Distributed Proxy Server Management: A Self-Organized Approach

K.C. Tsui (tsuikc@comp.hkbu.edu.hk)
Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Kowloon, Hong Kong

M.J. Kaiser (mjk@gmx.it)
Department of Computer Science and Engineering, University of Connecticut, Storrs, CT 06269-3155, USA

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

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. A self-organized approach to manage a distributed proxy system called Adaptive Distributed Caching (ADC) is proposed. We model each proxy as an autonomous agent that is equipped to decide how to deal with client requests using local information. Experimental results show that our ADC algorithm is able to compete with typical hashing-based approaches. This paper gives a full description of the self-organizing distributed algorithm, with a performance comparison based on hit rate and hop count. Additional evaluation of the performance with respect to the request routing table size is also presented.

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

1. Introduction

The Internet is growing exponentially and web caches have been shown to be a feasible way to reduce the overall network traffic [3, 6, 18] . Web servers store web objects, which are requested by clients spread over the global network. A web cache, or proxy, is usually placed between clients making requests and web servers servicing the requests. It will try to resolve the needed object by its local cache. The Internet traffic is expected to reduce and the response to user request is expected to improve [1, 11].

Proxies that are not able to resolve an incoming request have to make a choice between either forwarding the request directly to the origin server or query a neighboring proxy. The idea that a proxy can forward requests to another peer leads to research in cooperative proxies and distributed proxy systems [8, 33]. Distributed proxy systems are based

on a set of proxies that when combined increase the overall storage space (cache) and, therefore, increase the chance of fulfilling incoming requests. Cooperative proxies try to combine their individual caches in such a way that maximum cache-usage is achieved while acting transparently as one single load-balanced proxy cache [10]. They lead to overhead problems like inter-proxy communication and the distribution of object storage location information.

Previous research on cooperative proxies can be found in hierarchical [33] and hashing approaches [20, 29], adaptive web caching [37], CacheMesh [34], WebWave [15] and the straightforward approach with a central coordinator [25].

Additional research for distributed systems covers areas on consistency between multiple proxies [14], the theoretical description of the underlying processes [21, 35] and attempts to build the system on the idea of economical models [31]. Issues such as performance [28, 30] and effectiveness of replication schemes [23] were also studied and different schemes for the internal representation of URL lists have been analyzed [13, 24]. Research in distributed caching can also be found in distributed object replication schemes [32], resource allocation [9] and server load balancing [36].

Another way to relieve network congestion is to install an 'reverse proxy' at the origin server end [5, 7]. 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 [4]. 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 'understands' 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.

The proposed method 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* [22]. 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.

This article will first describe some work related to proxy servers. Details of the *Adaptive Distributed Caching* (ADC) algorithm 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 are a lot of work on improving this base strategy.

Existing cooperative proxy systems can be organized in hierarchical and distributed manners [12, 26]. 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) [10]. 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 [37] and *CacheMesh* [34] 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 [16] for a more detailed discussion on the limitations of the pre-fetching approach.

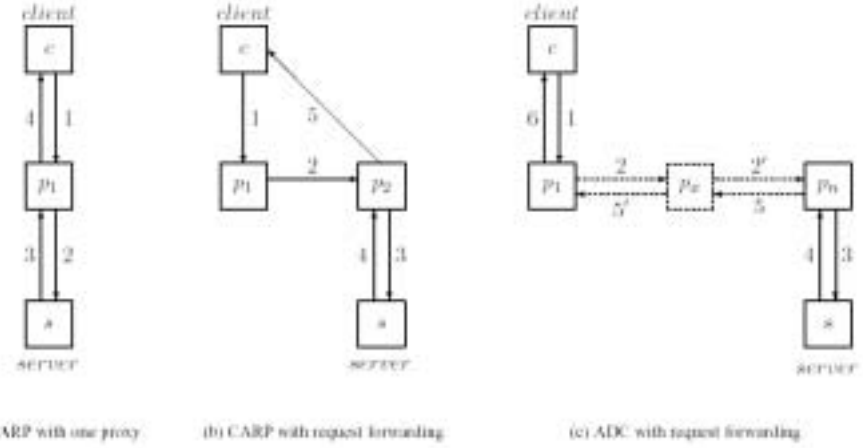
Others have work on structuring the proxy servers so as to improve the chance of locating the required object in one of the proxy servers. Common technique is to arrange a host of proxy servers in a hierarchical manner so that some proxy server, not necessarily on the same local area network, is the proxy of many other proxy servers while serving as the proxy of some local users [27]. 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 [36] 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 [20].

3. Adaptive Distributed Caching

In this section, we will introduce the core components of the adaptive distributed caching (ADC) algorithm. In essence, the algorithm combines the advantages of hierarchical distributed caching (allowing multiple copies of the same object) and of hashing based distributed caching. ADC was developed out of the ideas behind the hashing based allocation and during our research we redefined specific components to reach the desired emergent behaviors.

