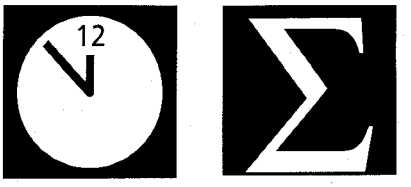


Hash Routing for Collections of Shared Web Caches

Keith W. Ross
University of Pennsylvania

Abstract

Shared Web caches, also referred to as proxy Web servers, allow multiple clients to quickly access a pool of popular Web pages. An organization that provides shared caching to its Web clients will typically have a *collection of shared caches* rather than just one. For collections of shared caches, it is desirable to coordinate the caches so that all cached pages in the collection are shared among the organization's clients. In this article we investigate two classes of protocols for coordinating a collection of shared caches: the ICP protocol, which has caches ping each other to locate a cached object; and the hash routing protocols, which place objects in the shared caches as a function of the objects' URLs. Our contribution is twofold. First, we compare the performance of the protocols with respect to cache-server overhead and object retrieval latency; for a collection of shared caches, our analysis shows that the hash-routing schemes have significant performance advantages over ICP for both of the performance metrics. The existing hash-routing protocols assume that the cache servers are homogeneous in storage capacity and processing capability, even though most collections of cache servers are vastly heterogeneous. Our second contribution is to extend a robust hash-routing scheme so that it balances requests among the caches according to any desired distribution; the extended hash-routing scheme is robust in the face of cache failures, is tunable for heterogeneous caches, and can have significant performance advantages over ICP.

Shared Web caches, also referred to as proxy Web servers, allow multiple clients to quickly access a pool of popular Web objects. They are increasingly used in organizations (corporations, universities, government agencies, Internet service providers, etc.) whose Web clients can be connected directly to a shared cache over a high-speed local area network (LAN). Because many such organizations are connected to the Internet over lower-speed congested links, the average response time for requested Web objects can be substantially reduced if a significant fraction of the objects are available in the organization's shared cache. Moreover, by servicing a document from its own storage, a shared cache reduces the load on the link from the organization to the Internet, on the links within the Internet, and on the originating Web server. Shared caches have been successfully used by universities, research centers, and even countries (e.g., New Zealand) [1]. Hit rates up to 50 percent are not uncommon for shared caches [1]. The hit rate is even higher when hits from the local caches in the client hosts (PCs and workstations) are included in the hit rate. Emerging push technologies which push popular Web pages into shared caches should further increase the hit rates.

An organization that provides shared caching to its Web clients will typically have a *collection of shared caches* rather than just one. A collection of shared caches is desirable for several reasons. First, it eliminates the single point of failure: if one or more of the caches goes down, then the clients will still be able to retrieve popular objects from the other caches in the collection. Second, each department within the organization may have its own shared cache, and the processing and storage capabilities of the caches may vary enormously; it is desirable to coordinate the departmental caches so that all cached objects can be shared among the organization's clients. Third, a single shared cache simply may not be able to handle the demand from the organization's clients, due to the limited processing power and storage capacity of a single shared cache. Collections of shared Web caches can also be organized into a hierarchical topology [1], for example, a collection at the organizational level, a collection at the regional Internet service provider (ISP) level, and a collection at the national level.

In this article we study protocols for coordinating a collection of shared caches. The focus of this article is on the important and popular caching topology in Fig. 1. In this single-tier topology, an organization has a collection of shared

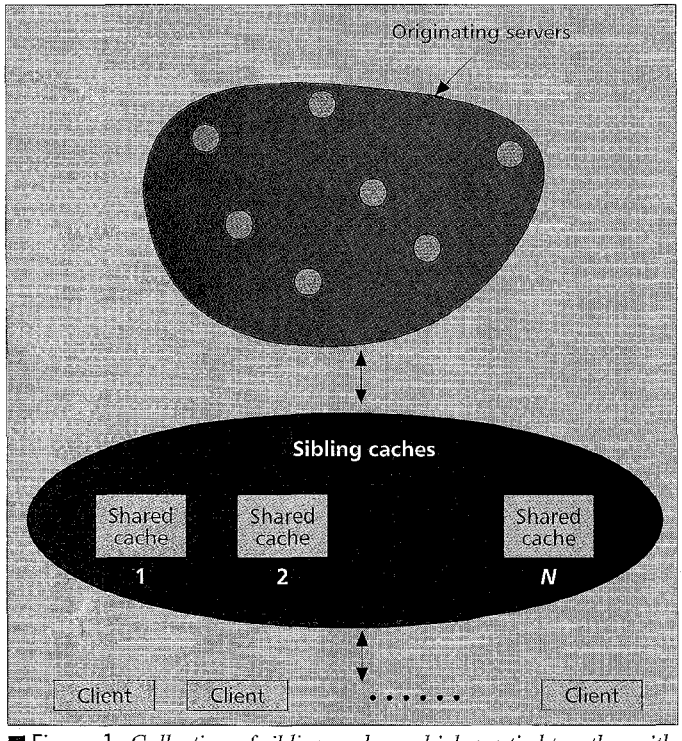
0890-8044/97/$10.00

caches which are tied together by a high-speed network. When a client requests an object, a search is first performed over the entire collection of shared caches; if the object is not found in the shared caches, one of the shared caches retrieves the object from the originating server and then forwards the object to the requesting client. All of the caches in the tier are *siblings*; that is, none of the caches are parents to any of the other caches in the tier. At the end of this article we will briefly discuss more complex hierarchical topologies.

