RC21474(96910)10MAY1999 Computer Science/Mathematics

Research Report

Controlled Local Replication for Hash Routing among Proxy Web Caches

Kun-Lung Wu, Philip S. Yu
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

IBM Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Controlled Local Replication for Hash Routing among Proxy Web Caches

Kun-Lung Wu and *Philip S. Yu*

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Correspondent: Kun-Lung Wu
Tel: 914-784-6615; Fax: 914-784-7455
email: klwu@watson.ibm.com

Abstract

This paper studies *controlled local replication* for hash routing, such as CARP, among a collection of loosely-coupled proxy web cache servers. Hash routing partitions the entire URL space among the shared web caches, creating a single logical cache. Each partition is assigned to a cache server. Client requests for non-assigned-partition objects are forwarded to sibling caches. Duplication of cache contents is eliminated and total incoming traffic to the shared web caches is minimized. However, the response times of client requests can increase substantially due to increased inter-cache traffic and CPU overhead. We propose a controlled local replication of non-assigned-partition objects in each cache server to effectively reduce the inter-cache traffic. Trace-driven simulations are conducted to study the performance impact of local replication. The results show that (1) regardless of cache sizes, with a controlled local replication, the average response time, inter-cache traffic and CPU overhead can be effectively reduced without noticeable increases in incoming traffic; (2) for very large cache sizes, a larger amount of local replication can be allowed to reduce inter-cache traffic without increasing incoming traffic; and (3) local replication is effective even if clients are dynamically assigned to different cache servers.

Index Term: controlled local replication, cooperative web caching, CARP, web proxy, hash routing.

1 Introduction

Collections of loosely-coupled web caches are increasingly used by many organizations to allow a large number of clients to quickly access web objects [20, 21, 15, 11, 10, 1, 14]. A collection of cooperating proxy caches have many advantages over a single cache in terms of reliability and performance. They can also be organized in a hierarchical way [20], such as a collection at the local level, a collection at the regional level and another collection at the national level.

In this paper, we study the performance issues of cooperating/shared [1] web caching among a collection of loosely-coupled proxy cache servers. We focus on a single-tier topology, where an organization has a collection of shared web caches connected together by a local area network (LAN) or a regional network, such as a metropolitan area network. Clients are configured to connect to one of the shared caches. When a client requests an object, the object is first searched for within the collection of caches; if not found, one of the shared caches fetches the object from the content server and then forwards the object to the client. All the caches act as *siblings* and no hierarchical order exists among them.

One popular approach to coordinating a collection of cooperating web caches is to use the *internet cache protocol*(ICP) [19]. ICP is an application-layer protocol running on top of user datagram protocol/internet protocol (UDP/IP). It is the engine used by the Harvest/Squid cache software to coordinate and share a hierarchy of web caches [3, 18]. In a single-tier collection of sibling caches, ICP works as follows. A client sends a request to its configured cache server. If this configured cache server cannot satisfy the request, it broadcasts a query using ICP to all the other sibling caches. If at least one sibling cache has the object, the configured cache server then sends an HTTP request for the object to the first sibling that responds to the query with an ICP hit message. Upon receiving the object, the configured cache stores a copy in its cache and forwards the object to the client. If no sibling cache responds after a time-out, then the configured cache server fetches the object from the content server. As a result, multiple copies of the same object can be simultaneously present in the sibling caches.

An alternative to ICP is to use a *hash routing* scheme. Hash routing, such as the cache

[1] In this paper, we use *cooperating web caching* and *shared web caching* interchangeably.

array routing protocol (CARP) [17], has been proposed to effectively coordinate a collection of cooperating web caches [16, 14, 17]. It is a deterministic hash-based approach to mapping an URL object to a unique sibling cache. Hashing thus partitions the entire URL space among the caches, creating a single logical cache spread over many caches. Each proxy cache is responsible for the web objects belonging to the assigned partition. No replication of any cached object exists among the sibling caches [17].

In a hash routing protocol such as CARP, the configured cache server computes a hash function based on the URL of the requested object. [2] If the requested object belongs to the assigned partition, the cache server either returns the object to the client from its cache, if found, or fetches it from the content server. On the other hand, if the requested object belongs to a non-assigned-partition, then the cache server forwards the request to a sibling cache. After receiving the object from the sibling cache, it returns the object back to the client. In order not to replicate any object among the caches, a cache server does *not* place an object on its own cache, if the object does not belong to the assigned partition [14].

Without replication, the effective cache size is the total aggregate of all sibling caches. Global cache hit rate is thus maximized and the total incoming traffic due to cache misses is minimized. However, the average response time for client requests can be significantly degraded because of CPU overhead for processing HTTP request/reply messages and object transmission delays between configured caches and sibling caches. Inter-cache traffic is needed even for repeated requests for the same non-assigned-partition object on the same cache server. Such delays, however, can be effectively reduced by a small amount of local replication. Namely, when a configured cache server receives an object from a sibling cache, a copy of the object is also placed on the configured cache, even though it is a non-assigned-partition object. We call this modified scheme as hash routing with controlled local replication. Local replication reduces the effective aggregate cache capacity and thus could increases the total incoming traffic from the content servers. But, it reduces the amount of traffic between sibling caches and thus the average request response times. It also reduces cpu workloads by reducing the demand for processing HTTP request/reply messages between sibling caches.

[2] Note that hashing can also be executed by every browser. However, we focus on server-executed hash routing in this paper.

In this paper, we study the performance impact of the amount of local replication and cache sizes on the performance of hash routing in terms of average response times, total incoming traffic, total inter-cache traffic and CPU utilization. We propose an effective cache management approach that controls the amount of local replication in order not to significantly degrade the overall caching effectiveness. We examine the trade-off between the increase in total incoming traffic from the content servers to the shared web caches and the reduction in inter-cache traffic. Trace driven simulations were conducted to evaluate the performance trade-offs. The results show that (1) regardless of cache sizes, a relatively small amount of local replication can effectively reduce the average response time, inter-cache traffic and CPU overhead without noticeable increases in incoming traffic; (2) if cache sizes are large, a larger amount of local replication can be allowed to reduce inter-cache traffic without even increasing incoming traffic; and (3) local replication is very effective even if clients are dynamically configured to connect to different cache servers.