In short, our proxies maintain minimal number of duplicate copies of the frequently requested objects to balance the user request load between the cooperative proxies. ADC allows the distributed proxies



(a) CARP with one proxy (b) CARP with request forwarding (c) ADC with request forwarding

Figure 1. Different ways to deal with a client request where a cache miss occurs at the first point of inception by CARP and ADC

to agree on the specific location of one object without the need for a central coordinator or a broadcasting protocol.

All proxies in the widely used CARP algorithm use the same hashing algorithm to find out which proxy is responsible for a particular request based on the URL of the needed object [10]. If a proxy that first encountered the request (p_1 in Figure 1a) is responsible for the object, it will fulfill the request either from its cache or fetch the object the server. Otherwise, it will forward the request to another proxy (p_2 in Figure 1b), which will act like p_1 except forwarding a reqest. The benefits of CARP are efficient request handling and no redundant data. However, it is susceptible to data loss due to infrastructure change in the proxy system, and potential uneven distribution of load due to exceptional high demand on certain sites.

The ADC proxies do not have a pre-defined request routing function and try to learn the routing table (*mapping tables*) by observing the experienced traffic. Any proxy that cannot fulfill the incoming request (p_1 in Figure 1b) will forward the request to another proxy, which in turn may continue to forward the request until a proxy (p_n) either has the requested objects in its cache or decides to fetch the objects from the origin server. The requested object will then travel in reverse direction along the path that the request has originally traversed and each proxy on the path will decide whether to cache the object (*selective caching*). Therefore, it is likely that there are duplicate copies of the web object.

The following subsections will first describe the mapping tables that is used as request routing tables and index to the cache content. Details of the core functions of ADC, namely, request forwarding, backtracking, and selective caching, will be then given.

Table Ia. A sample single-table

OBJ-ID	PROXY	LAST	AVG	HITS
www.xy634	[5]	9952	0	1
www.xy34	[4]	9953	0	1
www.xy123	[1]	9954	0	1
www.xy64	[2]	9955	0	1
www.xy53	[1]	9956	123	432

3.1. Mapping Tables

The mapping table is a local data structure within a proxy that is used to resolve the object location by the forwarding process (see section 3.2). In a more abstract sense we see the mapping table as a direct replacement of the static hashing function used in hashing approaches to map object IDs (URLs) onto a unique location.

It is the main objective of our algorithm to allow all existing mapping tables to agree on a unique location for each object without a broadcasting protocol or central coordinator. In our first attempt of the algorithm allowed the table to grow infinitely, keeping track of all previously experienced objects, which usually leads to out of memory problems and performance issues [17]. In our latest extension we introduced a way to limit the mapping table while still keeping the performance at the previously attained level [18].

3.1.1. *Single-Table*

The single-table is used to simply keep track of the current flow of requests. Each unknown object will become a new entry on the top of the table, displacing the oldest entry at the bottom of the table - the well-known LRU algorithm. It is a requirement of the single-table that it is large enough so that requests with at least two hits can occur.

Table Ia shows a simple example of a single-mapping table. The first column contains the general object ID - object URLs in our case. The second column contains the current assigned location for the specific object. This information is local and might vary between different proxies. Objects for which a particular proxy is responsible will have the value 'THIS' in this column of its single-table. The third column takes a marker, which stores the time when this object was last requested. The fourth column keeps track of the average time between two requests of this object and the last column simply keeps track of the number of times a specific object has been requested.

Table Ib. A sample multiple-table

OBJ-ID	PROXY	LAST	AVG	HITS
www.xy64	[8]	2252	70	2
www.xy55	THIS	4253	75	2
www.xy13	[1]	4154	83	34
www.xy644	THIS	6555	90	2
www.xy52	[4]	3356	123	42

Table Ic. A sample cache-table

OBJ-ID	PROXY	LAST	AVG	HITS
www.xy6	[3]	1152	2	434
www.xy5	THIS	5453	5	342
www.xy33	[2]	5254	6	211
www.xy44	THIS	6755	10	432
www.xy2	THIS	8356	15	43

It should be pointed out that we intentionally do not use the HITS value to compute the average request time but focus solely on the time difference between the last two requests. In any adaptive system, changes in the recent past need to be considered and the HITS value would allow objects that were highly requested in the past but within a very short time span to remain unnecessarily long in the local cache.

When an existing entry in the single-table experiences another hit, migration from the single-table to the multiple may occur depending on two factors: the time difference between the latest two requests - a reasonable first approximation of the average object request frequency; and the request frequency of the objects in the multiple-table.

3.1.2. *Multiple-Table*

The multiple-table (Table Ib) is restricted in size and contains only objects that were requested more than once. The organization of the multiple-table is the same as the single table and is ordered by the average request time of the objects. This order allows the simple identification of the object with the worst average time and quick insertions/deletions based using binary search.

