Using Name-Based Mappings to Increase Hit Rates

David G. Thaler and Chinya V. Ravishankar
Electrical Engineering and Computer Science Department
The University of Michigan, Ann Arbor, Michigan 48109-2122
thalerd@eecs.umich.edu ravi@eecs.umich.edu

***Abstract*— Clusters of identical intermediate servers are often created to improve availability and robustness in many domains. The use of proxy servers for the WWW and of Rendezvous Points in multicast routing are two such situations. However, this approach can be inefficient if identical requests are received and processed by multiple servers. We present an analysis of this problem, and develop a method called the Highest Random Weight (HRW) Mapping that eliminates these difficulties. Given an object name and a set of servers, HRW maps a request to a server using the object name, rather than any *a priori* knowledge of server states. Since HRW always maps a given object name to the same server within a given cluster, it may be used locally at client sites to achieve consensus on object-server mappings.**

We present an analysis of HRW and validate it with simulation results showing that it gives faster service times than traditional request allocation schemes such as round-robin or least-loaded, and adapts well to changes in the set of servers. HRW is particularly applicable to domains in which there are a large number of requestable objects, there is a significant probability that a requested object will be requested again, and the CPU load due to any single object can be handled by a single server.

HRW has now been adopted by the multicast routing protocols PIMv2 and CBTv2 as its mechanism for routers to identify Rendezvous Points/Cores.

1 Introduction

In the usual client-server model, clients access object data or services that are made available by servers. A single-server system is not robust, however, and often provides insufficient resources to handle a large number of requests. Thus, *clusters* of equivalent servers can be used to increase service availability and to lower the workload on individual servers.

In this paper, we investigate how a client may map a request for a particular object to a server, so as to minimize response time. In particular, we will limit our scope to domains with the following characteristics:

- All requests for the same class of objects are handled by a cluster of servers with equivalent functionality and capacity.
- The set of servers in the cluster is known to clients prior to issuing requests, and all clients see the same set of servers. For example, this information may be statically configured, looked up at client startup time, or periodically distributed to clients.
- There is some notion of a "hit rate" at a server, so that a server responds more quickly to a duplicate request than a first-time request. This is clearly true when the server functions as a cache. It is also true if clients need to locate the server which was assigned to do a particular task.
- The benefit of object replication across servers is negligible. That is, the CPU load due to any single object can be handled by a single server.

The above characteristics cover a wide variety of domains. Some examples include:

Real-time producer-consumer systems For example, in multicast routing protocols such as CBT [1] and PIM [2], receivers' routers request data for a specific session by sending a join request towards the root of a distribution tree for that session, and sources send data to a session via the root of its tree. The root thus takes on the role of a server, with receivers and sources becoming clients. Sources and receivers must rendezvous at the root for effective data transfer.

Client-side WWW proxy caches In the World Wide Web (WWW), pages can be cached at proxy servers [3, 4]. All outbound client requests can then go through a local proxy server. If the proxy server has the page cached, the page is returned to the client without accessing the remote provider. Otherwise, the page is retrieved and cached for future use.

Note that *server*-side proxy caches often do not fall within the relevant class of domains. When proxy caches are placed near servers to handle inbound requests, the number of requests for a single object may represent enough CPU load to make object replication more effective.

Task delegation In task brokering systems, tasks may be delegated to various servers. Any clients which desire to interact with such a task must then contact the server running that task.

For these domains, we will present an efficient algorithm which maps requests to servers such that requests for the same object are sent to the same server, while requests for different objects are split among multiple servers. We will refer to this concept as a "name-based" mapping.

It is possible to view the case where all clients send requests for the same object to the same server as defining *affinities* between objects and servers in a cluster (Figure 1). A number of studies (e.g., [5, 6, 7]) have examined the related notion of "cache-affinity" scheduling in the context of shared-memory multiprocessors, in which tasks are sent to processors which already have data cached. This achieves higher cache hit rates at the possible expense of load balancing.

Several fundamental differences exist, however, between multiprocessor systems and distributed systems which limit the applicability of cache-affinity scheduling. First, multiprocessor systems typically assume centralized schedulers, whereas clients in decentralized distributed systems independently choose servers and direct requests to them. A centralized scheduler in a distributed system also represents an undesirable performance bottleneck and a single point of failure.

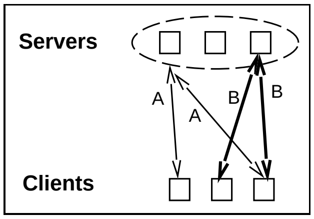


Figure 1: Object-Server Affinities

Second, a centralized scheduler can maintain up-to-date processor load information, but such information is expensive to obtain in distributed systems, since scheduling may be done locally at clients distant from the servers (in terms of latency). Finally, in some multiprocessor systems, a request in progress may be migrated to another processor as part of the scheduling algorithm. Such migration is usually undesirable or even impossible in distributed systems, and migrated requests must typically be restarted.

Although cache-affinity scheduling algorithms are not directly applicable in our domains, our goals are similar: to increase the cache hit rates and thus reduce latency by using appropriate scheduling. We highlight the importance of using a sensible policy for reducing replication and improving hit rates by directing requests for a given object to the same server.

This paper makes two major contributions. First, it gives a general model for mapping requests to servers: Section 3 describes the model and typical goals of mapping functions, covers previous work, and shows how common mapping functions fit within our model. Second, we present our "name-based" mapping function: Section 4 develops this notion, Section 5 provides an analysis of our mapping function, and Section 6 describes efficient implementations. Finally, applications in two popular domains are examined as case studies in Section 7.

2 Goals for mappings

Algorithms for mapping requests to individual servers within clusters have typically concentrated on two goals: load balancing and low mapping overhead. We argue in this paper that a number of other goals are important as well. We discuss these goals in this paper, and proceed to develop the notion of name-based mappings. We also propose a new mapping method called Highest Random

Weight mapping that meets all the criteria discussed in this section.

2.1 Traditional goals

We begin by considering the two goals on which conventional mapping algorithms have focussed.

Goal 1 (Low overhead): *The latency introduced in picking a server within a cluster must be as small as possible.*

Schemes which require purely local decisions have low overhead, while schemes requiring an additional exchange of network messages have high overhead. Also, for an algorithm to be applicable to many domains, the mapping function must be portable and fast enough to use in high-speed applications.

Goal 2 (Load balancing): *To guarantee uniform latency, requests should be distributed among servers so that each server sees an equal share of the resulting load (over both the short and long term) regardless of the object size and popularity distributions.*

To ensure that this goal is realized, one must first define what load balancing means, and consider how loads are generated. It is well-known (e.g., [8]) that patterns of requests can be very bursty when the request stream includes machine-initiated requests, so that the arrival process of requests is not, in general, Poisson. The Packet Train [9] model is a widely-used alternative to the traditional Poisson arrival model, and appears to model such request patterns well. We therefore adopt this model, and assume that requests arrive in batches, or "trains".

For load balancing over short time intervals, we focus on the load resulting from the arrival of a single train of requests. In this case, we desire that the load resulting from each train (in terms of amount of resources consumed) be split nearly equally among the available servers.

After the individual requests in a packet train are assigned to servers, there are many possibilities for the load on any single server. We can model the load l on a given server as the value of a random variable $\mathbf{l}$. The random variable $\mathbf{l}$ may be viewed as representing a load distribution defined over an infinite population of identical servers, so that the load on any one server is a single value sampled from this distribution. We can similarly model the loads within a cluster of size m as a sample of size m from this distribution. For the algorithms we analyze in this paper, we may consider this sample to be the values of m independent arbitrarily distributed random variables $\mathbf{l}_1, \mathbf{l}_2, \cdots, \mathbf{l}_m$ with identical means and variances.

Let the mean and variance of the distributions of the $\mathbf{l}_i$ be μ and σ^2 respectively. If $l_1, l_2, \cdots, l_m$ are the load values sampled from the $\mathbf{l}_i$, we are interested in defining a measure to ensure that these sampled values are as close to each other as possible. The sample standard deviation s provides a good measure of dispersion. However, the value of s depends on the magnitudes of the sample values l_i, whose expectation is the population mean. We therefore normalize s by dividing by the population mean, and deal with the ratio s/μ. Our goal will be to minimize this ratio.

Consider an incoming packet train that contains N requests $r_1, r_2, \cdots r_N$. Since these requests represent loads, we simply treat them as loads. Let these requests be drawn from individual distributions with mean μ and variance σ^2. Let there be m servers $S_1, S_2, \cdots, S_m$, and let each server be assigned $k = \lfloor N/m \rfloor$ requests. The resulting load l_i on server S_i is thus a sum of k random variables $r_{i1} + r_{i2} + \cdots r_{ik}$. We begin with the following lemma.

Lemma 1 *Let $\mathbf{l}_i$ be a random variable whose values $l_i = r_{i1} + r_{i2} + \cdots r_{ik}$ represent the load on server S_i. The distribution of $\mathbf{l}_i$ is approximately normal with mean $k\mu$ and variance $k\sigma^2$ as $N \rightarrow \infty$.*

Proof: The result follows immediately from the Central-Limit Theorem. See any standard book dealing with sampling theory, [10], for example. □

Hence, for sufficiently large packet train sizes, loads on all servers are normally distributed. We will now show that loads in a server cluster become balanced if the coefficient of variation (ratio of mean to standard deviation) of the load distribution on servers tends to zero.

