A Study of the Performance and Parameter Sensitivity of Adaptive Distributed Caching

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

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

Abstract. A self-organized approach to manage a distributed proxy system called *Adaptive Distributed Caching (ADC)* has been proposed previously. 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 hits and hops rate. Additional evaluation of the parameters caching-, multiple- and single-table size is also presented.

Keywords. Distributed Caching, Adaptive Proxy, Self-Organization, Internet Server

I. INTRODUCTION

The Internet is growing exponentially and web caches have been shown to be a feasible way to reduce the overall network traffic [12]. Web Servers store objects (documents), which are requested by clients spread over the global network. A web cache, or also known as web proxy, is usually placed between the requesting clients and the resolving origin server. Client requests are usually directed to the proxy server, which will try to resolve the needed object by its locally cached data.

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 [27]. Distributed proxy systems are based on the idea that a set of multiple proxies combined increase the overall storage space for cached documents and increase therefore the chance to resolve 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 [4] which leads to problems like inter-proxy communication to avoid the storage of equal objects and the distribution of information about stored projects for the allocation process.

Previous research on cooperative proxies can be found in the area of hierarchical [27] and hashing approaches [13][29], adaptive caching [17], CacheMesh [28],

WebWave [9] and the straightforward approach with a central coordinator [18][26].

Additionally research for distributed systems cover areas for consistency between multiple proxies [8], the theoretical description of the underlying processes [14][29] and attempts to build the system on the idea of economical aspects [24]. Aspects like performance [21][23] and effectiveness of replication schemes [15] were evaluated and different schemes for the internal representation of URL lists analyzed [7][16]. Parallel applications for research in the area of distributed caching can be found in distributed object replication schemes [25], resource allocation [5] and sever load balancing [1].

II. PREVIOUS WORK

1. Central Coordinator [26]

In our first approach we introduced the self-organizing approach of proxy load balancing by the usage of a central coordinator in front of all running proxies. We have shown in that the system is well able to react to adapt the load distribution in regard to the individual performance characteristics of every proxy. The central coordinator collectively receives all incoming requests for the proxy system and assigns the request to the currently best performing proxy without considering previously stored objects. Additionally it learns from the response time the validity of its choice and adapts the internal performance values based on a simple reinforcement learning algorithm. The major drawback in our first approach lay in the fact that the central coordinator creates the clear bottleneck situation for the overall system due to the fact that all requests and feedbacks have to pass the coordinator. Additionally, the system left space for improvement in regard to more specific request forwarding considering previously stored objects.

2. Self-Organized Adaptive Proxies (SOAP) [10]

Based on our experiences in the approach with a central coordinator we introduced the idea to place the reinforcement learning component in front of every proxy (essentially as part of every proxy) allowing the proxies to receive requests directly with the individual learning and evaluation of forwarding decisions. Therefore each proxy received an internal table to map the URL of a specific object onto one proxy of the total set of running proxies. The feedback system was based on the simple response time evaluation. We had to learn that such a system requires a large number of requests for each object ID to allow the whole system to learn the same mapping. Each mapping table contained one entry for a specific URL domain (category) and the decision-making component mapped each category onto one proxy location. The drawback of the provided solution lay in the fact that our solution was not able to deal ideally with bottleneck situations were, for example, only requests for one category were injected. The major lesson we learned in our work is the importance of selective caching. The idea behind selective caching as described later, is the fact that each proxy decides individually for each object if it will cache the data or discard the received information.

3. Unlimited Adaptive Distributed Caching [11]

In our next step we tried to overcome the drawbacks of SOAP and its domains by a direct mapping of each object onto exactly one location. In abstraction we can look at the mapping table as a written hashing function with a direct object-ID to location relationship. This idea introduced a new problem, the mapping table that stores the URL mappings needed to be very large to be able to store an entry for every experienced object-ID and we accepted this drawback by letting the table grow indefinitely. Avoiding the idea of information broadcasting, we attacked the problem by means of information multicasting along the forwarding path. Each resolved object bypasses all proxies that forwarded the request previously. It is important for our system that multiple proxies focus on the same location for a specific object. Our work has shown that the algorithm is well able to provide the wanted functionality and was well able to compete with classical hashing approaches.

4. Adaptive Distributed Caching with a realistic Mapping Table [12]

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. Whenever a previously un-requested object is experienced by one of the proxies, it will receive an entry in the mapping table so that future requests for it can use the stored information for the computation of the request frequency. To allow the system to learn from new requests, 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. As a consequence of our work we, split up the existing mapping table into a single-, multiple- and caching table. Elements move back and forth between the three tables in accordance to the average time-difference between equal requests.