Once the multiple-table is filled, newly arrived objects from the single-table need to have an average request time lower than the worst

case currently residing in the table before it will be placed at the appropriate position. Objects removed from the multiple-table will be put back into the single-table as a regular entry; giving it the chance to be hit again later. An object will be considered for migrating from the multiple-table to the caching-table when it is requested again after an entry has been created in the multiple-table.

3.1.3. *Caching-Table*

The caching table (Table Ic) in an ADC proxy keeps track of all currently cached objects. This table is very similar to the multiple-table, with the exception that the table entries represent actually stored objects. Similar to the multiple-table, this table is also ordered by the average request value in column four and new objects have to outperform at least the worst case in (the last row) the table and will be placed in the appropriate position within it. Elements that drop out of the bottom of this table move back to the multiple-table which gives them the chance to be hit again in the near future or to drop out completely over time.

Notice that some entries in the caching-table has the 'PROXY' entry set to another proxy ID. These objects are cached (i.e. a duplicate copy is made) during the selective caching process (see section 3.4) and the ID corresponds to the proxy that actually supplied the requested object, either from its cache or from the origin server.

3.2. Request Forwarding

Every proxy in ADC has its own mapping table and each request comes with a global unique ID (usually based on the clients IP address and an internal request counter). ADC proxies handle a request by first look into its cache. Ideally, the requested object is found and returned to the requester (i.e. a client or another proxy). Otherwise, it triggers the request forwarding mechanism. Upon receipt of a reply, either from another proxy or the origin sever, further processing is required. The complete pseudocode is depicted in Figure 2.

Request forwarding describes the idea that during the search process, unresolved requests will be forwarded to a more suitable proxy object or the origin server. In general, the decision for a forwarding location is based upon either previously learned data stored in the local mapping tables or a random selection over the set of known proxies. In cases where the same proxy got selected twice during the forwarding process, the doubly-hit proxy will always forward the request to the origin server to prevent a loop from happening. Similarly, a maximum number of forwarding can be set to avoid endless forwarding.

```
Request Handling:
begin
  if (object in cache)
    update caching-table
    return object
  else
    // forward request
    store request info
    if   (hop_count > max_hop_count) or (seen request twice)
         send request to origin server
    else
         if   (object in single-table or multiple table)
              retrieve proxy id
         else
              host ← randomly select a proxy id or origin server
         endif
         send request to host
    endif
    // process reply
    perform selective caching
    if   (responsible proxy is THIS)
         return object to client
    else
         if   (object obtained from origin server)
              mark object with proxy id
              return object to requester (upstream proxy or client)
         endif
    endif
  endif
end
```

Figure 2. The pseudocode for handling a request in a ADC proxy server

The retrieved object will then traverse the same path back to the requesting client and all proxies on the way have the option to cache the data. We call this process *backtracking*. In regard to the internal data structure, it is important that every proxy stores information about every forwarded request as long as the backtracking process is not completed.

3.3. Multicasting by Backtracking

Backtracking is the process a retrieved object travels back from the downstream proxy. This is the method by which all ADC proxies agree

on the location for a specific web object in the absence of a central coordinator or a broadcasting protocol. The assumption is that the proxy which retrieves the web object concerned will mark the package with its ID and all other proxies on the return path will receive and accept this information. Once the mapping tables are updated, subsequent requests for the same object can be directed to the right location.

3.4. Selective Caching and Aging

Selective caching was introduced in our previous work to allow each proxy to autonomously specialize on a specific set of cached data [17]. In hierarchical and hashing systems, every proxy stores all passing objects regardless of its future significance and usually uses the LRU algorithm as the cache replacement strategy. This approach has the drawback that it creates a high cache fluctuation rate with minimal reliability in regard to the cached content. Proxy agents based on ADC keep track of the average request frequency of all requested objects based on the last two requests experienced. The learned data, in the form of time gap between two requests, will be used to decide whether the new data should be cached or not. A newly arrived object will only be cached if its average request time is smaller than the worst case currently residing in the cache.

As mentioned before, we introduced selective caching as a mean to focus on the frequently requested objects, and preliminary work has shown that our ADC algorithm works better with selective caching and an ordered table than a table based on a typical LRU algorithm [19]. An object will only be cached if it is able to move from the multiple-table into the caching-table by having an average request time shorter than the worst case.

To make sure that old objects will expire, we introduced a simple object aging strategy by computing the object average time with a focus on the passed time since the last request.

$$T_{age} = \frac{T_{average} + (T_{last} - T_{now})}{2} \tag{1}$$

The advantage of this equation is that it is simple and incurs minimal computational cost. It gives the currently requested objects a lower age (allowing them to stay longer in the table) and represents the actual average value for the next request. Essentially it can be seen as a moving average with a focus on the current time. It should be noted that all object age at the same pace and that an established table order remains the same during the aging process. New objects use the current age of the existing objects to place itself into the appropriate position.