One popular protocol which allows Web clients to coordinate and share a hierarchy of Web caches is the Internet Caching Protocol (ICP), which is defined in an Internet Draft [2, 3]. ICP is an application-layer protocol which runs on top of User Datagram Protocol/Internet Protocol (UDP/IP). It is the engine of the Harvest caching software, which has been adopted by many organizations, including the country of New Zealand and the National Laboratory for Applied Network Research (NLANR). (The Squid caching software is the free derivative of Harvest.) In brief, in the context of the single-tier hierarchy ICP works as follows. The Web client requests a page from one of the shared caches. If the shared cache cannot satisfy the request, it simultaneously queries all the other sibling caches to see if any of them have the object. This querying is not done over Hypertext Transfer Protocol (HTTP) but over the customized ICP application-layer protocol. If at least one of the siblings has the object, the shared cache requests the object (using HTTP) from the first sibling cache to respond with an ICP hit message; upon receiving the object, the shared cache stores a copy of the object and sends the object to the client. If none of the siblings have the object, the shared cache fetches the object from the originating Web server, places a copy in its cache, and delivers the object to the client. One of the properties of ICP is that multiple copies of the object can be present in the collection of shared caches.

One alternative to ICP is to use *hash routing schemes* to retrieve objects from a hierarchy of shared caches. Hash routing schemes for collections of Web caches have recently been proposed by several research teams [4–8], and one hash routing scheme forms the basis of the Cache Array Routing Protocol (CARP), defined in an Internet Draft [7]. A simple hash routing scheme operates as follows. All the clients store a common hash function which maps URLs to a hash space.[1] The hash space is partitioned, and each set in a partition is associated with one of the sibling caches. When a client desires an object, it first hashes the object's URL and then requests the object from the sibling cache whose set contains the hash value. If the cache cannot satisfy the request, it retrieves the object from the originating server, places a copy in its cache, and forwards the object to the client. One of the important properties of all hash routing schemes is that at most one copy of an object resides in the collection of caches. Thus, with hash routing the caches in the collection are effectively aggregated (rather than replicated, as with ICP), leading to higher hit probabilities.

Simple hash routing has, however, a serious drawback. If one of the caches fails, or if a new cache is added to the collection, then the hit probability will drop dramatically and instantaneously. The hit probability will eventually climb back to its original level, but this could take days or even weeks, depending on the nature of the request traffic. To resolve this problem, Thaler and Ravishankar [4] and Smith and Valloppillil [5] have independently developed a *robust* hash-routing scheme that overcomes this *disruption* problem. The robust hash-routing scheme of Smith and Valloppillil is part of CARP [7, 9].



■ Figure 1. *Collection of sibling caches, which are tied together with a high-speed network (e.g., Ethernet or ATM technologies).*

The contribution of this article is twofold. First, we provide simple analytical models for several coordination protocols, enabling a performance comparison of ICP with respect to the hash-routing schemes. We provide this analysis for two performance metrics: *overhead at a shared-cache server, and average latency to satisfy a client request for an object*. For both metrics we show that a hash-routing scheme, such as CARP, performs favorably compared to ICP. Our second contribution is to extend robust hash routing to the case of *heterogeneous caches*, that is, to caches with differing processing power and storage capacity. This extension enables the hash-routing scheme to distribute the numbers of objects among the caches according to any desired distribution; caches with larger processing capacity and/or storage capacity can be allocated a larger fraction of the objects in a precise manner. This extension to heterogeneous servers is also part of CARP [7, 9].

This article is organized as follows. In the second section we describe ICP and two implementations of a simple hash-routing scheme. In the third section we provide an analytical performance comparison of ICP and the simple hash-routing schemes. In the fourth section we briefly review how a hash routing scheme can be made robust. In the fifth section we extend the robust schemes to handle a collection of heterogeneous caches. In the sixth section we briefly discuss multi-tier hierarchies of caches. We summarize in the final section.

Shared Web Caching Protocols

An organization such as a corporation, university, or ISP often has a high-speed network (at rates of 10 Mb/s and up) which interconnects the organization's clients and servers. When a client requests a Web page from one of the servers

[1] *A hash function is a deterministic mapping which maps inputs to the hash space. The resulting outputs of the hash function are seemingly random, with the outputs uniformly scattered over the hash space. Small changes in the input create arbitrary "random" changes in the output.*

on the high-speed network, the response is often instantaneous because the network is fast and uncongested. But when a client requests a page from a remote server, the response is often slow and annoying. Several factors can contribute to this slow response: a relatively low-speed link existing between the organization and the distant Web server; congested links between the organization and the Web server; or the Web server itself being overloaded with requests. These factors can collude, further exacerbating the response time. Web page caching can help reduce the delay, or latency, for the remote pages. There are two types of Web caching mechanisms that are extensively used in practice: local client caching and shared caching. In this article our attention is focused on shared caching.

If an organization has a single shared cache on its high-speed network, all its clients send all their HTTP request messages directly to the shared cache. If the shared cache has a requested object, the shared cache forwards the object to the requesting client, in which case the latency is typically small since the organization's network is fast and uncongested. If the shared cache does not contain the requested object, the shared cache retrieves the object from the originating server and then forwards the object to the requesting client; the shared cache also stores a copy of the object, so it will be able to rapidly deliver the same object to any other client that requests it in the future. The cache is said to be *shared* because clients share all the objects in the cache. When the shared cache's storage becomes full, the storage device must replace objects according to some replacement policy. A popular replacement policy is *least recently used*, whereby a newly requested object replaces the object in the cache that has not been requested for the longest period of time; see [10] for an overview of Web cache replacement policies.

As discussed earlier, there is great incentive for an organization to have more than one shared Web cache. In this section we discuss two important classes of protocols for coordinating multiple shared caches.

Internet Caching Protocol

ICP is based on pinging the caches in an attempt to locate objects [1–3]. ICP is an application-layer protocol which runs on top of UDP/IP. An ICP message contains a 20-byte header plus a variable-sized payload; the payload typically contains a URL.

Each Web client is configured to connect to a particular shared cache. (This is typically done by configuring the browser in the client so that its proxy server is a specific shared cache. In this manner, all the client's HTTP requests are sent directly to the shared cache.) When a cache receives a request from a Web client, it will first check itself to see if it can satisfy the request. If the object does not exist in its cache, the cache queries *all* the sibling caches. A cache will query the sibling caches by sending a query message to each; the query messages are sent to siblings in parallel. Upon receipt of a query message, a sibling cache will extract the URL and check to see if the object is in its own cache. If a match is found, the sibling cache will return a hit message to the requesting cache; if the requested object is not in the sibling cache, the sibling cache will return a miss message. The query, hit, and miss messages are all part of the ICP application-layer protocol.