Lemma 2 *Let $l_1, l_2, \cdots, l_m$ be a sample of size m from a normal distribution with mean μ and standard deviation σ. If $s = \sqrt{\sum_{1 \leq i \leq m} (l_i - \bar{l})^2/(m-1)}$ is the sample standard deviation, $(s/\mu) \rightarrow 0$ as $(\sigma/\mu) \rightarrow 0$.*

Proof: Let $\bar{l}$ be the sample mean, and let $y_i = l_i - \bar{l}$. By the usual definition of sample variance, we have

$$\frac{(m-1)s^2}{\sigma^2} = \left(\frac{y_1}{\sigma}\right)^2 + \cdots \left(\frac{y_m}{\sigma}\right)^2$$

The sum on the right side follows the χ^2 distribution [10] with $m-1$ degrees of freedom[1]. Thus, we may write $s^2 = \sigma^2[\chi^2_{m-1}/(m-1)]$. Therefore, $(s/\mu)^2 = (\sigma/\mu)^2[\chi^2_{m-1}/(m-1)]$. Since the χ^2 distribution is bounded, $(\sigma/\mu)^2[\chi^2_{m-1}/(m-1)] \to 0$ as $(\sigma/\mu) \to 0$. Hence $(s/\mu)^2$ vanishes as well. □

We can now combine these lemmas to conclude that for sufficiently large packet train sizes: (1) the load distribution on servers is approximately normal, and (2) the loads in a server cluster become balanced if the coefficient of variation of this normal distribution tends to zero. We will find this result useful in studying the properties of request mapping algorithms in subsequent sections.

2.2 Additional goals for mappings

This section provides the motivation for our name-based method for assigning objects to servers. We motivate our approach by discussing the issues of replication and disruption.

2.2.1 Replication and Rendezvous Issues

A request mapping algorithm can reduce retrieval latency not just by balancing loads and maintaining low overhead, but also through a third mechanism: minimizing replication of work. Poorly-designed mapping schemes can cause several different servers to do the same work, lowering efficiency. Replication of work arises when client requests for the same object are sent to multiple servers, causing them each to retrieve and cache the same object separately, for example.

Such replication is particularly unacceptable in real-time producer-consumer domains, where a producer sends real-time object data to a server. Since consumers retrieve object data from servers, producers and consumers must *rendezvous* at the server for successful information transfer. That is, producers and consumers must select the same server independently and simultaneously. Real-time data requires low end-to-end latency, so a 100% hit rate for server selection is *required* for a successful rendezvous between producer and consumer. Long latency would otherwise result, violating real-time deadlines, since the server selected by the consumer must first obtain the data from the producer. Section 7.1 describes one example of this effect.

A 100% hit rate for server selection can only be achieved when the producer sends its information to all servers in a cluster (wasting resources), when consumers send requests to all servers (again wasting resources), or when all producers and consumers send data and requests for the same object to the same server. The last of these options may be viewed as defining *affinities* between objects and servers.

Replication can also reduce hit rates in caching schemes by decreasing the effective cache size of the servers in the cluster. For example, a study conducted in 1992 [11] reported that a 4 GB cache was necessary for intermediaries to achieve a cache hit rate of 45% for FTP transfers. Thus, if we used four servers and a mapping scheme which allows replication, *each* server would require a 4 GB cache to achieve a 45% hit rate, rather than only a 1 GB cache each. On the other hand, in a balancing scheme that avoids replication, each server would see requests for one fourth of the objects. Intuitively, its response time and cache hit rates would then be the same as if there were only 1/4 the requestable objects, and the probability of finding a requested object in the cache will thus be greater. As we will see in Section 7.2, this scheme allows each server to get an equivalent hit rate with only a 1 GB cache.

If the growth of replication is slow, however, it is unlikely to be a matter for concern. However, as Theorem 6 of Section 5.5 demonstrates, replication can get quickly out of hand, and an object can be expected to become replicated in all m servers within about $m + m \ln m$ requests.

An important factor to consider is that the latency for the server to retrieve the object from the remote provider is far longer than the latency to retrieve the object from the server's cache. We thus formulate the following additional goal for a mapping algorithm:

Goal 3 (High hit rate): *The mapping scheme should attempt to increase the probability of a hit.*

2.2.2 Minimizing Disruption

Whenever a server comes up or goes down, the current object-server affinities may change. This leads to another goal:

[1] There are only $m-1$ independent terms in the sum on the right side. We can choose $m-1$ values arbitarily, but the last value is now fixed because $l_1 + \cdots + l_m = m\bar{l}$ must hold.

Goal 4 (Minimal Disruption) *Whenever a server comes up or goes down, the number of objects that are remapped to another server must be as small as possible.*

In multicast routing, for instance, this goal minimizes the number of sessions liable to experience data loss as a result of changes to distribution trees. For distributed caching systems, this maximizes the likelihood that a request for a previously-cached object will still result in a cache hit.

A parameter of particular significance for schemes to map objects to servers is the *disruption coefficient* δ, which we define as the fraction of the total number of objects that must be remapped when a server comes up or goes down.

Theorem 1 (Disruption Bounds) *For* every *mapping which evenly divides objects among servers, the disruption coefficient* δ *satisfies*

$$\frac{1}{m} \leq \delta \leq 1,$$

where m *is the number of active servers.*

Proof: Let N be the number of objects cached within the cluster. We first observe that no more than N objects can be disrupted, giving 1 as the upper bound on disruption. Next, if objects are evenly divided among servers, then there are N/m objects assigned to each server at any time. When one or more servers go down, all N/m objects assigned to each server that goes down *must* be reassigned to another server, regardless of any other changes. Thus, disruption $\delta \geq \frac{N/m}{N} = \frac{1}{m}$.

When one or more servers come back up, all objects which were previously mapped to each server *must* be reassigned to it when it comes back up. This is because the set of active servers is then identical to the set of active servers before the server(s) originally went down, at which time it had N/m objects. Since clients making a purely local decision cannot distinguish between the two cases, the previous mapping must be restored. Thus, N/m objects must be reassigned to each new server, regardless of any other changes. Hence, again, disruption $\delta \geq \frac{N/m}{N} = \frac{1}{m}$. □

3 A Model for Mapping Requests to Servers

Any scheme which maps a request r^k for an object k to a specific server in a cluster can be logically viewed as picking the server which minimizes (or maximizes) some value function f. Let $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ be a cluster of m servers. A typical mapping function $\mathcal{F}$ thus selects a server S_i at time t such that:

$$\mathcal{F}_t(r^k) = S_i : f_t(i) \leq f_t(j), \quad j \neq i \qquad (1)$$

where i and j are indexes of servers.

A number of mapping functions have been used in practice that attempt to realize one or more of the goals discussed in Section 2. The goal of balancing loads is particularly significant from our point of view since it directly influences response times.

We now present some commonly-used value functions, and discuss how well each of them achieves the above goals.

3.1 Static Priority Mapping

In a static priority scheme, the server list is statically ordered, e.g., $f(i) = i$ in the model described above. Clients simply try contacting each server in order until one responds. While this does provide fault tolerance, the entire load will typically fall on the highest-priority server, potentially causing long response times during times of heavy use. In addition, the cache space available is under-utilized, since the space available at the other servers is not used.

3.2 Minimum-Load Mapping

Sending a request to the least-loaded server divides requests between servers so as to keep the load low on each server, providing faster service times. Here, f is some measure of the current load, i.e., $f_t(i) =$ (load on S_i at time t), thus requiring an additional mechanism (either periodic or on-demand) to determine which server currently has the lowest load. Making this determination is non-trivial, since clients are constantly issuing requests, and load information may be out of date by the time it is acquired and used by a client. In the worst case, all clients issue requests to the same, previously idle, server, resulting in a very high load.

Minimum-Load Mapping is, however, the approach taken in many existing systems. For example, Cisco's LocalDirector [12], which redirects WWW requests to one of a set of local servers, periodically queries the servers for status information, thus potentially using out-of-date information. In the Contract Net protocol [13], servers are queried for load information when a request is ready, introducing additional latency.

3.3 Fastest-Response Mapping

In the fastest-response scheme, a client pings the servers, and picks the one that responds first. Thus, $f_t(i) =$ (response time for S_i). When all servers are equally distant, this mapping is similar to the least-loaded scheme (with the same advantages and disadvantages), since the server with the least load typically responds first. The Harvest [4] web cache implementation and the Andrew File System (AFS) [14] both use this method.

3.4 Round-Robin Mapping

A simpler scheme is round-robin, where successive requests are sent to consecutive servers. For example, when a name in the Domain Name Service (DNS) resolves to multiple IP addresses, DNS returns this list of IP addresses, rotated circularly after each request. If clients use the first address on this list, requests will be sent to the various IP addresses in round-robin fashion[2], thereby balancing the number of requests sent to each [15]. The NCSA scalable web server configuration [16] is one example of such use of DNS.