4. Experimentation

To validate and verify the *Adaptive Distributed Caching* algorithm described above, we ran multiple simulations over two sets of artificially created client requests using the polygraph benchmarking tool[1], which is mostly used for the real-time evaluation of hardware proxy systems for and it allows us the specific settings of different parameters to create an artificial request pattern which is supposed to be close to that in real-life.

4.1. Experimental Setup

In the first set of tests, we compare the performance of the ADC algorithm to the widely-used hashing-based CARP [35] using cache hit rate and hop count. The second set of tests aim to test the sensitivity of the ADC algorithm according to the size of the mapping tables and performance is evaluated according to cache hit rate, hop count and execution time. In all the tests, object size is ignored and are assumed to be uniform. The number of ADC or HASH proxies is 5 and there are 525 servers providing web objects to 250 clients. The maximum number of requests per second is set at 100. Therefore, the maximum possible number of requests in the generated data files is 266.5 million.

4.2. Request Patterns

Both created files come with a set of almost 4 million requests, which is equivalent to around 12 hours of traffic from the 250 clients, and is divided into three main phases. The load in each phase gradually increases from 10% to 100% at the initial 20 minutes and wise vera at the last 20 minutes. Figure 4 shows the request throughput in each phase [2] . Phase 1 (fill), with around 864k requests, covers a simple fill phase with only 9% request repetitions. Phase 2 (top1) with around 1.5 million requests offers requests and repeats itself in Phase 3 (top2). There is an idle time covering 108k requests where the workload is kept at only 10% of the maximum workload. For performance assessment of industrial strength proxy servers using the polygraph tool, only the middle section of top2 is used.

The two request patterns differ only in the document popularity model. For scenario 1 data file (Figure 3a), it has a set of popular documents (the *hot set*), which is equal to 1% of the total number of requested documents at any instant (the *working set*). The probability of requesting an object from the hot set is 10%. Scenario 2 data file (Figure 3b) has the same hot set size but the probability of accessing

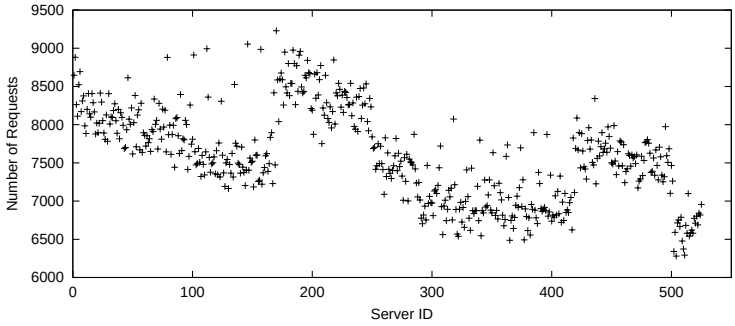


Figure 3a. Number of requests per server for Scenario 1 request pattern

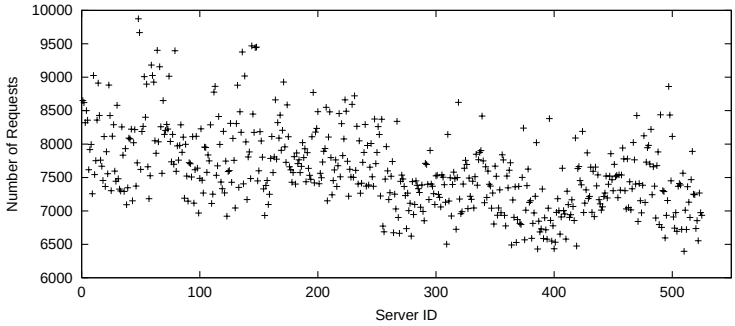


Figure 3b. Number of requests per server for Scenario 2 request pattern

the hot set is 90%. This result of this variation is a difference in the maximum achievable hit rate.

4.3. Performance Metrics

We used cache hit rate and hop count to measure the performance of the algorithms. When we study the parameter sensitivity in relation to table size, we also included simulated execution time.

Cache hit rate refers to the percentage of time a request can be located in the local cache of any proxy server in the proxy system. It can serve as an indicator as how much network traffic can be saved by employing a proxy. The maximum achievable hit rate depends on the cache size and the traffic pattern. Therefore, unless exact hardware/software is used, direct comparison with published results is not useful. However, for our purposes, relative performance can be measured.