III. ADAPTIVE DISTRIBUTED CACHING

In the following sections we introduce the core components of the adaptive distributed caching 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 (fast allocation through global agreement). As shown in the previous chapter, the algorithm developed out of the ideas behind the hashing based allocation and during our research we redefined specific components to reach the wanted global emergent attributes.

In short, our proxy objects maintain multiple copies of the frequently requested documents to balance the user request load between the cooperative proxies and reduce the number of copies in situations where only few requests for a particular object are experienced. In both cases the algorithm allows the distributed proxies to agree on the specific location of one object without the need for a central coordinator or a broadcasting protocol.

The core of the ADC algorithm can be divided into four parts that allow the global stabilization, in combination with the peer proxies: (1) Request Forwarding and Looping, (2) Multicasting by *Backwarding*, (3) Mapping Tables and (4) Selective Caching with Aging.

1. Request Forwarding and Looping

Request Forwarding and Looping 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 were random request forwarding resolves in an loop, meaning that the same proxy got selected more than once during the forwarding process, the doubled hit proxy will always forward the request to the or gin server and therefore terminate the search process. As a second termination criterion, to avoid endless request forwarding over a large set of proxies, a maximum number of forwarding can be set. After reaching the forwarding maximum the next proxy will end the search process by forwarding the request to the origin server. It should be kept in mind at this point that in our system we don't expect the loss of messages and that always either one of the proxy objects or the actual origin server will finally resolve the request. The retrieved object will then traverse the same path back to the requesting client, a process we call this *backwarding*, and all proxies on the way have the option to cache the data. In regard to the internal data structure, it is important that every proxy stores information about every forwarded request as long as the backwarding process is not completed. Each request comes with a global unique ID (usually based on the clients IP address and an internal request counter), which is used to give each proxy the option to identify forwarding loops.

2. Multicasting by Backwarding

Every running proxy has its own *mapping table* and unknown objects are usually resolved by random forwarding based search over the total set of running proxies. In this paragraph we talk about the idea of data and information dissemination through multicasting on the backwarding path. Essentially, in our work we tried to find a technique that allows proxy objects to agree on the same location for a specific document without the use of a central coordinator or a broadcasting protocol. Looking at the set up of the proxy system based on a hierarchical structure we can see that every object will be passed down along the hierarchy from the root to the leave proxy. In our forwarding process, each object will pass all previously passed proxies. This backwarding can be seen as a simple technique for data dissemination based on multi-casting to a selected group of proxy objects. We use this technique to enforce all proxies in this multi-casting group to agree on the same location for a specific object. In essence, one proxy will either resolve or even cache the object, starting at the end of the forwarding path. This proxy will mark the package with its address and all other proxies on the backwarding path will receive and accept this information. Each individual proxy on the return path may cache the object if it fits their selective caching criteria and use the information about the resolving proxy agent to update the location in their mapping table. Essentially, the longer the backwarding path, the more proxies will

agree on the same location for the specific object. If in the future the same object is requested starting at a different proxy, the forwarding path only has to hit one of the previously used proxies to quickly find the agreed location. Over the backwarding path a different set of proxies will learn about the location and more and more proxies will stabilize on the same information without any further communication. The resolved objects usually come in sizes of up to n Kbytes and the additional information of the resolving proxy location creates a negligible overhead.

3. Mapping Tables

The mapping table is a local data structure within every proxy that is used to resolve 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. 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 drawbacks. In our latest extension we introduced a way to limit the mapping table while still keeping the performance at the previously attained level.

3.1 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, 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. 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).

Figure 3 shows a simple example of a single-mapping table. The first column contains the general object ID in our case the object URL. The second column contains the current assigned location for the specific object. This information is local and might vary between different proxy agents. Objects for which a particular proxy is responsible will have the value THIS in this column of that proxy's single-table. The third column takes a marker, which stores a local time value of 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 this a specific object has been requested.

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 to remain unnecessarily long in the local cache.

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.xy452	Proxy[4]	9958	522	434
www.xy2	Proxy[1]	9959	0	1
www.xy32	Proxy[2]	9960	0	1
www.xy29	Proxy[4]	9961	0	1

Figure 1: A Sample Single-table

3.2 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 equal to 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.

OBJ-ID	PROXY	LAS	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.xy52	This	3458	323	44
www.xy32	Proxy[1]	7859	553	65
www.xy232	This	3260	766	2
www.xy299	Proxy[4]	3261	874	54

Figure 2: A Sample Multiple-Table

3.3 Caching Table

The caching table in an ADC proxy keeps track of all currently cached objects. 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.