There exist many papers on the general issues of proxy web caching, such as [2, 4, 21, 15, 11, 10, 1, 14]. However, none of them deal specifically with the performance issues of controlled local replication for hash routing for a collection of shared web caches. Local replication in hash routing for shared web caches is similar to shared virtual memory [9], distributed file system [13] and remote caching [8] in that they all try to use multiple caches to cooperate on caching. However, there are distinct differences among them. Most importantly, these prior works do not partition the object space among the cache servers. Shared virtual memory and distributed file system decide which workstation to cache a shared object on a demand basis from program executions and file access patterns, respectively. In remote caching, the objective is to find a sibling cache that can cache an object once the object is about to be evicted from another cache. In contrast, the hash partition for which a cache server is responsible is generally fixed, unless the hash function is changed.

The paper is organized as follows. Section 2 describes the details about hash routing with controlled local replication, including the cache management approach to effectively controlling local replication. Section 3 presents the simulation model, system parameters and workload characteristics. Section 4 then shows our performance results from the trace-driven simulations. Finally, Section 5 provides a summary of the paper.

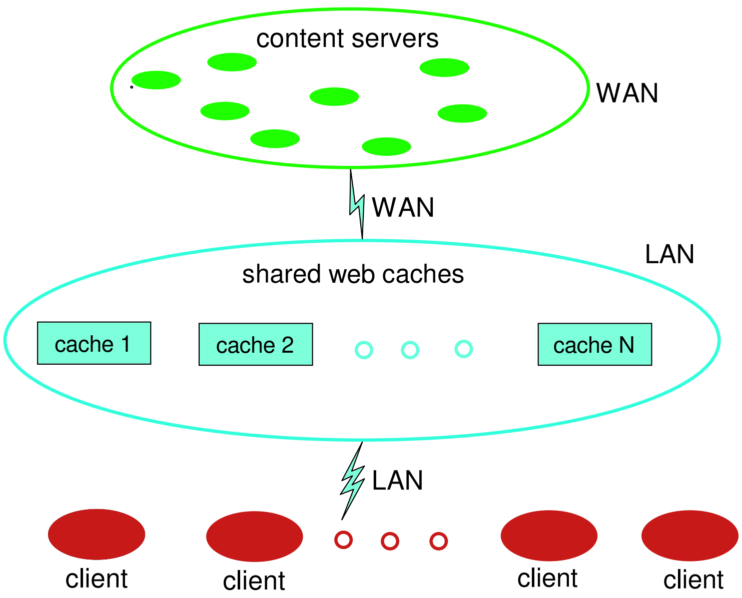


Figure 1: A collection of loosely coupled shared web caches connected by a LAN.

2 Hash routing with controlled local replication

Fig. 1 shows the architecture of a collections of loosely coupled cooperating web caches, which are connected by a LAN or a regional area network. The cooperating web caches are siblings; namely no hierarchical order exists among them. Each client is configured to connect to one of the sibling caches. The client to cache server configuration can either be static or dynamic. We study both cases in this paper. For static configuration, a client connects to the same cache server throughout the simulations. For dynamic configuration, a client may be configured to a different server after a period of time. Details about the dynamic configuration will be described in Section 4.4.

With local replication, there are two kinds of objects in each cache. One belongs to the assigned partition and the other belongs to any of the non-assigned partitions. Objects that do not belong to the assigned partition represent locally replicated objects. To control the amount of local replication, we use two LRU stacks. Fig. 2 shows the implementation of two LRU stacks to control local replication. One is called *local* LRU stack and the other called *regular* LRU stack. Objects belonging to the assigned partition of a cache server can be in either local or regular LRU stack, but locally replicated objects (i.e., objects not in

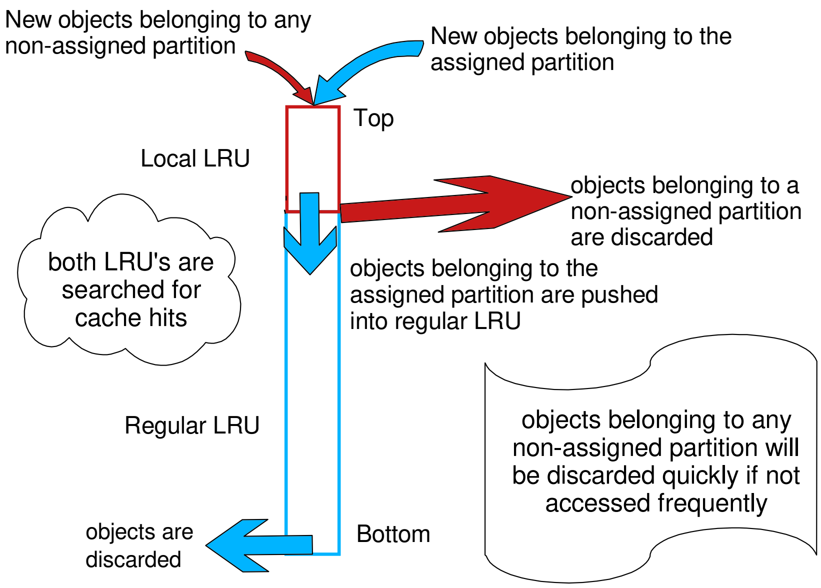


Figure 2: An implementation of two LRU stacks to controlled local replication.

the assigned partition) can only be in the local stack. The aggregate size of the two stacks amounts to the total cache size. When checking if an object is present in a cache, both LRU stacks are searched. On a cache hit, the object is moved up to the top of the corresponding LRU stack.