Hop count measures the number of inter-machine communication, i.e. client/proxy, proxy/proxy, and proxy/server communication. For the hashing algorithm, the possible combination sequence of communications are (shown graphically in Figure 1a & b):

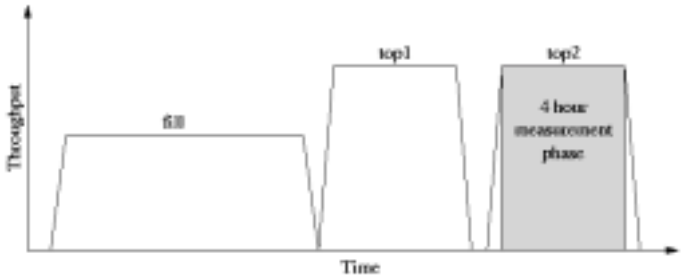


Figure 4. Throughput of the polygraph generated traffic pattern over time

- client/proxy/client (local cache lookup)
- client/proxy/-server/proxy/client (origin server lookup)
- client/proxy1/proxy2/client (cache lookup from non-directly connected proxy)
- client/proxy1/-proxy2/server/proxy2/client (origin server lookup from non-directly connected proxy)

Therefore, the average hop count should be bounded between 2 and 5, and an ideal hop count will be between 2 and 3. As for ADC, the number of hops is always an even number because the route will be traversed twice. The lower bound for ADC's hop count is two, and the ideal hop count is between 2 and 4.

4.4. ADC versus Hashing

We compare the performance of our ADC algorithm to the performance of a common hashing algorithm by hit rate - the percentage of reuse, and hop count - the number of intervening proxies/server before the requested object reaches the requester. The system runs with 20k entries for the single- and the multiple-table and 10k entries for the caching-table in each of the 5 running proxies.

4.4.1. *Hit Rate*

Figure 5a and 5b show the average hit rate as a moving average over 5,000 request intervals for scenario 1 and 2 traffic patterns respectively. The general observation is that during the fill and top1 phases, ADC takes longer time to achieve the hit rate hashing achieved. However, in the measurement taking phase (top2), ADC performs as well as the hashing algorithm.

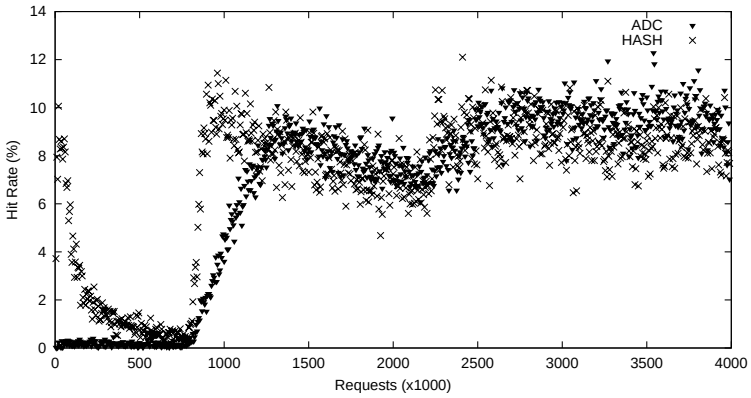


Figure 5a. Hit rates achieved by ADC and hashing for scenario 1 traffic pattern

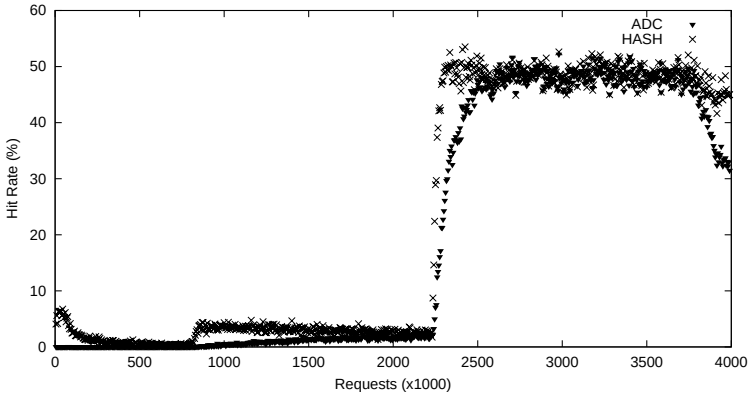


Figure 5b. Hit rates achieved by ADC and hashing for scenario 2 traffic pattern

4.4.2. *Hop Count*

While ADC and hashing achieve similar cache hit rate, it is necessary to see if ADC is doing so at the expense of increased latency. Figure 6a and 6b show the comparison of the average number of hops needed to resolve a request.

For scenario 1 traffic, hashing requires on average close to 5 hops to resolve a request. That means hashing needs to contact the origin server via the second proxy most of the time. On the other hand, ADC needs on average 2 additional hops than hashing. That means ADC needs to consult one more proxy than hashing about half of the time. This result, on one hand, allows the ADC algorithm to search for a specific object more flexibly than the hashing algorithm. On the other hand, ADC has longer systems response than the hashing algorithm.