OBJ-ID	PROXY	LAST	AVG	HITS
www.xy6	Proxy[3]	1152	2	434
www.xy5	This	5453	5	342
www.xy33	Proxy[2]	5254	6	211
www.xy44	This	6755	10	432
www.xy2	This	8356	15	43
www.xy3	This	8357	33	1
www.xy52	This	2258	37	434
www.xy23	Proxy[2]	1259	38	22
www.xy32	This	6360	44	42
www.xy99	Proxy[3]	7361	65	124

Figure 3: Caching Table

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 [26]. 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 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.

As mentioned before, we introduced selective caching as a mean to focus on the more important often requested objects, and preliminary work has shown that our algorithm works better with the approach of selective caching and an ordered table than a table based on a typical LRU algorithm. 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 objects average time with a focus on the passed time since the last request.

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

Figure 4: Moving Average over last two requests

The advantage of this equation is that it is simple and comes with a 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 objects age at the same pace and that an established table order remains the same during the aging process. Newly entering objects use the current age of the existing objects to place itself into the appropriate position.

IV. ALGORITHM

The algorithm for ADC is implemented in every running proxy with an equal setting without any further modifications or fine-tuning. Each proxy essentially reacts to two types of events: an incoming request or an incoming reply, otherwise it stays idle. The following sections describe the core parts of the algorithm with additional descriptions.

1. Receive Request

Each proxy receives incoming requests and trys to resolve it by means of its locally cached data or through smart forwarding to a more suitable location. Looking at the meta-code in Figure 5 we can clearly see this decision-making as the if-statement in line 3. If the request could be resolved by the local data, the proxy will fetch the stored object, update the respective entry in the mapping table data structure and return the object to the requesting peer. On the other side, were the object was not available in the local cache, the proxy stores the information for the backwarding and forwards the request to the origin server if a loop got detected or the current request exceeds maximum number of forwarding hops. In all other cases, the proxy looks up the stored forwarding information and uses it to transfer the request to a better peer.

```
Receive_Request()
      Local_time++
      Object = Request.getRequestedObject()
      IF (Object is locally cached)
            Update_entry(Object,this)
            Return getData(Object) to Request.getSender()
      ELSE
            Store_Backwarding(Request)
            Request.setSender(this)
```

```
IF(Loop_Detected(Object) OR Request.isMaxHops()
      Forward Request to Origin Server
ELSE
      Forward Request to Forward_Addr(Object)
```

Figure 5: Receive_Request()

Figure 6 describes the general selection of the forwarding address. It should be pointed out that in case where no entry for the requested object exists, the proxy will select a random peer over the total set of all known proxies including itself. Additionally, the counter for the received requests represents the local clock of the proxy and is used for the later described average computation.

```
Forward_Addr (Object)
      Entry = find Entry in Multiple or Single Table (Object)
      IF (Entry == NULL)
            Return Random Selection (Set of Proxy Locations)
      ELSE
            Return Entry.location
```

Figure 6: Forward_Addr()

2. Receive Reply

The second major event for every proxy system is the arrival of a reply from a previously forwarded request along the backwarding path. It is assumed in our system that every request got finally resolved by either one of the proxies or the origin server, therefore each reply package carries the data for the requested object and as described earlier the information about the resolving proxy. Figure 7 shows this aspect in the second if statement. The first if-clause simply checks if the resolving proxy got already set, a NULL value stays for the data from the origin server and the current proxy will be assigned as the official resolver. After updating the internal mapping table, the proxy checks if the received data was stored in the local cache and sets itself as resolving proxy if no other proxy has previously cached the same data. This focus on only one caching location is necessary to allow the system to agree faster on one location for a specific object. Finally the object continues its way along the backwarding path.

```
Receive_Reply()
      IF (reply.getResolver() == null)
            Reply.setResolver(this)
      Update_Entry(Object,reply.getResolver())
      IF(isLocallyCached(Object) AND reply.notCached())
            Reply.setResolver(this)
            Reply.setCached(this)
      Backward Reply
```

Figure 7: Receive_Reply()

3. Update Entry

In both sub-procedures for receive request and reply, we can identify the method *Update_Entry* as the core procedure to update the internal mapping table data structure. The *Update_Entry* table is accessed each time when the information for a resolved object passes the proxy, or directly when the proxy was able to resolve the request using its locally cached data. Looking at Figure 8 we can see that the *Update_Entry* method contains essentially four parts due to the fact that the method first needs to identify the table that contains the related entry and searches for it in the order, caching table, multiple-table and single-table. If not found a new entry will be created and placed into the single-table.

The first part checks the caching table for the wanted information and updates the entry in accordance to the new assigned location and average time. The caching table is ordered and the updated entry will be placed respectively.

