A Name-Based Mapping Scheme for Rendezvous

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

November 13, 1996

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 [1] in multicast routing are two such situations. However, this approach is 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, HRW maps it to a server within a given cluster 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, and where there is a significant probability that a requested object will be requested again.

HRW has now been adopted by the multicast routing protocol PIMv2 [2] as its mechanism for clients to identify Rendezvous Points.

1 Introduction

In the usual client-server model, clients access object data and services that are exported by data providers. However, this model is not robust and often provides insufficient service bandwidth when used with a single provider. Instead, *clusters* of equivalent providers can be used to increase service availability and to lower the workload on individual providers. Such clusters may be formed by grouping providers by location or by functionality, or, more typically, both. Thus, we define a cluster as a set of providers with similar functionality and/or location. For example, the set of fileservers for a specific AFS volume [3] is a cluster.

In this paper, we will refer to the actual exporters of objects and services as *providers*, and use the term *server* in a more generic sense (see Section 1.1). We will also use the term *object* generically to refer to both to concrete entities such as files, and more abstract services exported by providers.

1.1 Proxies and Rendezvous Points

Grouping providers into clusters increases robustness, but several practical problems remain in this model. First, since communication still occurs directly between clients and providers, retrieval latencies can be long, and the bandwidth requirements may extend over larger network regions. Second, in heterogeneous environments such as today's Internet, clients and providers may not even be allowed to communicate directly because of firewalls and other security constraints. Finally, a direct application of this model requires the clients and providers to know each other's identities *á priori*, making it difficult to scale the model to large numbers of clients and servers.

These problems are often solved by adding one or more levels of indirection, placing a cluster of *proxy servers* between providers and clients. These proxy servers then function as *rendezvous points* where clients and providers can meet to exchange information. Proxies can function as object caches (say, in a WWW application), or as stand-ins for providers (say, in a multimedia application).

Figure 1 shows three possible rendezvous schemes. In a lazy scheme, the proxy server functions as a cache for the clients, and retrieves an object from a provider only when it doesn't have a local copy. For example, in the World Wide Web (WWW), pages can be cached at proxy servers [4, 5]. All 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.

In an eager-server scheme, the proxy server knows the provider's identity, and requests objects in anticipation of client requests. For example, FTP mirror sites request and cache a set of objects from a master site (the provider, in this case), and then use their cache to service later requests from clients.

In an eager-provider scheme, the provider initiates contact with the proxy, and provides it with the object data. In this case, the proxy server need not know the provider's identity. For example, in multicast routing protocols such as CBT [6] and PIM [1], 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 becoming clients, and sources the providers.

The term "server" is typically associated with an entity which operates in lazy mode. In our model, providers and proxies can be either lazy or eager. For convenience, we will use the generic term "server" to refer to the individual members of clusters.

1.2 Issues

A request must first be mapped to a cluster, and then to a server within the cluster. We continue to distinguish between a provider, which is the site where the object originates, and a server, which is

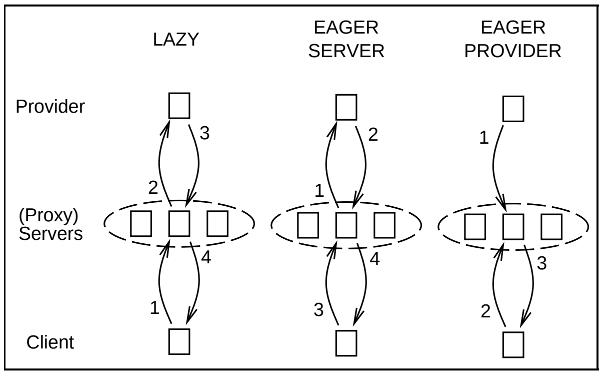


Figure 1: Rendezvous Schemes

generically a member of a cluster.

Servers within a cluster are functionally equivalent, so picking a server within a cluster is typically an issue of performance, not of functionality. However, different clusters can have different functionality, and a request must be sent to a cluster appropriate to the request's class. Thus, the task of picking clusters is very much dependent on the nature of the request.

The request class is typically determined by attributes such as the client's location, a requested object's class, or the provider's location. Typically, an object request includes such information, so identifying its class is hence usually straightforward. Traditional methods for resolving requests have not always clearly separated the issues of identifying clusters and servers within clusters. Thus, name servers may return a list of service providers in response to a query.

This focus of this paper is on picking servers within clusters, not on the specifics of mapping requests to clusters. We treat the mapping of requests to clusters as an issue closer to traditional work in name resolution, and an issue orthogonal to our primary concern here.

There are several possible approaches to obtaining the server list for a given cluster. One option is for clients to be *statically configured* with the list of servers. For example, instead of configuring a WWW browser with a single HTTP proxy, it might be configured with a list of proxies. This approach is also used in the CBT [6] multicast routing protocol.

Since manual configuration can make management time-consuming when the list changes, it is often more reasonable to have all clients dynamically obtain the list of servers from a common location. Thus, a second possibility is for clients to use a *name service* to resolve the list of servers in a cluster.

A good example of such a lazy (or "pull") scheme for retrieving this list is the Domain Name System (DNS) [7]. With DNS, a client can resolve a hostname into a list of IP addresses. Thus, if each IP address refers to one server, a lookup on a well-known hostname representing the request class can retrieve the list of server addresses.

