Dynamic proxy-cache multiplication inside LANs

Claudiu Cobârzan, László Böszörményi

Institute of Information Technology
University Klagenfurt
Technical Report No TR/ITEC/05/2.02
February 2005

Abstract

Proxy-cache deployment in the LANs has become a current practice aimed at increasing the availability of the data while also reducing client perceived latency, reduce the load on origin servers as well as the external network bandwidth consumption. As the load increases, due to an increase in client's requests for both cached and non-cached data, it often happens that one single proxy-cache can not handle all the incoming requests. For those situations, when request dropping and cache replacement becomes necessary, we propose an alternative, namely proxy-cache splitting. Our solution is to dynamically deploy additional proxy-caches inside the LAN, and divert towards them some of the requests addressed to the original proxy-cache(s). By doing this we can achieve even better response time, load balancing, higher availability and robustness of the service than in the case in which a single proxy-cache is used.

Contents

1 Introduction

The constant increase in both volume and demand of multimedia data in the Internet, tends to stress the existing infrastructure. The main factors are the characteristics of multimedia data (e.g. size, bandwidth requirements) which highly differ from those of typical web data. The traditional way to cope with such situations is to deploy proxy-caches at LAN edges. There is a vast literature regarding both web and video caching with a lot of attention dedicated to cache replacement strategies. Under certain situations a single proxy-cache does not suffice, so multiple proxy-caches have to be used. Cooperative caching has been introduced for web caches, e.g. Harvest [5] and Squid [18], and for video caches as well, e.g. by Brubeck and Rowe [4] and MiddleMan [1].

Our paper proposes a novel proxy-cache system that is able to "spawn" new proxies via split operations whenever the actual situation demands it. Examples of such situations include extremely high load and severe storage constraints on the proxy. In those cases one additional proxy-cache in the LAN would help lower the load on already running proxy-caches as well as increase the capacity of the "federate" cache. Figure 1 shows such an example with (a) a single proxy-cache servicing the clients of the LAN and (b) the same LAN with three running proxy-caches after two consecutive split operations.

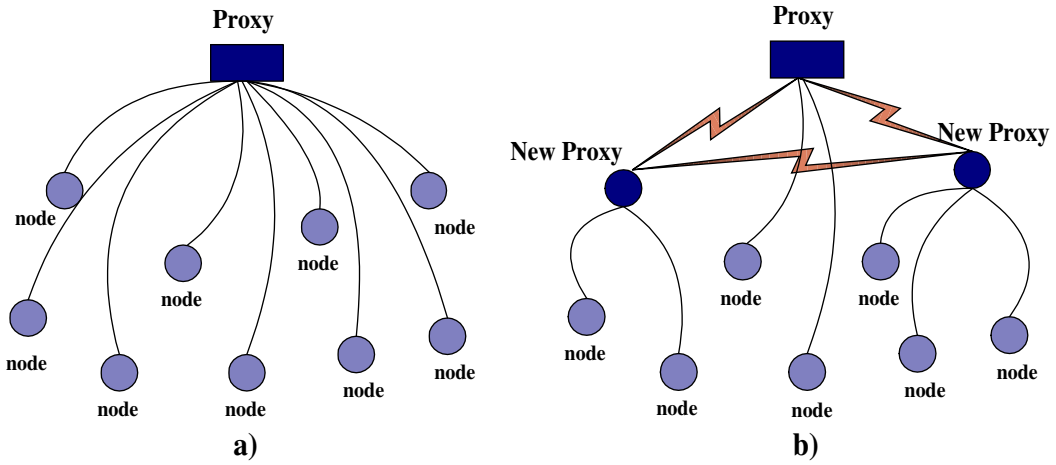

Figure 1: Example of a) LAN with one proxy-cache; b) LAN after the split operation has been performed twice

The system we propose dynamically adjusts the number of running proxies in the LAN, depending on the load and on client request patterns, by either spawning new proxies on periods with high activity or putting them in a "hibernate" state or even stopping them on periods with low activity.

The reminder of this paper is structured as follows. Section 2 describes the proxy-cache split idea in more detail by presenting a first model of the proposed system and two proxy split scenarios. A best-case/worst-case analysis regarding the additional costs in terms of latency and transferred data is also presented. Section 3 outlines the related

work and Section 4 concludes the paper while also presenting directions for future work.

2 Proxy-cache splitting

The aim of our work is to provide a way to achieve better service for clients requesting video content, especially a better response time and higher availability and robustness. Usually this is done by deploying a proxy-cache node inside the network. There are situations when deploying a single proxy-cache does not suffice, for example when servicing large, popular content to many clients, or when the volume of requested data puts the proxy-cache under constraints (cpu, mem, storage, etc.). In those cases requests have to be rejected in order to lower the load on the machine and cache replacement has to be performed in order to free disk space. We state that in some circumstances it would be more beneficial to just deploy an additional proxy-cache inside the LAN, that could take over some of the load on the existing proxy-cache(s) and by doing so, avoid both request dropping and cache replacement. On the other hand, if current and maybe predicted future load could be handled by a smaller number of proxy-caches than those currently active, then some of them could enter a "hibernating" state or could be shut down (stopped). When hibernating, a proxy-cache could only service requests if the desired data is already available in the local cache (no forwarding towards origin servers) or not even service requests at all, but just wait for a reactivation command. Both cases of "hibernation" have certain advantages over shut-down, especially in the situation a future proxy split is needed, at the cost of consuming resources on the host machine during the idle period.