When all servers are up, a true Round-Robin scheme maps the n^{th} request sent to the cluster to the n^{th} server (modulo m). Thus, $\mathcal{F}(r_n) = n \pmod m$. To get an ordered list for robustness, this is logically equivalent to assigning weights in our model as:

$$\begin{aligned} f_0(i) &= i \pmod m \\ f_n(i) &= (f_{n-1}(i) - 1) \pmod m \end{aligned}$$

where n denotes the number of previous requests sent, and m is the number of servers.

We now demonstrate formally that round-robin achieves load balancing when the request rate is high. As discussed earlier, we will use the packet-train model [9] for our analysis.

Let N be the number of requests in the batch or train. Let $\mathbf{r}$ be a random variable describing the service time for one request. Let $\mathbf{l}_i$ be a random variable describing the total service time for all requests in the batch which are mapped to a given server S_i. (Note that $\mathbf{l}_i$ is independent of the queue discipline.)

Theorem 2 (Round-Robin Load Balancing): *Let $\mathbf{l}_i$ be a random variable describing the total processing required for all requests mapped to a given server S_i. If N requests are assigned to m servers in a round-robin manner, then the square of the coefficient of variation of $\mathbf{l}_i$ is given by:*

$$CV[\mathbf{l}_i]^2 = \left(\frac{m}{N}\right) CV[\mathbf{r}]^2, \tag{2}$$

and hence, when $\mathbf{r}$ has finite variance,

$$\lim_{N\to\infty} CV[\mathbf{l}_i] = 0.$$

Proof: Since requests are assigned round-robin, each server will get exactly (when N is a multiple of m) N/m requests. Since the value of $\mathbf{l}_i$ for a given server is the sum of the service times of the requests mapped to it, we get:

$$E[\mathbf{l}_i] = (N/m)E[\mathbf{r}] \tag{3}$$

Since service times of individual requests are independent and identically distributed, the variance is also additive, giving:

$$Var[\mathbf{l}_i] = (N/m)Var[\mathbf{r}] \tag{4}$$

Equation (2) directly follows from (3) and (4), since $CV[\mathbf{l}_i]^2 = Var[\mathbf{l}_i]/E[\mathbf{l}_i]^2$. □

Lemma 2 now guarantees that the load is balanced when N is large, and the load is therefore significant. This observation applies both to long-term load balancing (where $N \to \infty$ as the time interval of interest grows), as well as short-term load balancing as the request rate increases (i.e., when large batches of requests arrive within a short period of time).

3.5 Random Mapping

Another way to balance the expected number of requests assigned to each server, is to send requests to a server chosen at random (e.g., $f(i) = random()$), as suggested in [14]. This is referred to in queueing theory as a *random split*.

As before, let N be the number of requests in the batch or train. Let $\mathbf{r}$ be a random variable describing the service time for one request. Let $\mathbf{l}_i$ be a random variable describing the total service time for all requests in the batch which are mapped to a given server S_i. (Note that $\mathbf{l}_i$ is independent of the queue discipline.)

[2] Note that DNS may not give true Round-Robin ordering when clients generate duplicate requests, due to DNS caching at clients.

Theorem 3 (Random-Split Load Balancing): *Let $\mathbf{l}_i$ be a random variable describing the total processing required for all requests mapped to a given server. If N requests are randomly assigned to m servers, such that the probability that a request will be mapped to a given server is $1/m$, then the square of the coefficient of variation of $\mathbf{l}_i$, is given by*

$$CV[\mathbf{l}_i]^2 = \left(\frac{m}{N}\right) CV[\mathbf{r}]^2 + \left(\frac{m-1}{N}\right) \quad (5)$$

and hence, when r has finite variance,

$$\lim_{N\to\infty} CV[\mathbf{l}_i] = 0.$$

Proof: Since the value of $\mathbf{l}_i$ for a given server is the sum of the service times of the requests mapped to it, we get:

$$E[\mathbf{l}_i] = (N/m)E[\mathbf{r}] \quad (6)$$

To find the second moment of the service time, let a server receive k requests $r_1, r_2, \ldots, r_k$. Then the square of the total expected service time is given by:

$$E[(r_1 + \cdots + r_k)^2] = kE[\mathbf{r}^2] + k(k-1)E[\mathbf{r}]^2$$

since request service times are i.i.d. We next observe that the number of requests k mapped to a given server is binomially distributed with success probability $p = 1/m$. Thus, we obtain:

$$E[\mathbf{l}_i^2] = \sum_{k=0}^{N} \binom{N}{k} \frac{(m-1)^{N-k}}{m^N} (kE[\mathbf{r}^2] + k(k-1)E[\mathbf{r}]^2)$$

This equation can be split into the terms for each moment of $\mathbf{r}$:

$$E[\mathbf{l}_i^2] = m^{-N}(A \cdot E[\mathbf{r}^2] + B \cdot E[\mathbf{r}]^2)$$

Letting $k' = N - k$, we can now solve for the coefficients A and B separately as follows:

$$\begin{aligned} A =& \sum_{k'=0}^{N} \binom{N}{N-k'} (N-k')(m-1)^{k'} \\ =& N \sum_{k'=0}^{N-1} \binom{N-1}{N-1-k'} (m-1)^{k'} \\ =& Nm^{N-1} \end{aligned}$$

applying the Binomial Theorem in the last step. Similarly,

$$\begin{aligned} B =& \sum_{k'=0}^{N} \binom{N}{N-k'} \frac{(m-1)^{k'}}{m^N} (N-k')(N-k'-1) \\ =& N(N-1) \sum_{k'=0}^{N-2} \binom{N-2}{N-2-k'} (m-1)^{k'} \\ =& N(N-1)m^{N-2} \end{aligned}$$

Putting these results back into the original equation, we obtain:

$$\begin{aligned} E[\mathbf{l}_i^2] &= (N/m)E[\mathbf{r}^2] + (N(N-1)/m^2)E[\mathbf{r}]^2 \quad (7) \\ Var[\mathbf{l}_i] &= E[\mathbf{l}_i^2] - E[\mathbf{l}_i]^2 \\ &= (N/m)E[\mathbf{r}^2] + (N(N-1)/m^2)E[\mathbf{r}]^2 \\ &\quad -(N^2/m^2)E[\mathbf{r}]^2 \\ &= (N/m)E[\mathbf{r}^2] - (N/m^2)E[\mathbf{r}]^2 \quad (8) \end{aligned}$$

Thus, for the square of the coefficient of variation, we get:

$$\begin{aligned} CV[\mathbf{l}_i]^2 &= Var[\mathbf{l}_i]/E[\mathbf{l}_i]^2 \\ &= \frac{(N/m)E[\mathbf{r}^2] - (N/m^2)E[\mathbf{r}]^2}{(N^2/m^2)E[\mathbf{r}]^2} \end{aligned}$$

Simplifying, and using the identity $CV[\mathbf{r}]^2 = (E[\mathbf{r}^2]/E[\mathbf{r}]^2) - 1$, we obtain Equation 5. □

Table 1 summarizes how well each function discussed above meets the desired properties.

4 Mappings Based on Object Names

Not every scheme which avoids replication has a low disruption coefficient. For example, the Static Priority mapping algorithm (Section 3.1) certainly avoids replication, since all requests are sent to the same server. However, its disruption coefficient is unity since every object gets remapped when the primary server fails. Disruption can be minimized by balancing the number of objects mapped to each server.

One way of accomplishing this goal is to use the name of the object to derive the identity of the server. Since this mapping is a purely local decision, its overhead remains low. Unlike conventional mapping schemes based on name servers, such a mapping is "stateless" since it depends only on the identities of the object and the cluster servers, and

Mapping	f	Balances	Overhead	Replication
Static Priority	$f(i) = i$	Nothing	Low	No
Least-loaded	$f_t(i) =$ load on S_i at time t	Load	High	Yes
Fastest-response	$f_t(i) =$ response time for S_i at time t	Resp.Time	High	Yes
Round-robin	$f_n(i) = (f_{n-1}(i) - 1) \pmod m$	# Requests	Low	Yes
Random	$f(i) = random()$	# Requests	Low	Yes
HRW	(see Section 4.1)	# Objects	Low	No

Table 1: Mapping Functions

not on the state at the cluster servers or that held in a name server. Such a stateless mapping can be viewed as a *hash function*, where the key is the name of the object, and the "buckets" are servers.

In our case, the number of buckets can vary over time as servers are added or removed. Second, it is possible for one or more of the servers to be down, so that an object must hash to another server when one goes down. Therefore, the output of such a hash function must be an *ordered list* of servers rather than a single server name. In some domains, such as PIMv2 [17], the list of servers is dynamically updated to exclude unreachable servers. In this case, it suffices for the hash function to map a name to a single server. We are interested in the more general case, however, and therefore define a stateless mapping as a function which, given a list of servers, maps an object name to a specific ordering of the server list.

A conventional hash function maps a key k to a number i representing one of m "buckets" by computing i as a function of k, i.e., $i = h(k)$. The function h is typically defined as $h(k) = f(k) \pmod m$, where f is some function of k (e.g., $f(k) = k$ when k is an integer). In our case, a key k corresponds to an object name, and a bucket corresponds to a server in a cluster. A serious problem with using a modulo-m function for mapping objects to servers, however, arises when the number of active servers in the cluster changes.

