ELSEVIER

Web caching with consistent hashing

David Karger [1], Alex Sherman [*,1], Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, Yoav Yerushalmi

MIT Laboratory for Computer Science, 545 Technology Square, Room 321, Cambridge, MA 02139, USA

Abstract

A key performance measure for the World Wide Web is the speed with which content is served to users. As traffic on the Web increases, users are faced with increasing delays and failures in data delivery. Web caching is one of the key strategies that has been explored to improve performance.

An important issue in many caching systems is how to decide what is cached where at any given time. Solutions have included multicast queries and directory schemes.

In this paper, we offer a new Web caching strategy based on *consistent hashing*. Consistent hashing provides an alternative to multicast and directory schemes, and has several other advantages in load balancing and fault tolerance. Its performance was analyzed theoretically in previous work; in this paper we describe the implementation of a consistent-hashing-based system and experiments that support our thesis that it can provide performance improvements.

Keywords: Caching; Hashing; Load balancing

1. Introduction

As the World Wide Web becomes a dominant medium for information distribution, mechanisms for delivering Web traffic efficiently and reliably are needed. However, today's data delivery methods are prone to unpredictable delays and frequent failures. Two main causes of these delays and failures are congested networks and swamped servers. Data travels slowly through congested networks. Swamped servers (facing more simultaneous requests than their resources can support) will either refuse to serve certain requests or will serve them very slowly. Network congestion and server swamping are common because network and server infrastructure expansions have not kept pace with the tremendous growth in Internet use.

Servers and networks can become swamped unexpectedly and without any prior notice. For example, a site mentioned as the 'cool site of the day' on the evening news may have to deal with a 10,000-fold increase in traffic during the next day. Thus, planning ahead is of limited benefit; the best schemes for handling load will adapt to changing circumstances.

1.1. Web caching

Caching has been employed to improve the efficiency and reliability of data delivery over the Internet. A nearby cache can serve a (cached) page

* Corresponding author.
[1] E-mail: {karger,asherman}@theory.lcs.mit.edu

quickly even if the originating server is swamped or the network path to it is congested. While this argument provides the self-interested user with the motivation to exploit caches, it is worth noting that using widespread use of caches also engenders a general good: if requests are intercepted by nearby caches, then fewer go to the source server, reducing load on the server and network traffic to the benefit of all users.

The most obvious approach, of providing a group of users with a single, shared caching machine, has several drawbacks. If the caching machine fails, all users are cut off from the Web. Even while running, a single cache is limited in the number of users it can serve, and may become a bottleneck during periods of intense use. Finally, two important limits arise on the hit rate a single cache can achieve. First, since the amount of storage available is limited, the cache will suffer 'false misses' when requests are repeated for objects which it was forced to evict for lack of space. Second, the limit on the number of users that the cache can serve works against the desire to aggregate requests from as many users as possible for caching purposes: typically, the more user requests are aggregated together, the better the hit rate becomes as one user requests objects already requested by other users.

1.2. Related work

To achieve fault tolerance, scalability, and aggregation of larger numbers of requests (which improves hit rates) several groups [2,4,5,7] have proposed using systems of several *cooperating caches*. These systems all share certain common properties. Every client selects one *primary* cache in the system. A request from the client goes to its primary cache. If the primary cache misses, instead of going directly to the content server, it tries to locate the requested resource in other cooperating caches. If it succeeds, a slow fetch of the resource from the content server is replaced by a fetch of the resource from a (presumably closer) cooperating cache. Thus, the other cooperating caches serve as a 'second level cache' to reduce the cost of misses in the primary cache.

The systems differ in precisely how data is located in the case of a primary cache miss. Some schemes broadcast a query to all other caches using multicast [7] or UDP broadcasts [2]. Besides consuming excess bandwidth with broadcast queries, the primary cache must wait for *all* cooperating caches to report misses before it contacts the content server; this can slow down performance on second-level cache misses. Other schemes use directories, either centralized [5] or repeatedly broadcast to support local queries [4]. Directory queries or transmissions also consume bandwidth, and centralized directories can become new points of failure in the system [5].

Another problem with all these systems is the duplication of data among caches. Any cache might be queried for any piece of data, which will cause it to store a copy. On a second-level hit, network bandwidth and time are wasted copying the data to another cache. Worse, these copies evict other pages that can be requested, reducing the number of cache hits. If cooperating caches are all 'near' each other, one might hope to effect a larger cache (with fewer false misses) by storing each object only at one or a few machines [9].