The distributed architecture we propose, assumes the deployment of two types of entities: the `dispatchers` and the `daemons`. The `dispatchers` are processes/threads that run at the same node as the proxy-cache and can be seen as front-ends of the proxy-caches which:

- *handle incoming requests* - serve them either from the local cache, or from the origin server; if this is not possible, the requests are forwarded to other active `dispatchers`/proxies in the LAN or they are discarded; if there are multiple `dispatchers` a request can be forwarded to, the best candidate is chosen (the one that has the requested object cached or if no cached copy exist in the "federate" cache, the one with the smallest load);

- *manage the proxy code* - archive the proxy code and send it to the location on which a new proxy-cache is to be spawned (using the `daemon` running on the selected target)

- *manage the "child" proxy-cache processes* - the `dispatchers` are responsible with stopping/ pausing/ restarting a "child" proxy-cache depending on various conditions (global load, volume of the clients' requests, volume of streamed/stored data, etc.)

The `daemon` processes/threads run in the ideal case on every node of the LAN and are responsible for:

- *managing clients' requests* - the `daemon` either directly receives, or it intercepts the client requests and then decides to forward them to the appropriate proxy-cache, depending on the local available knowledge about the global state of the proxy-caches "federation" (this may include information on the load of the particular proxy-caches it uses as well as information on the data cached by those proxies);

- *managing the proxy code* - the `daemon` receives/compiles the code sent by a `dispatcher` that initiates a proxy split operation;

- *managing the local proxy-cache process* - stops/ pauses/ restarts it, either as a result of incoming requests from its "parent" proxy, or depending on specific local conditions (load, storage capacity).

In the case the `daemon` thread/process hosted on a certain node crashes, the clients from that node still have access to the federate cache as long as the proxy-cache(s) set as default in the client's browser is/are still running. This is also true for the clients from nodes with no `daemons` at all. The `daemon` is essential in the proxy-cache splitting process, as it is used by the "parent" to transfer its code if it is not already available at the selected node.

2.1 The model of the proposed distributed proxy-cache architecture

We consider that the number of nodes in the LAN is **n**. Let **P** be the set of available proxy-caches (there is at least one running proxy cache in the LAN):

$$P = \bigcup_{i=1}^{k} P_i, k = |P|, 1 \leq k \leq n$$

A proxy-cache $\mathbf{P_i}$ is defined as follows:

$$P_i = (maxResources_i, minResources_i, LC_i), i = 1..k$$

where:

- $maxResources_i$ - represents the maximum amount of resources that can be used by the proxy-cache:

  $$maxResource_i = (maxCpu_i, maxMem_i, maxCapacity_i, maxLan_i)$$

  namely the maximum amount of CPU power, memory, storage space and external bandwidth.

- $minResources_i$ - represents the minimum amount of resources that have to be used in order to serve any client's request. It is defined in a similar mode with the $maxResources_i$:
$$minResource_i = (minCpu_i, minMem_i, minCapacity_i, minLan_i)$$

- LC_i - the content of the local cache
$$LC_i = \{c_{ij}, j = 1..q, q = \text{the number of cached objects}\}$$
An object c_{ij} is defined as:
$$c_{ij} = (size(c_{ij}), timeLastAccess(c_{ij}), hitCount(c_{ij}), qualityValue(c_{ij}))$$
where:
  - $size(c_{ij})$ - the size of the object
  - $timeLastAccess(c_{ij})$ - the last time the object has been requested
  - $hitCount(c_{ij})$ - the number of times the object has been requested
  - $qualityValue(c_{ij}) \in [0..1]$ - the measure of the object's quality (based on the actual characteristics of the video object, such as resolution, color information, etc.)

  The $qualityValue$ is a relative value that shows the degree in which the cached object matches the desired quality for a certain class of users. A value equal or close to 1 corresponds to the objects that have exactly or almost the desired quality, while values close to 0 are assigned to objects that show the most drastic difference between actual and desired quality. High absolute quality does not necessarily mean that the $qualityValue$ is close to 1. For example, if the vast majority of the users have only limited display size, say 800x600, a video object encoded at 1280x1024 will have a qualityValue closer to 0 than to 1, because further operations (e.g. transcoding) have to be performed in order to deliver the object to the requesting clients.

  For each object c_{ij}, a *utility* value can be computed using a function $u : LC_i \rightarrow \mathbf{R}$:
$$u(c_{ij}) = const_1 * size(c_{ij}) + const_2 * \frac{1}{timeLastAccess(c_{ij})} + \\ + const_3 * hitCount(c_{ij}) + const_4 * qualityValue(c_{ij})$$
  where $const_1, const_2, const_3, const_4 \in [0, 1]$ and $const_1 + const_2 + const_3 + const_4 = 1$ ($u(c_{ij})$ is computed as a weighted average of the different characteristics of the cached video object).

  Those constants can be fixed when the proxy-cache is started and remain the same during the run period of the proxy-cache. Another possibility that needs further investigation would be to dynamically modify those values when traffic conditions, load level, request rate, etc. reaches certain values, in order to maximize the byte hit ratio. The utility value of the cached objects is used to decide which objects get discarded when performing cache replacement.

