Adaptive Distributed Caching

Markus J. Kaiser, Kwok Ching Tsui and Jiming Liu

Department of Computer Science
Hong Kong Baptist University
Kowloon Tong, Kowloon, Hong Kong

Email: mjk@gmx.it, {tsuikc, jiming}@comp.hkbu.edu.hk

Abstract - This paper introduces an adaptive algorithm for distributed caching based on the idea of autonomous proxy caches without the usage of a central coordinator or broadcasting protocol. We will show, that the algorithm outperforms existing approaches based on hashing algorithms in hot-spot scenarios and common power-law request patterns.

I. INTRODUCTION

The Internet is growing exponentially and web caches have been shown to be a feasible way to reduce the overall network traffic. A cache is usually placed between the requesting clients and the resolving origin server, storing transferred objects for future retrieval.

If a proxy is not able to resolve an incoming request it can either forward the request directly to the origin server or query a neighboring proxy for the needed object. The idea that a proxy can forward requests to a different proxy cache lead to research in the area of cooperative proxies. Cooperative proxies try to combine their individual caches in such away that a maximum cache-usage is achieved while acting transparently as one single load-balanced proxy cache [3].

A. Hierarchical vs. Hashing

Common approaches for cooperative systems encompass mostly solutions based on hierarchical cache structures or classical hashing algorithms [4]. While in the hierarchical structure unresolved requests get forwarded to sibling caches, using ICP, and to the parent cache up the hierarchy with a high number of additional querying messages, the hashing approach, like found in CARP [7], computes the exact location of the requested object and forwards the request to maximal one additional proxy minimizing the number of querying messages to one [5][6].

On the other side, the hierarchical approach leads usually to multiple copies of the same object in different locations, minimizing the overall cache usage but allows good load balancing in hot-spot situations. The hashing approach assigns one exact location to each object, leading to a maximum of cache usage but lacks the flexibility to load-balance hot-spots through multiple copies.

Additionally, both algorithms are more or less flexible to cope with changes in the underlying infrastructure like the removal or addition of proxy resources. In the hierarchical structure the changed infrastructure needs to be represented in a changed proxy hierarchy, through user interference. Some solutions for hashing based algorithms are able to slowly cope with changes in the proxy set but in general the predefined hashing function does not fancy changes in the infrastructure.

B. Ideal Cooperation

This short comparison shows the major objectives of a distributed cooperative proxy environment: *Object Allocation*, *Cache Usage*, *Load Balancing* and *Reactivity towards the Infrastructure*.

1) Object allocation, describes the challenge of finding the exact location of a cached object with a minimum number of messages or forwarding hops (a hop occurs for a message between client/proxy, proxy/proxy and proxy/server) (Figure 1)

2) Cache-Usage, the goal is to maximize the usage of the overall cache space in all cooperative proxies through minimization of redundant data. Ideally, each object uses the cache space of exactly one proxy with a low fluctuation to promote ideal allocation.

3) Load Balancing, describes the idea of ideal content dissemination in regard to the current request pattern. In contrast to maximized cache-usage, hot-spot situations demand the storage of multiple copies of the same object in different locations.

4) Reactivity towards the infrastructure, an ideal distributed proxy system should be able to scale with newly added resources, and furthermore should fail gracefully with the removal of resources.

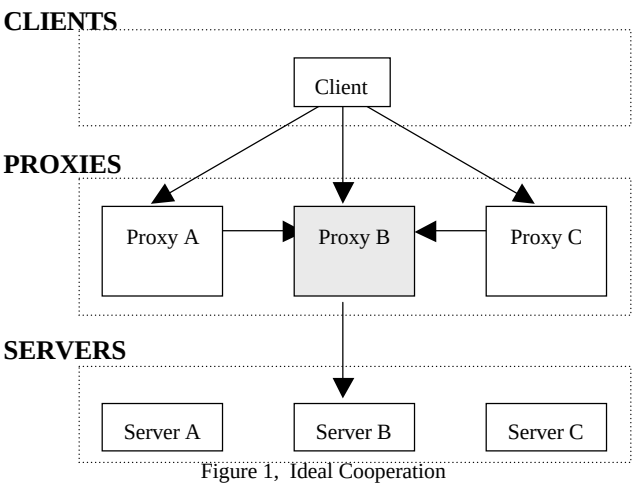

Figure 1, Ideal Cooperation

C. Problem Definition

A cooperative caching system is based on a set of M individual proxies with a total cache capacity C. A set of N different objects, where usually N > C, needs to be assigned to the total cache-space in such a way that each object has ideally one location allowing a fast allocation and maximal cache-usage (for simplification purposes we assume all objects of equal size).