The second part of the method checks the multiple-table, if the object was not found in the caching table. If found in the multiple-table, the entry will be updated in regard to average time and location but this time the proxy checks if the new average time is lower than the worst case in the caching table. If that is the case, the last element in the caching table will move into the ordered multiple-table and the selected element from the multiple-table will move into the ordered single-table.

The third part checks the single-table for the wanted object and is similar to the second part. If found, the entry will be updated and the proxy checks if the new average value is smaller than the worst average value of the multiple-table, if that's the case the element from the single-table will be placed into the ordered multiple-table and the last element of the multiple-table will be placed at the top of the single-table. It should be reminded again that the single-table is based on a simple LRU algorithm while entering elements will be placed on the top of the table and leaving elements drop out on the bottom.

Finally, the fourth part takes care of the situation, where the object was not found in one of the three mapping tables and creates a new entry. The new entry will be initialized with the assigned values and placed on top of the single-table, the last element of the single-table drops out.

```
Update_Entry (Object, Location)
// PART 1 : CHECK AND UPDATE CACHED TABLE
      IF (Entry = RemoveEntry(CachedTable,Object)!= NULL)
            Entry.calcAverage(Local_Time)
            Entry.location = Location
            InsertOrdered(CachedTable, Entry)
            RETURN

// PART 2 : CHECK AND UPDATE MULTIPLE TABLE
      IF (Entry = RemoveEntry(MultipleTable,Object)!= NULL)
            Entry.calcAverage(Local_Time)
            Entry.location = Location
            IF(Entry.getAverage < CachedTable.WorstAverage())
```

```
                IF (CachedTable.Size == MaxCachedSize)
                      Temp = RemoveLastEntry(CachedTable)
                      InsertOrdered(MultipleTable, Temp)
                InsertOrdered(CachedTable, Entry)
          ELSE
                InsertOrdered(MultipleTable, Entry)
          RETURN

// PART 3 : CHECK AND UPDATE SINGLE TABLE
      IF (Entry = RemoveEntry(SingleTable,Object)!= NULL)
          Entry.calcAverage(Local_Time)
          Entry.location = Location
          IF(Entry.getAverage<MultipleTable.WorstAverage())
                IF (MultipleTable.Size == MaxMultipleSize)
                      Temp = RemoveLastEntry(MultipleTable)
                      InsertOnTop(SingleTable, Temp)
                InsertOrdered(MultipleTable, Entry)
          ELSE
                InsertOnTop(SingleTable, Entry)
          RETURN

// PART 4 : CREATE NEW ENTRY
      Entry = Create new Entry (Object, Location)
      IF (SingleTable.Size == MaxSingleSize)
          RemoveLastElement(SingleTable)
      InsertOnTop(Entry)
```

Figure 8: Update_Entry()

4. Calculate Average

The *Calc_Average* function, used in the *Update_Entry* method, computes the new average time value for a specific entry based on the current local time. Essentially, the function contains three parts with the important last line, where each object receives the time-stamp for the last access. The average value for a newly created object will always be assigned with 0, while the second time when the object got accessed, the *local_time* and the *timestamp* value is used to compute the approximate average rate. In all other cases, we compute an average time-difference between two requests based on the earlier described formula for a simple moving average over two values.

```
Calc_Average()
      IF (hits == 0)
          average = 0;
      ELSE
          IF (hits == 1)
```

```
            average = local_time - timeStamp;
      ELSE
            average=(average+(local_time-timeStamp))/2;
hits++;
timeStamp = time;
```

Figure 9: Calc_Average()

V. EXPERIMENTATION

To validate and verify the described algorithm for adaptive distributed caching, we ran multiple simulations over a set of artificially created client requests. We used the polygraph application as benchmarking tool to create a set of around 4 million requests. In all tests we compared the results of the ADC algorithm to the widely used distributed caching based on a hashing algorithm like CARP and evaluated the performance in regard to hits rate and parameters like HOPS and execution time.

1. Settings

The current experimentations allows the variation of the following five parameters: algorithm, number of proxies, size of single-table, size of multiple-table and size of caching table. Additional parameters like maximum number of hops and changes of the infrastructure can be used but were not applied in our latest work and are part of our future concentration. All tests are based run on the multi-agent platform Carolina [19] with each running agent implements one proxy. Our group had access to a set of 8 equal Pentium III machines and we distributed the agents in such a fashion that each host run exactly one ADC-agent. It should be pointed out at this point that our application only focuses on the handling of requested URLs and the proxies will not cache and transfer the actual objects data.