New objects to be cached are pushed into the top of the local LRU stack. Conceptually, the bottom of the local LRU stack in Fig. 2 represents a threshold in the cache. Any non-assigned-partition object will be discarded once it is pushed beyond the threshold. In our implementation, the objects at the bottom of the local LRU are always replaced by any new objects to be cached. However, if a replaced object belongs to the assigned partition, then a copy of it is pushed into the top of the regular LRU stack, which in turn will push out other objects at the bottom of the regular LRU stack.

This two-LRU cache management effectively controls local replication without degrading overall caching effectiveness. Objects belonging to a non-assigned partition can stay in the cache only if they are frequently referenced. If there is little reference locality to the non-assigned-partition objects, they will be quickly replaced by the assigned-partition objects. In other words, the existence of the local LRU stack does not automatically reduce the cache capacity for the assigned-partition objects. Thus, the locally replicated objects can effectively capture reference locality without unnecessarily occupying cache space.

The modified hash routing protocol with controlled local replication works as follows. Each client is configured to a particular cache server, called *direct server* in this paper. When a client sends an HTTP request to its direct server, it first examines if the object can be found in its own cache by checking both the local and regular LRU stacks. If yes, the object is returned to the client. If not, then the configured direct server computes a hash function based on the request's URL and sends an HTTP request to a unique sibling cache that is responsible for the hash partition to which the URL belongs. This unique sibling cache is called the *partition owner* of the requested object. The partition owner in turn tries to satisfy this request from its cache. If the object can be found, then the partition owner sends the object back to the direct server via an HTTP reply message. If not, the partition owner fetches the object from the content server, places a copy on its cache, and then sends it back to the direct server [3]. Once the direct server receives the object, it places a copy of the object on its cache and then forwards the object back to the client. In the worst case, the object is sent from the content server, to the partition owner, to the direct server and finally to the client, all through HTTP messages. In summary, in the modified hash routing protocol, (a) the direct server first checks its cache for the requested object upon receiving a request from a client; (b) the direct server also stores a copy of an object on its own cache upon receiving it from the partition owner; and (c) separate LRU stacks are used to manage controlled local replication of non-assigned partition objects in each cache server.

3 Simulation implementation

3.1 System model

We implemented a trace-driven simulator that models the modified hash routing with local replication among a collection of sibling caches. Table 1 shows the definition of various system parameters and their default values used in our simulations. Traces were used to drive a collection of 8 sibling cache servers. For each cache server, we implemented a CPU server and a buffer manager. For the CPU server, we implemented a FIFO service queue. The service time for processing an HTTP message, a request or a reply, is T_{http}. The CPU

[3] Note that the direct server can also be the partition owner of an object. In that case, no HTTP messages would be forwarded.

Notation	Definition (Default values)
N	total number of sibling caches (8)
T_{http}	CPU overhead for an HTTP request/reply message (0.02 sec)
T_{cache}	CPU overhead for looking up an object from its cache or storing an object into its cache ($0.5 \times T_{http}$)
T_{hash}	CPU overhead for computing URL-based hashing ($0.1 \times T_{http}$)
T_{lan}	mean message delay from one cache to a sibling cache (0.22 sec)
T_{wan}	mean message delay between a cache and a content server (2.2 sec)
Q	CPU queueing delay
α	threshold (as a percentage of cache size) used for preventing a large object from being cached (0.5%)

Table 1: System parameters.

service time for looking up an object from its cache or storing an object into its cache is T_{cache}. And the CPU service time for computing the hash function based on a request's URL is T_{hash}. We assumed that $T_{cache} = 0.5 \times T_{http}$ and $T_{hash} = 0.1 \times T_{http}$ [4]. The mean message delays for the LAN and WAN are T_{lan} and T_{wan}, respectively. In our simulations, we assumed that the ratio of T_{http}:T_{lan}:T_{wan} is approximately 1:10:100 [5]. We chose 0.02 second as the default T_{http} so that the average CPU utilization during the peak hours will be in the range of 50% $\sim$ 70%.

For each buffer manager, we implemented two LRU stacks: one local LRU and one regular LRU. The sum of the two stack sizes is the total cache size for the server. To prevent an extremely large object from replacing many smaller objects, we set a threshold α for a cached object [1]. Because our emphases are on the impacts of local replication on total inter-cache traffic and total incoming traffic from content servers, we assumed that there was no access delay incurred by the buffer manager in our simulations.

To compute the average request response time, we assumed that the mean network delay of a message between sibling caches is T_{lan} and that between a cache and a content server is T_{wan}. We had an FIFO queue on each CPU to account for the CPU queueing delay, Q. Simulation for response time begins when a request is submitted to the direct server according to its timestamp on the traces. It takes the direct server T_{http} seconds to process

[4]We varied the sizes of T_{cache} and T_{hash} relative to T_{http} and found that the results were not sensitive to T_{cache} and T_{hash}.

the HTTP request. If local replication is allowed, T_{cache} seconds are required to search if the requested URL is in its cache. If yes, the object is returned to the client and the request finishes after T_{http} seconds for the direct server to process an HTTP reply message. Hence, if a request is completed by a cache hit on its direct server, the response time is as follows:

$$T_{direct-server-hit} = 2 \times T_{http} + T_{cache} + Q. \tag{1}$$

But, if the requested object is not found in the direct server and it is an assigned-partition object, then the direct server would fetch the object from the content server. The response time will be:

$$T_{direct-server-miss} = 4 \times T_{http} + T_{hash} + 2 \times T_{cache} + 2 \times T_{wan} + Q. \tag{2}$$

The direct server receives an HTTP request message, checks its local cache and does not find the object, computes the hashing, and sends an HTTP request to the content server. A round-trip message delay in the WAN is added.