In the worst-case scenario, all objects are requested with a uniform distribution, not allowing a LRU cache to focus on a stable set objects. Recent work has shown, that the experienced request pattern on a proxy server, follows the power-law distribution [11], where a small number of most objects gets requested most of the time and a high number of objects gets requested very rarely. In the other extreme the small set of mostly requested objects, hot documents, is located on the same proxy, leading to an bottleneck scenario in pre-assigned hashing algorithm with lack of maintaining multiple objects.

An ideal cooperative proxy environment maintains multiple copies of the same hot object on different locations to load balance the proxy latency. Therefore we are looking for an adaptive distributed algorithm that is able to maximize the average hit-rate and minimizes the average number of hops needed to find the document. Additionally, This constraint creates a conflict between the storage of multiple objects decreasing the average number of hops needed to find an object

D. Related & Previous Work

As described earlier, the change from a general almost uniform request pattern to a more focused hotspot situation, is the major problem, that existing approaches fail to cope with. A hierarchical structure always maintains multiple copies, good for load balancing, but minimizes the cache-usage and comes with a costly search algorithm [9]. Hashing always maintains exactly one copy of each object, allows a minimum search but is not able to avoid a proxy overload in hotspot situations through multiple copies [8].

In our previous work we tried to cope with the problematic through the simple usage of a central coordinator [1], which assigns incoming requests accordingly to a specific proxy, maintaining ideal load balancing with non-existing search costs. Due to the fact, that the central coordinator establishes a single-point of failure and is usually not able to scale with a growing number of proxies we focused in our second work on the creation of adaptive proxy servers [2], that will self-organize themselves to minimize the search cost and which is able to load-balance hotspots. We build the approach on the idea of data clustering in regard to the objects URL-main-domain were over time a specific proxy became responsible for a certain number of domain data. We experienced that in hot spot situation, when only data of one domain is requested each proxy will maintain copies of the requested objects, but the approach was not able to minimize redundant data between proxies.

F. Proposal

In the following chapters we introduce an adaptive distributed caching algorithm based on the principles of self-organization in autonomous objects. We will show that the algorithm is able to reduce the number multiple copies in situations where N > C and the request pattern is not emphasizing on a group hot documents. On the other side, if such a hotspot situation occurs, N < C, our algorithm will allow the overall system to maintain multiple copies to balance the load between the cooperative proxies.

Looking at the special case of N=C, after a learning period, the system of adaptive proxies should stabilize in such a way, that each proxy becomes devoted for a exact set of objects, and all proxies will agree on the same location for a specific object, minimizing the search costs, without the usage of an central coordinator or a broadcasting protocol.

F. Self-Organization

The major challenge of our approach lies in the constraint that proxies are not allowed to exchange knowledge directly but have to make autonomous decisions based on local picture [12][13]. Each autonomous proxy has to be proactive towards the ideal system state through request–response evaluation and request pattern observation. For this purpose the proposed algorithm comes with two adaptive components, which will reinforce each other in the global picture: *selective caching* & *request forwarding*

1) Selective Caching, instead of using the very common LRU algorithm to replace cache content with every arriving object, the adaptive proxy will try to stabilize its cache content in the currently most often requested objects. This stabilization allows neighboring proxies to make good assumptions about the cached content [10].

2) Request Forwarding, assigns an unresolved request to the most suitable proxy, which is most often returning a cache hit. The more proxies agree on the same location, the more the targeted proxy will experience a change in its request pattern and cache the highly requested objects.

The last step will lead to a further stabilization of the targeted cache and will lead to reinforcement through the attraction of more requests for the cached data. The overall system will move to a stable self-organized state in regard to the current request pattern.

II. ADAPTIVE DISTRIBUTED CACHING ALGORITHM (ADC)

Each proxy contains an implementation of the exact same algorithm with the following components: Mapping Table, Request Forwarding, Selective Caching, Cache Replacement.