1.1 Algorithm

We simulated the new ADC algorithm and one simple hashing algorithm based on the widely used CARP approach [29]. A proxy in the CARP algorithm tries to resolve incoming requests by means of its locally cached data and forwards the unresolved request in accordance to a globally known hashing function assigning the requested object to a specific location in the total set of known proxies. If the second proxy cannot resolve the forwarded request, the request will be assigned to the origin server. After the request got resolved the second proxy will store the received data replacing existing information based on the LRU algorithm and forward the request directly to the requesting client, bypassing the first proxy.

1.2 Number of proxies

The number of running proxies is represented by the number of running proxy agents, were each agent receives its individual process for execution. In our test we were able to show that a simulation running on a powerful one Gigabyte memory machine returns the same results as a run spread over a distributed set of

machines were each hosts runs exactly one proxy agent. As shown later, the single CPU approach outperforms the distributed approach in relation to execution time due to the fact that no network communication is involved. For our tests we are able to run any number of proxy agents by means of parallel processes or a maximum of eight proxies when each proxy receives its individual host.

1.3 Single-table Size

Represents the number of equal-sized objects that can be mapped by the single-table. As described earlier, this value is most significant for the performance and hit rate of the overall algorithm. The larger the table size, the larger the part of the request pattern that is temporarily stored for evaluation. On the other side, the larger the table, the more memory is used by the application and needs to be shifted around during the memory access. Our experimentation shall show how different values for the single-table influence the outcome of the system.

1.4 Multiple-table Size

Represents the number of equal-sized objects that can be mapped by the multiple-table. Ideally this table should be as large as the total amount of available cached objects minus the locally cached objects allowing a one-to-one mapping for the content of all proxies but in real system the multiple-table will only be able to trace a fraction of all known objects. Changing this parameter, by increasing and decreasing the table will show how different values affect the overall systems performance. It is strongly assumed that the ideal value for the mapping table size is highly dependent on the characteristics of the experienced request pattern.

1.5 Caching Table Size

Represents the number of equal-sized objects that can be cached in one proxy. The larger the local caching table the more objects can be stored. Due to the fact, that each proxy is limited in its usable resources like RAM and disc space, no proxy is able to store the total of all experienced objects. The cache table will in accordance with our algorithm only store the most often requested objects and changes in the table size shall provide an idea for its relation to the systems performance.

1.6 Request Pattern

For the evaluation of our system, we needed to access to a set of requests, which we can run against our proxies to evaluate the overall performance. We looked into different online available log files of server and proxy systems but due to a lack of description that could allow a third person to repeat our test cases and a lack of large request files we decided to use the widely used polygraph proxy benchmarking tool [32]. The polygraph application 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 the in real-life experienced request distribution of the Internet. The created file comes with a set of almost 4 million requests and is divided into three phases. Phase 1 with around 1.0 million requests covers a simple fill phase with almost no

request repetitions. Phase 2 with around 1.5 million requests offers requests and repeats itself in Phase 3.

```
/* PolyMix-4 workload  */
TheBench.peak_req_rate = 100/sec;
FillRate = TheBench.peak_req_rate;
ProxyCacheSize = 1MB;
```

Figure 10: Polygraph Settings

2. ADC versus Hashing

In this section of the experimental part we compare the performance of our ADC algorithm to the performance of a common hashing algorithm by experienced hits and hops. 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.

2.1 Hit Rate

In the first evaluation Figure 11 we compare the plain hit rate experienced for the earlier described polygraph request pattern of around 3.99 million requests. The diagram shows clearly the three mentioned phases, a fill phase up to around 900k requests, the request phase I and the request phase II. The diagram shows the average hit rate as a moving average over the last 5000 requests and the total number of requests. It is clearly visible that, the ADC algorithm drags after the Hashing algorithm to reach its high values in phase I, but is then after the learning phase is finished quite able to outperform the hashing algorithm by a minimal margin. Further tests, with a repetition of the request pattern and a system with pre-learned information shall be shown in the future work.