We use **D** to denote the set of `dispatchers`:

$$D = \bigcup_{i=1}^{k} D_i, k = |P|, 1 \leq k \leq n.$$

As each `dispatcher` corresponds to a certain proxy-cache, there is a function f (bijection), $f : D \rightarrow P, f(D_i) = P_i, \forall i \in \{1, .., k\}$ (a proxy P has exactly one `dispatcher` D).

One `dispatcher` $\mathbf{D_i}$ is defined as follows:

$$D_i = (P_i, GC, GU, siblings_i), \forall i \in \{1, ..., k\}, k = |P|, 1 \leq k \leq n$$

where:

- P_i - the corresponding proxy
- GC - the content of the global cache (viewed as the union of all local caches)

$$GC = \bigcup_{i=1}^{k} LC_i, \forall i \in \{1, ..., k\}, k = |P|, 1 \leq k \leq n$$

- GU - the *utility* values for the objects in GC

$$GU = \bigcup_{i=1}^{k} LU_i, \forall i \in \{1, ..., k\}, k = |P|, 1 \leq k \leq n$$

  where LU_i = the set of utility values for the objects in LC_i
  $LU_i = \{u(c_{ij}, \forall c_{ij} \in LC_i, i \in \{1, ..., k\}, j \in \{1, ..., q\}, q = |LC_i|)\}$

- $siblings_i$ - the rest of the running proxies/dispatchers

$$siblings_i = P \setminus \{P_i\}$$

We denote with **A**, the set of `daemons`, ideally running on each node of the LAN.

$$A = \bigcup_{i=1}^{n} DA_i$$

There is a function g (bijection), $g : [1..n] \rightarrow A, g(i) = DA_i, \forall i \in \{1, .., n\}$ which assigns each node in the LAN a running `daemon`.

One `daemon` $\mathbf{DA_i}$ is defined as follows:

$$DA_i = (P\prime_i, LOAD(P\prime_i), \bigcup_{p \in P\prime} MLC_p(m)), \forall i \in \{1, ..., n\}$$

where:

- $P\prime_i$ - a subset of the proxy-cache set P ($P\prime_i \subseteq P$)
- $LOAD(P\prime_i)$ - the load of the proxy-caches in the subset $P\prime$

$$LOAD(P\prime_i) = \bigcup_{p \in P\prime} LOAD(p)$$

  where $LOAD(p)$ represents the current load of the proxy $p \in P\prime$ (different formula can be used to compute the load)

- $MLC_p(m)$ - the most "useful" m objects stored in the cache $p \in P\prime$

$$MLC_p(m) = \bigcup_{i=1}^{m} c_{ij}, u(c_{ij}) \geq u(c_{ij+1}), \forall j \in \{1, ..., m-1\}$$

  where c_{ij} represents the cached object and $u(c_{ij})$ the value returned by the utility function defined above for the object.

We denote $request_i(o)$ a request for the object o issued by a client/`daemon` at node i and we define it as following:

$$request_i(o) = (o, UP)$$

where:

- o - the object being requested
- UP - the user preferences (UP = NULL if no preferences specified)

If specified, the user preferences must be in the range supported by the terminal capabilities. It would make no sense to request a video encoded at a 24bit color depth if the terminal can display a maximum of 256 colors. Of course, if the user preferences are specified disregarding the terminal capabilities or they are not specified at all, it is possible that the retrieved content is useless because it can not be visualized.

2.2 Proxy splitting scenarios

As mentioned before we intend to perform a splitting operation under two conditions: when the proxy-cache is under storage constraints or under load constraints. The question is how to decide that a splitting operation is more appropriate than performing cache replacement or reject the incoming requests? We propose the following two condition:

2.2.1 In the case of storage constraints

If $\forall i \in \{1, .., k\}, \forall m, n \in \{1, .., q\}(m \neq n), k = |P|, q = |LC_i|$

$$|u(c_{im}) - u(c_{in})| < \delta \tag{1}$$

then perform splitting, else perform cache replacement.

Cached objects	Size (MB)	Time of last access	Hit count	Quality value
c_1	100	-1	60	1
c_2	100	-5	70	1
c_3	100	-10	80	1
c_4	100	-20	90	1
c_5	100	-30	100	1

Table 1: Characteristics of the cached objects

Configuration	$const_1$	$const_2$	$const_3$	$const_4$
$conf_1$	0.25	0.25	0.25	0.25
$conf_2$	0.10	0.40	0.40	0.10
$conf_3(LRU)$	0	1	0	0
$conf_4(LFU)$	0	0	1	0

Table 2: Values for the coefficients used by the *utility* function u

In other words, splitting is performed when all the cached objects are essentially "equally" useful - the difference between the utility of all the objects in the cache is smaller than a certain fixed limit δ. This means that all cached objects have more or less the same utility and discarding any of them as result of a cache replacement operation would not be advisable. A better option would be to preserve all those objects in the cache. The condition could be relaxed, if considering that not for all, but for a certain fraction of the cached object set, the above mentioned condition holds.

If the condition does not hold, than cache replacement should be performed with regard to the utility of the objects. As an observation, if $const_1 = const_3 = const_4 = 0$ then the cache replacement strategy (CRS) becomes LRU (Least Recently Used), and if $const_1 = const_2 = const_4 = 0$, the CRS becomes LFU (Least Frequently Used).

We present a short example of how the values of those constants could influence the decision of making either a split operation or perform cache replacement. Consider that the cache contains only 5 objects with the characteristics described in Table 1. Consider now Table 2 with four value configurations for the constants that appear in the definition of the utility function (see Section 2.1). The graphical representation of the utility values corresponding to the data in Table 1 and Table 2 can be seen in Figure 2.