1.3. Our work

In this paper we suggest an approach that does away with all inter-cache communication, yet allows caches to behave together in one coherent system. Cache Resolver, the distributed Web caching system that we developed, eliminates inter-cache communication on a miss by letting clients decide for themselves which cache has the required data. Instead of contacting a primary cache that, on a miss, locates the desired resource in another cache, a user's browser directly contacts the one cache that should contain the required resource. Browsers make their decision with help of a hash function that maps resources (or URLs) to a dynamically changing set of available caches.

Hashing provides several benefits over broadcast- and directory-based schemes. A machine can locally compute exactly which cache should contain a given object. A unicast suffices to get the object or determine that it is not cached, decreasing network usage compared to other schemes. It also discovers misses faster than multicast schemes that must wait for all caches to respond. It avoids the maintenance and query overhead associated with directory based schemes. It does not create new points of failure for

the system — indeed, our scheme exhibits substantial reliability properties that we will discuss later.

Although the above advantages seem enough to motivate a hash-based solution, we also explored another aspect: it lets us push the page-location task down to the individual clients. A directory-based location scheme involves too much overhead to involve every browser, but a hash-based scheme is easy to implement at the browser level. Thus, prior schemes assumed that each client always contacted one fixed primary cache which, on a miss, contacted other caches to see if they had the page. We let the browser decide directly which cache to contact. This removes the need for an intermediary cache, thus improving response time. Furthermore, since all clients contact the same cache for a given page, our caching system suffers only one miss per page, regardless of the number of cooperating caches, rather than one (primary-cache-) miss per cache per page. This reduces the miss rate. The miss rate is further reduced because we avoid making redundant copies of a page; this leaves more space for other pages to be kept in the cache and hit.

While a hash-based scheme has several attractive properties, non-trivial issues must be considered to implement it properly. A theoretical paper developed a tool called *consistent hashing* to address some of these issues [6]. This is the basis of our current implementation work.

A similar proposal for the *Cache Array Routing Protocol (CARP)* appeared in an Internet Draft [3]. CARP is used in the Microsoft Proxy Cache [8]. CARP shares many of the intuitions of our approach, though it has not been justified theoretically as in [6]. An important difference between our proposal and that of CARP is how our hash function is implemented. Current browsers do not have all the functionality needed to support such a scheme. CARP places all consistent hashing responsibility on the browser, which creates some drawbacks we will discuss later. We instead make unusual (but correct) use of the Domain Name System (DNS) to support the browsers' use of hash functions. With modification to browsers, consistent hashing could be implemented entirely inside them without recourse to DNS; however, the DNS approach exhibits some benefits which may make it the correct one for the long term.

1.4. Paper overview

In Section 2 of the paper, we describe consistent hashing in more detail. In Section 3, we describe the implementation of our Web caching system that uses consistent hashing. We compare our system to other Web caching systems in Section 4. We mention other positive aspects of our Web caching system, such as fault tolerance and load balancing, in Section 5. We conclude in Section 6.

2. Consistent hashing

Our system is based on *consistent hashing*, a scheme developed in a previous theoretical paper [6]. Here, we motivate consistent hashing and describe its simple implementation. After outlining its theoretical justification, we describe experiments showing that it also works well in practice.

2.1. Needs

The goal of our system is to let any client perform a local computation that maps a URL to the particular cache that contains it. Hashing is a commonly used tool for this purpose. For example, given a set of 23 caches numbered $0, \ldots, 22$, we might hash URL u to cache $h(u) = 7u + 4 \bmod 23$ (we can think of the URL u as a bit-string which represents a large number). A common intuition about hash functions is that they tend to distribute their inputs 'randomly' among the possible locations. Such a random distribution will intuitively be even, meaning that no one cache becomes responsible for handling a disproportionate share of requested pages. This load-balancing feature of hashing is highly desirable in our application, since a disproportionately loaded cache will become the bottleneck for the entire system.