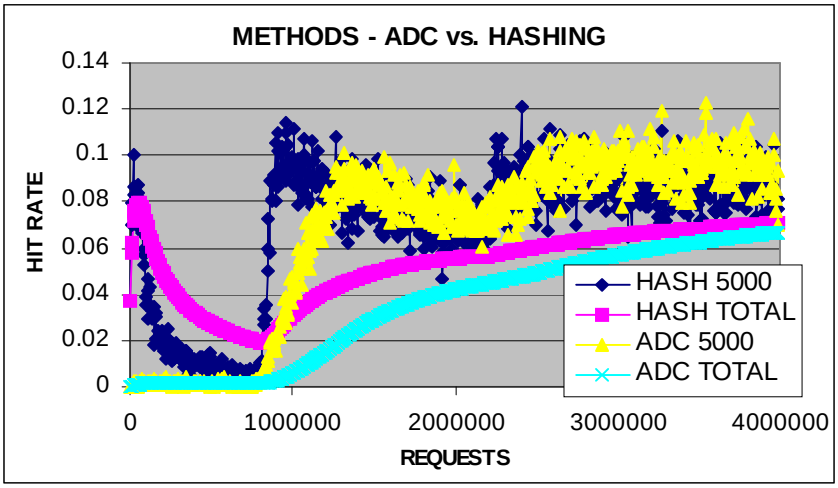

Figure 11: Hit Rate – ADC vs. Hashing

2.2 Hops Rate

Figure 12 shows the comparison of the average number of hops needed to resolve a request. A hop is regarded as the message transfer between client-proxy, proxy-proxy and proxy-server. We can safely claim that on average, the ADC algorithm needs two more hops than the hashing algorithm to resolve an incoming request. This result, on one side, allows the ADC algorithm to search for a specific object more flexibly than the hashing algorithm. On the other side, ADC has longer systems response than the hashing algorithm.

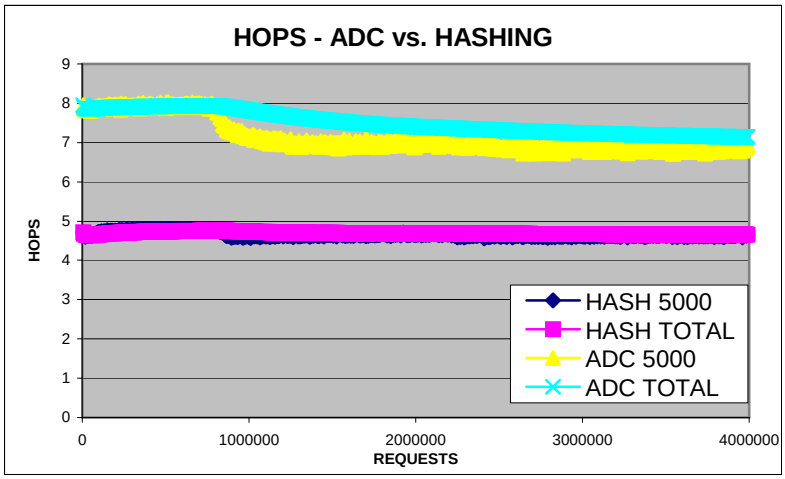

Figure 12: Hops – ADC vs. Hashing

3. Changing Table Size

Our experiments with different table sizes were focused on the size of 5k to 30k for the Caching, Multiple and Single-table. In the evaluation we focused on the three parameters, average Hits, Hops and Simulation Time needed for the 3.99 million requests. The static settings for all simulations were 10k for the caching table and 20k for the single and multiple-table. For example, when we changed the values for the caching table, we kept the size of the single and multiple-table at 20k entries.

3.1 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 observed the overall experienced hit-rate. Looking at Figure 13 we can clearly identify that the size of the caching table is mostly responsible for the value of 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, did not allow the system to improve the hit-rate above the around 0.7 for the caching table size of 10k. Interestingly, even a single-table size of 5k was still able to capture enough requests to allow the system to reach the same 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, but increasing it over the 10k did not significantly improve the number of experienced hits. Future work, will focus

more on the area of under 10k but is safe to say at this point, that the ideal parameters for the three tables is highly dependent on the nature of the experienced request pattern.

