ADAPTIVE DISTRIBUTED CACHING WITH MINIMAL MEMORY USAGE

Markus J. Kaiser, Kwok Ching Tsui and Jiming Liu

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

ABSTRACT

We have shown previously that our Adaptive Distributed Caching (ADC) algorithm is able to compete with typical hashing based approaches in the realm of distributed proxy systems. In our first approach, we allowed the algorithm to run with the assumption of unlimited system resources (e.g. physical RAM). In this paper we introduce an extension to ADC, which gives the algorithm the capability to perform in more realistic environments with limited memory. We will discuss the arising problem, introduce the new ADC and provide experimental results, which will show that our algorithm is able to achieve the performance characteristic of our previous work even with limited resources.

1. INTRODUCTION

The Internet is growing exponentially and web caches have been shown to be a feasible way to reduce the overall network traffic [6]. A cache is usually placed between the requesting clients and the resolving origin server, storing transferred objects for future retrieval and reducing the overall request latency for cached objects [8].

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 for the needed object. The idea that a proxy can forward requests to another peer lead to research in the area of cooperative proxies and distributed proxy systems [7]. 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 [11].

Previous research on cooperative proxies can be found in the area of hierarchical [7] and hashing approaches [9], adaptive caching [2], CacheMesh [1] and our previously introduced algorithms - SOAP [5] and ADC [3].

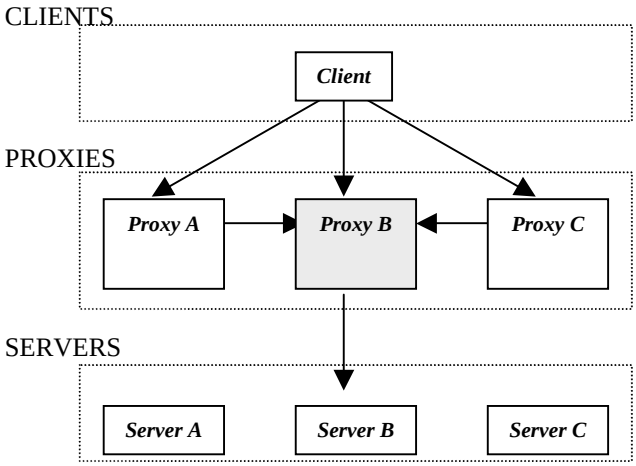

Figure 1. Ideal Cooperation

1.1. Hierarchical vs. Hashing

Common approaches for cooperative proxy systems encompass mostly solutions based on hierarchical cache structures or classical hashing algorithms [9]. While hierarchically organized proxies forward unresolved requests to sibling and parent caches using ICP, hashing proxies, like CARP [10], map the object-ID (URL) of an unresolved request onto the same proxy peer based on a globally known hashing function.

1.2. Adaptive Distributed Caching (ADC)

Preliminary work on ADC is covered by our studies in proxy load balancing by a central coordinator [4] and SOAP, which introduced the idea of adaptive forwarding by evaluating the response time for each requested object [5]. We introduced in our latest work an algorithm for adaptive distributed caching (ADC) based on a set of autonomous proxy agents. We have shown that our algorithm is well able to find a balance between optimal cache usage and distribution of hot documents and reaches good values for the object allocation results that are comparable with the near-ideal hashing algorithm.

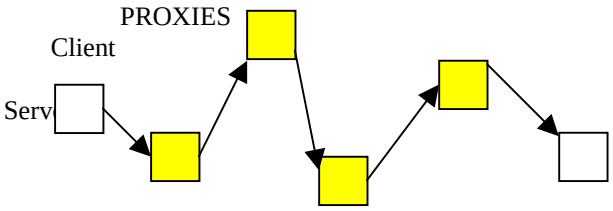

Figure 2. Request Forwarding Path

The reminder of this paper is organized in the following way: we will first restate the core components of the ADC algorithm and its limitations, introduce the latest changes and improvements to overcome the problem of memory limitation in the following section, followed by a set of experimentations for evaluating our modifications. The paper will be concluded by a final discussion and future work.

2. ADAPTIVE DISTRIBUTED CACHING THROUGH PROXY AGENTS

2.1. Previous Work

In our latest paper, we introduced an adaptive distributed caching algorithm that combines the advantages of hierarchical distributed caching (allowing multiple copies of the same object) and of hashing based distributed caching (fast allocation through global agreement). Our proxy agents maintain multiple copies of the frequently requested objects to balance the user request load between the cooperative proxies when hot documents experience a high request rate and reduce the number of copies in situations where only few requests are experienced. In both cases the algorithm allows the distributed proxy agents to agree on the specific location of one object without the need for a central coordinator or a broadcasting protocol.