Unfortunately, standard hashing has several drawbacks when applied to a caching system. Perhaps the most obvious is that caching machines will come up and go down over time. Consider what happens when a 24th cache is added to the system just described. A natural change is to begin using the hash function $h'(u) = 7u + 4 \bmod 24$. Unfortunately, under such a change, essentially *every* URL is remapped to a new cache. This has the effect of flushing all those

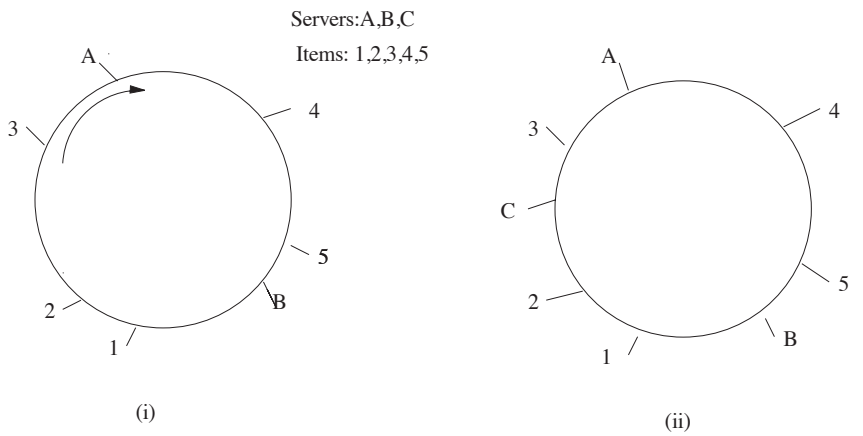

Fig. 1. (i) Both URLs and caches are mapped to points on a circle using a standard hash function. A URL is assigned to the closest cache going clockwise around the circle. Items 1, 2, and 3 are mapped to cache *A*. Items 4, and 5 are mapped to cache *B*. (ii) When a new cache is added, the only URLs that are reassigned are those closest to the new cache going clockwise around the circle. In this case, when we add the new cache, only items 1 and 2 move to the new cache *C*. Items do not move between previously existing caches.

URLs from the caching system: if our system looks for a URL in the new location, the fact that it is cached in the old is useless — we get a miss. This problem is exacerbated by the fact that information propagates through the Internet asynchronously. At any one time, different clients will have different information about what caches are up or down. We refer to the set of caches that a given machine knows about as its *view*, and observe that at any time, many different views will pervade the system. This has two potential drawbacks. If each view causes a URL to map to a different cache, we will soon find that each URL is stored in all caches — precisely the problem we were trying to avoid. Furthermore, with these multiple views in place it becomes difficult to argue that all caches will receive the same amount of load — the different views could steer too much load to one cache, even if each view in isolation appears to balance load appropriately.

Thus, it is critical for our hash function to map items *consistently*: regardless of the existence of multiple, changing views of the system, each item should be mapped to only a small number of machines, and in such a way that all machines get roughly the same load of items.

2.2. Consistent hashing

A simple-to-implement *consistent hash function* [6] satisfies the needs described in the previous section. Choose some standard *base* hash function that maps strings to the number range $[0, \ldots, M]$. Dividing by M, think of it as a hash function that maps to the range [0, 1], which can in turn be thought of as the unit circle [2]. Each URL is thus mapped to a point on the unit circle. At the same time, map every *cache* in the system to a point on the unit circle. Now assign each URL to the first cache whose point it encounters moving clockwise from the URL's point. An example is shown in Fig. 1.

Consistent hashing is easy to implement. All the 'cache points' can be stored in a binary tree, and the clockwise successor to a URL's point can be found (after hashing the URL point) via a single search in the binary tree. This supports consistent hashing to n caches in O(log n) time; an alternative implementation [6] that breaks the circle into equal-length intervals and 'groups' the cache points according to interval can improve this lookup to constant time, regardless of the number of caches. It should be noted that the scheme proposed in CARP [3,8] takes time linear in the number of caches, and is thus much less scalable to large numbers of caches.

For technical reasons detailed in [6], it is quite important to make a small number of *copies* of each cache point — that is, to map several copies of each

[2] For convenience, we will refer to a circle with unit circumference as a unit circle. Usually a unit circle denotes a circle with a radius of one.

cache to different 'random' points on the unit circle. This produces a more uniform distribution of URLs to caches.

2.3. Analysis

We now explain why the consistent hashing scheme just described will have the properties we desire. This is an intuitive summary of formal arguments [6].