It can be seen that the decision to perform either cache replacement or a split operation highly depends on the value configuration of the coefficients. For example, if $\delta = 15$ and the proxy is under storage constraints, cache replacement will be performed if configuration 3 or 4 are used, but a split operation will be initiated if configuration 1 or 2 are considered.

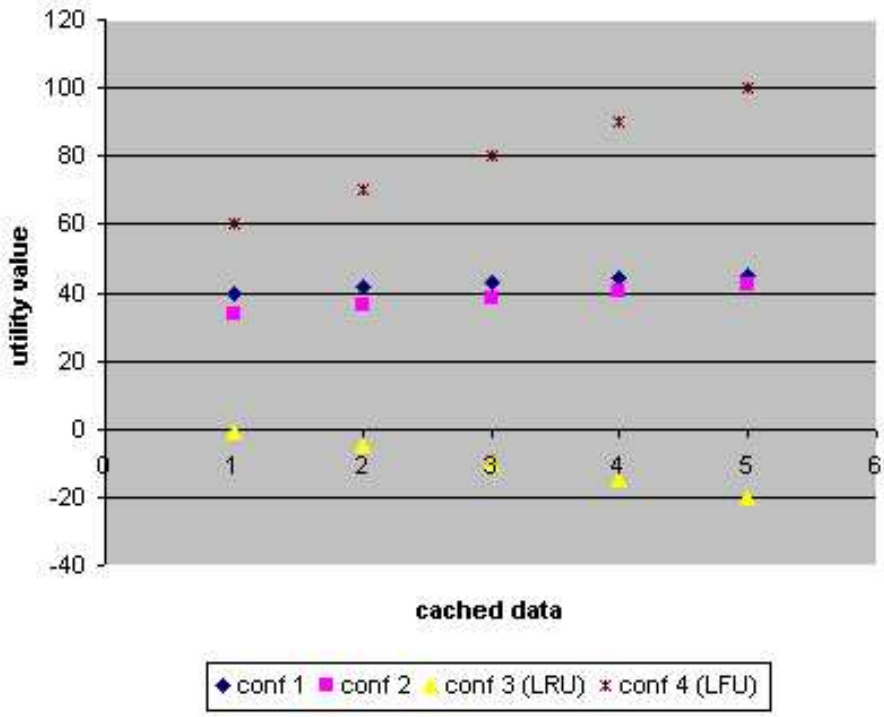

Figure 2: The utility values for the cache configuration from Table 1 computed using the constants values from Table 2

2.2.2 In the case of load constraints

When servicing a request for a certain object c_{ij} a certain amount of resources must be available. If $\forall P_i \in P$ we have $available_i < minResource_i$ than a particular request r_i is discarded and the particular time t_i is marked.

If $\forall i \in \{1, .., p-1\}$ (p fixed) we have

$$t_{i+1} - t_i < \xi \tag{2}$$

(the time interval between p consecutive discarded requests is smaller than a fixed threshold ξ), than we make a split operation.

It is to investigate in a real time environment how different values for δ and ξ influence the dynamic of the system.

2.3 Cache replacement after splitting

If one local cache, say P_i, is under storage constraints while other running proxies are not, requests incoming at P_i could be redirected to its siblings so that cache replacement

is avoided and potential "valuable" objects are preserved in the federate cache.

If all local caches are under storage constraints, and if condition (1) does not hold (a new proxy split is not necessarily advised), cache replacement has to be performed in order to provide storage space for incoming requests. There are two possibilities: either perform cache replacement independently for each cache, or perform cache replacement considering the available information regarding the utility of the objects cached at sibling sites.

If for example, there are two running proxies, and the utility of all the objects in P_1 is greater than the utility of those in P_2, then P_1 may not perform cache replacement, but let P_2 do it. This would require P_1 to signal P_2 to start discarding objects. One elegant solution is to just redirect the first request coming to P_1 towards P_2 and by doing so force the cache replacement, instead of explicitly demanding it via message passing.

2.4 Proxy-to-proxy communication

We consider that in the proposed system, the following types of messages are transmitted:

1. *request* (from a client to the proxy, or from a proxy to another proxy)
2. *response* (from a proxy to a client)
3. *cache information* (information on objects added/removed from a cache)
4. *proxy-cache code* (duplicate the proxy code)

A *send* operation is defined as follows:

$$send(source, destination, MSG\_TYPE, MSG\_CONTENT, MSG\_LEN, UP)$$

where:

- *source* - identifies in a unique way in the system, the initiator of the communication
- *destination* - identifies in a unique way in the system, the recipient of the message
- *MSG_TYPE* - indicates the type of the message (request/response/cache info/proxy-cache code)
- *MSG_CONTENT* - the effective content of the message (rtsp request, rtsp response, etc.)
- *MSG_LEN* - the size of the message
- *UP* - the defined user preferences

When a proxy chooses to forward a client request towards another proxy, it only has to indicate in the source field the client from which it received the original request. The receiving proxy will treat the request as it would have originated from the client. We do this in order to prevent the response being forwarded through multiple proxies on it's way back to the client. The increased complexity at the `dispatcher` side when processing a (forward) request is worth because of the save in streaming capability that would be required when forwarding a response towards a client across multiple proxies.