If the requesting cache receives a hit message from a sibling, it immediately issues an HTTP request to the first sibling cache that returned the hit message. If none of the sibling caches contain the requested object, the requesting cache waits until all the miss messages have been received (up to a configurable timeout whose default is 2 s); the requesting cache then retrieves the object from the originating Web server in the Internet. Once a cache obtains the object from a sibling cache or from the originating Web server, a copy of the object is placed in its own storage; the cache then forwards the object (within an HTTP response message) to the requesting client.

An important property of ICP is that there may be more than one copy of any given object in the collection of sibling caches. Replication occurs because a cache may obtain a copy of an object from a sibling cache in order to satisfy a client's request. This property creates a mirroring effect, whereby objects (especially popular objects) tend to get replicated across all the sibling caches. One unfortunate consequence of this property is that only a fraction of the total sibling storage capacity contains unique objects; the remainder of the capacity is wasted with multiple copies of the objects. One nice consequence of this property is that if one of the sibling caches fails, most of its cached objects will be available in the remaining sibling caches.

As the sibling caches become perfectly mirrored, the probability of receiving a hit message from a sibling cache approaches 0. A similar observation is made in [4], in which the rate of replication is shown to be large for a more general class of caching schemes.

ICP generates additional network traffic (i.e., ICP messages) which must be processed by the caches. We refer to this additional traffic as overhead. We shall quantify the extent of the overhead in the third section; in particular, we show that with ICP a shared cache will have to process many more ICP messages than HTTP messages.

We mention here that our description of ICP as a pinging protocol is consistent with the descriptions in the literature and in the Request for Comments (RFC) [2]. However, some implementations of ICP permit a form of simple hash routing; for example, a simple form of hash routing can be found in the code of NetCache 3.1 [11]. Nevertheless, the majority of ICP-based caching systems use pinging rather than hash routing.

Hash Routing Protocols

The hash routing (HR) schemes are deterministic hash-based approaches for mapping an object to a unique sibling cache [4–8, 11]. Hashing distributes the URL space among the sibling caches, creating a single logical cache spread over many caches.

The *simple HR scheme* works as follows. Let $h(\cdot)$ be a hash function which maps URLs to a hash space H. The hash space could be, for example, the set of all 32-bit binary numbers. The same hash function $h(\cdot)$ is stored in all the clients. The hash space, H, is partitioned into N sets, where N is the number of siblings. When a client desires an object, the client calculates $h(u)$, where u is the object's URL. The value of the hash function, $h(u)$, will belong to one of the sets in the partition. If $h(u)$ belongs to the nth set, the client sends an HTTP request message to the nth sibling cache. The sibling cache then examines its own cache to see if it has the object. If it does, it returns the object to the client within an HTTP response message. If it does not, the sibling cache sends an HTTP request message to the originating Web server in the Internet. When the sibling cache receives the object from the originating Web server, it places a copy of the object in its own cache and forwards the object to the client as part of an HTTP response message.

One important property of HR schemes is that the collection of sibling caches never contains more than one copy of an object: objects are not replicated across the sibling caches as with ICP. (This property may be briefly violated if a new shared cache is added to the collection, as discussed later.) Thus, hash routing creates a single logical cache spread across the collection of sibling caches; the effective storage capacity

of the collection of siblings is the sum of the storage capacities of the individual caches. Another important property, a key component of CARP, is that all communication among clients, caches, and originating Web servers is over HTTP; no additional application-layer protocol is needed [6].

We refer to the implementation just described as HR-client because the hashing of the URLs is done by the clients. This approach requires modification of existing Web client software, and therefore may be difficult to implement because of legacy Web clients that do not hash. A variation on the simple HR scheme, which we call HR-cache, is to instead implement the hashing function in the shared caches [5, 6]. This approach would allow existing Web browsers to fully participate in the protocol without modification. In HR-cache, each Web client is configured to connect to a particular shared cache, as with ICP. When a client requests an object, the HTTP request is sent to its configured shared cache. The configured shared cache hashes the object's URL and determines which unique sibling cache might have the object; the configured cache then requests the object from the appropriate sibling cache (which might be itself). If the sibling cache does not have the object, the sibling cache fetches the object from the originating Web server, places a copy in its cache, and forwards the object to the configured cache. Once the configured cache receives the object from the sibling cache, it forwards the object to the client, but does *not* place a copy of the object in its own cache (unless the configured cache is the appropriate sibling cache). Thus, with HR-cache, the configured cache initiates the HR algorithm on behalf of the requesting client; in the worst case, the object is sent from the originating server, to a sibling cache, to the configured cache, and finally to the client.

A minor variation of HR-cache uses the Domain Name Service (DNS) to dynamically assign a cache to a client [5, 6]. In this variation the client is not statically configured to a shared cache. When a client wants to send an HTTP request message to the sibling caches, it first queries DNS for an IP address. As the queries for IP addresses arrive from the clients to the DNS server, the DNS server rotates the IP addresses of the sibling caches. Therefore, each time a client requests an object, one of the caches is chosen effectively at random to initiate the HR-cache protocol on behalf of the client. This variation helps better distribute the load among the sibling caches. There is an analogous variation for ICP.

In our definition of hash routing we stated that the hash function takes into account the entire URL object. This implies that objects (HTML text, images, applets, etc.) within the same page will likely be scattered over the N cache servers. The scattering helps to spread the load of frequently requested pages over the N caches. An alternative to hashing the entire URL is to only hash the hostname portion of the URL, in which case all the objects in a particular page are routed to the same sibling cache. In fact, with hostname hashing all the objects of an entire site are routed to the same sibling cache. The sibling caches that house the most popular sites could have a relatively high request load. Moreover, because HTTP/1.0 establishes a separate Transmission Control Protocol (TCP) connection for each object, hostname hashing does not generate fewer TCP connections than URL hashing. Therefore, for HTTP/1.0, we feel that URL hashing is superior to hostname hashing. On the other hand, HTTP/1.1, with its persistent connections, may give a reduced number of TCP connections for hostname hashing. Although beyond the scope of this article, it would be interesting to study the benefits and drawbacks of hostname routing within the context of persistent connections.