Our argument draws from the intuition that the base hash function will map both URLs and cache points 'randomly' to the unit circle. Consider what happens when we add a new cache c to the system. This cache is mapped into the unit circle, and 'steals' certain URLs from other caches. Which URLs are stolen? Those that lie near c on the circle. But by our intuition, URLs are random on the circle. Thus few of them are likely to be near c, meaning that few will be stolen. Observing that URLs which are not stolen by the new cache do not move, we deduce our first desired property: when we add a new cache, few URLs change. Thus most of the cached items persist as hits in the modified system.

A similar argument applies to the problem of multiple views. A cache will get stuck with an item only if it is the closest cache to the item in *some* view. But if the cache is far from the item, then (by the intuition about random placement of caches) it becomes quite likely that in *every* view, some cache is closer. This prevents the cache from being stuck with the item. It follows that only caches close to an item will have to hold it. But the intuition about random placement of caches tells us that only a few caches end up near any one item. In other words, even when there are many caches, only a few caches are ever responsible for any one item.

The intuitions are formalized in the following theorem of [6]. The theorem refers to a *good* base hash function (the one used to map URLs and caches to the unit circle). In [6] it is shown that a random *universal hash function* constructed according to certain principles is good. In practice, standard hash functions that mix well (e.g., MD5) will most likely suffice.

Theorem 2.1 ([6]). *Consider a system with m caching machines and c clients, each with a view of an arbitrary set of half the caching machines. If $\Omega(\log m)$ copies of each caching machine are made and the copies and URLs are mapped to the unit circle using a good basic hash function, then the following properties hold:*

Balance: *In any one view, URLs are distributed uniformly over the caching machines in the view.*

Load: *Over all the views, no machine gets more than $O(\log c)$ times the average number of URLs.*

Spread: *No URL is stored in more than $O(\log c)$ caches.*

Consistent hashing is therefore a good hashing scheme to use in the dynamically changing and uncertain domain of the Internet.

2.4. Implementation results

We list a few brief statistics that demonstrate the usefulness of consistent hashing. First, consistent hashing is a relatively fast operation. For our testing environment, we set up a cache view using 100 caches and created 1000 copies of each cache on the unit circle. We timed the dynamic step of consistent hashing on a Pentium II 266 MHz chip. (The dynamic step includes URL string evaluation, base hash function evaluation, and traversing a binary tree.) On average each call to the hash function took 20 μs. This is about 0.1% of the total time necessary to transmit a 20-kB file from a local cache over 10 Mbit Ethernet. The value of 20 μs can be significantly reduced if we use a bucket array in the underlying cache view representation as opposed to the regular binary tree currently used by our implementation.

We also took some measurements to show that consistent hashing balances well among the caches, as its low load property stipulates. We used a week's worth of logs from the `theory.lcs.mit.edu` Web server. During that period of time, a total of 26,804 unique URL requests were made. We ran these unique URLs through our hash function with a varying number of caches to see how well it would distribute the files among the caches.

Caches	Avg. entries in cache	Std. Dev.	Std. Dev. as % of mean
3	8934	246	2.7
5	5360	173	3.2
8	3350	112	3.4
10	2680	68	2.6

The above data shows that the standard deviation of the number of entries is quite low: around 3% of the mean. These numbers improve when the data set is larger.

In a second experiment, we mapped 1500 names to caches using multiple views, involving 80 machines of which 5 were variably up or down in each view. The total number of (item,cache) pair rose to 1877, only a 25% increase on the base number.

3. Our system

As outlined above, the system we aim to implement seemed simple: a few standard, non-interacting Web caches combined with some hashing logic on the browsers. But when we began to implement our system, we quickly discovered that current browsers are not flexible enough to support consistent hashing unaided. Thus, in order to build a system compatible with current browsers, we made extensive use of the Domain Name System (DNS) to support consistent hashing. Our Web caching system, called the Cache Resolver, consists of three major components: the actual cache machines for storing the content, users' browsers that direct requests toward *virtual caches*, and the domain name servers (also known as resolution units) that use consistent hashing to resolve the virtual caches into specific physical addresses of the cache machines.