If a request is forwarded to a partition owner and it is a cache hit, the response time will be:

$$T_{partition-owner-hit} = 6 \times T_{http} + T_{hash} + 3 \times T_{cache} + 2 \times T_{lan} + Q. \tag{3}$$

In this case, the partition owner receives an HTTP request message, checks its own cache and find the object, and sends an HTTP reply message to the direct server together with the object. The direct server then receives an HTTP reply message, places a copy of the object in its cache, and sends an HTTP reply message back to the client. Note that the queueing delay, Q, is the total delay that might occur on both servers. If local replication is not allowed, we can save $2 \times T_{cache}$ in Eq. 3 by the direct server. But, there would be no possibility of local hits for non-assigned-partition objects.

Finally, if the forwarded request cannot be serviced by the partition owner, the object will be fetched from the content server and the response time will be as follows:

$$T_{partition-owner-miss} = 8 \times T_{http} + T_{hash} + 4 \times T_{cache} + 2 \times T_{wan} + 2 \times T_{lan} + Q. \tag{4}$$

In this case, a round-trip message delay in the WAN and a round-trip delay in the LAN are added to the response time in Eq. 4. Two more T_{http} are added to account for sending and receiving the HTTP request to and from the content server by the partition owner. A total of $4 \times T_{cache}$ are needed because the direct server and the partition owner both have a cache miss and both place a copy of the object in their own caches.

In order to capture the majority of traffic, we measured the total incoming traffic from the content servers by counting the total object sizes that must be fetched from the web servers to the shared web caches. Total incoming traffic is caused by collective cache misses by the shared web caches. More incoming traffic represents less caching effectiveness. For inter-cache traffic, we counted the total object sizes transmitted from the partition owner to the direct server. However, the total number of messages exchanged among the sibling caches was also used to measure the total inter-cache traffic. More inter-cache traffic increases client response times.

3.2 Workload characteristics

Proxy traces collected between 08/29/1996 and 09/09/1996 by Digital Equipment Corporation (now merged with Compaq Computer Corporation) were used to drive our simulations [12]. The sizes of the 12 days' traces varied from about 300,000 to over 1,300,000 entries. Each trace entry contains timestamp, client, server, size, type, url and other information. The client, server, and url were already mapped into unique integers to protect privacy. These traces were used to simulate requests made to different proxy servers in our simulations.

To determine the proper cache sizes for our simulations, we calculated through simulations the maximal buffer size needed for each proxy cache if no replacement were to be required. We called this *MaxNeeded* [1]. Fig. 3 shows the trace sizes (in terms of total entries) and the size of *MaxNeeded* for each trace. For *MaxNeeded*, we show the maximum, minimum and the average of the 8 sibling proxy caches. We assumed clients are configured to connect to the 8 proxy servers in a round-robin fashion statically. Namely, they are uniformly configured to the cache servers. For the 6 days where traces are large (over 1 million entries), the average *MaxNeeded* is about 700–750M bytes per cache. In other words, the *MaxNeeded* for the entire trace is about 5G to 6G bytes. The smallest *MaxNeeded* is about 200M bytes per cache. As a result, for most of our simulations, we assumed 350M bytes per cache, or about 50% of the *MaxNeeded* for the large traces, as the default cache size for each proxy cache. We also used 10% and 90% of *MaxNeeded* in our simulations.

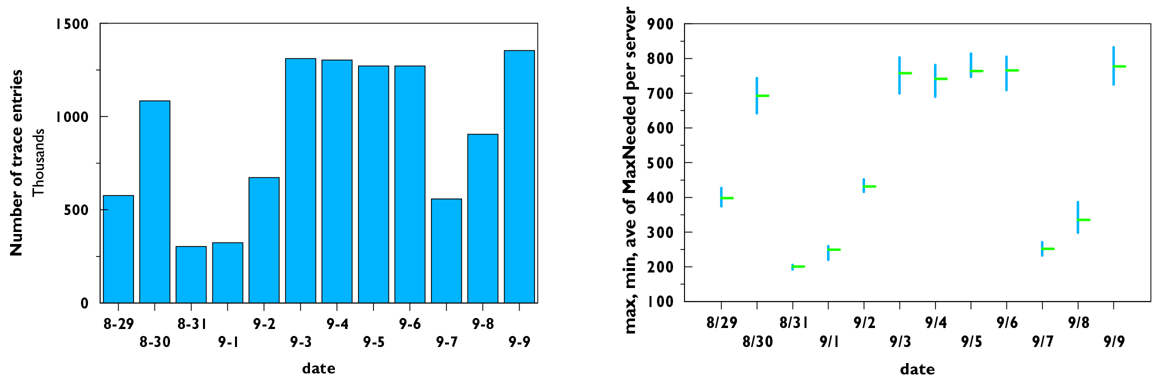


Figure 3: Trace sizes and *MaxNeeded* for each trace.

4 Results

4.1 Hash routing vs. independent caches

We first compare a simple hash routing protocol, such as CARP, with independent caches. In the case of independent caches, each cache server services all the requests sent to it by the configured clients. There is no forwarding of requests to other caches. And, there is no cooperation among the caches. Hence, same objects may be present in all of the shared caches, resulting more total incoming traffic to the collection of caches. However, there is no inter-cache traffic and client response times are faster.

Simple hash routing based on URL eliminates object duplication among the shared web caches. The total incoming traffic to the collection of shared proxy caches is thus minimized, compared with independent caches. But, because of the increased CPU overhead and inter-cache traffic caused by processing additional HTTP messages and object transmissions between cache servers, the average response times for client requests can increase substantially.