Regarding the UP (*user preferences*), if none are defined, than the existing objects will be returned in their original form, unless transcoding is required in order to cope with the specified TC (*terminal capabilities*). Specifying both UP and TC introduces additional complexity in the system but also helps to efficiently use existing resources (bandwidth, storage space, processor power) and achieve a great degree of satisfaction among served clients. It is still an open question how different quality variants of the same object should be treated by the system. As an observation, it can be added that the *qualityValue* of the cached objects should be computed while taking into account the user preferences/terminal capabilities of the clients that are being served. If a massive shift in terms of UP/TC is detected (e.g. many and highly active mobile clients, many clients with low resolution displays, etc.) than it would make sense to recalculate the *qualityValue* as well as the *utility* value of the cached objects. This is because the cache replacement process is performed based on the computed *utility* values, and for example, it would make no sense to favor high quality objects when most of the clients have low terminal capabilities/user preferences or vice-versa.

2.5 Additional costs induced by the proposed architecture: best-case/worst-case scenarios

Inside the system, the message exchange cost can be viewed with regard to the required time to transmit a message, with regard to the amount of data that is transferred, or as a combination of the two (both time and data volume).

In the following, we give a short analysis of the best/worst case scenarios from the point of view of the latency perceived by the client. Another possible best/worst case analyses would be with regard on the amount of data transferred within the system.

2.5.1 Latency

The delay perceived by the client depends on the delay introduced by the LAN communication, the one introduced by the WAN communication, as well as on the delay introduced by searching the local caches and the server repository.

We make the following notations:

- *Delay* - the total delay as perceived by the client

- d_{lan} - the delay introduced when transmitting a message (request/response) in the LAN;
- d_{wan} - the delay introduced when transmitting a message (request/response) in the WAN;
- d_{cache} - the delay introduced when searching the local cache (depends on the number of cached objects);
- d_{server} - the delay introduced when searching the server repository/performing admission control;
- $d_{timeOut}$ - the time out interval fixed for the proxy-server communication

We propose the following forwarding algorithm for requests passed from one proxy to another inside the LAN: when a proxy receives a request, it first checks the local cache and returns the appropriate object in case of a hit. Otherwise (local miss) it checks the list with cached objects at siblings sites in order to see if the requested object is cached in the federate cache. If it does, it marks the request and sends it to the appropriate sibling. The decision to forward a request to a certain proxy is made based on the locally available information on the global state of the federate cache. It may happen that this information is outdated and that by the time a forwarded requests reaches the sibling, the requested object does not exist anymore on the sibling site. The *worst case* would be when a request received by a `dispatcher` D_i is forwarded from one dispatcher to the other until returns to D_i. In this case, supposing that the client didn't cancelled the request, it is forwarded by D_i to the origin server S. In the case the client has really bad luck, the server is down or it can't serve incoming requests.

Suppose there are k active proxy-caches, and using the above mentioned notations, we distinguish the following two *worst cases*, when it comes to the user perceived latency:

- bouncing request and server down

$$Delay = (k+1)d_{lan} + kd_{cache} + d_{timeOut}$$

- bouncing request and server can't serve incoming requests

$$Delay = k(d_{lan} + d_{cache}) + 2(d_{lan} + d_{wan}) + d_{server}$$

The *best case* is of course when the first proxy receiving the client request, can serve it from the local cache. In this case we have:

$$Delay = 2d_{lan} + d_{cache}$$

Assuming the following two configuration, *conf1* with $d_{lan} = 0.1, d_{cache} = 0.001, d_{wan} = 0.5, d_{server} = 0.005$ (in seconds each), and *conf2* with $d_{lan} = 0.01, d_{cache} = 0.001, d_{wan} =$

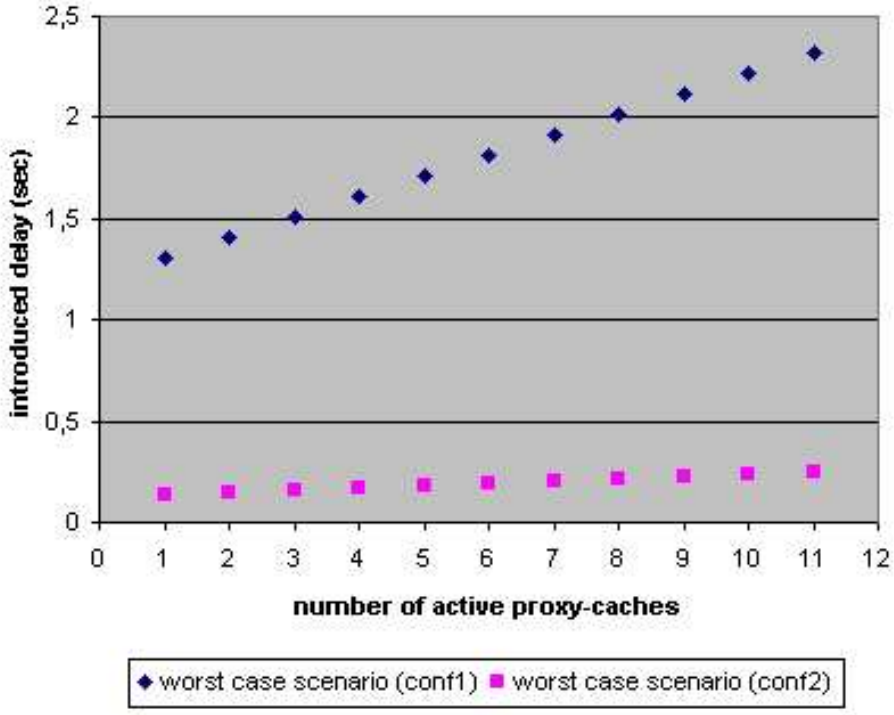

Figure 3: The maximum delay that could be introduced when servicing a request for the values $d_{lan} = 0.1, d_{cache} = 0.001, d_{wan} = 0.5, d_{server} = 0.005$ corresponding to conf1 and for $d_{lan} = 0.01, d_{cache} = 0.001, d_{wan} = 0.05, d_{server} = 0.005$ corresponding to conf2

$0.05, d_{server} = 0.005$ (also measured in seconds) the maximum introduced delay in the case up to 11 proxy-caches are active inside a LAN is showed in Figure 3.