Performance Comparison of ICP and Hash Routing Schemes

In this section we compare the performance of the HR protocols with that of ICP. We shall perform this comparison in the context of the common single-tier hierarchy of Fig. 1. The reader should be careful not to extrapolate the conclusions for the single-tier hierarchy to more complex topologies; see the sixth section.

We consider two performance metrics: *overhead at a caching server* and *average latency to satisfy a client request for an object*. Throughout this section we shall make the following assumptions:

- The ICP caches are perfectly mirrored. This is a reasonable assumption for steady-state operation [4, theorem 6], and it simplifies the analysis. We will also briefly consider in this section the case of the sibling caches being only partially mirrored.
- All IP addresses exist in the DNS caches of the shared caches and Web clients (i.e., we ignore DNS lookups).
- Local client Web caching is ignored.

The last two assumptions can be relaxed at the expense of a more complicated and less insightful analysis.

For ICP, denote P_{ICP} for the steady-state probability that a requested object is present in the collection of sibling caches; denote P_{HR} for the analogous probability for the hash routing schemes. The hit probabilities P_{ICP} and P_{HR} depend on a myriad of factors, including the storage capacities of the caches, user request patterns, and object replacement strategies for full caches. As mentioned earlier, for large storage capacities hit probabilities of 50 percent are not uncommon [1]. Because hash routing effectively aggregates the individual caches, we can safely assume that $P_{HR} > P_{ICP}$. Let N denote the number of sibling caches.

Let H denote the amount of effort required by a cache to process an HTTP message; for simplicity we assume that the effort to process any HTTP request or response message is H. The quantity H also includes the effort to establish and maintain the TCP connection associated with an HTTP response-request pair. With persistent HTTP (e.g., HTTP/1.1 [12]), TCP connections to each of the N caches could be kept permanently open, reducing the cache server effort H.

Traffic Processing at a Shared Cache

In this subsection we determine the expected load placed on a cache server by a single client HTTP request. First, consider HR-client. Mark one of the caches, which we refer to as the *marked cache*. We now calculate the marked cache's expected effort for a client HTTP request. With probability $1/N$, the hash of the URL directs the request to the marked cache. If the request is not directed to the marked cache, the expected effort is zero. Throughout the remainder of this paragraph, suppose that the request is directed to the marked cache. With probability P_{HR}, the object is present in the cache, in which case the marked cache must process one HTTP request from the client and one HTTP response to the client, giving a total effort of $2H$. The probability is $1 - P_{HR}$ the object is not in the cache, the marked cache must fetch the object from the originating server; in this case the marked cache must process one HTTP request from the client, one HTTP request to the originating server, one HTTP response from the originating server, and one HTTP response to the client, giving a total effort of $4H$. Therefore, the expected effort of HR-client is

$$E_{HR\text{-}client} = 1/N[P_{HR} \cdot 2H + (1 - P_{HR}) \cdot 4H] = H/N[4 - 2P_{HR}] \quad (1)$$

Having analyzed HR-client, we now proceed to analyze ICP and HR-cache. To this end we define the overhead ratio of each of the protocols as

$$OH_X = \frac{E_X}{E_{HR-client}},$$

where E_X is the expected overhead of the protocol (ICP or HR-cache). We shall see that HR-client requires the least amount of cache server effort. The overhead ratio compares the effort of scheme X to that of HR-client. For example, if the overhead ratio for scheme X is 3, scheme X requires three times the effort of HR-client.

Now consider ICP. Because all the siblings are assumed to be perfectly mirrored for ICP, P_{ICP} is the probability that any specific cache has the object. We calculate the marked cache's expected effort associated with an arbitrary client HTTP request. Let I denote the effort required to process an ICP message (query, hit or miss). The probability is $1/N$ that the client is configured to the marked cache; the probability is $(N - 1)/N$ that the request arrives at one of the other sibling caches. Assuming that the client is configured to the marked cache, with probability P_{ICP} the marked cache will have the requested object, in which case the marked cache processes one HTTP request from the client and one HTTP response to the client, giving an effort of $2H$; with probability $1 - P_{ICP}$, in addition to processing a request from the client and response to the client, the marked cache has to send and receive $N - 1$ ICP messages, and retrieve the object from the originating server using HTTP, giving an effort of $4H + 2I(N - 1)$. Now assuming that the client is configured to one of the other sibling caches, the marked cache will, with probability $1 - P_{ICP}$, receive and send an ICP message; and with probability P_{ICP} the marked cache will not get involved in the transaction. Therefore, the expected effort of the ICP protocol is

$$\begin{aligned} E_{ICP} &= \frac{1}{N}[P_{ICP}\cdot 2H + (1-P_{ICP})(4H + 2I(N-1))] \\ &\quad + \frac{N-1}{N}(1-P_{ICP})2I \\ &= \frac{H}{N}[4-2P_{ICP}] + \left(\frac{N-1}{N}\right)(1-P_{ICP})4I \end{aligned} \tag{2}$$

Using $P_{HR} > P_{ICP}$, we obtain

$$OH_{ICP} > 1 + 2(N-1)\frac{I(1-P_{HR})}{H(2-P_{HR})}. \tag{3}$$

We see that the overhead ratio for ICP grows linearly and without bound as the number of caches increases. Thus, ICP pinging scales poorly as the number of siblings increases.