If a simple modulo-m hash function were used to map objects to servers, then when the number of active servers changes from m to $m-1$ (or vice versa), all objects would be remapped except those for which $f(k) \pmod m = f(k) \pmod{m-1}$. When $f(k)$ is uniformly distributed, the disruption coefficient will thus be $\frac{m-1}{m}$; i.e., almost all objects will need to be reassigned. Clearly, a better scheme is needed.

4.1 Highest Random Weight Hashing

We now introduce a new mapping algorithm, which we call **Highest Random Weight (HRW)**. HRW operates as follows. An object name and server address together are used to assign a random "weight" to each server. The servers are then ordered by weight, and a request is sent to the active server with the highest weight.

Thus, HRW may be characterized by rewriting Equation 1 as follows:

$$\mathcal{F}_t(r^k) = S_i : Weight(k, S_i) \geq Weight(k, S_j), \quad i \neq j. \tag{9}$$

where k is the object name, S_i is the IP address of server i, and $Weight$ is a pseudo-random function of k and S_i.

5 Properties of HRW

We now analyze the HRW algorithm described above and examine how well it satisfies the requirements outlined in Sections 3 and 4.

5.1 Low overhead

It is easy to see that HRW requires no information beyond server and object names. This allows clients to make an immediate decision based on purely local knowledge.

In real-time producer-consumer domains, it must be possible to change the server in use without requiring the data transfer to start over. In such domains, some optimizations are desirable when the server changes, if a client is receiving a large number of objects simultaneously.

First, when a server goes down, clients must reassign all objects previously mapped to that server. For each of those objects, if the list of weights $Weight(k, S_i)$ has been preserved, this list can be used directly to reassign the object to its new maximum-weight server. Alternatively, the implementation could trade speed for memory by recalculating the weights for each server and not storing $Weight(k, S_i)$.

Second, when a server comes up (and the lists of weights have not been preserved), recalculation of all weights for all objects can be avoided simply by storing the previously winning weight. Then, when the server S_i comes up, the implementation need only compute $Weight(k, S_i)$ for each object k in use, and compare it to the previously winning weight for k. Only those objects for which S_i yields a higher weight need be reassigned.

5.2 Load balancing

Let there be a set $\mathcal{S} = \{S_1, S_2, \cdots, S_m\}$ of m servers. Let $O = \{o_1, o_2, \cdots, o_K\}$ be the universe of requestable objects. Some objects are likely to be requested more frequently than others, so let o_i have popularity p_i, defined as the probability that an incoming request is for object o_i. Let $M : O \rightarrow \mathcal{S}$ be a mapping of objects to servers that partitions the objects equally among the servers.

The actual run-time load experienced by any server S_i clearly depends on the popularities of the objects mapped to it. The more popular the objects mapped to S_i, the greater the load it sees. By analogy with object popularities, we may define the popularity of a server S_i as a random variable $\mathbf{q}_i$, whose value equals the sum of the popularities of the objects mapped to it. Thus, $\mathbf{q}_i$ represents the probability that a request will be sent to S_i. If $q_1, q_2, \cdots, q_m$ are server popularity values, we must have $\sum_{1 \leq i \leq m} q_i = 1$ since $\sum_{1 \leq i \leq K} p_i = 1$.

Let the mapping M assign objects $o_{i1}, o_{i2}, \cdots, o_{ik}$ to server S_i. Since M assigns all objects to servers, we can view $o_{i1}, o_{i2}, \cdots, o_{ik}$ as a selection of size k from the set O, such that any object o_i is selected with the probability $1/K$. We can similarly model server popularities by viewing $p_{i1}, p_{i2}, \cdots, p_{ik}$ as a sample of size k from $p_1, p_2, \cdots, p_K$, taken without replacement. In this case, the population mean $\bar{p} = (p_1 + p_2 + \cdots + p_K)/K = 1/K$, since the p_i sum to 1. Let the population variance be σ_p^2.

We now prove two theorems that characterize the load-balancing properties of HRW. Theorem 4 states that the coefficient of variation of $\mathbf{q}_i$ vanishes as the number of objects K becomes large. Theorem 5 states that the amount of processing done by each server is balanced when both K and N are large.

Theorem 4 (Hash-Allocation Request Balancing): *Let K objects be partitioned among m servers using HRW, with each server receiving exactly $k = K/m$ objects. If p_i, $\bar{p}$, σ_p^2, and $\mathbf{q}_i$ are as defined above, then the square of the coefficient of variation of $\mathbf{q}_i$ is given by*

$$CV[\mathbf{q}_i]^2 = \left(\frac{m-1}{K-1}\right) CV[p]^2, \tag{10}$$

and hence, when p has finite variance,

$$\lim_{K \rightarrow \infty} CV[\mathbf{q}_i] = 0.$$

Proof: Let objects $o_{i1}, o_{i2}, \cdots, o_{ik}, k = K/m$ be mapped to S_i. As before, we can treat the object popularities p_i as a sample of size k, and write $\mathbf{q}_i = p_{i1} + p_{i2} + \cdots + p_{ik}$. If $\bar{\mathbf{p}}_i = (p_{i1} + p_{i2} + \cdots + p_{ik})/k$ is the sample mean, then $\mathbf{q}_i = k\bar{\mathbf{p}}_i$, so we have $E[\mathbf{q}_i] = kE[\bar{\mathbf{p}}_i]$, $Var[\mathbf{q}_i] = k^2 Var[\bar{\mathbf{p}}_i]$. We know from sampling theory [10] that $E[\bar{\mathbf{p}}_i] = \bar{p} = 1/K$, so

$$E[\mathbf{q}_i] = k\bar{p} = (K/m)(1/K) = (1/m) \tag{11}$$

We also know from sampling theory [10] that $Var[\bar{\mathbf{p}}_i] = (K-k)(\sigma_p^2/k)/(K-1)$. Since $Var[\mathbf{q}_i] = k^2 Var[\bar{\mathbf{p}}_i]$, we can substitute and simplify to get

$$Var[\mathbf{q}_i] = \left(\frac{m-1}{K-1}\right)\left(\frac{K^2}{m^2}\right)\sigma_p^2 \tag{12}$$

We can now substitute $E[\mathbf{q}_i] = 1/m, \bar{p} = 1/K$, and rearrange to obtain Equation 10.

Theorem 5 (Hash-Allocation Load Balancing): *Let K objects be randomly partitioned among m servers using HRW, with each server receiving exactly K/m objects. Let N be the request train size, and let the service time $\mathbf{r}$ of requests and p both have finite variance. Then, if $\mathbf{l}_i$ is a random variable representing the amount of processing done by server S_i,*

$$\lim_{N \rightarrow \infty} \lim_{K \rightarrow \infty} CV[\mathbf{l}_i] = 0 \tag{13}$$

Proof: From Theorem 4, the coefficient of variation of $\mathbf{q}_i \rightarrow 0$ as $K \rightarrow \infty$ for any server. From Equation 11, $E[\mathbf{q}_i] = (1/m)$ is independent of K, so that $Var[\mathbf{q}_i] \rightarrow 0$ as $K \rightarrow \infty$, and hence $\mathbf{q}_i \rightarrow (1/m)$. We can now apply Theorem 3, and Equation 13 follows immediately. □

Using Lemma 2, we can now conclude that the processor loads are balanced when the conditions of Theorem 5 are met. That is, the load balancing effectiveness *increases* as the demand increases.

5.3 High hit rate

It is easy to see that HRW avoids replication, thus potentially giving a higher hit rate, as long as clients have a consistent list of servers for each cluster. Again, we assume that the server list is known to clients a priori. For example, it may be statically configured, resolved at client startup time, or periodically distributed to clients. See [18] for further discussion of these issues.

5.4 Minimal disruption

When a server S_i goes down, all objects which mapped to that server must be reassigned. All other objects will be unaffected, and so the optimum disruption bound is achieved. The randomizing property of HRW allows the reassigned objects to be evenly divided among the remaining servers, thus preserving load balancing.

When a server $S_i \in \mathcal{S}$ comes back up or when a server $S_i \notin \mathcal{S}$ is added to the set $\mathcal{S}$, then the objects which get reassigned to it are exactly those that yield a higher weight for S_i than for any other server. This again achieves the optimum disruption bound of $1/m$.

Thus, we have shown that HRW achieves the minimum disruption bound.

5.5 Comparing HRW with Other Mappings

It is instructive to compare the performance of HRW qualitatively with that of other mappings, particularly with the Round-Robin and Random mappings, which are also stateless. Section 7 presents an empirical comparison of HRW with other mappings.

The Round-Robin and Random mappings do an excellent job of balancing loads, as Theorems 2 and 3 demonstrate. However, balancing server loads is not the primary criterion for favoring a mapping. Ultimately, it is often more important to optimize response time. For the application domains of our interest, server load balancing is an important goal only to the extent that it helps optimize response time.

Optimizing response time means reducing both the expected value as well as the variance of the response time. A serious problem with the Round-Robin and Random mappings in caching domains we consider is that decreases in response time due to load balancing tend to be counterbalanced in practice by significant increases in retrieval latency due to cache misses. Each cache miss requires a retrieval from a remote provider, an operation that may be orders of magnitude more expensive than retrieval from a local cache.