A. *Mapping Table*

The proxy stores information for each requested object in the mapping table. The table rows represent the entry for one object, while the table columns represent the attributes: *Object ID, Location ID, Stability, Time Stamp, Average Time.*

```
Table.getCurrentAverage ( objectID, time ) {
  return ( ( 3 * Table.getAverageTime ( objectID ) )
      + ( time – Table.getTimeStamp ( objectID ) ) ) / 4;
}
Table.updateAverage ( objectID, time ) {
   value = ( ( 3 * Table.getAverageTime ( objectID ) )
      + ( time – Table.getTimeStamp ( objectID ) ) ) / 4;
  Table.setAverage ( objectID, value );
  Table.setTimeStamp ( objectID, time );
}
```

Figure 2, Adaptive Mapping Table Functions

1) Object ID, a unique identifier for the whole system for this particular object. In an Internet Proxy environment, the object URL usually represents the unique identifier. For our simulations, we used a plain numerical value to identify different objects.

2) Location ID, is the unique identifier for a specific location (proxy ID). When the local proxy was not able to resolve the request, it will forward the request to the location specified by the location ID in the mapping table. When the location ID is equal to the current proxy, the request will be forwarded to the origin server.

3) Stability, this value represents a probability value for the likelihood to use the defined location or to do a random forwarding. The higher the stability value, the more likely it is, that the request will be forwarded to the assigned location (Table 1).

Value	Situation
0.02	Initial value for an unknown object
0.98	Assigned value after first initialization

TABLE 1: STABILITY VALUES

4) TimeStamp (T_{last}), is a local time value representing the last time when this object was requested. In our simulation, each proxy counts the number of received requests and the current time is always the current request number.

5) Average Time ($T_{average}$), represents an estimate for the average time between two requests. To make the system adaptive to different request patterns a simple non-stationary formula is used. Figure 2, shows the major update functions where the update formula will be used.

$$T_{current} = \frac{3 \bullet T_{average} + (T_{now} - T_{last})}{4} \quad (1)$$

B. *Request Forwarding*

When an incoming request was not resolved by the locally cached data, an alternative route needs to be selected based on the information in the mapping table. The proposed algorithm will simply accept the assigned location with a probability of the stability value. If the assigned location is equal to the current proxy or if a loop got detected, the request will be forwarded directly to the origin server.

```
IF RequestTable.containsRequest ( requestID ) {
     Forwarding = THIS;
} ELSE {
     RequestTable.addRequest ( requestID );
     IF rand.probability < Table.getStability ( objectID ) {
          Forwarding = Table.getLocation ( objectID );
     ELSE
          Forwarding = select random proxy;
}
IF ( Forwarding == THIS ) {
     Forwarding = origin server;
}
```

Figure 3, Request Forwarding Algorithm

A newly requested object will always be initialized with a very small stability value (0.02) leading to a high chance for a random search across multiple proxies to resolve the request. In our current algorithm, we did not limit the number of hops for the search path and the search will continue until either the origin server is randomly selected or a path loop is detected.

Loop Detection occurs through the fact, that each request uses a unique request ID and all forwarding proxies will store information about the unresolved request. When a proxy experiences the same request ID twice, a loop has occurred and the request will be forwarded to the origin server. The request table is used to allow resolved objects to traverse the same path back to the requesting client. In case of a loop occurrence additional data needs to be stored to avoid that a double incoming request does not update the proxy cache and mapping table twice.

C. *Response Evaluation*

After either a proxy or the origin server resolved the request, the package traverses the same way back to the requesting client and leaves a feedback trail in each passing proxy. Each proxy will individually decide whether to store or discard the received data before forwarding the resolved object to the next proxy on the feedback path.

Before the package migrates its way back, it will be marked with the ID of the resolving proxy or the ID of the proxy

which forwarded the request to the server. In this way, all proxies on the forwarding path will be informed about the most suitable location of this object, for future requests. In case, where the request got resolved by the server and not by a proxy, the first proxy on the way back that caches the object will overwrite the current resolver ID with its own ID, to inform the remaining proxies on the response path about its ability to resolve future requests much faster (Figure 4).

```
IF  (Response.getResolver() == null)  | |
   (Response.isCached() && Table.getCached(objectID)) {
      Response.setResolver ( THIS );
      Response.setCached (Table.getCached(objectID) );
      Table.setLocation ( objectID, THIS )
} ELSE {
     Table.setLocation ( objectID, Response.getResolver () );
}
```

Figure 4, Response Evaluation

Essentially the information distribution is very similar to a multicasting to a selected group of proxies with two advantages, first the multicast message is not initialized from a central point but forwarded from proxy to proxy, and secondly a message with the resolved objects is sent anyways and there is no need to initialize additional messages to distribute the information.