Scenario 2 traffic is generated such that a high cach hit rate can be achieved. The reduction in hop count also reflects this fact. Hashing

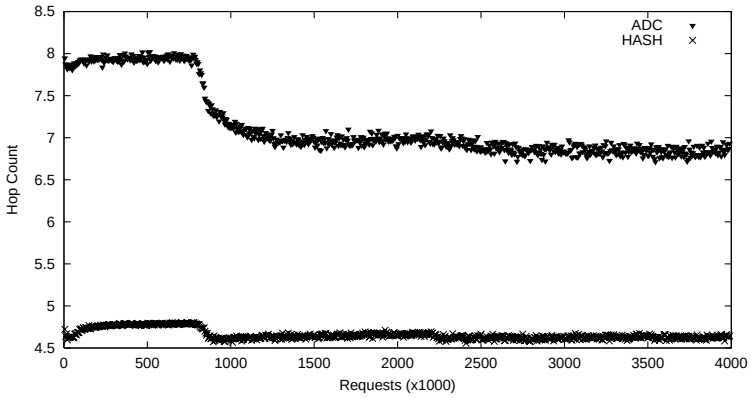


Figure 6a. Hop counts of ADC and hashing for scenario 1 traffic pattern

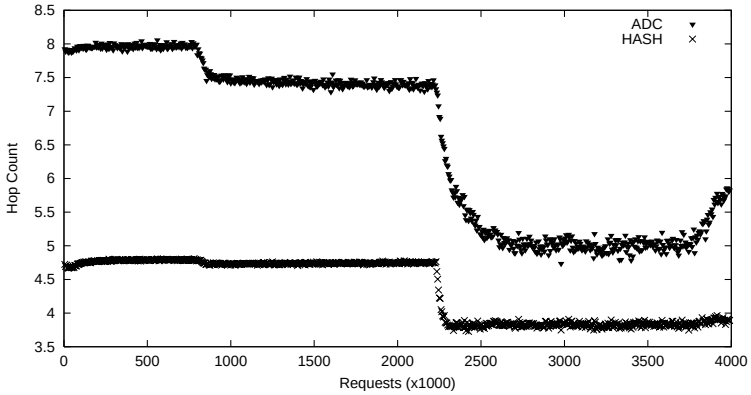


Figure 6b. Hop counts of ADC and hashing for scenario 2 traffic pattern

require on average only 4 hops. That means the second proxy only need to contact the origin server about half of the time. ADC needs only one additional hop than hashing during the top2 phase, which can be accounted for by the additional proxy/proxy transfer before reaching the client. There is no need for ADC to employ more than two proxies to resolve a request. It is clear that ADC can optimally duplicate the frequently requested objects and cut down on the need to contact the origin server for the required web objects.

4.5. Changing Table Size

This section reports our experiments that focused on varying the size for the caching-, multiple- and single-table between 5k and 30k, and we varying the size of one table at a time. The default table sizes for all simulations were 10k for the caching table and 20k for the single and multiple-table. We used average hit rate, hop count and simulation

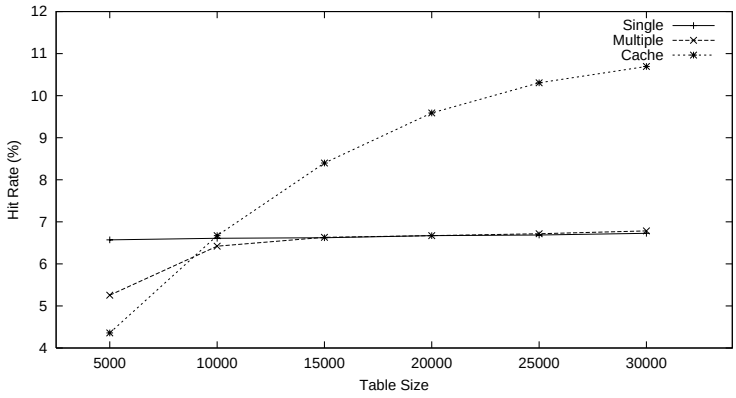


Figure 7a. Hit rate and Hop count by various table size combinations (using scenario 1 traffic)

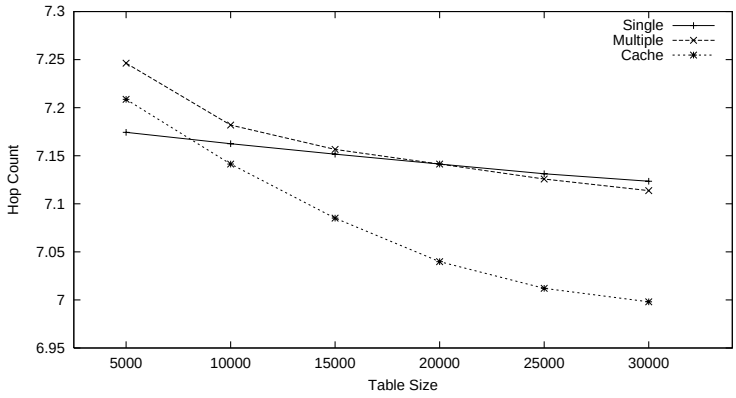


Figure 7b. Hit rate and Hop count by various table size combinations (using scenario 1 traffic)

time needed for the 4 million requests in scenario 1 as our evaluation criteria.