It can be seen from the above example that, if the time needed for a cache and server look-up remains the same (constant load on both cache and server), the introduced latency can vary widely depending on both local and external network conditions. The example shows the "jump" in additional latency corresponding to a 10 time increase in both local and external latency (the values were measured using the `ping` utility for the same hosts during different times of the day).

It can be easily seen that even without constraints regarding the available external bandwidth, it is highly probable for the maximum number of active proxy-caches to be limited by the additional latency that would be induced in the worst case scenario. This holds especially if the network conditions are not very good (we have high induced latencies for both LAN and WAN) as the delay is highly dependable on those conditions.

Recent studies (see [3, 9]) have shown that hit rate and byte hit rate, for both web and multimedia data, grow in a log-like fashion as a function of cache size when classical replacement strategies like LRU, LFU, LFU-DA, etc. are considered. If we apply this knowledge to our previous example with 11 active proxy-caches, with a storage capacity of 50GB each, we get a graphic like the one in Figure 4.

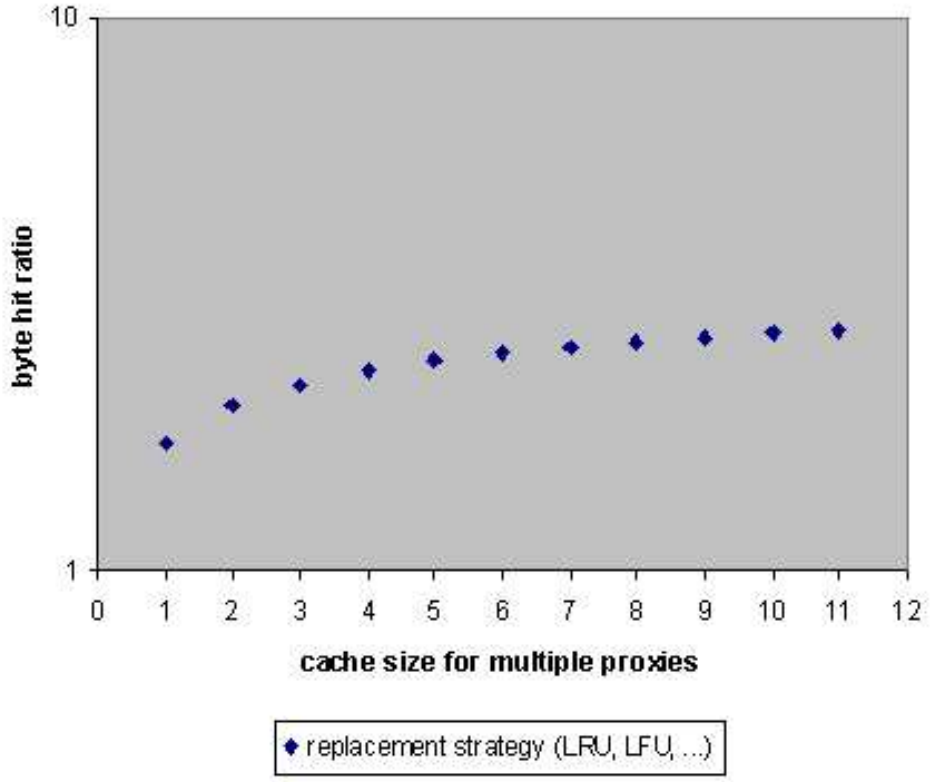

Figure 4: Variation of the bit rate and byte hit rate as function of cache size

From both Figure 3 and Figure 4 we can deduce that the best *costs/benefits* ratio is achieved when a relative small number of split operations are performed. In this case the client perceived latency could be reduced since the amount of data transferred from the external WAN is smaller than the amount of data transferred from the much faster LAN. We intend to validate this assumption in a real time environment, once our implementation of the system (which is based on the existing implementation of the QBIX proxy-cache [15, 17]) is completed. We also want to see if the above mentioned assumption holds when cache replacement is done considering also the size and the quality of the objects.

2.5.2 Transferred data

When we speak about data transferred inside the system we can make the following difference:

- data transferred from the proxy-cache to the client in response to a client request
- data transferred from the proxy-cache to one arbitrary node when doing a split operation
- "maintenance information" when more than one proxy-cache exists inside the LAN

The *best case* scenario, is the one in which a single proxy-cache exists inside the LAN. In this case, the last two situations mentioned above can not occur.

In the case there is a single proxy-cache in the LAN, but a split operation is necessary, the proxy-cache code has to be transferred from the existing proxy-cache to a suitable node in the local network. This operation induces of course some stress on the underlying network because of the volume of transferred data.