Finally, we consider HR-cache. The analysis of this protocol is a bit more complicated. The probability is $1/N$ that the client is configured to the marked cache; it is $(N - 1)/N$ that the client is configured to one of the other sibling caches. Let A denote the expected effort given that the client is configured to the marked cache; let B denote the expected effort given that the client is configured to one of the other sibling caches. Focusing on the calculation of A, with probability $1/N$ the request is for an object mapped to the marked cache; in this case the effort is $2H$ if the marked cache has the object, and $4H$ if the marked cache does not have the object and fetches it from the originating server. With probability $(N - 1)/N$ the request is for an object mapped to one of the other sibling caches; in this case, the effort of the marked cache is $4H$. Thus,

$$\begin{aligned} A &= \frac{1}{N}[P_{HR}\cdot 2H + (1-P_{HR})\cdot 4H] + \frac{N-1}{N}4H \\ &= 4H - \frac{2P_{HR}H}{N} \end{aligned}$$

Now, focusing on the calculation of B, the probability is $1/N$ the request is for an object mapped to the marked cache; it is P_{HR} that the marked cache has the object and returns it directly to the requesting cache; it is $1 - P_{HR}$ that the marked cache does not have the object and requests it from the originating server. Thus,

$$\begin{aligned} B &= \frac{1}{N}[P_{HR}\cdot 2H + (1-P_{HR})\cdot 4H] \\ &= \frac{1}{N}(4H - 2P_{HR}H) \end{aligned}$$

The expected effort for HR-cache is therefore

$$\begin{aligned} E_{HR-cache} &= \frac{1}{N}A + \frac{N-1}{N}B \\ &= \frac{H}{N}(4-2P_{HR}) + \frac{4H}{N}(1-1/N) \end{aligned}$$

The overhead ratio for HR-cache is

$$OH_{HR-cache} = 1 + \frac{2(1-1/N)}{2-P_{HR}} \tag{4}$$

In the worst case ($N = 1$ and $P_{HR} = 1$) the overhead ratio for HR-cache is 3. In the more likely case of large N and $P_{HR} = 0.5$, the overhead ratio is 2.33. Therefore, we can expect the cache server effort of HR-cache to be two to three times greater than the effort for HR-client.

In conclusion, compared to HR-client, HR-cache places little additional effort on the cache servers. In contrast, ICP can introduce significant additional processing overhead, particularly when the number of sibling caches is large. With ICP, for each requested object that is not present in the sibling caches, each cache will have to process at least two ICP messages, a property which is true even if the sibling caches are not perfectly mirrored. Therefore, with respect to cache server load, ICP performs poorly respect to hash routing, even if the caches are far from being perfectly mirrored.

Average Latency

We now determine the average latency to satisfy a client request for the ICP and hash-routing schemes. For this latency analysis we make the following additional assumption: When an object is requested by a shared cache, the object must be fully received before it can be forwarded (to either a client or another shared cache). The validity of this assumption depends on the specific implementation of shared caching; the analysis can be modified for shared caching systems which forward packets while the object is being received.

We now define several expected round-trip delays between host pairs. Let delay_{CS} be the expected delay to request and transfer an object from a sibling cache to a client assuming that the requested object is in the sibling cache. The clock for this delay begins when the client requests the object and ends when the client receives the entire object; the delay includes TCP connection establishment, propagation delays, transmission delays, server processing delays, queuing delays, and TCP slow-start delays. Similarly, define delay_{SS} and delay_{SO} to be the expected delay to request and transfer an object between two sibling caches and between the sibling cache and originating server, respectively. The delays delay_{SS} and delay_{SO} also include all the components named above. Finally, let $\text{delay}_{\text{ping}}$ be the expected pinging delay in ICP. The clock for this delay begins when a sibling cache issues an ICP query until when the sibling cache receives the first ICP hit message; if none of the siblings has the object, this delay is until all the ICP miss

messages have been received or the ICP timeout (whose default value is 2 s), whichever comes first. Because ICP runs over UDP, $\text{delay}_{\text{ping}}$ does not include a delay component for TCP connection establishment; also, the processing and transmission components of this delay are relatively small because ICP messages are simple and short.

Now consider the average latency for ICP. A client requests an object from its configured shared cache, and this shared cache will eventually deliver the object to the client. This will account for delay_{CS}. The probability that the shared cache will not have the object is $1 - P_{ICP}$. In this case, the shared cache will send ICP query messages to all the sibling caches and will receive ICP miss messages from all the siblings (since the siblings are perfectly mirrored in steady state); the shared cache will then request the object from the originating server. Thus, the average latency is

$$T_{ICP} = \text{delay}_{CS} + (1 - P_{ICP})[\text{delay}_{ping} + \text{delay}_{SO}].$$

Similarly, it can be shown that the average latency for HR-client is

$$T_{HR\text{-}client} = \text{delay}_{CS} + (1 - P_{HR})\text{delay}_{SO},$$

and the average latency for HR-cache is

$$T_{HR\text{-}cache} = \text{delay}_{CS} + (1 - 1/N)\text{delay}_{SS} + (1 - P_{HR})\text{delay}_{SO}.$$

Let us now compare the average latencies for ICP, HR-client, and HR-cache. Typically, the sibling caches are in close geographic proximity and connected by high-speed links; therefore, we may safely assume $\text{delay}_{SO} >> \text{delay}_{\text{ping}}$ and $\text{delay}_{SO} \gg \text{delay}_{SS}$. Under these conditions, the average latency for ICP is

$$T_{ICP} \approx \text{delay}_{CS} + (1 - P_{ICP})\text{delay}_{SO},$$

and the average latency for the two HR schemes is

$$T_{HR} \approx \text{delay}_{CS} + (1 - P_{HR})\text{delay}_{SO}.$$

The ratio of these two latencies is

$$\frac{T_{ICP}}{T_{HR}} \approx \frac{\text{delay}_{CS} + (1 - P_{ICP})\text{delay}_{SO}}{\text{delay}_{CS} + (1 - P_{HR})\text{delay}_{SO}}. \quad (5)$$

Consider Eq. 5 for the two most common scenarios:

- The clients are connected to the network of sibling caches through high-speed LAN links, as is typically the case in a LAN office environment. For this case we typically have $\text{delay}_{SO} \gg \text{delay}_{CS}$.
- The clients are connected to the network of sibling caches through low-speed modem links, as is often the case for residential access to an ISP. In this case, the client-sibling delays may be comparable or even substantially greater than the sibling-origin delays.