2.2. Algorithm

The core of the ADC algorithm can be divided into two local techniques that allow global stabilization: *Request Forwarding* and *Selective Caching* with its components *Mapping Table* and *Self-Organization by Multi-Casting*. The same version of the algorithm is installed on every running proxy.

2.2.1. Selective Caching & Aging
Selective Caching was introduced in our previous work to allow each proxy to autonomously specialize on a specific set of cached data [3]. In hierarchical and hashing systems, every proxy stores all passing objects regardless of its future significance and uses the LRU algorithm as cache replacement strategy. This approach 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 three experienced requests. 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 discarded. A newly arrived object will only be cached if its average request time is smaller than the worst case currently residing in the cache.

2.2.2. Mapping Table
The mapping table is a local data structure within every proxy for resolving the object location to be used by the forwarding process using the object ID (URL). In a more abstract sense we can see the mapping table as a direct replacement of the static hashing function, used in hashing approaches, to map object IDs onto their 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. The size of the mapping table is the main focus of this paper. In our previous work we allowed the table to grow infinitely, keeping track of all previously experienced objects, which usually leads to out of memory problems and performance drawbacks. In the following section we will introduce a way to limit the mapping table while still keeping the performance at the previously attained level.

3. MAPPING TABLE FOR LIMITED SYSTEM RESOURCES

As previously mentioned, our algorithm for adaptive distributed caching assumes infinite resource capacity for the local mapping table in every proxy. This scenario is highly unrealistic and it was the ultimate goal of this paper to identify an extension to ADC so that it is more suitable for a realistic situation while keeping the performance characteristics achieved previously. Different approaches and modifications have been tried out to overcome the stated obstacle and the introduction of additional single and multiple mapping tables appeared as the most suitable to achieve our goal.

3.1. Problems Encountered

At this point let us give an example of the underlying problems that arise when we limit the mapping table to an infinite size. Whenever a previously un-requested object is experienced by one of the proxies, it should have an entry in the mapping table so that future requests for it can use the stored information for the computation of the request frequency. Essentially each mapping table entry should exist long enough to allow the occurrence of another hit.

OBJ-ID	PROXY	LAST	AVG	HITS
www.xy634	Proxy[5]	9952	0	1
www.xy34	Proxy[4]	9953	0	1
www.xy123	Proxy[1]	9954	0	1
www.xy64	Proxy[2]	9955	0	1
www.xy53	Proxy[1]	9956	123	432
www.xy343	Proxy[7]	9957	0	1
www.xy29	Proxy[4]	9961	0	1

Figure 3. A Sample Single Table

In our previous algorithm, each newly created entry will remain in the mapping table forever, leading to a growing mapping table. Depending on the amount of unique requested objects, the system will eventually run out of memory.

In the case of a limited-size mapping table, it will fill up over time and existing entries will sooner or later be replaced by new requests. As described earlier, we use the average time between two requests to decide whether an object should be stored or discarded. Therefore, each newly created entry should stay long enough in the mapping table so that a repeat request can occur to allow the algorithm to compute the average request time.

3.2. New Algorithm

To deal with the above problem, we found it useful to part the mapping table into a section for objects that are requested more than once and another one for new objects that are only requested once. The underlying idea can be stated as follows.

3.2.1. Single Table

The single table is used to simply keep track of the current flow of requests. Each unknown object will receive a new entry on the top of the table and 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. Preliminary studies have shown that the minimum table size is highly dependent on the experienced request pattern and more work is necessary to characterize this relationship. When an existing entry in the mapping table experiences another hit, the time difference between the two requests will be used as a first approximation of the average object request frequency and the object will move from the single table to the multiple table (described below). The LAST column represents the local time value, for the last time when the specific object was requested while the AVG value represents the average time between two requests of the same type (Figure 3).