Since performing name service lookups on demand delays the actual request, this resolution could be optimized by having the client periodically request the server list, or by having an authoritative source periodically advertise the server list to interested clients ("push" name service).

"Push" schemes, however, are less useful when client lifetimes are short compared with the distribution periods. In such a case, a client may not even have acquired the server list when it needs to make an object request. They are, however, more suitable when clients are long-lived and object data is real-time. In such cases, push schemes eliminate the latency of retrieving the list at request time.

Since reducing latency is often a main goal when designing distributed services, schemes which incur less network communication are preferable.

This paper makes two major contributions. First, it gives a model for mapping requests to servers: Section 2 describes the model and typical goals of mapping functions, and Section 3 shows how common mapping functions fit within this model. Second, we present a "name-based" mapping function: Section 4 defines 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 A Model for Mapping Requests to Servers

Having identified the cluster that a request maps to, and having obtained the list of servers within that cluster, one must choose a server that will handle the request.

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:

$$\mathcal{F}_t(r_k) = S_i : f_t(i) \leq f_t(j), \quad j \neq i \tag{1}$$

We saw in Section 1.2 how to obtain the server list C for a given cluster. In this section and the rest of this paper, we will discuss methods for mapping requests to servers within a cluster.

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 for mappings

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

Goal 1 (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.*

We measure load balancing effectiveness by the *coefficient of variation* of the loads across the set of servers. This measure is defined as the ratio between the standard deviation and the mean. Thus, if L is the distribution of loads across the set of servers, $CV[L] = \sigma[L]/E[L]$.

To obtain the mean and variance of the distribution L, we will treat the loads across the m servers as a sample of size m from the load distribution for a single server. Theorem 1 below allows us to obtain the mean and variance of such a sample from the mean and variance of the original distribution. Thus, we need only consider the characteristics of the distribution of load on a single server.

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 (in terms of amount of resources consumed) be equally split among the available servers for each train. To measure load balancing effectiveness, we will therefore begin by determining the coefficient of variation of the load seen by each server over a batch of N requests. Thus, if s is a random variable describing the amount of processing done by a server, we will be interested in $CV[s]$. Theorem 1 relates this quantity to the distribution of loads across the servers.

Theorem 1 (Sample mean and variance) *Let $X_m = x_1, x_2, \cdots x_m$ be a sample of size m drawn from a distribution X with mean μ and variance σ^2. Let $\hat{\mu} = (x_1 + x_2 + \cdots + x_m)/m$ and $\hat{\sigma}^2 = ((x_1 - \hat{\mu})^2 + (x_2 - \hat{\mu})^2 + \cdots (x_m - \hat{\mu})^2)/m$ be the mean and variance of the sample X_m, respectively. Then, $E[\hat{\mu}] = \mu$, and $E[\hat{\sigma}^2] = \frac{(m-1)}{m}\sigma^2$.*

Proof: See any standard book dealing with sampling theory, [10], for example. □

We are ultimately interested in the conditions when the variance of the loads across the m servers tends to zero. Theorem 1 tell us that this happens precisely when the variance of the load on a single server tends to zero. Thus, the load is perfectly balanced among servers when

$$CV[s] = 0 \tag{2}$$

since the expected value of s is non-zero. When the load is low, this parameter is less important, but increases in significance as the load (i.e., $E[s]$) increases.

Goal 2 (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 using an 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.

2.2 Additional goals for mappings

This section provides the motivation for our hash-based method for assigning objects to servers. We motivate our approach by discussing the issues of duplication 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, raising efficiency concerns. Replication of work arises, for example, when client requests for the same object are sent to multiple proxy servers, causing them each to retrieve and cache the same object separately.

Such replication is particularly unacceptable in eager-provider domains such as multicasting, where a provider sends real-time object data to a proxy server. Since clients retrieve object data from proxies, the provider and client must *rendezvous* at the proxy for successful information transfer. That is, the client and provider must select the same proxy simultaneously. Real-time data requires low end-to-end latency, so a 100% hit rate at the proxy is *required* for a successful rendezvous between provider and client. Long latency would otherwise result, since the proxy selected by the client must first retrieve the real-time data from the provider.

A 100% hit rate at the proxy can only be achieved either when the provider multicasts its information to all proxy servers in a cluster (wasting resources), or when all clients send requests for the same object to the same proxy server. As discussed towards the end of this section, the second of these options may be viewed as defining *affinities* between objects and servers.

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

Goal 3 (Rendezvous): *The mapping scheme should attempt to increase the probability of a cache hit through rendezvous, thereby decreasing retrieval latency.*

Replication can also reduce hit rates in lazy and eager-server caching schemes by decreasing the effective cache size of the servers in the cluster. For example, a study conducted in 1992 reported [11] 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 proxy to get an equivalent hit rate with only a 1 GB cache.

We thus formulate the following goal, which has much in common with the last one:

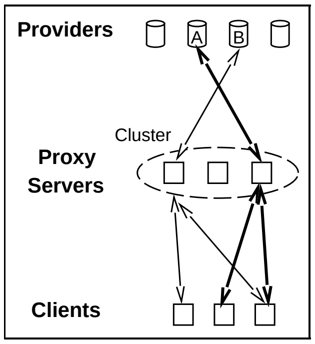


Figure 2: Object-Server Affinity

Goal 4 (Reducing Replication): *The mapping scheme should attempt to increase the effective cache size by mapping all requests to the same server, reducing replication and increasing cache hit rate.*

If the growth of replication is slow, however, it is unlikely to be matter for concern. However, as Theorem 7 of Section 5.2 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.