4.5.1. *Hits by Table Size*

In the first part of the evaluation we changed each table size from 5k to 30k in the steps of 5k and observe the overall cache hit rate. We can identify from Figure 7b that the size of the caching table is mostly responsible for the overall hit rate. Naturally, the more cache is available in the system, the more hits are experienced in the test-run. Increasing both the single and the multiple-table beyond 10k did not improve the hit rate above 7%. Interestingly, even a single-table size of 5k was still able to capture enough requests to allow the system to reach the same

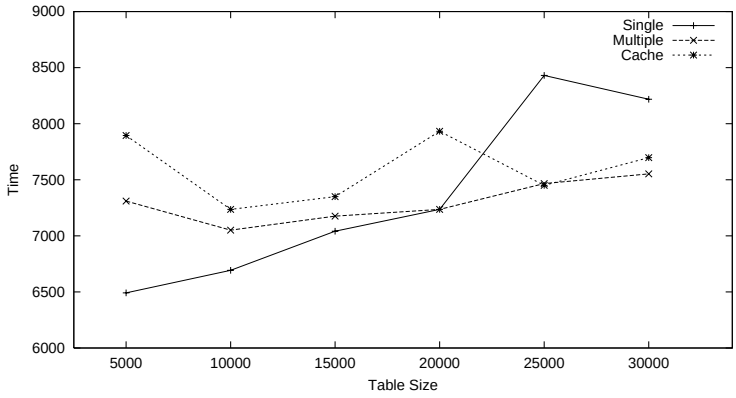


Figure 8. Execution time (in seconds) of the simulations for various table size

number of hits as a single-table size of 30k. A multiple-table of under 10k has a negative impact on the overall hit rate.

4.5.2. *Hops by Table Size*
Figure 7a depicts the average number of hops needed to resolve a request for various table sizes. The general trend is that larger table sizes results in less hops. However, the difference is actually less than 0.15 hops for the single- and multiple-table. The biggest difference resulted in increasing the caching-table size is still just hop, and can be regarded as not significant in comparison of the average number of around 7 hops. The least radical decline can be observed from the changes in the single-table size. This result is consistent with the hit rate results above: the bigger the caching-table the better. If the size of the single-table is big enough to handle the amount of requests a proxy encounters plus those entries demoted from the multiple-table, further increase in size will be benefit the system, as these objects are not cached.

4.5.3. *Time by Table Size*
Figure 8 shows the simulation time in seconds. It can be observed that increasing the single- and multiple-table slows down the overall execution time while increasing caching-table has no significant impact. It seems the most time consuming part of ADC lies with the access mechanism of the single- and multiple-table.

While the single-table is based on the LRU algorithm, locating a candidate for removal during replacement requires ADC to search a linked list element by element. Insertion and deletion at the ordered multiple-table is mostly operated by binary search algorithms. Both access schemes are extremely time consuming and a better data structure is needed to provide speed-ups in the future versions of this algorithm.

Constant paging and context switches occur frequently during the experiments due to limited amount of memory and causes significant delay. Algorithms like MD5 [24] could be used in future to reduce the amount of memory required for storing the URLs in the mapping tables.

5. Conclusions

We have presented in the article the new design of the algorithm called *Adaptive Distributed Caching*, which aims to allow a system of proxy servers to self-organize without the need for central coordination or complicated communication protocol. Based on the experiments we conducted using two artificial (but quite realistic) data files, it is confident that ADC is able to achieve the same level of performance as a well-known hashing algorithm. The extra effort required is just one to two more hops, which can be considered small amount of network latency comparing to the latency incurred on the Internet. We also presented evaluation results regarding various parameters like caching-, multiple- and single-table sizes. We can conclude that a three-table setup only requires a small storage overhead, as small single- and multi-table are required.

The results further show that ADC requires a learning period before it can match the performance of the common hashing algorithm. The strength of ADC should be further studied in the situations where changes both in the proxy system infrastructure and different traffic pattern such as the Wisconsin proxy benchmark [2]. Additionally, we plan to implement the ADC algorithm in a real proxy system such as the freely available Squid server [3].

Acknowledgements

This project is partially funded by grants from the Hong Kong Baptist University (FRG/01-02/I-37 & FRG/02-03/II-39).