OBJ-ID	*PROXY*	*LAST*	*AVG*	*HITS*
www.xy64	Proxy[8]	2252	70	2
www.xy55	This	4253	75	2
www.xy13	Proxy[1]	4154	83	34
www.xy644	This	6555	90	2
www.xy52	Proxy[4]	3356	123	42
www.xy433	Proxy[8]	7557	313	4
www.xy299	Proxy[4]	3261	874	54

Figure 4. A Sample Multiple Table

3.2.2. Multiple Table

The multiple table is also restricted in its size and contains only objects that were requested more than once ordered by their average request time. Once the table is filled, newly arriving objects from the single table have to have a lower average value than the worst case currently residing in the table before it will be placed at the appropriate position. Removed objects from the multiple table will be placed into the single table as a regular entry, giving it the chance to be hit again later. The forwarding address for elements with the THIS value marks objects for which this proxy itself is responsible when future requests arrive and unresolved queries need to be forwarded to the origin server. The general structure of the multiple table is the same as that of the single table and it should be pointed out that the table is always ordered in ascending order of the fourth column (average request time). This order allows the simple identification of the object with the worst average time and quick insertions/deletions based using binary search.

3.2.3. Cached Table

The cached table in a proxy within the ADC architecture keeps track of all currently cached objects there. This table is very similar to the previously described 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.

3.3. Noisy Hits

Splitting up the mapping table from our previous work into a single and a multiple table has overcome the obstacle of limited resources but has introduced a new problem of *Noisy Hits*. When we order the mapping table based on the average request time, an object that has been requested only two times but within a very short time frame and never again will be wrongly placed into the mapping table (or even the caching part in the worst case)

and uses up resources that could otherwise be assigned to more frequently requested objects. Different approaches have been tried out to filter out the noisy hits. When considering simplicity of the algorithm and computational cost, we decided that each object has to be hit at least twice to enter the multiple table, and at least three times before moving them into the caching table. With this approach, we are losing a small percentage of the overall hit-rate by compromising objects that were requested only once or twice, but it allows the algorithm to focus on the more important, well-established objects.

4. EXPERIMENTATION

To validate/verify the new algorithm described above, we ran multiple simulations over a set of artificially created request patterns based on the polygraph test bed [15] and compared the results to those of a common hashing based algorithm.

4.2. ZIPF Test Set

In our previous paper [3], we introduced a simple artificially created request pattern based on ZIPF distribution over a set of 10,000 unique objects with 500,000 requests. We simulated a set of 10 proxies with a total cache size of 1,000 objects (each proxy is able to store 100 objects).

For the simulation with the ZIPF data, we select 1000 entries as the average multiple table size, a value that was chosen based on preliminary studies, and compare the results of ADC to the two hashing algorithms. Figure 8 shows that for our own ZIPF distribution, the ADC algorithm is well able to compete with both hashing approaches.

4.3. POLYGRAPH Test Set

For the second session of experimentations, we tried to create artificial data that represents closer the request patterns experienced on a real system and used the widely used Polygraph tool to build a file with 2.3 million requests. Figure 5 shows the source code of the polygraph language. We used the standard Polymix 4 file without the initial fill-phase. Essentially, the final request file pattern contains two peak request rates over a set of approximately 1.5 million requests with some request gaps in between.

4.3.1. Hit-Rate

We ran the Polymix 4 file against ADC and both hashing algorithms, and describe in this section the observed hit rate.

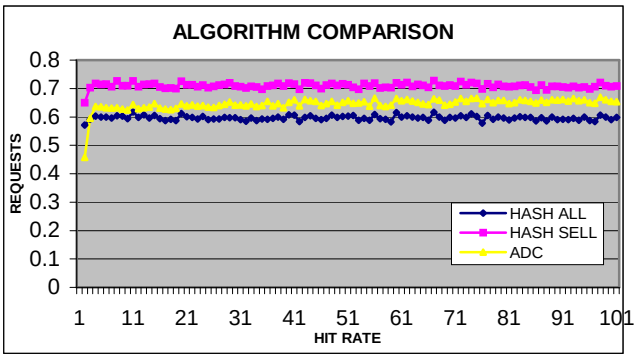

Figure 5. Algorithm comparison

```
/* PolyMix-4 workload  */

TheBench.peak_req_rate = 100/sec;
FillRate = TheBench.peak_req_rate;
ProxyCacheSize = 1MB;
```