Cache-Affinity Scheduling

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 2). A number of studies (e.g., [12, 13, 14]) 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.

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 (in terms of latency) from the servers. 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

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:

Goal 5 (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 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 2 (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 $\delta \geq \frac{N/m}{N} = \frac{1}{m}$. □

Finally, in some domains, it is useful to include an additional desirable property:

Goal 6 (State aggregation): *The mapping must allow "similar" objects (however defined) to map to the same server.*

This goal will be explored in more detail in Section 5.1.4.

3 Commonly-used mapping functions

A number of mapping functions have been used in practice that attempt to realize one or more of the above goals. The goal of balancing loads is particularly significant from our point of view since it directly influences response times. The coefficient of variation of the distribution of loads across servers will continue to be our primary measure of load balancing effectiveness.

We now present some of the 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 in Section 2. 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.

However, this is exactly the approach taken in many existing systems. For example, Cisco's LocalDirector [15], which redirects WWW requests to one of a set of local servers, periodically queries the servers for status information. In the Contract Net protocol [16], servers are queried for load information when a request is ready.

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 [5] web cache implementation and the Andrew File System (AFS) [3] 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, thereby balancing the number of requests sent to each [17].

To get an ordered list, round-robin is logically equivalent to assigning weights in our model as:

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

where r 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 r be a random variable describing the service time for one request. Let s be a random variable describing the total service time for all requests in the batch which are mapped to a given server. (Note that s is independent of the queue discipline.)

Theorem 3 (Round-Robin Load Balancing): *Let s be a random variable describing the total processing required for all requests mapped to a given server. If N requests are assigned to m servers in a round-robin manner, then the square of the coefficient of variation of s is given by:*

$$CV[s]^2 = \left(\frac{m}{N}\right) CV[r]^2, \tag{3}$$

and hence, when r has finite variance,

$$\lim_{N \to \infty} CV[s] = 0.$$

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 s for a given server is the sum of the service times of the requests mapped to it, we get:

$$E[s] = (N/m)E[r] \tag{4}$$

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

$$Var[s] = (N/m)Var[r] \tag{5}$$

Equation (3) directly follows from (4) and (5), since $CV[s]^2 = Var[s]/E[s]^2$. $\square$

Theorem 1 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 random server (e.g., $f(i) = random()$), as suggested in [3]. 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 r be a random variable describing the service time for one request. Let s be a random variable describing the total service time for all requests in the batch which are mapped to a given server. (Note that s is independent of the queue discipline.)

Theorem 4 (Random-Split Load Balancing): *Let s 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 s, is given by*

$$CV[s]^2 = \left(\frac{m}{N}\right) CV[r]^2 + \left(\frac{m-1}{N}\right) \tag{6}$$

and hence, when r has finite variance,

$$\lim_{N\to\infty} CV[s] = 0.$$

Proof: Since the value of s for a given server is the sum of the service times of the requests mapped to it, we get:

$$E[s] = (N/m)E[r] \tag{7}$$

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

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

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:

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

This equation can be split into the terms for each moment of r:

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

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

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

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	$f_r(i) = (f_{r-1}(i) - 1) \pmod N$	# Requests	Low	Yes
Random	$f(i) = random()$	# Requests	Low	Yes
HRW	(see Section 4.1)	# Objects	Low	No

Table 1: Mapping Functions

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:

$$E[s^2] = (N/m)E[r^2] + (N(N-1)/m^2)E[r]^2 \tag{8}$$

$$\begin{aligned} Var[s] &= E[s^2] - E[s]^2 \\ &= (N/m)E[r^2] + (N(N-1)/m^2)E[r]^2 - (N^2/m^2)E[r]^2 \\ &= (N/m)E[r^2] - (N/m^2)E[r]^2 \end{aligned} \tag{9}$$

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

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

Simplifying, and using the identity $CV[r]^2 = (E[r^2]/E[r]^2) - 1$, we obtain Equation 6. $\square$

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

4 Mappings Based on Object Names

Not every scheme which avoids duplication has a low disruption coefficient. For example, the Static Priority mapping algorithm certainly avoids duplication, 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 not on the state at the cluster servers or that held in a name server. Such stateless mapping can be viewed as a *hash function*, where the key is the name of the object, and the "buckets" are servers. There are, however, two important differences between our use and typical use of hash functions.

First, 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 [2], 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. However, we are interested in the more general case, 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 changed 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. A logically equivalent scheme would be to use the **Lowest Random Weight (LRW)**, but without loss of generality, we analyze only HRW.

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

$$\mathcal{F}_t(r_k) = S_i : Weight(k, S_i) \geq Weight(k, S_j), \quad i \neq j. \tag{10}$$

where k is the object name, S_i is the 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 Section 2 and Section 4.

5.1 Load balancing

Let K be the total number of requestable objects. Let p be a random variable describing the popularity of an object (i.e., the probability that a request is for the given object). Let m be the number of servers. Let q be a random variable describing the popularity of a server (i.e., the probability that a request will be sent to that server). In general, q will be the sum of the popularities of all the objects mapped to that server.

We now prove two theorems that characterize the load-balancing properties of HRW. Theorem 5 states that HRW balances the number of requests received by each server when K is large. Theorem 6 states that the amount of processing done by each server is balanced when both K and N are large.