In the case of a "global" miss, the requested object has to be requested from an origin server. If it is cached at an arbitrary proxy, this proxy has to announce it's siblings about the new entry, which again induces additional costs.

The *worst case* scenario is when as a result of the announcements of new cached objects, new splitting operations become necessary.

We make the following assumptions:

- the initial cost in terms of transferred data for a message of type request/response is the same, and we note it as c_{reqRes};
- the cost in terms of transferred data for the proxy-cache code transfer is $c_{codeTrans}$
- the cost in terms of transferred data for the "maintenance" information (entries added or removed from a local cache) is c_{maint}

Consider also the following notations:

- n - the number of nodes in the LAN
- k - the number of proxy-caches/`dispatchers` in the LAN
- P_{max} - the maximum number of proxy-caches that can run at the same time in the LAN (obtained as a result of multiple measurements)
- $DATA$ - the amount of data transferred in the system

Now also consider the following *worst case* situation:

1. all the nodes in the LAN make a request for an independent object (this costs $2(n-k)c_{reqRes}$)
2. when receiving the requests and serving them, the changes in the local caches have to be announced to all the siblings (this costs $(k-1)(n-k)c_{maint}$)
3. those requests/responses can also trigger the remaining $(P_{max}-k)$ of split operation (costs $(P_{max}-k)c_{codeTrans}$)
4. supposing that each proxy announces the `daemons` about the new cached objects we have additional costs $(k(n-k)c_{reqRes})$

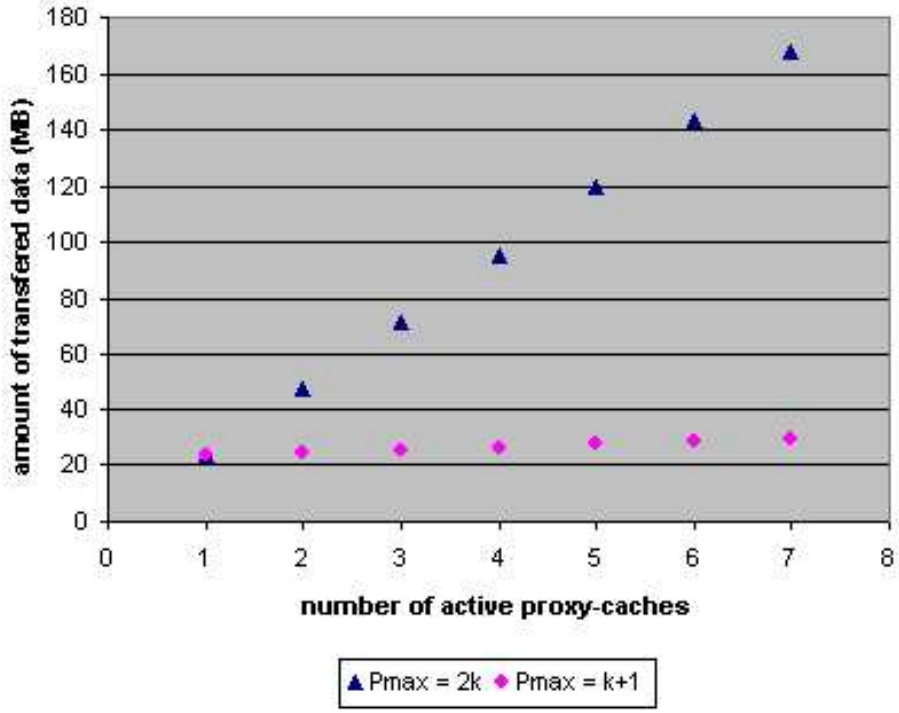

Figure 5: Maximum amount of data transferred in the worst case situation described above

So, the total cost in terms of transferred data, when each client in the LAN makes exactly 1 request for a non-cached object is:

$$DATA = (n-k)[(k+2)c_{reqRes} + (k-1)c_{maint}] + (P_{max} - k)c_{codeTrans}$$

Considering $c_{reqRes} = 0.001, c_{maint} = 0.01, c_{codeTrans} = 23$ (values in MB, estimated for the ViTooKi system [17]) the maximum amount of data that is additionally transferred is shown in Figure 5.

As you can see, there is a big difference in the amount of transferred data depending on the value of P_{max}. $P_{max} = 2k$ means that there is no coordination between the running proxies regarding the decision to perform a split operation. In this case, if the conditions are right, it may happen that all running proxies decide to perform a split operation in the same time, so the number of running proxies actually doubles. This is not the most fortunate decision, not only from the point of view of the additional transferred data. On the other hand, if the running proxies take a common decision ($P_{max} = k + 1$) at the cost of supplemental inter-proxy communication, and perform a single split operation, the additional amount of transferred data remains virtually the same, regardless of the number of active proxies. As it can be seen in Figure 5, the cost of the additional inter-proxy communication needed in order to make a common decision, from the point of view of additional transferred data, is very small and can be ignored.

3 Related work

The last few years have brought an increasing interest in video caching as a result of the rising popularity and availability of multimedia content on the Web. The vast majority of the research concentrates on partial video caching, approach that considers specific parts of videos or is done with respect to the quality of the videos. Examples of proposals for partial video caching include caching of a prefix [16], caching of a prefix and of selected frames [10], caching of a prefix combined with periodic broadcast [7], caching of hotspot segments [6]. Other approaches consider the caching of a prefix based on popularity [11], segment-based prefix caching [19] or variable sized chunk caching [2].