If the client-sibling delays are larger than the sibling-origin delays, the ratio of the latencies will be approximately 1; that is, the latency for ICP will be roughly the same as the latency for hash routing. If, on the other hand, we are operating in the first scenario (which is very common), then

$$\frac{T_{ICP}}{T_{HR}} \approx \frac{1 - P_{ICP}}{1 - P_{HR}}.$$

Thus, in the first scenario, if the miss rate of ICP is twice that of HR, the average latency of ICP will be twice that of HR.

There is some empirical evidence that shows that increasing the cache size beyond a critical amount does not significantly increase the hit probability [13]. This implies that if each ICP cache size is at least as large as this critical value, the hit rates for ICP and HR will be approximately the same. However, it may be costly to provide the requisite storage capacity to each sibling. For example, it may be desirable to use RAM (as opposed to disk) for Web caching in order to minimize retrieval latencies out of a cache server. Also, as large audio and video objects become more prevalent in the Web, it will become more difficult to store all the most commonly referenced objects in a single ICP cache. For these reasons, as the Web evolves we expect hash routing, with its factor of N increase in effective storage capacity, to have significantly higher hit probabilities, implying significantly lower average latency for the common first scenario.

We also mention that we have tacitly assumed the delay between a cache and the originating server, delay_{SO}, to be independent of the hit probability. In actuality, delay_{SO} may decrease as the hit probability increases, because the links between the caches and the originating servers become less congested. This phenomenon further increases the gap in latency performance between the HR schemes and ICP.

Robust Hash Routing

We have argued that simple HR schemes have significant performance advantages over ICP. Nevertheless, they have a critical flaw: whenever a cache comes up or goes down, the fraction of objects in incorrect caches can be large [4, 5]. Thaler and Ravishankar [4] refer to this fraction as the *disruption coefficient*. For simple modulo-N hash functions, they show that the disruption coefficient is close to 1. In the simple HR scheme described in this article, based on a partitioned hash space, the disruption coefficient is 0.5, which is less than 1 but still too large.

To see that the disruption coefficient is 0.5, recall that the hash space is partitioned into N sets; assume that these sets are contiguous and of equal size. Without loss of generality, assume that the hash space is the interval [0, 1]. Now suppose we add a cache to the collection, increasing the number of caches to $N + 1$. Then it is easily seen that only the following intervals in the hash space will agree with the updated mappings:

$$\left[0, \frac{1}{N+1}\right], \left[\frac{1}{N}, \frac{2}{N+1}\right], \left[\frac{2}{N}, \frac{3}{N+1}\right], \ldots, \left[\frac{N-1}{N}, \frac{N}{N+1}\right].$$

Adding the lengths of these intervals gives

$$\sum_{i=0}^{N-1} \frac{N-i}{N(N+1)} = \frac{1}{N(N+1)} \sum_{n=1}^{N} n = \frac{1}{2}.$$

Thus, after adding a new cache, only 50 percent of the cached objects are in the correct caches. A similar calculation shows that when a cache goes down, again, only 50 percent of the cached objects are in the correct caches. Therefore, with this simple scheme the hit probability will be cut in half immediately after a disruption.

To overcome this unfortunate reduction in hit probability after a disruption, robust hashing schemes have recently been proposed [4, 5]. With *robust hashing*, for each sibling cache, the URL and sibling cache name are used *together* to generate a hash value or score; the object is then mapped to the sibling cache with the highest score. More specifically, let $h(u, c)$ be a hash function which maps a URL u and a cache-server name c to an ordered hash space. For a given u, robust hashing calculates the scores $h(u, c_1), \ldots, h(u, c_N)$ for each of N sibling caches; it then routes u to sibling cache n which has the highest $h(u, c_n)$ value. Thus, for robust HR-client, for a given URL u the client calculates the N scores and requests the object from the cache with the highest score. For HR-cache, the cache finds the highest score on the client's behalf. The

cache-server name can be its hostname, its IP address, or something informal, such as MarketingCache.

With simple hash routing, a minor disruption causes a large fraction of the cached objects to reside in incorrect caches. With robust hashing, the large majority of objects remain resident in the correct caches after a disruption. If a cache fails, all the cached objects remain in their correct caches; if the number of caches is increased from $N - 1$ to N, only the fraction $1/N$ of the cached objects reside in an incorrect cache. Thus, robust hashing gracefully adapts to disruptions in the number of sibling caches.

Thaler and Ravishankar [4] examine robust hashing in great detail for homogeneous caches, and discuss several specific hash functions. Smith and Valloppillil [5] also provide a robust hashing function, which is implemented in CARP [7].

Heterogeneous Caches

Up to this point we have tacitly assumed that all of the caches have the same processing power and storage capacity. Because an organization will likely want to add additional caches to satisfy increasing demand, and because processing and storage are improving at a phenomenal pace, the assumption of homogeneous caches is unrealistic. We expect most collections of caches to be heterogeneous, and the processing power and storage capacity to vary greatly among the caches.

It is therefore important to extend the hashing schemes to the case of heterogeneous caches, with which it is no longer desirable to map the fraction $1/N$ of URLs to each cache. Instead, caches with more processing power and storage capacity should get a larger fraction of the URLs.

Let $p_1, \ldots, p_N$ be given arbitrary target probabilities for each cache. (Each p_n is positive, and $p_1 + \ldots + p_N = 1$.) If a cache has target probability p_n, we desire the fraction p_n of URLs to be mapped to it. In this section we address the question, how do we modify the hashing schemes so that the URLs are mapped to the various caches with the desired target probabilities? For the simple hashing scheme, whereby the hash space is partitioned into N sets, this problem has a simple solution: we simply partition the hash space so that the size of the nth set is the fraction p_n of the size of the entire hash space. However, the solution to this problem is less straightforward for the robust hashing scheme. Recall that the robust hashing scheme is preferable to the simple scheme, since it gracefully adjusts to disruptions such as the addition or deletion of caches.