Theorem 5 (Hash-Allocation Request Balancing): *Let K objects be randomly partitioned among m servers, with each server receiving exactly K/m objects. Let p and q be random variables characterizing the object and server popularities, as defined above. Then the square of the coefficient of variation of q is given by:*

$$CV[q]^2 = \left(\frac{m}{K}\right) CV[p]^2, \tag{11}$$

and hence, when p has finite variance,

$$\lim_{K\to\infty} CV[q] = 0.$$

Proof: Since the value of q for a given server is the sum of the popularities of the objects mapped to it, we get:

$$E[q] = (K/m)E[p] = (1/m) \tag{12}$$

Since popularities of individual objects are i.i.d., the variance is also additive, giving:

$$Var[q] = (K/m)Var[p] \tag{13}$$

Equation 11 directly follows from 12 and 13, since $CV[q]^2 = Var[q]/E[q]^2$. □

Theorem 6 (Hash-Allocation Load Balancing): *Let K objects be randomly partitioned among m servers, with each server receiving exactly K/m objects. Let N be the request batch size, and let the service time r of requests and p have finite variance. Then, if s is the amount of processing done by a server,*

$$\lim_{N\to\infty}\lim_{K\to\infty} CV[s] = 0 \tag{14}$$

Proof: From Equation 12, $E[q] = (1/m)$, and from Theorem 5, the coefficient of variation of $q \to 0$ as $K \to \infty$ for any server, so that $q \to (1/m)$. We can now apply Theorem 4, and Equation 14 follows immediately. $\square$

Using Theorem 1, we can now conclude that the processor loads are balanced when the conditions of Theorem 6 are met.

5.1.1 Low overhead

It is easy to see that the algorithm as described requires no additional information to be exchanged. This allows clients to make an immediate decision based on purely local knowledge.

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

When a server goes down, clients must reassign all objects previously hashing 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 $Weight(k, S_i)$.

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.

5.1.2 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. There are two possible methods for achieving such consistency quickly.

First, we could require participation from servers themselves, and have them maintain consistent lists. In this case, HRW could be run at the servers, and one server could forward or redirect client requests to another server. This method, however, incurs additional latency.

Alternatively, we could require clients to arrive at consistent lists, and try to minimize the the convergence time for clients (and providers in eager-provider domains) to reach consistency in their server lists. One option is for clients to periodically resolve the server list amongst themselves, in which case the convergence time is equal to the resolution period. In this case, the resolution period should be chosen such that the convergence time is acceptable. On the other hand, when server lists are obtained from a "push" name service, this convergence time is simply the time until all clients receive a new sever list from the name service. A more detailed analysis of convergence time can be found in [18].

5.1.3 Minimal disruption

When a server S_i goes down, all objects which hashed to that server will be reassigned (i.e., $\{k : Weight(k, S_i) > Weight(k, S_j) \quad \forall S_j \in \mathcal{S}, S_j \neq S_i, S_j \text{ is up}\}$). All other objects will be unaffected, and so the optimum disruption bound is achieved. The objects reassigned will also 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 in the set $\{k : Weight(k, S_i) > Weight(k, S_j) \quad \forall S_j \in \mathcal{S}, S_j \neq S_i, S_j \text{ is up}\}$. As above, this again achieves the optimum disruption bound of $1/N$.

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

5.1.4 Allowing state aggregation

One simple approach to achieving state aggregation when similarity is evident from the object name is to use a filter $F(k)$ on the object name in place of k in Equation 10, such that similar names give the same weights and hence map to the same server. For example, in multicast routing, $F(k)$ might mask out the lowest two bits of a multicast address k, so that each group of four consecutive multicast addresses map to the same rendezvous point. As another example, in web proxy caching, $F(k)$ could be just the hostname portion of the URL to aggregate by remote server, or all but the last component to aggregate by directory.

It must be noted, however, that a tradeoff exists between state aggregation and the granularity of load balancing. That is, the more objects are aggregated, the courser the granularity over which the load can be divided.

5.2 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 3 and 4 demonstrate. However, balancing server loads is not the primary criterion for favoring a mapping. Ultimately, it is often more important to optimize response latency. For the application domains of our interest, server load balancing is an important goal only to the extent that it helps optimize response latency.

Optimizing response latency means reducing both the expected value of as well as the variance of the response time. A serious problem with the Round-Robin and Random mappings is that improvements in response latency 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 7 (Replication growth): *Let $S = \{S_1, S_2, \cdots S_m\}$ be a cluster of m servers, and let $r_1, r_2, \cdots$ be a series of requests for object k. For each such request r_i, randomly select a server S_j, and assign r_i to S_j. If p requests are processed before all m servers cache the object k, then*

$$E[p] = m\left(\ln m + \gamma + \frac{1}{2m} + O(\frac{1}{m^2})\right), \tag{15}$$

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

Proof: We can view the progression to full replication as a series of stages, with l servers caching the object k in stage l. Stage m is thus the final stage.

Let n_i be the number of requests required for the system to transition from stage $(i-1)$ to stage i. Clearly, $n_1 = 1$. By definition,

$$p = \sum_{i=1}^{m} n_i \tag{16}$$

During stage 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 stage-i request is sent to a server that does not cache the object is $(m-i+1)/m$. Thus n_i, the number of stage-i requests follows a geometric distribution, so that

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

Using linearity of expectation in Equation 16, 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

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

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.