Fig 4 shows the trade-off between the average response times and total incoming traffic for simple hash routing as compared with independent caches. The cache size for each proxy server is 350M bytes, representing about 50% of its average *MaxNeeded* (about 700M). We showed the response times and incoming traffic during the peak hours between 5:00 and 11:40 am on the 9-4-1996 trace. Each data point on the response time represents the average of all the requests finished during the subsequent 20-minute interval. For example, the data

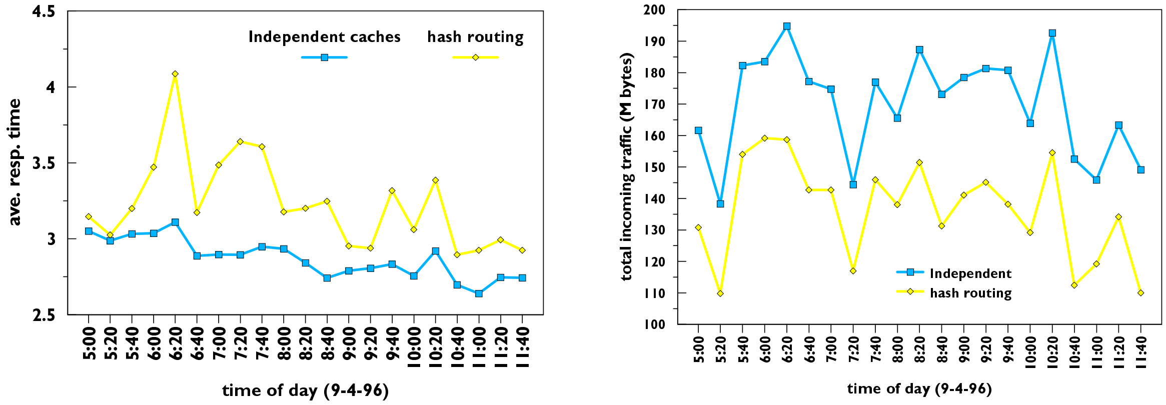


Figure 4: The trade-offs of response time and incoming traffic between hash routing and independent caches.

point at 5:00 am represents the average of requests finished between 5:00 and 5:20 am. The total incoming traffic is the total object sizes that were to be fetched from the content servers during the 20 minute interval by all cache servers. Note that we did not include the HTTP messages in the traffic calculation. As shown in Fig. 4, simple hash routing uses the aggregate cache more efficiently, thus reducing the total incoming traffic. However, such cache efficiency is achieved at the expense of higher average response time due to increased inter-cache traffic.

4.2 The impact of local replication

Fortunately, inter-cache traffic can be effectively reduced by proper introduction of local replication of non-assigned-partition objects in the hash routing protocol. With local replication, the configured cache server (the direct server) first looks into its own cache, then forwards, if necessary, the request to a sibling cache (the partition owner) on a cache miss. To see the impact of local replication, we simulated the modified hash routing protocol with the amount of local replication ranges from 0% to 100% of the proxy cache size. Note that 10% local replication means replication of objects in a cache can be up to 10% of the cache size. Because object replacements occur at the bottom of the local LRU stack, locally replicated objects (those not belonging to the assigned partition) will quickly be replaced by new objects if these replicated objects are not referenced in the immediate future.

Fig. 5 shows the average response times and the total incoming traffic of the modified hash

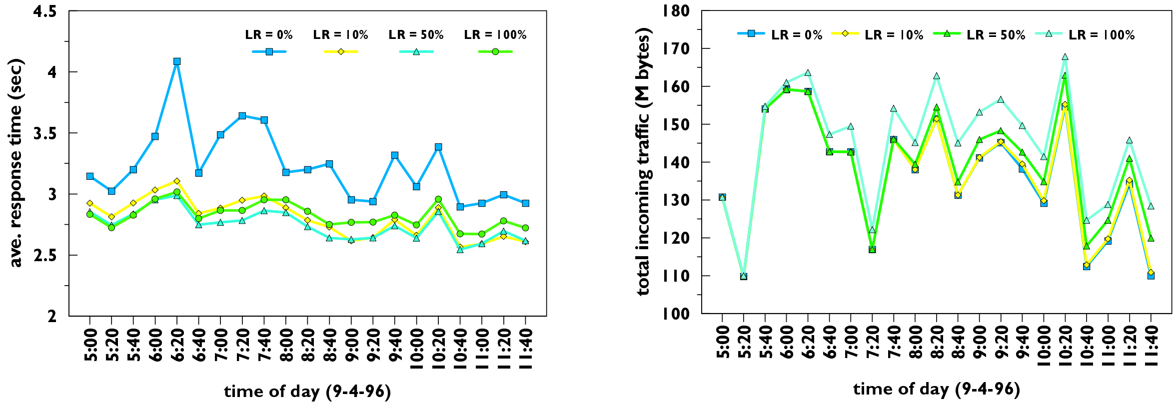


Figure 5: The impact of local replication on response time and incoming traffic.

routing with different amounts of local replication. Fig 6 shows the total inter-cache traffic and the corresponding CPU utilization. The results were for the peak hours between 5:00 and 11:40 am of 9-4-1996 trace. The cache size was 350M bytes per cache. In general, the average response time, total inter-cache traffic and CPU utilization reduces as the amount of local replication increases, but the total incoming traffic increases due to replication of objects among the caches. With a relatively small amount of local replication, such as 10% of cache size from Fig. 5, the average response time, the inter-cache traffic and CPU utilization can be substantially reduced without any noticeable increase in total incoming traffic. Note that in Fig. 5 the total incoming traffic for the cases of $LR = 0\%, 1\%$ and 10% [5] are all indistinguishable. In other words, there is no noticeable increase in total incoming traffic while substantial improvements in average response times, inter-cache traffic and cpu utilization can all be achieved for those cases. However, comparing the cases of $LR = 100\%$ and $LR = 50\%$ with that of $LR = 0\%$, the improvements in the average response time, inter-cache traffic and cpu utilization are achieved at the expense of a moderate increase in total incoming traffic. In other words, the negative impact of local replication starts to appear as the amount of local replication becomes large. These results demonstrate that, with properly controlled amount of local replication of non-assigned-partition objects, substantial performance improvements can be achieved without noticeable increases in incoming traffic.

We also simulated all 12 days' traces for different amounts of local replication. Fig. 7 shows the impact of local replication on the daily total incoming traffic and total inter-

[5] LR stands for Local Replication.