In the robust hashing scheme, for a given URL u we calculate a hash value $h_n = h(u, c_n)$ for each of N caches. We then map the URL to the cache that has the highest h_n. This scheme will map $1/N$ of the URLs to each cache. To deal with target probabilities, we introduce multipliers $x_1, \ldots, x_N$ and multiply each h_n with the respective x_n; we then map the URL to the cache that has the largest $Z_n = x_n h_n$ value. If the multipliers are different, the fractions of URLs routed to the caches will no longer all be the same.

But how do we choose the multipliers so that the fraction of URLs routed to cache n is p_n for $n = 1, \ldots, N$? It is easily seen that simply setting $x_n = p_n$, $n = 1, \ldots, N$ does not give the desired weights (and can be grossly incorrect). The following provides a simple recursive algorithm to determine the weights.

Theorem 1 — Let $p_1, \ldots, p_N$ be given target probabilities. Reorder the caches so that $p_1 \le \cdots \le p_N$. Let

$$x_1 = (Np_1)^{1/N}$$

and let $x_2, \ldots, x_N$ be calculated recursively as follows:

$$x_n = \left[\frac{(N-n+1)(p_n - p_{n-1})}{\prod_{i=1}^{n-1} x_i} + x_{n-1}^{N-n+1} \right]^{\frac{1}{N-n+1}}.$$

Then the robust hash algorithm with multipliers $x_1, \ldots, x_N$ will route the fraction p_n of URLs to the nth cache for $n = 1, \ldots, N$.

Proof — Let $x_1, \ldots, x_N$ be an arbitrary set of nonnegative multipliers satisfying $x_1 \le x_2 \cdots \le x_N$. Let $h_1, \ldots, h_N$ be the hash values associated with each of the N caches. Because the h_ns are outputs of a hash function, they can be taken to be independent, uniformly distributed random variables. Without loss of generality, we take each h_n to be uniformly distributed over $[0, 1]$. Let $Z_n = x_n h_n$ be the nth multiplied hash value. Note that the Z_ns are independent and that Z_n is uniformly distributed over $[0, x_n]$. Let $Z_{(n)} = \max(Z_1, \ldots, Z_{n-1}, Z_{n+1}, \ldots, Z_N)$. Let q_n be the probability that n has the largest multiplied hash value, that is, $q_n = P(Z_{(n)} \le Z_n)$. Conditioning on $Z_n = x$, we obtain

$$\begin{aligned} q_n &= P(Z_{(n)} \le Z_n) \\ &= \frac{1}{x_n} \int_0^{x_n} P(Z_{(n)} \le x)dx \\ &= \frac{1}{x_n} \int_0^{x_n} \prod_{i \ne n} P(Z_i \le x)dx \\ &= \frac{1}{x_n} \int_0^{x_n} \frac{\prod_{i=1}^{N} P(Z_i \le x)}{P(Z_n \le x)} dx \\ &= \int_0^{x_n} \frac{1}{x} \prod_{i=1}^{N} P(Z_i \le x)dx \\ &= \sum_{j=1}^{N} \int_{x_{j-1}}^{x_j} \frac{1}{x} \prod_{i=1}^{n} P(Z_i \le x)dx, \end{aligned} \tag{6}$$

where $x_0 = 0$. We must now get an explicit expression for the product in the above expression.

For $x_{j-1} \le x \le x_j$,

$$P(Z_i \le x) = \begin{cases} 1 & i \le j-1 \\ \dfrac{x}{x_j} & i \ge j \end{cases}$$

Thus, for $x_{j-1} \le x \le x_j$,

$$\begin{aligned} \prod_{i=1}^{N} P(Z_i \le x) &= \left[\prod_{i=1}^{j-1} P(Z_i \le x) \right] \left[\prod_{i=j}^{N} P(Z_i \le x) \right] \\ &= \prod_{i=j}^{N} \frac{x}{x_i} = \frac{x^{N-j+1}}{\prod_{i=j}^{N} x_i}. \end{aligned} \tag{7}$$

Inserting Eq. 7 into Eq. 6 gives

$$\begin{aligned} q_n &= \sum_{j=1}^{n} \int_{x_{j-1}}^{x_j} \frac{x^{N-j}}{\prod_{i=j}^{N} x_i} dx \\ &= \sum_{j=1}^{n} \frac{1}{\prod_{i=j}^{N} x_i} \frac{1}{N-j+1} (x_j^{N-j+1} - x_{j-1}^{N-j+1}) \\ &= \left(\prod_{i=1}^{N} x_i \right)^{-1} \sum_{j=1}^{n} \frac{(\prod_{i=1}^{j-1} x_i)(x_j^{N-j+1} - x_{j-1}^{N-j+1})}{N-j+1}. \end{aligned}$$

We have one degree of freedom. Set

$$\prod_{i=1}^{N} x_i = 1.$$

Then

$$q_n = \sum_{j=1}^{n} \frac{\left(\prod_{i=1}^{j-1} x_i\right)\left(x_j^{N-j+1} - x_{j-1}^{N-j+1}\right)}{N-j+1}. \quad (8)$$

From Eq. 8 we have

$$q_n = q_{n-1} + \frac{\left(\prod_{i=1}^{n-1} x_i\right)\left(x_n^{N-n+1} - x_{n-1}^{N-n+1}\right)}{N-n+1}. \quad (9)$$

The desired result follows by setting $q_n = p_n$, $n = 1, \ldots, N$, and solving for x_n in Eq. 9.

The algorithm in the theorem requires negligible computational effort and produces the correct multipliers. As a simple example, suppose that $N = 3$ and $p_1 = p_2 = 1/81$ and $p_3 = 79/81$; then the algorithm gives $x_1 = x_2 = 1/3$ and $x_3 = 9$. Once we calculate the multipliers, we can multiply them all by the same positive constant and use these scaled multipliers instead (if we so desire). However, we must start with x_1 as defined in Theorem 1 if we are to use the recursion in Theorem 1; a different value of x_1 will provide incorrect results.