Some of this effect arises from replication of data elements in the server cache. In an intuitive sense, replication decreases the effective cache size for the cluster as a whole, since replicated objects are held in more than one server, thus wasting cache space. However, replication is likely to be problem only if it grows quickly enough. We now demonstrate that in the absence of a deliberate effort to control it, replication can quickly get out of hand.

Theorem 6 (Replication growth for random selection): *Let $\mathcal{S} = \{S_1, S_2, \cdots S_m\}$ be a cluster of m servers, and let $r_1^k, r_2^k, \cdots$ be a series of requests for object k. For each such request r_i^k, randomly select a server S_j, and assign r_i^k to S_j. If p requests are processed before all m servers cache the object k, then*

$$E[p] = m\left(\ln m + \gamma + \frac{1}{2m} + O(m^{-2})\right), \qquad (14)$$

where $\gamma = 0.57721\cdots$ is Euler's constant.

Proof: We can view the progression to full replication as a series of phases, with phase i being the duration between $(i-1)$ servers caching the object and i servers caching it. The process begins with phase 1 and ends after phase m.

Let n_i be the number of requests in phase i. Clearly, $n_1 = 1$. By definition,

$$p = \sum_{i=1}^{m} n_i \qquad (15)$$

During phase i, there are $(i-1)$ servers which cache the object, and $m-i+1$ servers that do not. Since requests are randomly assigned to servers, the probability that a given phase-i request is sent to a server that does not cache the object is $(m-i+1)/m$. Thus n_i, the number of phase-i requests follows a geometric distribution, so that

$$E[n_i] = \frac{m}{m-i+1}$$

Using linearity of expectation in Equation 15, we get $E[p] = \sum_{i=1}^{m} E[n_i] = \sum_{i=1}^{m} m/(m-i+1)$. After changing the summation index appropriately, this reduces to

$$E[p] = m \sum_{i=1}^{m} \frac{1}{i}$$

It is well-known (see [19], for example) that

$$\sum_{i=1}^{m} \frac{1}{i} = \ln m + \gamma + \frac{1}{2m} - O(m^{-2}).$$

Substituting above, the theorem follows. □

It is clear that full replication is achieved rather quickly. With 10 servers in a cluster, the expected number of requests to reach full replication is about 30. Therefore, the effective cache size of the cluster is reduced by a factor of 10 after an average of about 30 requests per object. In contrast, the replication factor in HRW is always zero. Thus, the effective cache size remains unchanged under HRW mappings.

5.5.1 Caching under HRW

Since the goal of a caching policy is to maximize the hit rate, a caching policy attempts to cache the set of objects most likely to be requested next. Since the future is unknown, the caching algorithm must predict this set in practice.

An "optimal" caching policy is one which maximizes the probability that the object named in the next request is in the cache. Let object k have size s_k, and let $p_k(t)$ be its popularity at time t. That is, $p_k(t)$ is the probability that an incoming request at time t is for object k. The expected hit rate for the next request is then equal to $\sum_{k \in \mathcal{C}} p_k(t)$, where $\mathcal{C}$ is the set of cached objects. Thus, the optimal set of objects to cache at time t in a cache of size C maximizes $\sum_{k \in \mathcal{C}} p_k(t)$, subject to the constraint that $\sum_{k \in \mathcal{C}} s_k \leq C$. This is an instance of the Knapsack problem, which is known to be NP-complete [20].

Cache replacement strategies can then be viewed as heuristics to solve this problem. Since the future is unknown, they must use local estimates of the $p_k(t)$'s based on statistics such as recency or frequency of reference, typically deriving them from past history.

If a request r^k for object k may be sent to any server in a cluster with equal probability, then each server will see the same set of object popularities $p_k(t)$. That is, the probability that the next incoming request is for object k is the same at all servers. We will refer to such mapping schemes as "non-partitioned mappings". These include all mappings previously discussed which allow replication (i.e., all except Static Priority and HRW). Conversely, we will refer to mapping schemes under which $p_k(t)$ is non-zero at exactly one server for each k as (completely-) "partitioned mappings". These include HRW and Static Priority.

Assuming equal cache sizes, all servers under a non-partitioned mapping will have the same expected hit rate under an optimal caching scheme, since they all see the same object popularities. However, as the number of servers grows, each server will see a smaller fraction of incoming requests, spaced farther apart, and so its estimates of $p_k(t)$ can degrade in quality. Thus, we expect the hit rate seen by non-partitioned mappings to decrease as the number of servers grows. As we will see in Section 7.2, trace-driven simulations have confirmed that this is indeed the case in practice.

Theorem 7 (Partitioning Non-Harmful) *Under an optimal caching scheme, the expected hit rate in a partitioned mapping will be greater than or equal to the expected hit rate in a non-partitioned mapping.*

Proof: At time t, let $\mathcal{C}_0$ be the set of objects cached at some server under a non-partitioned mapping and an optimal caching scheme using a cache of size C. Let $P_0 = \sum_{k \in \mathcal{C}_0} p_k(t)$ be the expected hit rate for a request sent to that server at time t. Under an optimal caching scheme, we know that P_0 is maximized, subject to $\sum_{k \in \mathcal{C}_0} s_k \leq C$. Without partitioning, all servers see the same set of object popularities, so P_0 is the same for all servers. Hence the expected hit rate of the entire cluster is also P_0.

Let $\mathcal{K}_i$ be the set of objects mapped to server S_i in a partitioned mapping. Let $P_i = \sum_{k \in \mathcal{C}_0 \cap \mathcal{K}_i} p_k(t)$ be the portion of P_0 due to objects which get mapped to S_i in a partitioned mapping. (Thus, $P_0 = \sum_{i=1}^{m} P_i$.) Let $\mathcal{C}'_i \subseteq \mathcal{K}_i$ be the set of objects cached at S_i under an optimal caching scheme using a cache of size C.

Let $P'_i = \sum_{k \in \mathcal{C}'_i} p_k(t)$ be the expected hit rate at server S_i under a partitioned mapping and an optimal caching scheme. We will now show by contradiction that $P'_i \geq P_i$, and hence $\sum_{i=1}^{m} P'_i \geq P_0$ (i.e., the hit rate of the cluster is not decreased under a partitioned mapping).

Assume that $P_i > P'_i$. Then there exists a set of objects $\mathcal{C}_i = \mathcal{C}_0 \cap \mathcal{K}_i$, with $\sum_{k \in \mathcal{C}_i} s_k \leq \sum_{k \in \mathcal{C}_0} s_k \leq C$, and $\sum_{k \in \mathcal{C}_i} p_k(t) = P_i > P_i' = \sum_{k \in \mathcal{C}_i'} p_k(t)$. Thus, P_i' was not the optimum solution to the Knapsack problem at S_i, and we have a contradiction. □

Thus, the expected hit rate under an optimal caching scheme, in *any* partitioning scheme will be greater than or equal to the expected hit rate under an optimal caching scheme in any non-partitioning scheme, and grows as the number of servers is increased since more objects can be cached. In addition, a server sees the entire request history for objects assigned to it, regardless of the total number of servers. Thus, the quality of estimates of $p_k(t)$ does not degrade as the number of servers increases.

Since caching policies depend on such estimates in practice, the hit rate is expected to increase with the number of servers in HRW, and decrease in all other mapping schemes which do not partition the set of requestable objects. Again, as we will see in Section 7.2, trace-driven simulations have confirmed that this is indeed the case in practice.

Thus, HRW allows the hit rate to be increased. For this effect to be significant, the maximum hit rate possible must also be significant, i.e., a significant number of requests must be for objects which were requested in the past. Combining this observation with the conditions in Theorem 5 giving good load balancing, we obtain the following corollary.

Corollary 1 (HRW Applicability): *HRW is particularly suitable for domains where there are a large number of requestable objects, the request rate is high, and there is a high probability that a requested object will be requested again.*

This condition is true in a wide variety of domains. We will study two of them, multicast routing and WWW caching, in more detail in Section 7.

6 Implementing HRW Mappings

The weight function is the crucial determinant of HRW performance. To achieve a high hit rate, all clients should use the same weight function. Based on an evaluation of different randomization schemes (see Section 6.1), we recommend a HRW scheme based on the weight function W_{rand} defined as:

$$W_{rand}(k, S_i) = (1103515245 \cdot ((1103515245 \cdot S_i \\ +12345) \text{ XOR } D(k)) + 12345) \pmod{2^{31}} \qquad (16)$$

where $D(k)$ is a 31-bit digest of the object name k, and S_i is the address of the i^{th} server in the cluster. Note that the length of the address does not matter, as only the low-order 31 bits are significant in modulo 2^{31} arithmetic.

This function generates a pseudo-random weight in the range $[0..2^{31}-1]$, and is derived from the original BSD `rand` function[3], where it corresponds to:

```
srand(Si);
srand(rand() ^ D(k));
Weight = rand();
```

This function can be implemented with only a few machine instructions[4], requires only 32-bit integer arithmetic, and exhibits a number of desirable properties, as we will see in Section 6.1. Thus, implementing HRW entails first computing $D(k)$, and then computing $W_{rand}(k, S_i)$ for each S_i. The time to do this is typically negligible compared to the network latency.