D. Selective Caching

To allow the cache to stabilize in a set of objects, we follow the idea of selective caching. Each returned object from a forwarded request has to fulfill a certain criteria to enter the cache. In the described algorithm we want the cache to stabilize in the most frequently used objects, even if the frequency ratios of the different objects have almost the same value. For comparison and evaluation, we use the average request time for a certain object plus the current time difference to the last request to allow object aging.

```
IF  ( Table.getCurrentAverage ( objectID, time )  <
     0.5 · Table.getCurrentAverage ( cacheMaxID, time )) {
            Cache.Remove (cacheMaxID );
            Table.setCached (cacheMaxID, FALSE );
            Cache.AddSorted ( objectID );
            Table.setCached ( objectID, TRUE );
}
```

Figure 5, Selective Caching

For a new object to enter the cache, it has to pass the cached object with the worst request frequency in regard to the current time. The focus on the current time allows objects to age and gives the system the ability to adapt to changing request patterns. Simulations have shown, that the stabilization process will speed-up if we use just a fraction of the worst frequency.

The cache is always sorted by the current time values of the stored objects. If a cache hit occurs on a specific object, the object will slowly climb to the top of the sorted cache while the other objects age and move to the bottom. If a new object is able to enter the cache, the cached object with the worst request frequency will be removed, leading to stabilization in the most frequently used objects.

III. EXPERIMENTATION

In the following experiments we compared our algorithm to two types of hashing algorithms and we will show that our approach is able to outperform both of them in significant points like hit-rate, number of hops and load balancing. We do not use the hierarchical approach for comparison, because, hashing comes with an ideal object to location resolution which is one of the major objectives of our work.

A. Algorithms

We compare our adaptive distributed caching (ADC) algorithm against two types of hashing algorithm deferring in their caching strategy.

1) Hashing with cache everything (H&C) as found in CARP. In this approach, each unresolved request will be forwarded to the location computed by the hashing function. After the object is returned, the forwarding proxy will always store every arriving object in its local cache based on the Least Recently Used Algorithm.

2) Hashing with cache assigned objects only (HNC), In this algorithm, each unresolved request will be forwarded to the location computed by the hashing function. After the object is returned, the forwarding proxy will only store objects, which are assigned to its location based on the hashing function. The replacement algorithm is based on the Least Recently Used Approach.

B. Attributes

After the simulation we will compare, the hit-rate, number of hops and the load variance. Each simulation measuring point encompasses 1000 different request.

1) Average Hit Rate, represents the number of requests that were resolved by the cooperative caching system in regard to the number of requests. The higher the average – hit rate the better performs the algorithm.

2) Average Hops Rate, represents the number hops that are needed to resolve a request. The distance between a client/proxy, proxy/proxy or proxy/server defines one hop. The lower the average hops rate, the less time it takes to resolve the requested object. The evaluation of this parameter will give us an idea if the algorithm also suits in global scenarios where the number of transferred messages matters.

3) Average Load Variance, represents the variance in regard the number of requests a proxy experiences in the

measuring period. The variance value is computed by the well-known statistical variance method where a very small variance value represents an almost equal load for all proxies

C. ZIPF - Distribution

In the first simulation, we ran, all three algorithms against a typical ZIPF-distribution over a set of 10000 web-objects. A set of 10 proxies provides a total cache space of 1000 objects.