The robust hash function with multipliers $x_1, \ldots, x_N$ in the above theorem is part of CARP [7]. The multipliers can be used for other HR applications, such as rendezvous points in multicast servers [4].

Multi-tier Hierarchies

We have presented an HR scheme which is robust in the face of cache failures, tunable for heterogeneous caches, and can have significant performance advantages over ICP for a single-tier hierarchy of sibling caches. But what about more complex multi-tier hierarchies?

Consider the following two-tier hierarchy. In a city and its surrounding suburbs there are several universities, research institutions, and corporate campuses; each of these organizations has its own collection of sibling caches; a common regional ISP provides Internet access to all these organizations; the regional ISP has its own collection of sibling caches; and the sibling caches in the regional ISP are parents to the collections of sibling caches in the organizations.

ICP has been designed to accommodate multi-tier hierarchies such as the one just described, but hash routing can accommodate multi-tier hierarchies as well. For simplicity, consider HR-client, and suppose that hash routing is performed at both tiers in the hierarchy. Specifically, if a queried organization cache does not have a requested object, it hashes the URL and queries one of the ISP caches: the hash result at the organizational cache determines which ISP cache is queried. Thus, with hash routing, there is no replication among an organization's siblings, and there is no replication among the ISP's siblings. There is, however, replication across organizations as well as between each organization and the ISP. With ICP, there is also additional replication within the collections of sibling caches at both the organization and ISP tiers. If, from each given organization cache, the response times to all the ISP caches is roughly equal, hash routing should still give superior performance with respect to the two performance metrics considered in this article. On the other hand, if the response times vary significantly, with the closest ISP cache giving the lowest response time, then ICP, which would parent an organization's caches to the nearest ISP cache, could have a lower average latency.

In summary, both ICP and hash routing can be adapted to a hierarchy of caches. Neither ICP nor hash routing has superior latency performance for all topologies. When placing caches in a complex topology, both the hierarchical organization of the caches and the caching protocol should be carefully considered.

Summary

In this article we have used simple but realistic analytical models to compare the performance of ICP and HR schemes for a single-tier hierarchy of sibling caches. We have found that ICP cache servers typically process many more messages than hash routing servers. We have also shown that, if a larger aggregate cache translates into significantly higher hit probabilities, hash routing gives clients significantly lower delays than ICP. As the Web evolves, we expect large caches to correlate with high hit rates due to increasing numbers of popular audio and video objects that will be stored on the sibling caches.

We expect most collections of shared caches to consist of servers with vastly different processing and storage capabilities. In this article we have also extended robust hash routing to the important case of heterogeneous caches. This extension allows Web traffic to be distributed in a precise and equitable manner among the Web caches. An interesting problem for further research is to dynamically adapt the target probabilities, $p_1, \ldots, p_N$, in order to optimize a performance measure such as overall response time.

Acknowledgments

I would like to thank Vinod Valloppillil of Microsoft Corporation for having discussed hash routing at length with me. I would also like to acknowledge discussions with Mike Sullivan of Lockheed-Martin in the early stages of this research.

References

[1] Yeager and McGrath, *Web Server Technology*, San Francisco: Morgan Kaufman, 1996.
[2] D. Wessels and K. Claffy, "Internet Cache Protocol Version 2," Internet Draft, http://ds.internic.net/internet-drafts/draft-wessels-icp-v2-00.txt.
[3] A. Chankhunthod *et al.*, "A Hierarchical Internet Object Cache," *Proc. 1996 Usenix Tech. Conf.*
[4] D. G. Thaler and C. V. Ravishankar, Using Name-Based Mappings to Increase Hit," to appear, *IEEE/ACM Trans. Networking*, 1997.
[5] B. Smith and V. Valloppillil, personal communication, Feb.–June, 1997.
[6] V. Valloppillil and J. Cohen "Hierarchical HTTP Routing Protocol," Internet Draft, http://www.nlanr.net/Cache/ICP/draft-vinod-icp-traffic-dist-00.txt.
[7] V. Valloppillil and K. W. Ross, "Cache Array Routing Protocol v1.0," Internet Draft, http://ds1.internic.net/internet-drafts/draft-vinod-carp-v1-02.txt.
[8] Super Proxy Script: How to Make Distributed Proxy Servers by URL Hashing," White Paper, http://naragw.sharp.co.jp/sps, Aug. 1996.
[9] E. Sullivan, "CARP Divvies Up the Duties," *PCWeek Online*, Sept. 1997; http://www.zdnet.com/pcweek/reviews/0915/15carp.html.
[10] J-C. Bolot and P. Hoschka, "Performance Engineering of the World Wide Web: Application to Dimensioning and Cache Design," *Proc. 5th Web Conf.*, Paris, France, 1996.
[11] P. Danzig, *Pers. Commun.*, Sept. 1997.
[12] R. Fielding *et al.*, "Hypertext Transfer Protocol – HTTP/1.1, RFC 2068 http://ds.internic.net/rfc/rfc2068.txt, Jan. 1997.
[13] D. Marwood and B. Duska, "Squid Proxy Analysis," http://www.cs.ubc.ca/spider/marwood/Projects/SPA/Report/Report.html, April, 1997.

Biography

KEITH W. ROSS (ross@seas.upenn.edu/~ross) is a professor in the Department of Systems Engineering at the University of Pennsylvania. As of January 1998 he will be a professor in the Multimedia Department at Eurecom, Sophia Antipolis, France. He received his B.S. from Tufts University (1979), his M.S. from Columbia University (1981), and his Ph.D. from the University of Michigan (1985). He joined the University of Pennsylvania in 1985. He was program chair of the 1995 INFORMS Telecommunications Conference. He is the recipient of numerous grants from NSF and AT&T, and has consulted for major telecommunication companies. His current research interests include multimedia networking, video on demand, Internet and Web protocols, traffic modeling for broadband networks, and asynchronous online learning.