5.2.1 Caching under HRW

Since the goal of a caching policy is to maximize the hit rate, a caching policy attempts to predict which objects are most likely to be requested next.

An "optimal" caching policy is one in which the probability that the next request will be for a cached object is maximized. Let each object k have a size s_k and a probability $p_k(t)$ of being the next requested. The parameter t emphasizes that p_k varies over time. The expected hit rate of 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 are those maximizing $\sum_{k \in \mathcal{C}} p_k(t)$, subject to the constraint that $\sum_{k \in \mathcal{C}} s_k \leq C$. This is an example of the Knapsack problem, which is known to be NP-complete [20].

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

We will refer to mapping schemes under which the actual $p_k(t)$'s are the same for all servers 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 $(p_k(t) > 0$ at server $S_i) \Rightarrow (p_k(t) = 0$ at $S_j), \forall j \neq i$ as (completely-) "partitioned mappings". These include HRW and Static Priority.

Assuming the cache size for each server is the same, all servers under a non-partitioned mapping will have the same expected hit rate available under an optimal caching scheme, since all parameters of the Knapsack problem are the same for all servers. However, as the number of servers grows, the estimates of $p_k(t)$ can degrade in quality, since each server bases its estimates on a smaller number of (and less recent) observations per object. 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 8 *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: 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 at that server. 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, P_0 is the same for all servers, since the Knapsack parameters are the same, and 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 $\sum_{k \in \mathcal{C}_i} p_k(t) = P_i \geq P'_i = \sum_{k \in \mathcal{C}'_i} p_k(t)$. Thus, P_i was not the optimum solution to the Knapsack problem at S_i, and we have a contradiction. $\square$

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, the number of observations per object remains constant as the number of servers increase, causing no degradation in quality of estimating $p_k(t)$.

Since typical caching policies attempt to approximate optimal solutions, the above reasons explain why 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.

As we have seen above, 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 6 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. 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:

$$W_{rand}(k, S_i) = (1103515245 \cdot ((1103515245 \cdot S_i + 12345) \text{ XOR } D(k)) + 12345) \pmod{2^{31}} \tag{17}$$

where $D(k)$ is a 31-bit digest of the object name k (or of some filter function $F(k)$), and S_i is the address of the i^{th} server in the cluster.

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

```
srand(Si);
```

[1] Since `rand` is no longer the same on all platforms, implementations should use Equation 17 directly.

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

This function can be implemented with only a few machine instructions, requires only 32-bit integer arithmetic, and exhibits a number of desirable properties, as we will see in Section 6.1.

In the unlikely event that two 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 can occur:

Theorem 9 *In the W_{rand} function, a tie between two servers S_i and S_j occurs* **if and only if** $S_i \equiv S_j \pmod{2^{31}}$. *That is, a tie can only occur if the IP addresses of the two servers differ only in the most significant bit.*

Proof: First, assume that $S_i \equiv S_j \pmod{2^{31}}$. Then it is easy to see from Equation 17 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:

$$A((AS_i + B) \text{ XOR } D(k)) \equiv A((AS_j + B) \text{ XOR } D(k)) \pmod{2^{31}} \tag{18}$$

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:

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

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. □

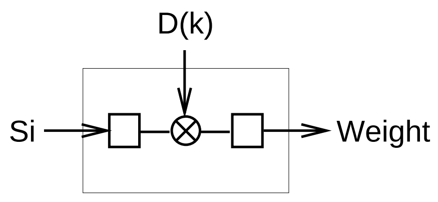
Thus, no tie-breaking rule is needed when the conditions of Theorem 9 are not met, such as when it is known that all servers are within the same network.

6.1 Comparing weight functions

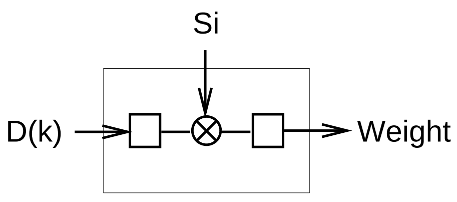
We now compare the performance of the W_{rand} function with that of other possible Weight functions to see how well it achieves load balancing. We consider several alternative weight functions.

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}. The second function we consider uses the Minimal Standard random number generator [22, 23], resulting in the weight function:

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



(a) W_{rand}



(b) W_{rand2}

Figure 3: Two-stage random weight functions

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

$$W_{rand2}(k, S_i) = (1103515245((1103515245 \cdot D(k) + 12345) \text{ XOR } S_i) + 12345) \pmod{2^{31}} \tag{19}$$

It can be shown that Theorem 9 applies to W_{rand2} as well, using a similar proof. Figure 3 depicts the relationship between W_{rand} and W_{rand2}, where each of the small boxes represents a randomizing filter.

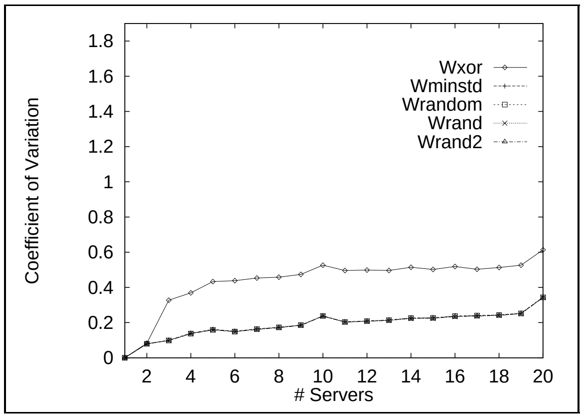
Finally we evaluate the option of performing an exclusive-OR over the four bytes of k and the four bytes of S_i to get a single one-byte result. We call this the W_{xor} weight function.