Notes

[1] Polygraph website: www.web-polygraph.org

[2] source: http://www.measurement-factory.com/results/public/cacheoff/N04/pm4-phases.png

[3] Squid website: www.squid-cache.org

References

1. G. Abdulla. *Analysis and Modeling of World Wide Web Traffic*. PhD thesis, Virginia Polytechnic Institute and State University, May 1998.
2. J. Almeida and P. Cao. Measuring proxy performance with the wisconsin proxy benchmark. Technical report, University of Wisconsin Department of Computer Science, April 1998.
3. G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine*, May 2000.
4. H. Bryhni, E. Klovning, and O. Kure. A comparison of load balancing techniques for scalable web servers. *IEEE Network*, 14(4):58–64, July/August 2000.
5. 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 WWWCaching Workshop*, 1999.
6. 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.
7. V. Cardellini, M. Colajanni, and P. S. Yu. Load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, May/June 1999.
8. C.-Y. Chiang, Y. Li, M. T. Liu, and M. E. Muller. On request forwarding for dynamic web caching hierarchies. In *Proceedings of the 20th International Conference on Distributed Computing Systems*, 2000.
9. J. Chuang. Resource allocation for stor-serv: Network storage services with QOF guarantee. In *Proceedings of the Nework Storage Symposium*, 1999.
10. J. Cohen, N. Phadnis, V. Valloppillil, and K. W. Ross. Cache array routing protocol v.1.1. September 1997.
11. B. M. Duska, D. Marwood, and M. J. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *Proceedings of USENIX Symposium on Internet Technology and Systems*, December 1997.
12. S. G. Dykes, C. L. Jeffery, and S. Das. Taxonomy and design for distributed web caching. In *Proceedings of the Hawaii International Conference on System Science*, 1999.
13. L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A sacable wide-area web caching sharing pool. In *Proceedings of ACM SIGCOMM Conference*, 1998.
14. J. Gwertzman and M. Seltzer. World-wide web cache consistency. In *Proceedings of USENIX Symposium on Internetworking Technologies and Systems*, pages 141–152, 1996.
15. A. Heddaya and S. Mirdad. WebWave: Globally load balanced fully distributed caching of hot published documents. In *Proceedings of 17th IEEE Conference on Distributed Computing Systems*, 1997.
16. Q. Jacobson and P. Cao. Potential and limits of web prefetching between low-bandwidth clients and proxies. In *Proceedings of Third International WWWCaching Workshop*, 1998.
17. M. J. Kaiser, K. C. Tsui, and J. Liu. Adaptive distributed caching. In *Proceedings of the 2002 Congress on Evolutionary Computation*, pages 1810–1815, May 2002.
18. M. J. Kaiser, K. C. Tsui, and J. Liu. Adaptive distributed caching with minimal memory usage. In *Proceedings of Simulated Evolution and Automated Learning Conference*, pages 360–364, September 2002.

19. M. J. Kaiser, K. C. Tsui, and J. Liu. Self-organized autonomous web proxies. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1397–1404, July 2002.
20. D. Karger, T. Leighton, D. Lewin, and A. Sherman. Web caching with consistent hashing. In *Proceedings of the WWW8 Conference*, 1999.
21. C. Lindemann and A. Reuys. *Modeling Web Proxy Cache Architectures*. Addison-Wesley, 1999.
22. 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.
23. J.-M. Menaud, V. Issarny, and M. Banatre. Improving effiectiveness of web caching. In *Recent Advances in Distributed Systems*, volume LNCS 1752. Springer Verlag, 2000.
24. A. J. Menezes, P. C. V. Oorscht, and S. A. Vanstone. In *Handbook of Applied Crytography*, page 816. CRC Press, 1996.
25. S. Paul and Z. Fei. Distributed caching with centralized control. In *Proceedings of the Fifth International Web Caching and Content Delivery Workshop*, 2000.
26. M. S. Raunak. A survey of cooperative caching. Technical report, December 1999.
27. P. Rodriguez, C. Spanner, and E. W. Biersack. Web caching architectures: Hierarchical and distributed caching. In *Proceedings of the Fourth International WWWCaching Workshop*, 1999.
28. P. Rodriguez, C. Spanner, and E. W. Biersack. Web caching architectures: Hierarchical and distributed caching. 2001.
29. K. W. Ross. Hash-routing for collections of shared web caches. *IEEE Network Magazine*, 11(7):37–44, Nov-Dec 1997.
30. A. Rousskov and V. Soloviev. On performance of caching proxies. In *Proceedings of the ACM SIGMETRICS Conference*, 1998.
31. M. Sinnwell and G. Weikum. A cost model-based online method for distributed caching. In *Proceedings of IEEE Conference on Data Engineering*, 1996.
32. R. Tweari, M. Dahlin, H. Vin, and J. Kay. Design considerations for distributed caching on the internet. In *Proceedings of the 19th IEEE Conference on Distributed Computing Systems*, 1999.
33. J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.
34. Z. Wang and J. Crowcroft. CacheMesh: A distributed cache system for world wide web. In *Proceedings of NLANR Web Cache Workshop*, June 1997.
35. A. Wolman, G. M. Voelker, N. Sharman, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 16–31, 1999.
36. K.-L. Wu and P. S. Yu. 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*, 1999.
37. L. Zhang, S. Michel, K. Nguyen, and A. Rosenstein. Adaptive web caching: Towards a new global caching architecture. In *Proceedings of the Third International WWWCaching Workshop*, 1998.