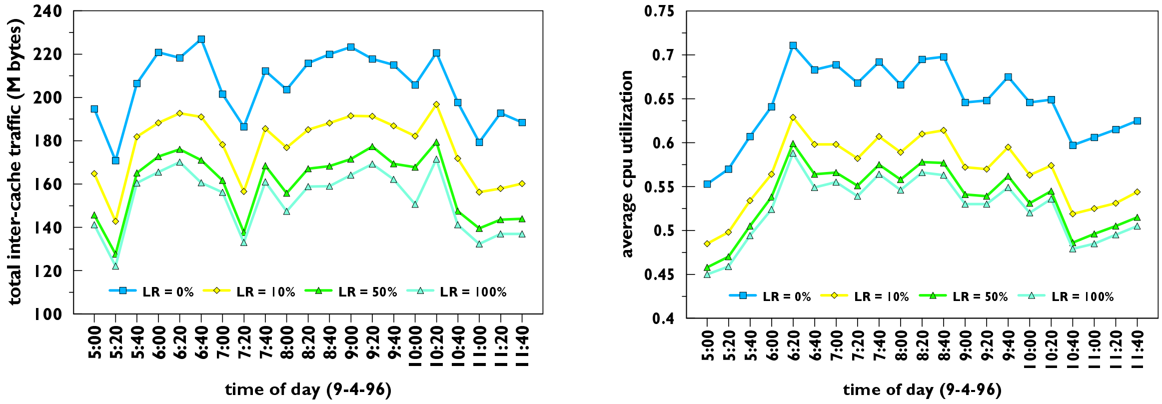


Figure 6: The impact of local replication on inter-cache traffic and cpu utilization.

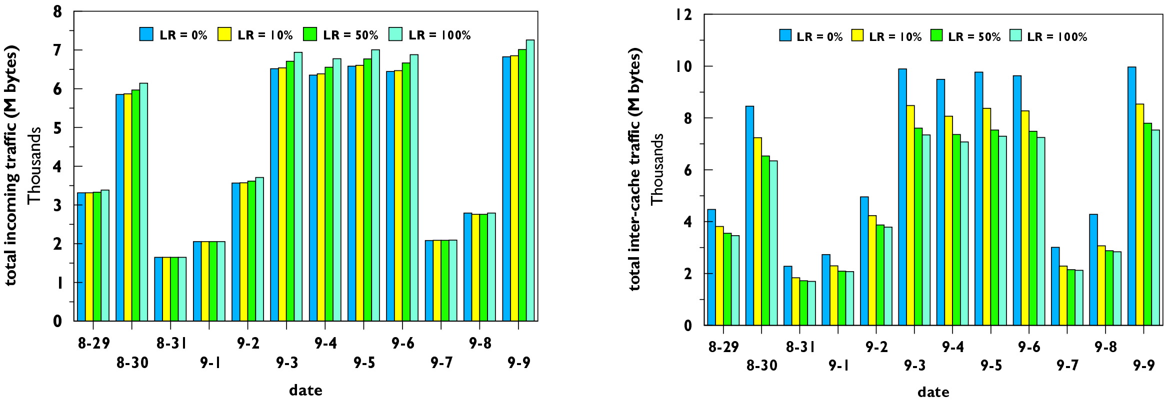


Figure 7: The impact of local replication on daily total incoming and inter-cache traffic.

cache traffic. For these simulations, the cache size was 350M bytes for each cache server. In general, we observed significant reductions in inter-cache traffic for all traces without noticeable increases in total incoming traffic for the case of 10% local replication. For some cases where *MaxNeeded* is smaller than 350M bytes (such as 8-31, 9-1, 9-7 and 9-8) (see Fig. 3), a larger local replication (e.g., 100%) can be used to reduce inter-cache traffic without any penalty in increased incoming traffic.

4.3 The impact of cache size

So far, we have been using 350M bytes as the default cache size for each proxy server. It is about 50% of the *MaxNeeded* for each server. From Fig. 7, we also noticed that, with large

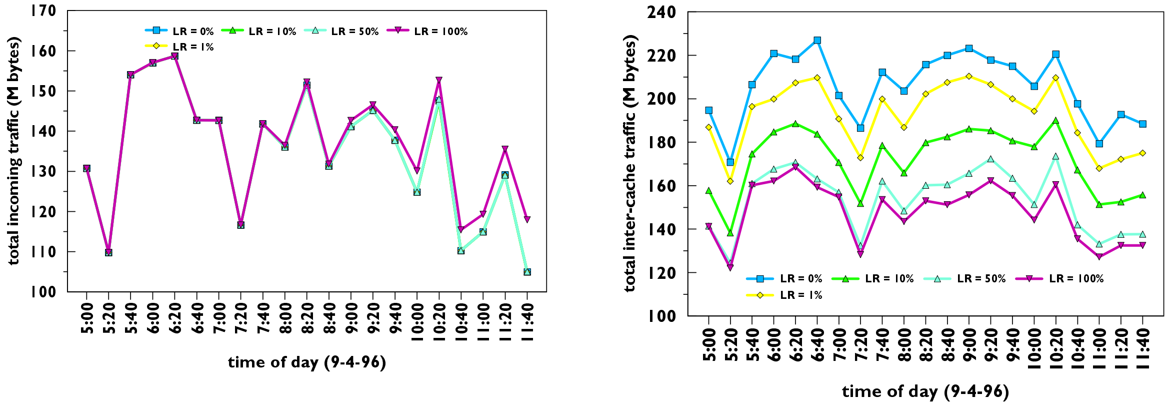


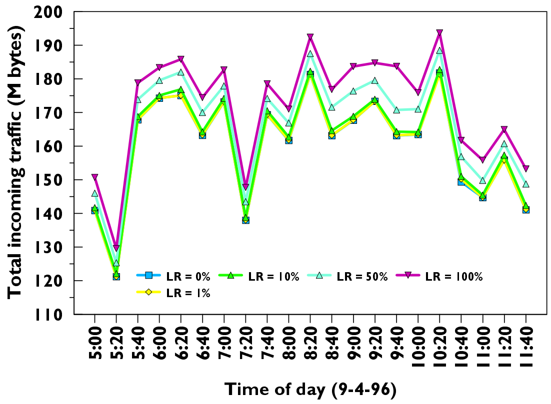
Figure 8: The impact of local replication when cache size is large.

cache sizes, more local replication can be allowed to improve performance without increasing total incoming traffic. Here, we examine the impact of different cache sizes on the various system performance.