In the unlikely event that multiple servers are assigned the same weight for a name k, ties can be broken by choosing the server with the highest S_i. The following theorem states exactly when such ties will occur:

Theorem 8 *In the W_{rand} function, a tie between two servers S_i and S_j occurs* **if and only if** $S_i \equiv S_j \pmod{2^{31}}$.

Proof: First, assume that $S_i \equiv S_j \pmod{2^{31}}$. Then it is easy to see from Equation 16 that $W_{rand}(k, S_i) = W_{rand}(k, S_j)$ since modulo-2^{31} congruence is preserved under addition, multiplication, and the XOR operation.

For the other direction, assume that $W_{rand}(k, S_i) = W_{rand}(k, S_j)$. Then, since modulo-$2^{31}$ congruence is preserved under subtraction:

$$A((AS_i + B) \text{ XOR } D(k)) \equiv \\ A((AS_j + B) \text{ XOR } D(k)) \pmod{2^{31}} \qquad (17)$$

where $A = 1103515245$, $B = 12345$. But A and 2^{31} are relatively prime, so a standard result from number theory [21] tells us that we may cancel A, leaving us with:

$$(AS_i + B) \text{ XOR } D(k) \equiv (AS_j + B) \text{ XOR } D(k) \pmod{2^{31}}$$

[3] Since `rand` is no longer the same on all platforms, implementations should use Equation 16 directly.

[4] E.g., 7 on an i386 with `gcc -O2`, not counting the digest.

Using the fact that modulo-2^{31} congruence is preserved under the XOR operation, then by repeating the procedures above, we finally get that $S_i \equiv S_j \pmod{2^{31}}$. This will be the case if and only if the low 31 bits of S_i and S_j are the same. □

Theorem 8 can be used to determine when ties can occur in a given domain. It guarantees, for example, that a tie can occur in IPv4 only if the IP addresses of the two servers differ only in the most significant bit. Thus, no tie-breaking rule is needed such as when it is known that all servers are within the same network.

6.1 Comparing weight functions

We now compare the performance of the W_{rand} function with that of other possible Weight functions to see how well it achieves load balancing. We consider several alternative weight functions.

The first competing weight function we consider is based on the Unix system functions `random` and `srandom` in place of `rand` and `srand`, resulting in a weight function we denote W_{random}[5]. The second function we consider uses the Minimal Standard random number generator [22, 23], resulting in the weight function:

$$W_{minstd}(k, S_i) \;=\; (16807((16807 \cdot S_i) \text{ XOR } D(k))) \pmod{(2^{31}-1)}$$

Our third alternative is to modify the W_{rand} function as follows:

$$W_{rand2}(k, S_i) = (1103515245((1103515245 \cdot D(k) + 12345) \text{ XOR } S_i) + 12345) \pmod{2^{31}} \qquad (18)$$

It can be shown that Theorem 8 applies to W_{rand2} as well, using a similar proof. Figure 2 depicts the relationship between W_{rand} and W_{rand2}, where each of the small boxes represents a randomizing filter.

Finally we evaluate the option of performing an 8-bit exclusive-OR over all the bytes of k and S_i to get a single one-byte result. We call this the W_{xor} weight function.

As a basis for comparing functions, we will assume that 100 randomly selected objects are being serviced at a time, and look at the coefficient of variation of the number of objects assigned to each server. Figure 3 shows the results of this simulation, with each point representing an average over 5000 trials.

Graphs (a) and (b) show the results using random addresses for servers, and model the performance when servers are distributed across different networks. In (b), object names which yield consecutive $D(k)$ values starting at $D(k) = \text{E0020001}_{16}$ were used. As can be seen, W_{rand2} exhibits the best performance.

Graphs (c) and (d) show the results using consecutive addresses for servers, starting with the arbitrarily chosen IP address 173.187.132.245. In (d), object names that yield consecutive $D(k)$ values starting at $D(k) = \text{E0020001}_{16}$ were again used. It is interesting to note that all methods but W_{rand} and W_{random} were sensitive to the number of servers. We also ran other experiments (not shown) with different starting server addresses, and observed that the same methods were sensitive to the starting IP address as well. W_{rand} and W_{random} remained relatively unaffected.

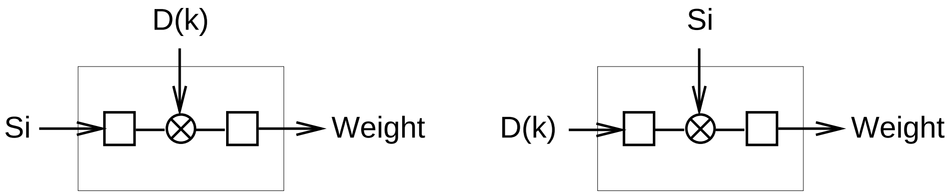
The relative performance of W_{rand} and W_{rand2} can best be understood by examining Figure 2, which represents the functions as two randomization stages separated by an XOR operator. If we fix the input to the first stage at a given value, and input a series of numbers x_i to the XOR operator, we would expect the input to the second stage to be significantly correlated with the series x_i. Whenever we input a sequential series of numbers to the XOR in our experiments, the input to the second stage will be correlated with this sequential series, lowering the degree of randomness of the $Weight$ value output. On the other hand, when the second input is also uniformly distributed, both functions perform similarly.

We also observed that W_{rand} and W_{rand2} were about 200 times faster than W_{random}, since `srandom` is computationally expensive.

We thus conclude that, of those Weight functions studied, W_{rand} and W_{rand2} give the best load balancing performance. The choice of which is most appropriate for use with HRW depends on the characteristics of the domain of use.

Finally, while we could have simulated many other types of pseudo-random functions, we note that finding a good pseudo-random function is hard [22]. Given the good performance of W_{rand} as a hash function, we felt that investi-

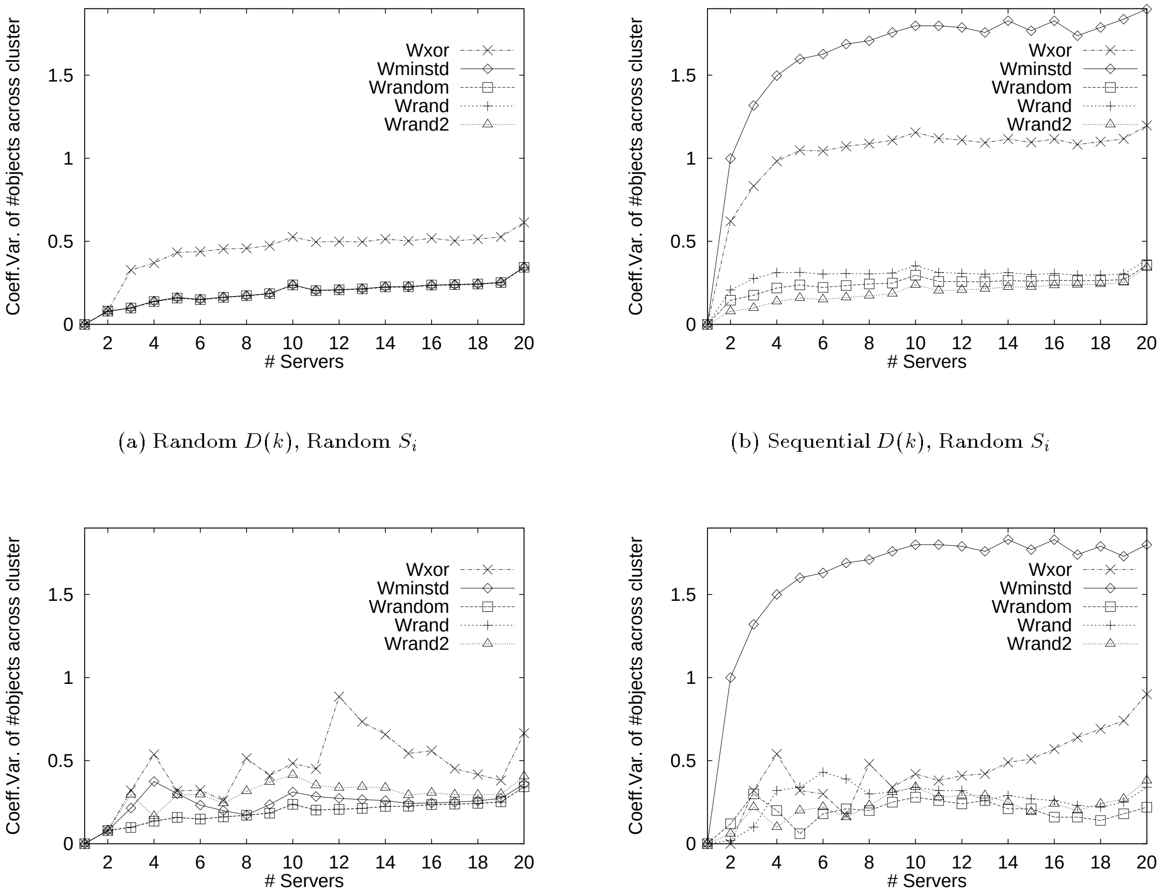
[5] The algorithm used by `srandom` is quite complex. As we will see, it does not perform significantly better than W_{rand} as a hash function, and is not worth describing here.