Quality based video caching proposals include periodic caching of layered coded videos [8], adaptive caching of layered coded videos in combination with congestion control [13], quality adjusted caching of GoPs (group of pictures) [14] or simple replacement strategies (patterns) for videos consisting of different quality steps [12].

Regarding distributed video caching we have among others, the work of Brubeck and Rowe [4] proposing multiple video servers accessible via the web and which manage tertiary storage systems as well as the MiddleMan [1] system which proposes a cooperative caching video server.

Our proposal, though having similarities with that in [1], differs from previous work by the fact that our system is dynamic and able to adjust the number of running proxy-caches in the LAN in a fully distributed fashion depending on a number of factors including current load, storage constraints, request patterns.

4 Conclusion and future work

We have presented a distributed proxy-cache architecture which aims at providing better service to LAN clients. The feature that distinguishes our proposal from those made in the past is the dynamic characteristic of our system which is able to adapt itself to changes in access, request and response patterns as well as to changes in network condition.

Future work will focus on finishing the implementation of the system, evaluating its performance in real-life situations and compare the performance with the case in which a single proxy-cache is used. Other points of interest are represented by the conditions triggering the split, hibernate and shut down operations. Another interesting problem is what happens in a system like the one we described, when multiple outgoing links with different capacities are available.

References

[1] Acharya, S., Smith, B.: Middleman: A Video Caching Proxy Server. In: Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video (2002)

[2] Balafoutis, E., Panagakis, A., Laoutaris, N., and Stavrakakis, I.: The impact of replacement granularity on video caching. In: IFIP Networking 2002. Lecture Notes in Computer Science, vol. 2345. Springer-Verlag, Berlin, Germany, (2002) 214-225

[3] Breslau, L., Cao, P., Fan, L., Phillips, G., and Shenker, S.: Web Caching and Zipf-like Distributions: Evidence and Implications. In: Proc. 21st Annual Conf. of the IEEE Computer and Communication Societies, (INFOCOM 99) New York, NY, (1999) 126-134

[4] Brubeck, D.W., Rowe, L.A.: Hierarchical Storage Management in a Distributed VOD System, In: IEEE MultiMedia, Fall 1996, Vol. 3, No. 3

[5] Chankhunthod, A., Danzig, P, Neerdaels, C., Schwartz, M., Worrell, K.: A Hirerchical Internet Object Cache. In: Proceedings of the 1996 USENIX Technical Conference (1996)

[6] Fahmi, H., Latif, M., Sedigh-Ali, S., Ghafoor, A., Liu, P., Hsu, L.H.: Proxy Servers for Scalable Interactive Video Support. In: IEEE Computer, 43(9): (2001) 54-60

[7] Guo, Y., Sen, S., Towsley, D.: Prefix Caching Assisted Periodic Broadcast for Streaming Popular Videos. In: Proceedings of ICC (International Conference on Communications) (2002)

[8] Kangasharju, J., Hartanto, F., Reisslein, M., Ross, K.W.: Distributing Layered Encoded Video through Caches. In: Proceedings of IEEE INFOCOM (2001)

[9] Lindemann, C., Waldhorst,O.: Evaluating the Impact of Different Document Types on the Performance of Web Cache Replacement Schemes. In: Proc. International Performance and Dependability Symposium (IPDS 2002), Washington, DC, (2002)

[10] Ma, W.-H., Du, D.H.-C.: Reducing Bandwidth Requirement for Delivering Video over Wide Area Networks with Proxy Server. In: IEEE International Conference on Multimedia and Expo, (2000) 991-994

[11] Park, H. S., Chung, K.D., Lim, E.J.: Popularity-based Partial Caching for VOD Systems using a Proxy Server. In: Workshop on Parallel and Distributed Computing in Image Processing, Video and Multimedia (2001)

[12] Podlipnig, S., Böszörmenyi, L.: Replacement strategies for quality based video caching. In: Proceedings of the IEEE International Conference on Multimedia and Expo (ICME). Vol. 2. IEEE Computer Society, Piscataway, NJ, (2002) 49-53

[13] Rejaie, R., Kangasharju, J.: A Quality Adaptive Multimedia Proxy Cache for Internet Straming. In: Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (2001)

[14] Sasabe, M., Wakamiya, N., Murata, M., Miyahara, H.: Proxy Caching Mechanisms With Video Quality Adjustment. In: Proceedings of the SPIE Conference on Internet Multimedia Management Systems (2001) 276-284

[15] Schojer, P., Böszörmenyi, L., Hellwagner, H, Penz, B., Podlipnig, S.: Architecture of a quality Based Intelligent Proxy (QBIX) for MPEG-4 Videos, World Wide Web Conference, (2003) 394-402

[16] Sen, S., Rexford, J., Towsley, D.: Proxy Prefix Caching for Multimedia Streams. In: Proceedings of the IEEE INFOCOM99. (1999) 1310-1319

[17] `http://vitooki.sourceforge.net/`

[18] Wessels, D.: Web Caching. O'Reilly, 2001

[19] Wu, K.-L., Yu, P.S., Wolf, J.L.: Segment-Based Proxy Caching of Multimedia Streams. In: Proceedings of the Tenth International World Wide Web Conference (2001)

Institute of Information Technology
University Klagenfurt
Universitaetsstr. 65-67
A-9020 Klagenfurt
Austria

http://www-itec.uni-klu.ac.at

University Klagenfurt