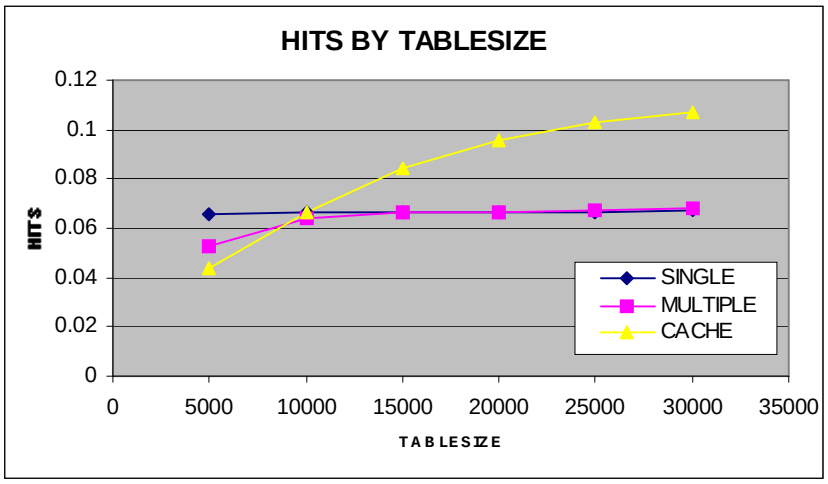

Figure 13: Hit Rates by Table Size

3.2 Hops by Table Size

Looking at the graph for the average number of hops needed to resolve a request (Figure 14), it should be pointed out that even if it appears the graph go through a bread spectrum of changes, of mostly declining nature, the values change only in the area of a ¼ hop, and can be regarded as not significant in comparison of the average number of around 7 hops. The most radical decline can be observed for changes in the single-table. Increasing the single-table, allows more objects to be stored in the system and requests are more likely to be resolved on an earlier base. Similar behavior can be observed for the multiple-table, more objects will be mapped. The slightly decreasing line for the single-table can be explained by the assumption that when old entries from the multiple-table move back into the single-table, they still keep their forwarding information and speed-up minimally the resolution of requests.

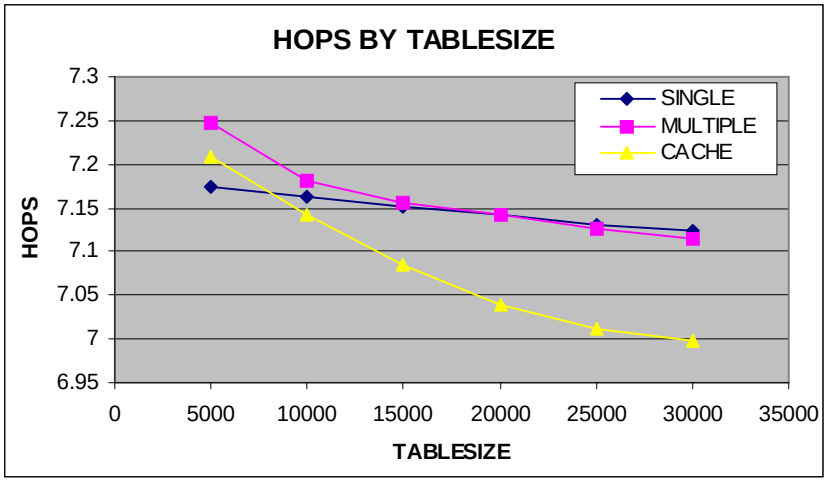

Figure 14: Hops by Table Size

3.3 Time by Table Size

Looking at the graph for the average time needed to resolve a request (Figure 15), it can be stated that increasing the single and multiple-table slows down the overall execution time while increasing caching table has no significant impact. An explanation for the behavior can be given by an in-depth analysis of the time consuming parts of the test-bed. On one hand, we have the actual ADC algorithm during the search process with its time-consuming parts during the access on the single and multiple-table.

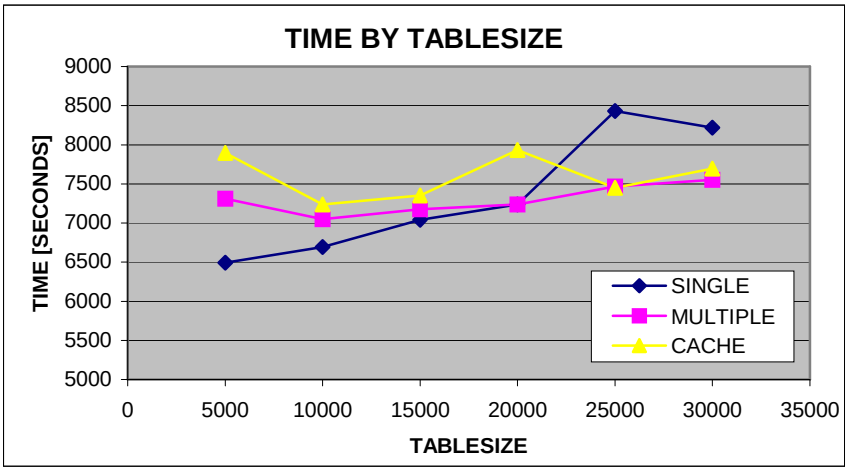

Figure 15: Processing Time by Table Size

While the single-table is based on the LRU algorithm with a linked list requires the element-wise search within the list to take out entries and replace them. Insertion and deletion at the ordered multiple-table is mostly operated by binary search algorithms. Both accesses are extremely time consuming and a more adapted data structure should provide speed-ups in the future versions of this algorithm. Another source of time delays can be found in the amount of used memory by each proxy, paging and context switches cause significant delays during the simulation of the test-bed (in this case all proxies running on one machine). Due to the fact, that most of the used memory is needed for the storage of the actual request URLs, algorithms like MD5 [16] should be used in the future to reduce the amount of required memory.