Figure 6. Polygraph Settings

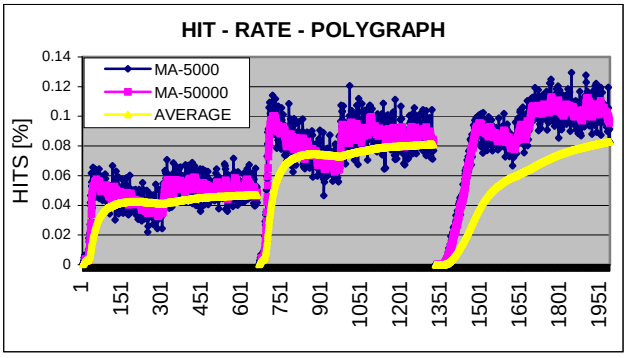

Figure 7. Hit Rate Polygraph

Figure 6 shows the hit rates in three sections. The first part covers the hashing algorithm with always caching; the second part the hashing with selective caching and the last part the ADC based approach. All three sections come with three graphs, a moving average over that 5,000 and 50,000 and all requests. It is clear that the updated version of the ADC algorithm (with single, multiple and caching tables) outperforms both hashing algorithms over the set of 2.3 million requests. It is also clear that the new ADC algorithm requires a certain learning phase, which comes with an additional overhead as can be observed in the slowly increasing graph of the total average hit-rate.

4.3.2. Hops Rate & Execution Time

In the second part of this section we take a look at the average hops rate, which represents the effort to search for a specific cached object. One hop is defined by the request

being forwarded from one proxy to another one towards the target server and on its way back. Our results show that both hashing approaches need on average around 5.5 hops to resolve a requested object. The ADC approach starts off with around 8 hops per request and decreases to an average of around 7 hops, an acceptable overhead in comparison to the ideal function based search procedure of hashing.

5. CONCLUSION

In this paper we extended the existing ADC algorithm so that it is capable of performing under resource restrictions like physical RAM. For this purpose we introduced a way to limit the mapping table by splitting it up into a table for single requested objects, multiple requested objects and cached objects. In the experimentation we compared the new algorithm to two typical hashing based algorithms using the artificial data sets of a ZIPF distribution over 10,000 objects and the 2.3 million requests received from the widely used benchmark tool called Polygraph. The results show that our modifications to ADC have no major performance drawbacks with respect to changes in the single table size and the new ADC is well able to outperform both hashing approaches in to the case of the polygraph benchmark data set. Additionally, the new ADC achieves competitive results under the categories of HOPS and execution time.

We will continue to focus on the distribution of the simulation over a set of multiple machines with a focus on changes in the infrastructure and overall execution time in relationship to the single and multiple table size. We will also look into the problem of noisy hits, the relationship between incoming request distribution and single table size, ways to shorten the learning period.

6. ACKNOWLEDGEMENT

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

7. REFERENCES

[1] Z. Wang, “Cachemesh: A Distributed Cache System for World Wide Web”, Web Cache Workshop, 1997.

[2] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, “Adaptive Web Caching: Towards a New Caching Architecture”, 3rd International WWW Caching Workshop, June 1998.

[3] M.J. Kaiser, K.C. Tsui, J. Liu, “Adaptive Distributed Caching”, Congress on Evolutionary Computation, IEEE 2002.

[4] K.C. Tsui, J. Liu, H.L. Liu, “Autonomy Oriented Load Balancing in Proxy Cache Servers”, Web Intelligence: Research and Development, First Asia-Pacific Conference, WI 2001, p.115-124.

[5] M.J. Kaiser, K.C. Tsui, J. Liu, “Self-organized Autonomous Web Proxies”, Conference on Autonomous Agents & Multi-agent Systems, IEEE 2002.

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

[7] P. Rodriguez, C. Spanner, E.W. Biersack, “Web Caching Architectures: Hierarchical and Distributed Caching”. 4th International Caching Workshop, 1999.

[8] A. Wolman, G.M. Voelker, N. Sharma, N. Cardwell, A. Karlin, H.M. Levy, “On the Scale and Performance of Cooperative Web Proxy Caching”, SOSP-17, 12/1999.

[9] K.W. Ross, “Hash-Routing for Collections of Shared Web Caches”, *IEEE Network Magazine*, 11, 7:37--44, Nov-Dec 1997.

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

[11] K.-L. Wu, 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

[12] C.-Y. Chiang, Y. Li, M.T. Liu, M.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

[13] J. Dilley, M. Arlitt, “Improving Proxy Cache Performance: Analysis of three Replacement Policies”, *IEEE Internet Computing*, Nov.-Dec. 1999.

[14] L. Breslau, P. Cao, L. 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.

[15] http://www.web-polygraph.org/