3.1. Caches

We installed a distribution of the Squid proxy cache package. A proxy cache replies with the data if it has a valid copy stored. Otherwise, it fetches the data from the original Web server and stores a copy as well. Squid uses an LRU replacement strategy. For load-balancing and fail-over purposes that will be discussed later, we run additional software on the same physical machines as the caches that monitors the squid process. When queried by other units of our system, this software replies with the status (dead or alive) and load information of the cache. When the squid is alive, the load represents the byte transfer rate at which the cache serves its Web requests over the last 30-s interval of service.

3.2. Mapping to caches

To implement browsers' consistent hashing, our first hope was to exploit the *autoconfiguration function* present in most commonly used browsers (Netscape 2.0x and higher and Internet Explorer 3.01 and higher). Users can specify a function written in JavaScript that will be invoked on every request and will select a list of proxy caches to be contacted based on the URL string being requested. The browser contacts the proxy caches in the order listed until it contacts one that responds with the data.

Unfortunately, autoconfiguration is too limited to support consistent hashing. The fundamental problem is that the autoconfiguration script is downloaded once, manually. Should the set of proxy caches change, the mapping function would become incorrect.

In order to get around this problem, we decided to use DNS. We could set up our own DNS servers, and modify them to support consistent hashing. This consistent hashing could be 'propagated' to browsers during name resolution. More precisely, we wrote an autoconfigure script that performs a *standard* hash of the input URL to a range of 1000 names that we call *virtual caches*. We then used DNS to map the 1000 virtual cache names onto actual cache IP addresses via consistent hashing.

In our testing, we discovered that some versions of the browsers under some operating systems will not reinvoke DNS even if they contain an expired resolution of a virtual name. If the cache identified by that resolution goes down, the browser will fail to load the page. To ameliorate this problem, our script actually returns a list of names (here, five names) instead of just one, to make up for the problem of 'broken' browsers. By returning a list, we allow these broken browsers a few more chances to achieve a working resolution of a physical name. The last value on the list is 'DIRECT' which is understood by the browser as an instruction to connect directly to the content server if all the other names fail.

3.3. DNS servers

The primary function of our DNS system is to resolve the virtual names generated by the users' JavaScript function to the actual physical IP ad-

dresses of the caches. We use a number of DNS servers each running a copy of unmodified BIND 8.0 distribution, an implementation of DNS protocol. BIND reads the mappings of virtual names to IP addresses from a *records* file, that is being updated dynamically by another program, called 'dnshelper'. Dnshelper monitors the set of caches, and runs consistent hashing to map all of the 1000 virtual names (see Section 3.2) to the range of only the live cache machines. If the set of available caches changes, 'dnshelper' signals BIND to reload the *records* file that contains new mappings.

The large number of virtual caches means that each will receive only a tiny fraction of the total system load. Consistent hashing guarantees that the number of virtual names assigned to each cache will be evenly distributed among the caches. These two facts together ensure that page load is uniformly distributed over caches. We describe more advanced load balancing strategies with consistent hashing in Section 5.2.

3.3.1. Discussion of DNS

Our use of DNS does in some sense violate our plan to place cache location functionality inside the browsers. However, it can be defended on several fronts. DNS is a standard tool for locating objects, and we are using it in that capacity instead of going to the effort of implementing our own protocol. Note that on any page request, a typical non-caching browser will perform a DNS resolution to find the page's server. We are simply replacing that resolution with one to identify a cache. Such a DNS resolution typically goes to a nearby DNS resolver, so should not add substantial latency to the page request. While our system now becomes dependent on proper functioning of DNS, any failure of DNS will likely bring the user's browsing to an end anyway, so we haven't really added a failure point to the system.

A second argument regarding DNS is that the per-request lookups can be done away with if our system proves itself. DNS is used only to create a mapping from a set of 1000 names to IP addresses. With minor modifications to browsers, such a mapping can easily be stored at the browser. Indeed, current browsers already maintain a cache of roughly 10 DNS entries; all we would have to do is raise this to 1000. Thanks to the properties of consistent hashing, the entries don't even need to be particularly up to date. Thus, a browser can be very lazy about updating its map, dropping a particular cache if it fails to respond, and lazily downloading updated lists of available caches when it happens to open a connection to a cache.

Although we *could* get rid of DNS, at present we see no reason to do so. In our experiments, DNS resolution never noticeably affected the performance of our system.