VI. CONCLUSION

In the presented paper, we focused on two major contributions to our previous work in the area of Adaptive Distributed Caching. The first part of the paper presented a clear description of the actual algorithm, which will allow any interested party to reproduce the work for further research. We presented at the second part of the paper comparisons of the ADC algorithm with a hashing algorithm based on a new 3.99 million request file, and evaluation results

regarding various parameters like caching, multiple- and single-table sizes. We focused on the evaluation of the overall hit and hops rate and have shown their impact on the overall systems performance. Additionally, we pointed out different areas where our work leaves space for improvement and gave a first feel for further evaluations. Additional future work, can be found in the creation of a real proxy system based on the free available Squid server [31], additional improvements on the algorithmic level, further test beds to understand the impact of individual system parameters, the performance comparison based on a new set of request patterns and an evaluation based on the Wisconsin Proxy Benchmark [1], the transfer of our results on new application areas like (server load balancing and resource allocation) and finally the creation of a theoretical framework to explain emerging attributes like merging and self-organization.

VII. REFERENCE

[1] J. Almeida and P. Cao, Measuring Proxy Performance with the Wisconsin Proxy Benchmark, Technical Report, University of Wisconsin Department of Computer Science, April 1998.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, Web Caching and Zipf-like Distributions: Evidence and Implications, Technical Report 1371, Computer Sciences Dept, Univ. of Wisconsin-Madison, April 1998.

[3] 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.

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

[5] J.C. Chuang, "Resource Allocation for stor-serv: Network Storage Services with QoS Guarantees", NetStorage'99, Network Storage Symposium Dec. 1999.

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

[7] L. Fan, P. Cao, J. Alineida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In Proceedings of ACM SIGCOMM Conference, 1998.

[8] J. Gwertzman and M. Seltzer, World-wide web cache consistency. USENIX Symposium on Internetworking Technologies and Systems, pages 141--152, January 1996.

[9] 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, May 1997.

[10] M. J. Kaiser, K. C. Tsui, J. Liu, Self-organized Autonomous Web Proxies, in Proceedings of the First International Joint Conference on Autonomous Agents & Multi-agent Systems, pp. 1397-1404, IEEE, 2002.

[11] M. J. Kaiser, K. C. Tsui, J. Liu, Adaptive Distributed Caching, in Proceedings of the IEEE Congress on Evolutionary Computation, pp. 1810-1815, IEEE, 2002.

[12] M. J. Kaiser, K. C. Tsui, J. Liu, Adaptive Distributed Caching with minimal memory usage, in Proceedings of SEAL 2002, to appear.

[13] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot

spots on the world wide web. In ACM Symposium on Theory of Computing, pp. 654-663, May 1997.

[14] C. Lindemann, and A. Reuys, Modeling Web Proxy Cache Architectures, 1999.

[15] J.-M. Menaud, V. Issarny, and M. Banatre. Improving Effectiveness of Web Caching, in Recent Advances in Distributed Systems, volume 1752 of LNCS. Springer Verlag, 2000.

[16] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, Handbook of applied Cryptography, CRC Press, ISBN:0-8493-8523-7, p. 816, October 1996.

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

[18] S. Paul and Z. Fei. Distributed caching with centralized control. In Proc. of the Fifth International Web Caching and Content Delivery Workshop, Lisbon, Portugal, May 2000

[19] L. Qiu, V. Padmanabhan, and G. Voelker, On the placement of web server replicas, in IEEE INFOCOM, Apr. 2001.

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

[21] P. Rodriguez, C. Spanner, and E. W. Biersack, Analysis of Web Caching Architectures: Hierarchical and Distributed Caching (2001).

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

[23] A. Rousskov and V. Soloviev, On Performance of Caching Proxies, in Proceedings of the ACM SIGMETRICS Conference, Madison, WI, June 1998.

[24] M. Sinnwell and G. Weikum, A cost-model-based online method for distributed caching, In Proceedings IEEE Conference on Data Engineering, Birmingham, GB, 1996.

[25] R. Tewari, 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, May 1999.

[26] 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.

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

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

[29] A. Wolman, G. M. Voelker, N. Sharma, 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 (SOSP '99), pp. 16-31, Kiawah Island Resort, SC, USA, December, 1999.

[30] K.-L. Wu, P. S. Yu, Load Balancing and Hot Spot Relief for Hash Routing among a Collection of Proxy Caches, in Proceedings of the 19th IEEE International Conference on Distributed Computing Systems.

[31] http://www.squid-cache.org/

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