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 4 shows the results of this simulation, with each point representing an average over 5000 trials.

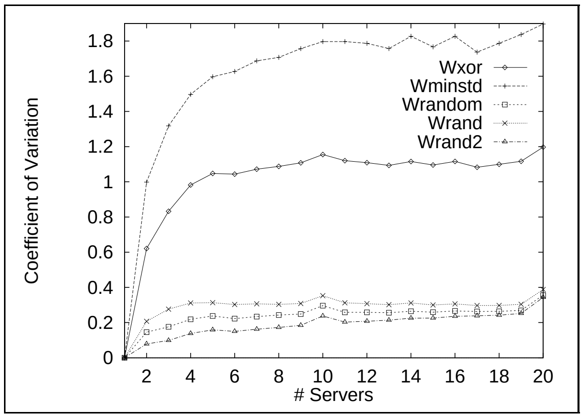
Graphs (a) and (b) show the results using random addresses for servers, and model the performance when servers are distributed across different networks. In (b), object names which yield consecutive $D(k)$ values starting at $D(k)$ = (hex)E0020001 were used. As can be seen, W_{rand2} exhibited the best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

Graphs (c) and (d) show the results using consecutive addresses for servers, starting with the arbitrarily chosen address 173.187.132.245. In (d), entity names that yield consecutive $D(k)$ values starting at $D(k) = (hex)E0020001$ 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 address as well. W_{rand} and W_{random} remained relatively unaffected.

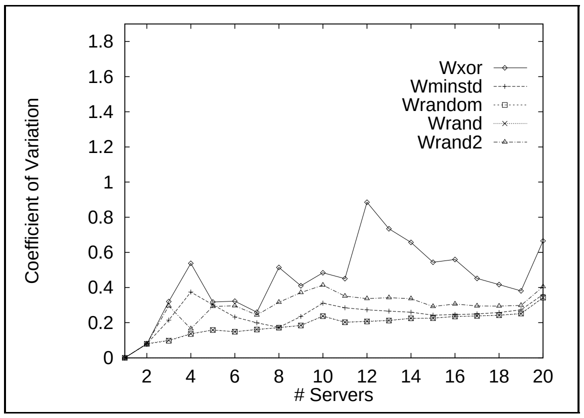
The relative performance of W_{rand} and W_{rand2} can best be understood by examining Figure 3, which represents the functions as two randomization stages separated by an XOR operator. If we fix the input to the first stage at a given value, and input a series of numbers x_i to the XOR operator, we 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.



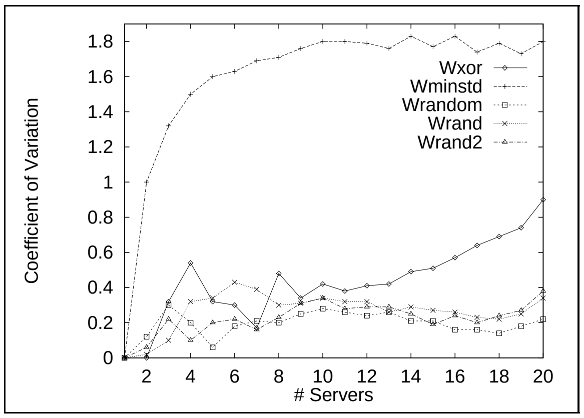
(a) Random objects, random servers



(b) Sequential objects, random servers



(c) Random objects, sequential servers



(d) Sequential objects, sequential servers

Figure 4: Coefficient of variation of weight functions

We also observed that W_{rand} and W_{rand2} were about 200 times faster than W_{random}, since W_{random} achieves a faster `random` at the expense of computationally expensive operations in `srandom`.

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.

7 Case Studies

To show how HRW applies to a wide variety of domains, we now examine two applications in more detail. The first is in an eager-provider domain, and the second is in a lazy-server domain.

7.1 Eager-Provider Case Study: Shared-Tree Multicast Routing

In shared-tree multicast routing protocols such as PIM [1] and CBT [6], 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, with other routers as clients and providers. Any host may be a sender, a receiver, or both.

An "object" in this domain is a multicast session identified by an IP multicast group address. The size of the object is unbounded. Routers with directly-connected receivers become the "clients", while routers with directly-connected senders become the "providers". 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.

Finally, the ability to allow state aggregation is important so that similar objects map to the same server. In particular, if related sessions use the same tree, they will experience similar delay behaviors, and synchronization between sessions becomes easier.

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 Sections 2 and 3. The original description of PIMv1 [1] allowed a separate cluster of servers (i.e., RPs) per object (i.e., session), causing the the state requirement for cluster information to grow rapidly as the number of objects grew. PIMv1 also 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 the cluster for that object. This resulted in undesirable complexity and resource consumption. In addition, the "name service" mechanism (for determining the cluster given an object) was left as an open issue, since introducing latency for lookups is undesirable.

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 still allowed a separate cluster per object. Thus, the potential state requirement remained large, and cluster "name service" remained an open issue. The Static Priority scheme also meant that the liveness of higher-priority servers in the cluster had to be tracked, incurring additional complexity.