4. Tests

We implemented our Web caching system because we believed that our design warranted a better performance than the existing Web caching system we studied. In this paper we compare our Cache Resolver against two such systems: the Harvest System and the CRISP described in Section 1.2. We demonstrate results that corroborate our hypothesis.

4.1. Test setup

To test our system, we used a network of seven machines connected to one 100-Mb switch. Three of the machines ran Squid, the proxy cache program. Another machine was designated as a Web server and was placed on a 10-Mb link in order to make data transfer with the Web server costly. Another machine ran a copy of BIND and was the designated domain name server using consistent hashing to resolve names from a virtual space of 1000 names and the three proxy caches. The sixth machine was used to run the test driver. The last machine was used either as a directory for the CRISP test or a parent cache in the Harvest System test.

4.2. Test driver

For the test driver, we used Surge [1], a Web load generating tool developed at Boston University. The designers of Surge studied Web traffic and developed their generator to simulate a proper distribution of object request sizes and frequencies. Prior to the test, Surge generates a database of files to be copied to the Web server. We generated a database of 1500 files of varying sizes with a total size of 34 Mb. Surge also generates a sequence of requests that it goes through

during the test, where each file could be requested more than once. The number of clients each running a specified number of threads is specified to Surge as an argument. Surge then simulates those clients that together go through a request schedule that Surge creates. Each request represents an object with a number of embedded Web files that may overlap among several objects and different files may be requested different numbers of times. The detailed mathematical distribution model used by Surge to simulate a Web server load is described in [1].

We modified Surge to run in two separate modes: *Common Mode* and *Cache Resolver Mode*. Common Mode represents a common cache setup where a set of users always use one local cache and when a miss occurs, that cache either fetches the data from another cache or from the primary content server. For the Common Mode, we run Surge with three clients, where each client talks to its own proxy cache. (A client in Surge runs many threads and represents a whole set of users.) The Cache Resolver Mode was designed to test our system. In this mode, each of the three clients executes a simple hash function similar to the autoconfiguration function utilized by our users. That function takes the URL requested as input and returns a set of virtual names. Surge clients proceed to resolve the names in that set to IP addresses through our DNS unit.

4.3. Results

We ran tests to compare the Cache Resolver with the Common Mode under various proxy cache configurations. We then analyzed proxy cache logs to compare miss rates for different test runs. We expected to see lower miss rates for the Cache Resolver Mode for two reasons. First of all, the Cache Resolver Mode assigns data to specific caches. Also, in these tests we varied cache capacities from 9 Mb to 36 Mb. Since the total database is 34 Mb, caches with smaller sizes will have higher miss rates, because data could be forced out of caches with LRU. In each test we ran, caches were cleared from previous data storage. The three caches were equal in their capacities for each test. Capacities took on values of 9, 12, 18, 24, 30 and 36 Mb for different tests. In the first set of tests we did not configure proxy caches to communicate among one another. They simply fetched data from the origin server on a miss. For this basic configuration, Fig. 2 shows higher miss rate displayed by the Common Mode than the Cache Resolver Mode. The difference in the miss rates is even higher with smaller cache capacities, where the data duplication of the Common Mode has even worse effect.

We also used our setup to test three other common cache system configurations. We tested Sibling Configuration, where all three caches were set up as siblings and used multicast protocol to check for data in other caches on a miss. We tested a Hierarchy Configuration, where we added another cache as a parent to the siblings in the style of the Harvest approach. Finally, we tested a CRISP configuration, where a central directory was set up to be queried by all the caches on a miss. For each system, we measured the primary cache miss rate (the miss rate