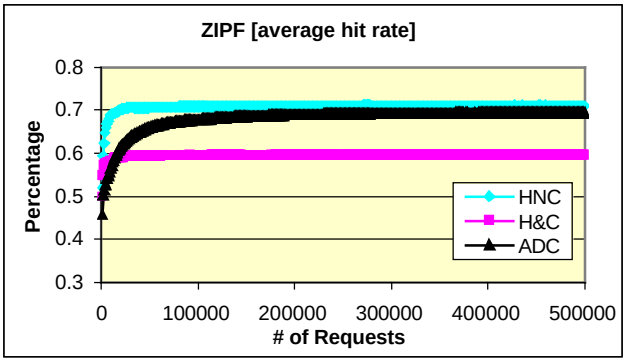

Figure 6, ZIPF - average hit rate

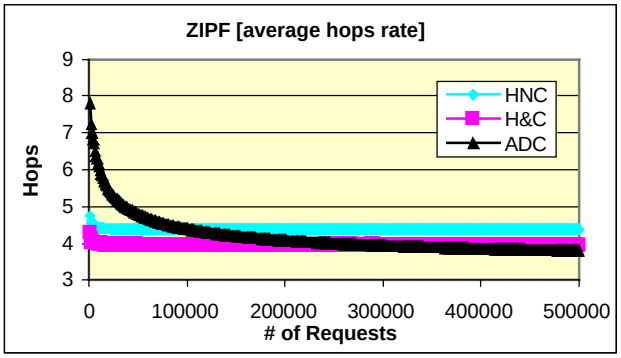

Figure 7, ZIPF - average hops rate

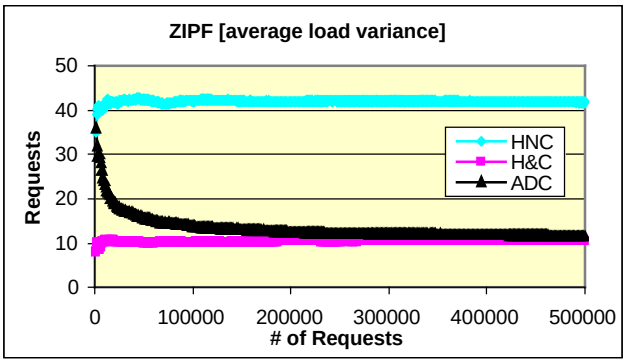

Figure 8, ZIPF – average load variance

1) Average Hit Rate (Figure 7), shows that the hashing approach with storage of only assigned objects (HNC) outperforms by far the hit-rate of the hashing approach with storage of all objects (H&C). Our algorithm (ADC) approaches quickly the good values of HNC and lies just a small percentage beneath its value.

2) Average Hops Rate (Figure 8), shows a reversed situation, were H&C outperforms slightly better than HNC. As visible from the diagram, ADC is even able to outperform H&C by a small percentage and probably much more in longer test runs due to the self-organizing nature of the algorithm.

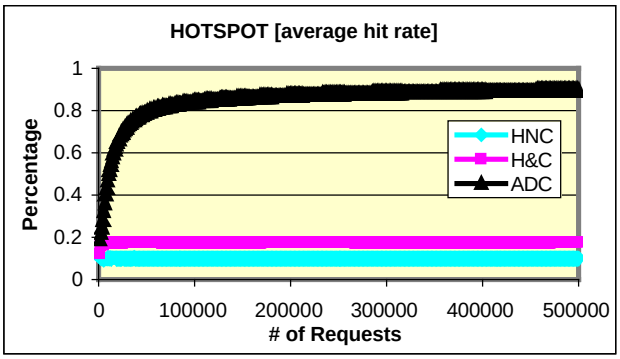

Figure 9, HOTSPOT – average hit rate

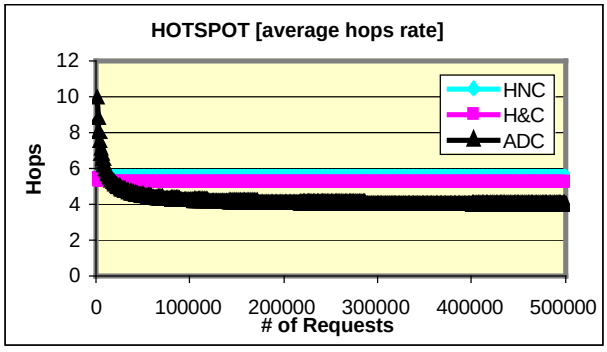

Figure 10, HOTSPOT – average hops rate

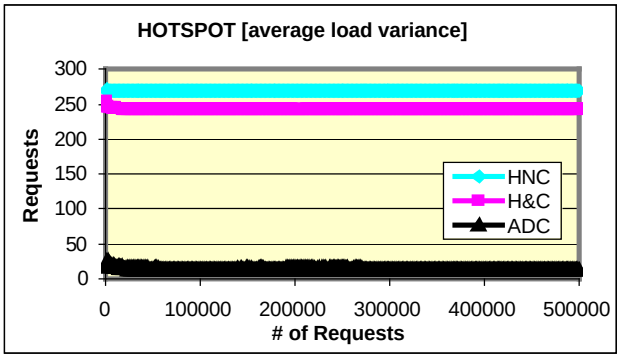

Figure 11, HOTSPOT – average load variance

3) Average Load Variance (Figure 9), shows clearly that our adaptive approach again very quickly approaching an almost optimal value and competes with the good load balance of H&C. HNC performs poorly in comparison to H&C and our algorithm.