To obviate the need to obtain cluster information per object, Handley and Crowcroft [24] first proposed the idea of using an "algorithmic" mapping to pick a server from a single cluster. PIMv2 [2] thus solves the problems just described by using a single cluster for a range of objects. (A single cluster may even be used for all objects, when the range is the entire object namespace). The cluster information is thus small enough that it can be periodically distributed to, and stored by all clients, servers, and providers. The need for any additional name service at request time is thus obviated, so that latency is kept low.

PIMv2 adopted our algorithm, HRW, as its mapping algorithm, with $D(k)$ employing an address mask to allow aggregating a small number of objects. The result is that the protocol complexity and state requirements are significantly lower than in PIMv1. Since multicast address allocation is done using a variety of methods, objects may be chosen randomly or sequentially as a result, 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 4 in which sequential objects started at multicast address 224.2.0.1 = (hex) E0020001, W_{rand2} was adopted as the weight function of choice in PIMv2.

7.2 Lazy-Server Case Study: WWW 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 [4, 5, 25]. 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., [26, 27]) have found that a cache hit rate of up to 50% can be achieved. Thus, since the number of possible objects is again large, while a significant concentration of requests exists, the conditions are appropriate for HRW.

Popular WWW browsers such as Netscape Navigator [28], NCSA Mosaic, and `lynx`, now allow specifying one or more proxy servers through which requests for remote objects are sent. Relying on a single proxy, however, does not provide any fault tolerance if the proxy goes down. For a robust deployment, multiple proxies are required. For example, instead of configuring a WWW browser with a single HTTP proxy, it might be configured with a list of proxies, or, alternatively, a single proxy hostname might map to multiple IP addresses.

When using multiple proxies, some criteria must be used by a client to select which one to use for a request. For example, different clients could be configured with different servers. This scheme, however, still fails to provide fault tolerance.

As the basis for simulating the performance of HRW compared with other allocation schemes, we used a trace of WWW requests. In this domain, 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 [29], 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. We first preload caches by simulating the caches with 60000 requests and an LRU replacement strategy (by which point, the caches were full). We then compute statistics over the next 100000 requests. In addition, we make the simplifying assumptions that all objects are cacheable, and that no objects are invalidated during the lifetime of the simulation (160000 requests).

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

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 [26] and [27].

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 32-bit digest of the object name obtained by computing its CRC-32 [30] checksum.

7.2.1 Simulation

Our simulator implemented the hash algorithm as described in Section 4, as well as a round-robin scheme, where each request was sent to the next server in succession, and a random allocation scheme, where each request was allocated to an random server. 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).

Figure 5 shows how the hit rate varies with the number of servers under each allocation scheme using 100 MB caches. The hit rate of HRW increases, approaching the maximum bound of 50%, since the effective cache size increases linearly with the number of servers. The other allocation schemes, however, give a lower hit rate 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. They exhibit similar hit rates since each assigns requests

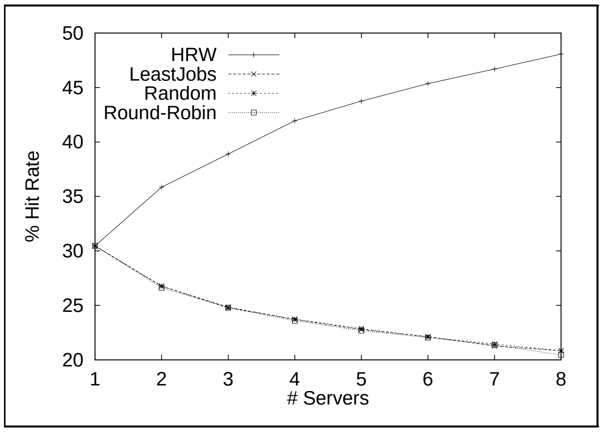


Figure 5: Hit rates of various allocation schemes

independently of where objects are cached. We observe that, by 6 servers, we have doubled the hit rate of the other schemes.

In figure 6, 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. 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. In other words, while other schemes give a hit rate that varies with the cache size on each server, the HRW hit rate varies with the total cache space available at all servers combined.

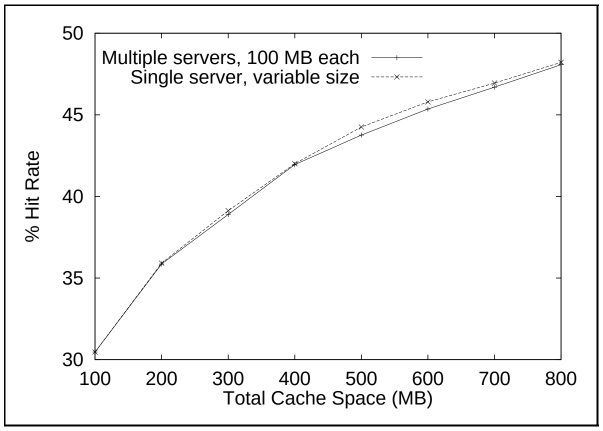


Figure 6: Hit rates of various total cache sizes under HRW

Figure 7 shows the resulting time the server took to retrieve the requested object (which was zero if the object was cached). By comparison, Glassman [27] 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 8 shows the expected speed improvement as seen by the client. We expect the improvement to be much more pronounced for