(a) W_{rand}

(b) W_{rand2}

Figure 2: Two-stage random weight functions



(a) Random $D(k)$, Random S_i

(b) Sequential $D(k)$, Random S_i

(c) Random $D(k)$, Sequential S_i

(d) Sequential $D(k)$, Sequential S_i

Figure 3: Weight functions compared. Y-axis is coeff. of var. of #serviced objects across cluster servers.

gating other weight functions was not necessary.

7 Case Studies

To show how HRW applies to a variety of domains, we now examine two applications in more detail.

7.1 Shared-Tree Multicast Routing

In shared-tree multicast routing protocols such as PIM [2] and CBT [1], receivers' routers request packets for a specific session by sending a "join session" request toward the root of a distribution tree for that session. Sources send data to a session by sending it toward the root of its tree. The root, known as a *Rendezvous Point* (RP) in PIM, and a

Core in CBT, thus takes on the role of a server. Routers with directly-connected senders and receivers become the "clients".

An "object" in this domain is a multicast session identified by an IP multicast group address. The size of the object is unbounded. Since session data is real-time, and may be sent from multiple sources, it is essential for clients and providers to determine the correct server (RP) quickly. Otherwise, real-time data sent by a host could overflow the local router buffers before it is able to identify the correct RP server. Low latency is also important to receivers who want to join a session in progress.

One example motivating the low latency requirement is known as the "bursty-source problem" [18], where a source periodically sends a burst of data, and then remains silent for a long time. An example is an application which broadcasts its current state once a day. If the source's router had to resolve the correct server via an exchange of network messages each time (since the number of sessions may be too large to maintain mapping state for idle sessions), then every burst could be missed and receivers would never get useful data.

In this application, the number of possible objects is large (2^{28} multicast addresses), and all receivers for the same session request the same object, causing a significant concentration of requests. The conditions of Corollary 1 are thus satisfied, and the situation is ideal for the use of HRW.

We focus on Sparse-Mode PIM in particular, since its evolution illustrates many of the concepts and goals discussed in Section 3. The original description of PIMv1 [2] did not specify any mapping algorithm for assigning join requests to servers. Since replication was not prevented, providers sent session data to all servers in a cluster. This resulted in undesirable complexity and resource consumption.

The next step, as the design of PIM evolved, was to specify a Static Priority scheme as the mapping algorithm. This avoided replication, reducing complexity and resource consumption, but meant that the liveness of higher-priority servers in the cluster had to be tracked, incurring additional complexity.

Finally, PIMv2 adopted our algorithm, HRW, as its mapping algorithm. The result is that the protocol complexity and state requirements are significantly lower than in PIMv1. Multicast address allocation can be done using a variety of methods, so that object names may be assigned randomly or sequentially, while servers are likely to be scattered among many subnets within the routing domain. Since these circumstances roughly correspond to graphs (a) and (b) of Figure 3, W_{rand2} was adopted as the weight function of choice in PIMv2.

7.2 WWW Client-side Proxy Caching

World Wide Web (WWW) usage continues to increase, and hence popular servers are likely to become more and more congested. One solution to this problem is to cache web pages at HTTP proxies [3, 4, 24]. Client requests then go through a local proxy server. If the proxy server has the page cached, the page is returned to the client without accessing the remote server. Otherwise, the page is retrieved and cached for future use. Various studies (e.g., [25, 26]) have found that a cache hit rate of up to 50% can be achieved. Thus, since the number of possible objects is large, and a significant concentration of requests exists, the conditions are appropriate for HRW.

Popular WWW browsers such as Netscape Navigator [27], NCSA Mosaic, and `lynx`, now allow specifying one or more proxy servers through which requests for remote objects are sent. A single proxy, however, does not provide any fault tolerance. For a robust deployment, multiple proxies are required. For example, a single proxy hostname might map to multiple IP addresses.

Some criteria must be used by a client to select a proxy to which to send a request. We now wish to compare various mapping algorithms through simulation, using an actual trace of WWW requests. In the discussion below, we will use the term "server" below to refer to the proxy rather than the actual server holding the object. This will allow the terminology to apply to the more general problem.

In the following simulations, the objects and object sizes are taken from the publicly-available WWW client-based traces described in [28], where all URLs accessed from 37 workstations at Boston University were logged over a period of five months. Since we are interested in the performance of a proxy scheme, we use only those URLs which referred to remote sites (not within the `bu.edu` domain) and were not found in the browser's own cache. Table 2 shows the characterization of the resulting data set used for simulation. Note that about 50% of the URLs requested were unique, giving an upper bound on the cache hit rate of around 50%, which agrees with the bound observed by [25] and [26].

Traced Item	Value
URLs Requested	186766
Mean object size	16599 bytes
Bytes Requested	3.1 Gbytes
Unique URLs Requested	93956
Mean unique object size	22882 bytes
Unique Bytes Requested	2.1 Gbytes

Table 2: Trace Summary

Since a unique object name k can, in general, have arbitrary length, and we wish to obtain a digest with which we can do 32-bit arithmetic, our simulation defined $D(k)$ to be the 31-bit digest of the object name obtained by computing its CRC-32 [29] checksum and discarding the most significant bit.

7.2.1 Simulation

Our simulator implemented HRW, a round-robin scheme, and a random allocation scheme. In a fourth scheme, similar to least-loaded allocation, a request was sent to the server with the least number of objects currently being serviced (with ties broken randomly). A fifth alternative, which fails to provide robustness, is to add more cache space to a single server, rather than adding another server; thus, all available cache space is combined at a single server.

We first preloaded server caches by simulating the caches with 60000 requests and an LRU replacement strategy (by which point, the caches were full). We then computed statistics over the next 100000 requests. In addition, we made the simplifying assumptions that all objects were cacheable, and that no objects were invalidated during the lifetime of the simulation (160000 requests).

Figure 4 shows how the hit rate varied with the number of servers under each allocation scheme using 100 MB caches. The hit rate of HRW increased, approaching the maximum bound of 50%, since the effective cache size increases linearly with the number of servers. The hit rate decreased for other allocation schemes, however, since the more servers there are, the less likely it is that a previous request for the same object was seen by the same server. Their hit rate curves were similar, since each assigns requests independently of where objects are cached. We observe that, by 6 servers, HRW's hit rate is double the hit rate of the other schemes.

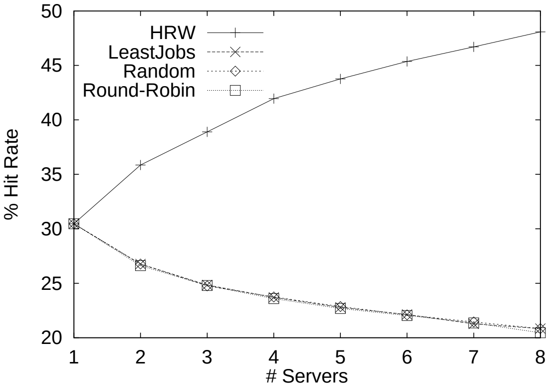


Figure 4: Hit rates of various allocation schemes

In Figure 5, we compare the effects of using HRW with multiple 100 MB servers, against those of combining all the available cache space into a single server (in which case all mapping schemes are equivalent). As can be seen, when HRW is used, adding another 100 MB server is indeed comparable to adding another 100 MB of space to a single server. HRW with multiple servers provides better availability, however. In other words, other schemes give a hit rate that depends on the cache size on *each* server, whereas the HRW hit rate depends on the *total* cache space available at all servers combined. The small difference observed between the two curves in Figure 5 is due to the fact that only whole objects are cached. That is, when there is enough space at all servers combined to hold an object, but not at any single server, HRW must evict an object from a cache.

Figure 6 shows the time the server took to retrieve the requested object (which was zero if the object was cached). By comparison, Glassman [26] found the average response time τ_{MISS} seen by a client for an uncached page to be between 6 and 9 seconds, compared with $\tau_{HIT} = 1.5$ seconds for a cached page, using a Digital Web relay. Using $\tau_{HIT} = 1.5$ and $\tau_{MISS} = 7.5$, Figure 7 shows the expected speed improvement as seen by the client. The line shown is based on the ratio of HRW's latency to that seen by Random. We expect the improvement to be much more pronounced for sites with low bandwidth connectivity to

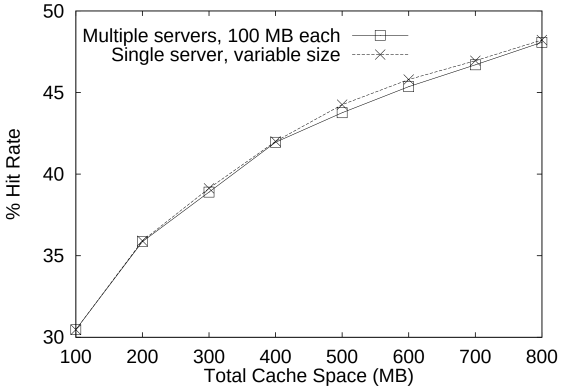


Figure 5: Hit rates of various total cache sizes under HRW

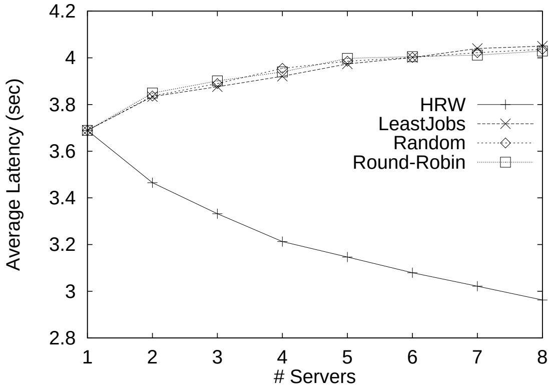


Figure 6: Latency of various allocation schemes

the outside world, such as New Zealand [30], since τ_{MISS} will rise while τ_{HIT} remains constant.