D. Uniform Hot Spot

In the second simulation, we ran, all three algorithms against a uniform – hot spot for a set of 1000 objects. A set of 10 proxies provides a total cache space of 1000 objects. To create a hot-spot situation, the 1000 objects are selected in such a way, that the two hashing algorithms will forward all unresolved requests to the same proxy, creating a bottleneck scenario.

1) Average Hit Rate (Figure 10), shows that load balancing of hot spots is the major strength of our proposed algorithm. While both hashing algorithms perform very poorly of fewer than 20% (100 objects per proxy represent a maximum of 10% of the total 1000 objects). ADC is able to maximize the total cache-usage through minimization of redundant objects allowing a hit-rate of over 90%.

2) Average Hops Rate (Figure 11), proofs that also regarding the average number of hops our adaptive algorithm outperforms the two hashing approaches by far. ADC is able to learn ideal mapping for each requested object and allows specific request forwarding.

3) Average Load Variance (Figure 12), the last figure shows more than clearly the load-balancing feature of ADC. With a value very close to zero, ADC is able to keep all proxies on an almost equal usage, while the overloaded proxy in the hashing approaches is the reason for the high load variance value.

IV.CONCLUSION

In this paper we introduced an adaptive algorithm for distributed caching based on request forwarding and response evaluation. We have shown that the algorithm, which is completely based on autonomous proxies, is more than able to outperform existing predefined hashing solutions for cooperative caching both in a general ZIPF request patterns and in hot-spot situations. The adaptive components of the algorithm allow the system to stabilize in the most suitable state in accordance to the current request pattern. Overall we can say, that our algorithm combines the advantages from both ideal mapping found in HNC with no multiple copies and load-balancing through redundant copies found in H&C. One limitation of the algorithm lies in the fact that it keeps a mapping table record for each requested object, which can lead to a large amount of system data. Future work will show how the limitation of the mapping table to a maximums size will affect the overall system behavior. Additionally, the algorithm needs to be tested in a real system with a real proxy request traces.

V.ACKNOWLEDGEMENT

This work is supported by Baptist University research grant FRG/01-02/I-03.

IV.REFERENCES

[1] Kwok Ching Tsui, Jiming Liu, Hiu Lo Liu, Autonomy Oriented Load Balancing in Proxy Cache Servers, Web Intelligence: Research and Development, First Asia-Pacific Conference, WI 2001, p.115-124

[2] Markus J. Kaiser, Kwok Ching Tsui, Jiming Liu, Self-organized Autonomous Web Proxies, AAMAS 2002

[3] Jia Wang, A survey of Web Caching Schemes for the Internet, ACM Computer Communication Review, 29(5):36--46, October 1999.

[4] Pablo Rodriguez, Christian Spanner, Ernst W. Biersack, Web Caching Architectures: Hierarchical and Distributed Caching. 4th International Caching Workshop, 1999

[5] Alec Wolman, Geoffrey M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, Henry M. Levy, On the scale and performance of cooperative Web proxy caching, SOSP-17, 12/1999

[6] Keith W. Ross, Hash-Routing for Collections of Shared Web Caches, IEEE Network Magazine, 11, 7:37--44, Nov-Dec 1997

[7] J. Cohen, N. Phadnis, V. Valloppillil, K.W.Ross, Cache array routing protocol v.1.1, Sept. 1997, Internet Draft

[8] Kung-Lung, Philip S. Yu, Load Balancing and Hot Spot Relief for Hash Routing among a Collection of Proxy Caches, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems

[9] Cho-Yu Chiang, Yingjie Li, Ming T. Liu, Mervin E. Muller, On Request Forwarding for Dynamic Web Caching Hierarchies, In Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00), Taipei, Taiwan, April 2000

[10] John Dilley, Martin Arlitt, Improving Proxy Cache Performance: Analysis of three Replacement Policies, IEEE Internet Computing, Nov.-Dec. 1999

[11] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, Scott Shenker, Web Caching and Zipf-like Distributions: Evidence and Implications, Technical Report 1371, Computer Sciences Dept, Univ. of Wisconsin-Madison, April 1998

[12] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz, Swarm Intelligence: From Natural to Artificial Systems

[13] Jiming Liu, Kwok Ching Tsui, Jianbing Wu, Introduction to autonomy oriented computation, In Proceedings of 1st International Workshop on Autonomy Oriented Computation, pages 1-11, 2001