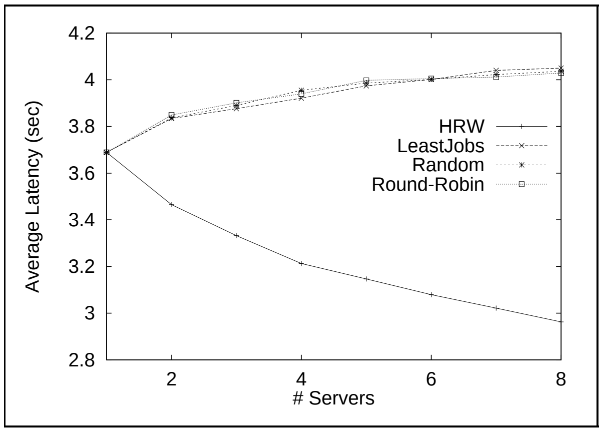


Figure 7: Latency of various allocation schemes

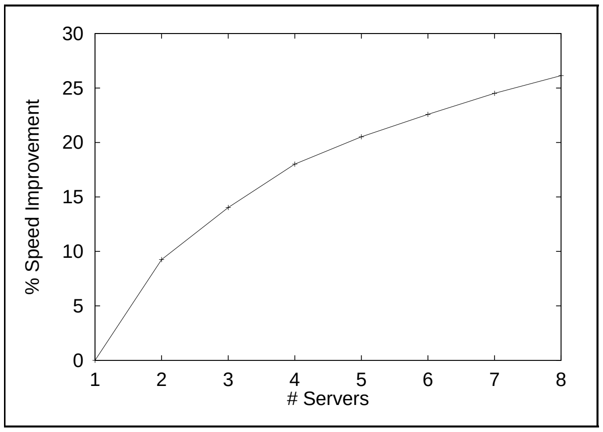


Figure 8: Speed improvement using HRW

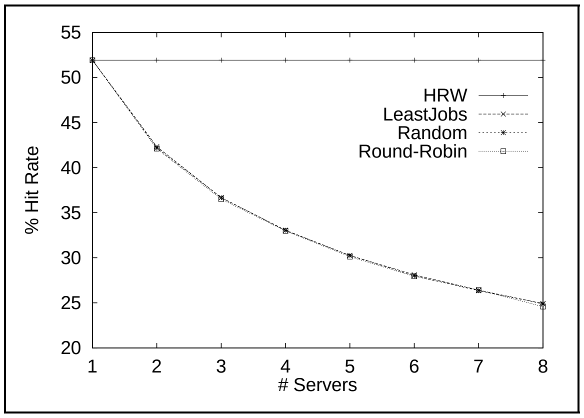
sites with low bandwidth connectivity to the outside world, such as New Zealand [31].

Figure 9 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. As shown, hash allocation again achieves a much higher hit rate and lower space requirement as the number of servers increases.

In summary, both browsers and proxy servers can achieve faster response time from clusters by using HRW. For example, when DNS finds multiple IP addresses for a given hostname, HRW may 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



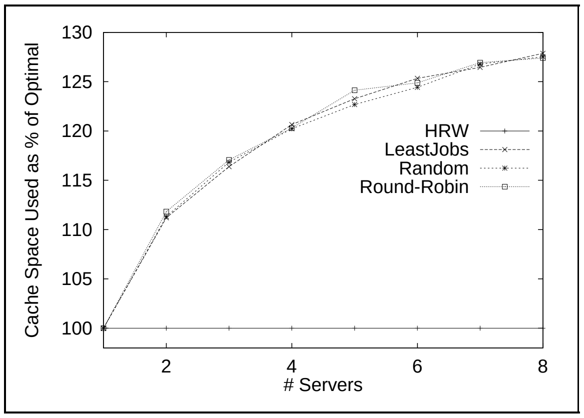


Figure 9: Time-based cache performance

mapping functions. Typical mapping functions permit replication, giving rise to a number of performance problems. 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, efficient implementation, and the ability to accomplish state aggregation. We then described an algorithm (HRW) which meets those needs using a purely local decision on the part of the client.

We have compared HRW to traditional schemes for assigning requests to servers, and have shown 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 have also shown that HRW is very useful in domains like multicasting where it is valuable for clients to independently deduce object-server mappings, and that HRW allows them to minimize overhead by relying on a purely local decisions.

Finally, we have 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 high, and there is a high probability that a requested object will be requested again.

HRW has already been applied to multicast routing, and has been recently incorporated as part of the PIM routing protocol [2]. 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.

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] 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.

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

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

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

[5] 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. Available from ftp://ftp.cs.colorado.edu/pub/cs/techreports/schwartz/Harvest.Conf.ps.Z.

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

[7] P. Mockapetris. Domain names - concepts and facilities, November 1987. RFC-1034.

[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] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] D. Estrin, A. Helmy, P. Huang, and D. Thaler. PIM RP paper. Work in progress.

[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] Mark Handley and Jon Crowcroft. Hierarchical PIM. In *Proceedings of the 34th IETF*, December 1995. Slides available at http://www.cs.ucl.ac.uk/staff/M.Handley/hpim.ps.

[25] 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.

[26] 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.

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

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

[29] 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.

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

[31] Donald Neal. The Harvest object cache in New Zealand. In *Fifth International World Wide Web Conference*, May 1996. Available at http://www.waikato.ac.nz/harvest/www5/.