Figure 8 shows the hit rate and cache space used when no limit exists on cache space. Again, since we assume that no objects are invalidated during the lifetime of the simulation, no time-based expirations were simulated, and hence no objects were evicted from any cache. As shown, hash allocation again achieves a much higher hit rate and lower space requirement as the number of servers increases, where the space requirement shown is ratio of the total amount of cache space used at the end of the simulation, over the combined size of all objects requested.

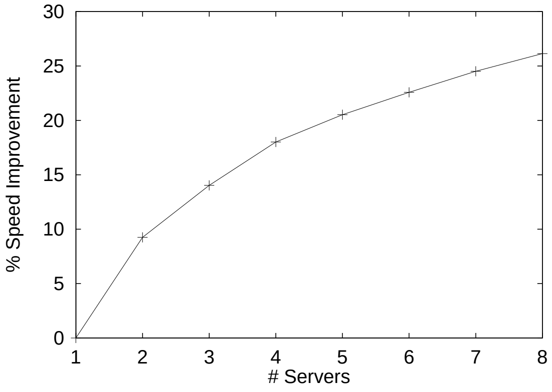


Figure 7: Speed improvement of HRW over Random

In summary, WWW clients can achieve faster response time from clusters of proxy servers by using HRW. For example, when a proxy hostname resolves to multiple IP addresses, HRW could be used to choose an appropriate address, rather than simply using the first address in the list.

8 Conclusions

We began with a model that views the mapping of requests to servers in a cluster as a minimization operation on a value function, and showed that this model adequately characterizes the behavior of typical mapping functions. Typical mapping functions permit replication, resulting in longer latencies and increased space requirements in the domains we consider. We argued that reducing replication would decrease latency and space requirements, and increase hit rates at cluster servers. In combination with the need for all clients to have the same view of which objects map to which servers, these considerations motivated the need for stateless mappings from objects to servers. We described various desirable properties of stateless mappings, including load balancing, minimal disruption as the set of active servers evolves, and efficient implementation. We then described an algorithm (HRW) which meets those needs using a purely local decision on the part of the client.

We compared HRW to traditional schemes for assigning requests to servers, and showed that in distributed caching, using a stateless mapping allows a higher cache hit rate for fixed-size caches, and a lower space requirement for variable-size caches. We also showed that HRW is very useful in real-time producer-consumer domains, where it is valuable for clients to independently deduce object-server mappings, and that HRW allows them to minimize overhead by relying on purely local decisions.

Finally, we provided empirical evidence that our algorithm gives faster service times than traditional allocation schemes.

HRW is most suitable for domains in which there are a large number of requestable objects, the request rate is

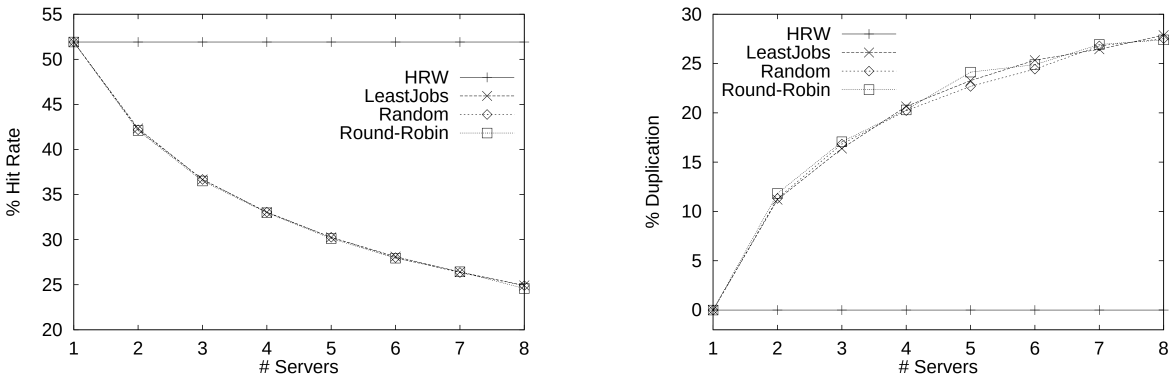


Figure 8: Time-based cache performance

high, there is a high probability that a requested object will be requested again, and the load due to a single object can be handled by a single server.

HRW has already been applied to multicast routing, where it has been recently incorporated by both the PIM [17] and CBT [31] protocols. HRW is also applicable to the World Wide Web. WWW clients could improve response time by using HRW to select servers in a cluster, rather than simply using the order presented by DNS. This improvement would be most significant at sites with low-bandwidth connectivity to the Internet using a cluster of proxy servers for outbound requests.

9 Acknowledgments

This work was supported in part by National Science Foundation Grant NCR-9417032. The authors wish to thank Liming Wei for suggesting simple weight functions, such as W_{xor}, for use with HRW to use for comparison. Thanks also go to the other PIM co-authors who provided the original motivation for and feedback on HRW, and to Boston University for making the web traces publicly available.

References

[1] Tony Ballardie, Paul Francis, and Jon Crowcroft. An architecture for scalable inter-domain multicast routing. In *Proceedings of the ACM SIGCOMM*, September 1993.

[2] Stephen Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. An architecture for wide-area multicast routing. In *Proceedings of the ACM SIGCOMM*, August 1994.

[3] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[4] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Proceedings of the Second International World Wide Web Conference*, pages 763–771, October 1994.

[5] Mark S. Squillante and Edward D. Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[6] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proc. 13th Symp. Operating Syst. Principles*, pages 26–40, October 1991.

[7] James D. Salehi, James F. Kurose, and Don Towsley. The effectiveness of affinity-based scheduling in multiprocessor network protocol processing (extended version). *IEEE/ACM Transactions on Networking*, 4(4):516–530, August 1996.

[8] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. In *Proceedings of the ACM SIGCOMM*, August 1994.

[9] Raj Jain and Shawn A. Routhier. Packet trains – measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):986–995, September 1986.

[10] Marek Fisz. *Probability Theory and Mathematical Statistics.* John Wiley & sons, Inc., 1963.

[11] Peter B. Danzig, Richard S. Hall, and Michael F. Schwartz. A case for caching file objects inside internetworks. Technical Report CU-CS-642-93, University of Colorado, Boulder, March 1993.

[12] Cisco Systems. Scaling the world wide web. Available from http://cio.cisco.com/warp/public/751/advtg/swww_wp.htm.

[13] Reid G. Smith. The Contract Net protocol: High-level communication and control in a distributed problem solver. *ACM Transactions on Computers*, pages 1104–1113, December 1980.

[14] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *IEEE Computer*, 23(5):9–21, May 1990.

[15] T. Brisco. DNS support for load balancing, April 1995. RFC-1794.

[16] Eric Katz, Michelle Butler, and Robert McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–164, 1994.

[17] Estrin, Farinacci, Helmy, Thaler, Deering, Handley, Jacobson, Liu, Sharma, and Wei. Protocol independent multicast-sparse mode (PIM-SM): Specification. *Internet Draft*, September 1996.

[18] D. Estrin, A. Helmy, P. Huang, and D. Thaler. A dynamic bootstrap mechanism for rendezvous-based multicast routing. Submitted to ACM/IEEE Transactions on Networking.

[19] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley, 2nd edition, 1973.

[20] Michael R. Garey and David S. Johnson. Computers and intractability: A guide to the theory of NP-completeness, June 1988.

[21] Ivan Niven, Herbert S. Zuckerman, and Hugh L. Montgomery. *An Introduction to the Theory of Numbers*. John Wiley & sons, Inc., 5th edition, 1991.

[22] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *CACM*, 31(10):1192–1201, October 1988.

[23] David G. Carta. Two fast implementations of the "minimal standard" random number generator. *CACM*, 33(1), January 1990.

[24] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of ACM SIGCOMM'96*, pages 293–305, August 1996.

[25] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Willians, and Edward A. Fox. Caching proxies: Limitations and potentials. *Proc. 4th International World-Wide Web Conference*, December 1995.

[26] Steven Glassman. A caching relay for the world wide web. *Computer Networks and ISDN Systems*, 27(2), November 1994.

[27] Netscape Communications Corp. Netscape Navigator software. Available from http://www.netscape.com.

[28] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Boston University, July 1995.

[29] Mark R. Nelson. File verification using CRC. *Dr. Dobb's Journal*, May 1992.

[30] Donald Neal. The Harvest object cache in New Zealand. In *Fifth International World Wide Web Conference*, May 1996.

[31] A. Ballardie. Core based trees (CBT version 2) multicast routing: Protocol specification. *Internet Draft*, May 1997.