Fig. 8 show the impacts of various amounts of local replication when the cache size is 630M bytes per server (about 90% of *MaxNeeded*). We show both the total incoming traffic and inter-cache traffic. Note that it is not distinguishable in total incoming traffic for the cases of 0%, 1%, 10%, and 50% local replication. In other words, there is no additional incoming traffic caused by local replication for those cases. However, there are significant improvements in the corresponding inter-cache traffic. Comparing Fig. 8 with Fig. 5 and Fig. 6, it is clearly shown that, with a large cache, a significant amount of local replication can be allowed to improve system performance without degrading the overall caching effectiveness.

Fig. 9 shows the total incoming traffic and total inter-cache traffic with various amounts of local replication when the cache size is only 70M bytes per server. Note that 70M bytes per server represents 10% of *MaxNeeded*. Even with a small cache size, local replication of about 10% of the cache size can still significantly reduces the inter-cache traffic without a noticeable increase in total incoming traffic. But, the increase in incoming traffic is more significant for a larger amount ($> 50\%$) of local replication.

To see the impact of various cache sizes on all the traces, Fig. 10 and Fig. 11 show the daily total incoming traffic and daily relative inter-cache traffic both in terms of total data amount and total number of messages exchanged. The amount of local replication for these cases was 10%. Obviously, the larger the cache size, the less the total incoming traffic is. All



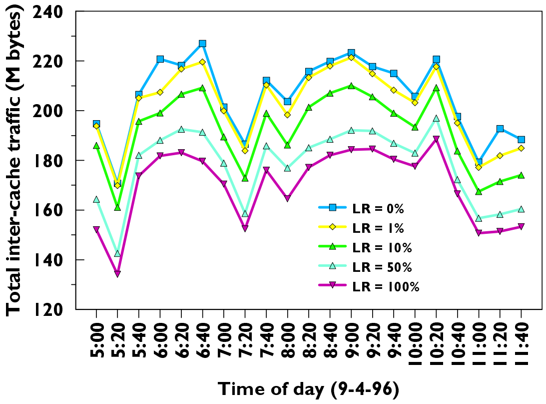


Figure 9: The impact of local replication when cache size is small.

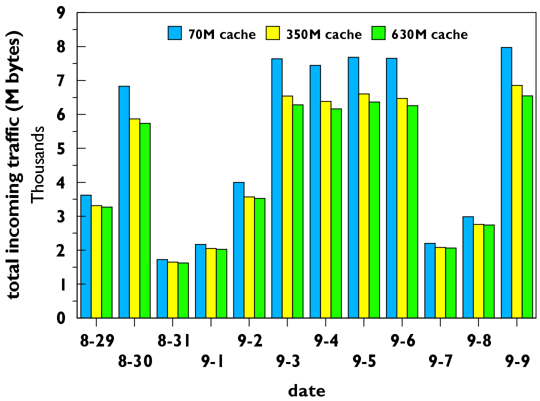


Figure 10: The impact of cache size on daily incoming traffic.

three caches sizes show inter-cache traffic savings in every trace and the savings are more significant as the cache size increases. In terms of total number of messages exchanged, a 350M-byte cache can save about 30% for all traces among 8 sibling caches. In terms of total data amount, the savings is about 15% for the case of 350M-byte cache. For the 9/7/1996 and 9/8/1996 traces, the savings is even larger than 50% in terms of total messages.

4.4 The impact of dynamic client configuration

Finally, we examine the impact of dynamic client configuration on the effectiveness of local replication. The basic idea of local replication is that object references by the clients connected to the same direct server exhibit reference locality. Hence, once an object is forwarded

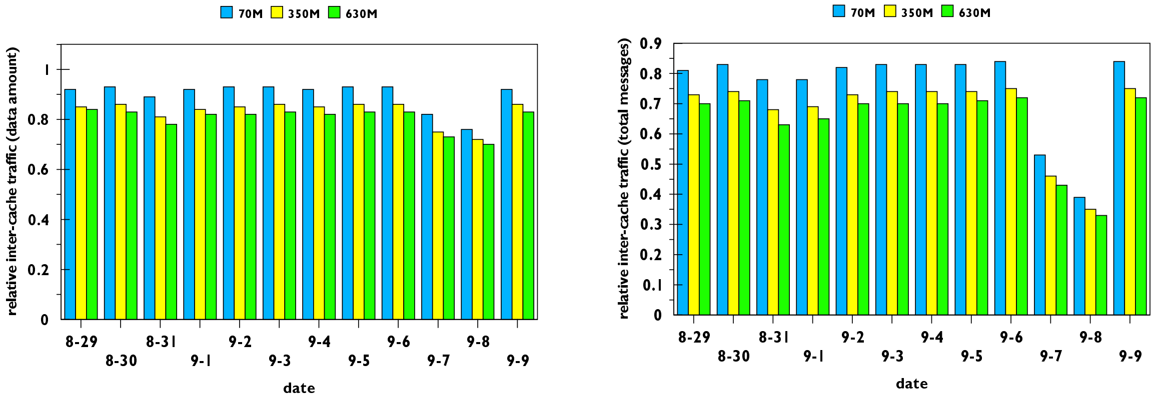


Figure 11: The impact of cache sizes on daily inter-cache traffic in total data amount and total messages.

from a sibling cache and replicated in the local cache, future references to the object can be satisfied locally without incurring inter-cache traffic. But, if client configuration changes dynamically, the effectiveness of local replication may degrade. A frequently used approach to dynamically configure a client to a collection of servers is the round-robin DNS (domain name server) approach [6, 7]. In this experiment, we assumed that there is a name server for the clients to ask for name-to-address resolution. In this case, the collection of shared web caches all share the same logical address. Each client asks the name server for the IP address of the proxy cache and sends the HTTP request to the IP address (the direct server). The IP address will be valid for TTL (time to life) seconds. After TTL seconds, a client would ask the name server again for a name-to-address resolution. The name server simply assigns, in a round-robin fashion, a new cache server to each of the coming name-to-address resolution request. The static configuration assumed in all of our simulations so far is equivalent to the case of TTL $= \infty$.

