Experimental results show that the distributed history method performs competitively when compared with the commonly used hashing algorithms. We can also conclude that the distributed history method copes very well with the hot spot situation at the expense of insignificant inter-proxy communication.

Load balancing under hot spot situation is perhaps just one of the many scenarios we find in real life. Hardware failure or addition of proxies are also likely to occur. We plan to test the three self-organized load balancing methods described in this article under more situations and using real web log data. In the current implementation, the routing table is of unlimited size. We are working on improving the efficiency of the distributed history method by limiting the size of the routing table (Kaiser et al., 2002a). Further work is also planned to implement the load balancing scheme in an experimental proxy system so that it performance can be assessed in real life situations.

Acknowledgements

The authors would like to thank Hiu Lo Liu for running some of the simulations reported here. This project is partially funded by grants from the Hong Kong Baptist University (FRG/00-01/I-03 & FRG/01-02/I-37).

References

Abdulla, G.: 1998, 'Analysis and Modeling of World Wide Web Traffic'. Ph.D. thesis, Virginia Polytechnic Institute and State University.

Barish, G. and K. Obraczka: May 2000, 'World Wide Web Caching: Trends and Techniques'. *IEEE Communications Magazine*.

Breslau, L., P. Cao, L. Fan, G. Phillips, and S. Shenker: 1998, 'Web Caching and Zipf-Lie Distributions: Evidence and Implications'. Technical Report 1371, Computer Science Dept., University of Wisconsin-Madison.

Bryhni, H., E. Klovning, and O. Kure: 2000, 'A Comparison of Load Balancing Techniques for Scalable Web Servers'. *IEEE Network* **14**(4), 58–64.

Bunt, R. B., D. L. Eager, G. M. Oster, and C. L. Williamson: 1999, 'Achieving Load Balance and Effective Caching in Clustered Web Servers'. In: *Proceedings of the Fourth International WWWCaching Workshop*.

Caceres, R., F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich: 1998, 'Web Proxy Caching: The Devil is in the Details'. *Performance Evaluation Review* **26**(3), 11–15.

Cardellini, V., M. Colajanni, and P. S. Yu: 1999, 'Load Balancing on Web-Server Systems'. *IEEE Internet Computing* **3**(3), 28–39.

Cohen, J., N. Phadnis, V. Valloppillil, and K. W. Ross: 1997, 'Cache Array Routing Protocol V.1.1'.

Duska, B. M., D. Marwood, and M. J. Feeley: 1997, 'The Measured Access Characteristics of World-Wide-Web Client Proxy Caches'. In: *Proceedings of USENIX Symposium on Internet Technology and Systems*.

Dykes, S. G., C. L. Jeffery, and S. Das: 1999, 'Taxonomy and Design for Distributed Web Caching'. In: *Proceedings of the Hawaii International Conference on System Science*.

Jacobson, Q. and P. Cao: 1998, 'Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies'. In: *Proceedings of Third International WWWCaching Workshop*.

Kaelbling, L. P., M. L. Littman, and A. P. Moore: 1996, 'Reinforcement Learning: A Survey'. *Journal of Artificial Intelligence Research* **4**, 237–285.

Kaiser, M. J., K. C. Tsui, and J. Liu: 2002a, 'Adaptive Distributed Caching'. In: *Proceedings of the 2002 Congress on Evolutionary Computation*.

Kaiser, M. J., K. C. Tsui, and J. Liu: 2002b, 'Self-Organized Autonomous Web Proxies'. In: *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*.

Karger, D., T. Leighton, D. Lewin, and A. Sherman: 1999, 'Web Caching with Consistent Hashing'. In: *Proceedings of the WWW8 Conference*.

Kohonen, T.: 2001, *Self-Organizing Maps*. Berlin: Springer-Verlag, 3rd edition.

Kohonen, T., S. Kaski, K. Lagus, J. Salojärvi, V. Paatero, and A. Saarela: 2000, 'Self Organization of a Massive Document Collection'. *IEEE Transactions on Neural Networks* **11**(3), 574–585. Special Issue on Neural Networks for Data Mining and Knowledge Discovery.

Liu, J. and K. C. Tsui: 2001, 'Introduction to Autonomy Oriented Computation'. In: *Proceedings of 1st International Workshop on Autonomy Oriented Computation*. pp. 1–11.

Raunak, M. S.: December 1999, 'A Survey of Cooperative Caching'. Technical report.

Rodriguez, P., C. Spanner, and E. W. Biersack: 1999, 'Web Caching Architectures: Hierarchical and Distributed Caching'. In: *Proceedings of the Fourth International WWWCaching Workshop*.

Sutton, R. S. and A. G. Barto: 1998, *Reinforcement Learning: An Introduction*. MIT Press.

Tsui, K. C., J. Liu, and H. L. Liu: 2001, 'Autonomy Oriented Load Balancing in Proxy Cache Servers'. In: *Web Intelligence: Research and Development. Proceedings of First Asia-Pacific Conference on Web Intelligence*. pp. 115–124, Springer-Verlag.

Wang, J.: 1999, 'A Survey of Web Caching Schemes for the Internet'. *ACM Computer Communication Review* **29**(5), 36–46.

Wang, Z. and J. Crowcroft: June 1997, 'CacheMesh: A Distributed Cache System for World Wide Web'. In: *Proceedings of NLANR Web Cache Workshop*.

Wu, K. and P. Yu: 1999, 'Load Balancing and Hot Spot Relief for Hash Routing Among a Collection of Proxy Caches'. In: *Proceedings of the 19th International Conference on Distributed Computing Systems*.

Zegura, E., K. Calvert, and S. Bhattacharjee: 1996, 'How to Model an Internetwork'. In: *Proceedings of INFOCOM 96*.

Zhang, L., S. Michel, K. Nguyen, and A. Rosenstein: 1998, 'Adaptive Web Caching: Towards a New Global Caching Architecture'. In: *Proceedings of the Third International WWWCaching Workshop*.