Here we examine the impact of TTL on the daily total savings in inter-cache traffic (see Fig. 12). A cache size of 350M bytes and a 10% local replication were employed for these simulations. Generally, as TTL increases, more significant savings in inter-cache traffic can be achieved. More importantly, even if TTL is as small as 5 minutes, substantial benefits of local replication can still be achieved for all traces.

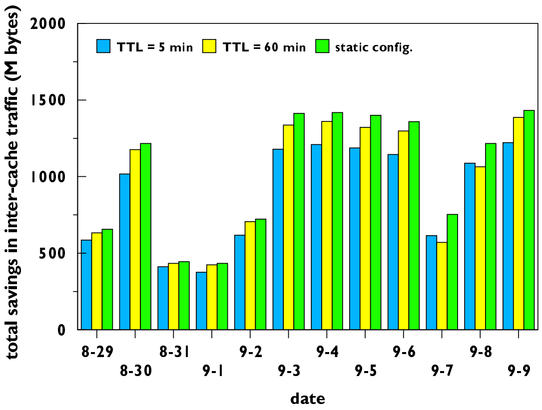


Figure 12: The impact of dynamic client configuration on the daily total savings of inter-cache traffic.

5 Summary

In this paper, we examined the performance issues of hash routing among a collection of shared web caches. A simple hash routing, such as CARP, maximizes the aggragate cache utilization by eliminating duplication of cache contents. But it penalizes client response times and cpu utilizations. We proposed an implementation of controlled local replication for non-assigned-partition objects in a hash routing protocol. We studied the effectiveness of the controlled local replication. Actual proxy traces were used to evaluate the performance impact of local replication on the caching effectiveness. The results showed that (1) a small amount of controlled local replication for non-assigned partition objects can greatly improve the performance of shared web caching without incurring noticeable increase in incoming traffic, even for a very small cache size; (2) for a large cache size, almost the entire cache can be used to store non-assigned-partition objects as well as assigned-partition objects to significantly improve the system performance without degrading the total caching effectiveness; and (3) local replication is also effective even if clients are dynamically configured to different cache servers.

References

[1] M. Abrams et al. Caching proxies: Limitations and potentials. In *Proc. of 4th Int. World Wide Web Conference*, 1995.

[2] C. C. Aggarwal et al. On caching policies for web objects. Technical report, IBM T. J. Watson Research Center, 1996.

[3] C. M. Bowman et al. The Harvest information discovery and access system. In *Proc. of 2nd Int. World Wide Web Conference*, pages 763–771, 1994.

[4] A. Chankhunthod et al. A hierarchical internet object cache. In *Proc. of 1996 USENIX Technical Conference*, 1996.

[5] M. D. Dahlin et al. Cooperative caching: Using remote client memory to improve file system performance. In *Proc. of 1st Symp. on Operating Systems Design and Implementation*, 1994.

[6] E. D. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–163, 1994.

[7] T. Kwan, R. McGrath, and D. A. Reed. NCSA's World Wide Web server: Design and performance. *IEEE Computer*, pages 68–74, 1995.

[8] A. Leff, J. L. Wolf, and P. S. Yu. Replication algorithms in a remote caching architecture. *IEEE Trans. on Parallel and Distributed Systems*, 4(11):1185–1204, Nov. 1993.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, 7(4):321–359, Nov. 1989.

[10] R. Malpani, J. Lorch, and D. Berger. Making world wide web caching servers cooperate. In *Proc. of 4th Int. World Wide Web Conference*, 1995.

[11] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance issues of enterprise level web proxies. In *Proc. of 1997 ACM SIGMETRICS*, pages 13–23, 1997.

[12] Digital Equipment Corporation (merged with Compaq). Digital's web proxy traces. http://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html, 1996.

[13] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Trans. on Computer Systems*, 6(1):134–154, 1988.

[14] K. W. Ross. Hash-routing for collections of shared web caches. *IEEE Network Magazine*, pages 37–44, Nov.-Dec. 1997.

[15] R. Tewari et al. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical report, UTCS TR98-04, Department of Computer Science, University of Texas at Austin, 1998.

[16] V. Valloppillil and J. Cohen. Hierarchical HTTP routing protocol. Internet Draft, http://ircache.nlanr.net/Cache/ICP/draft-vinod-icp-traffic-dist-00.txt, Apr. 1997.

[17] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. Internet Draft, http://ircache.nlanr.net/Cache/ICP/draft-vinod-carp-v1-03.txt, Feb. 1998.

[18] D. Wessels. Squid internet object cache. http://squid.nlanr.net/Squid/, 1998.

[19] D. Wessels and K. Claffy. Internet cache protocol version 2. Internet Draft, http://ds.internic.net/internet-drafts/draft-wessels-icp-v2-00.txt.

[20] N. J. Yeager and R. E. McGrath. *Web Server Technology: The Advanced Guide for World Wide Web Information Providers.* Morgan Kaufman, 1996.

[21] P. S. Yu and E. A. MacNair. Performance study of a collaborative method for hierarchical caching in proxy servers. *Computer Networks and ISDN Systems*, 30:215–224, 1998.