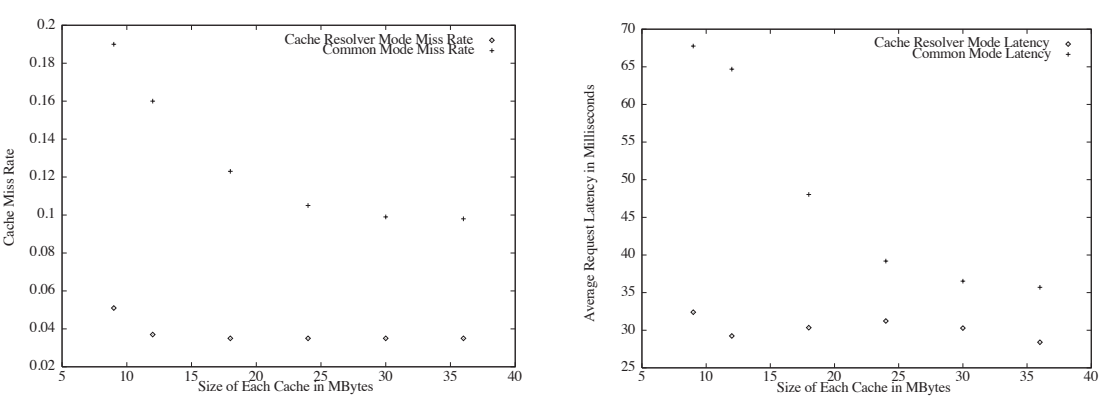

Fig. 2. Common Mode vs. Cache Resolver Mode. *X*-axis represents cache sizes in Mb of all the caches used in the test and *Y*-axis is the cache miss rate (left figure) and average latency in milliseconds (right figure).

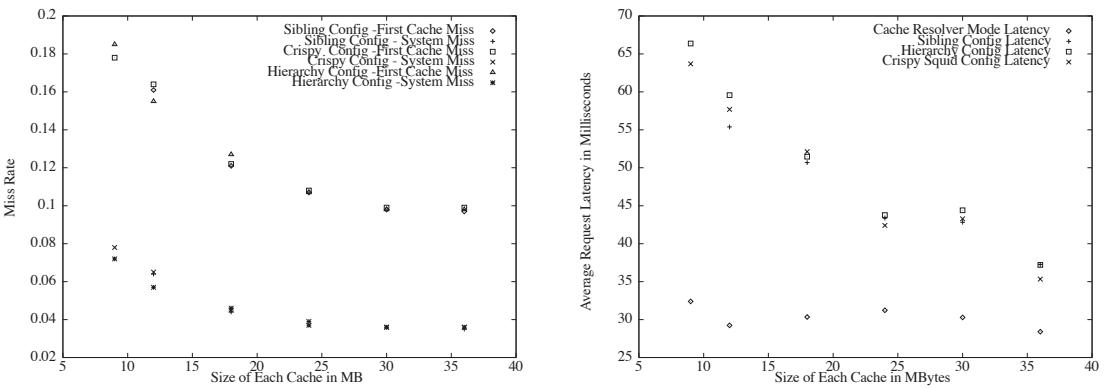

Fig. 3. Miss rates and latencies of three additional configurations.

at the first cache queried by users) and system miss rate (the miss rate of the whole cache system, when no cache had the required data). Fig. 3 shows that the primary cache miss rate for all three configurations resembles the miss rate of the Common Cache Mode displayed in Fig. 2, and the system miss rate for all of the configurations is almost as good as the Cache Resolver Mode's miss rate. For these three systems, there is a penalty associated with the primary cache miss, namely, additional inter-cache communication to check who has the data and inter-cache data transfer. In addition, each inter-cache data transfer results in data duplication which leaves less room in the cache's disk and, most importantly, RAM for fast user service. When the user is left to decide which cache to turn to via a hash function, the penalties associated with extra communication in the critical loop is avoided.

5. Extensions

In this section, we discuss extensions to our basic system that provide locality to users, load balancing and fault tolerance.

5.1. Locality

Regardless of caching scheme, user latency is greatly influenced by the proximity of the cache servers. Our system ensures that users are always served by the caches in their physically local regions. Our caches are split among geographical regions and the users are served only from the caches in their region. We place the knowledge of determining the users' geographical region inside the JavaScript function in the users' browsers. The JavaScript function is customizable: when users download it, they are given a choice of regions. The virtual names generated by the JavaScript function take on the following form: `A456.ProxyCache3.com`, where '456' is the hash of the URL and '3' represents the variable geographical region.

We then split our DNS system into a two-layer hierarchy. The top layer DNS servers resolve for the part of the name that contains geographical information and direct the user's DNS resolver to a set of bottom layer DNS servers that correspond to the user's geographical region. The bottom layer DNS servers are placed physically in a specific geographical region of caches and they resolve virtual names in terms of IP addresses of only those caches.

When resolving the name such as `A456.Proxy Cache3.com`, the user's DNS resolver is first directed to a top layer DNS that can resolve the second part of the name: '`proxyCache3`' and then that DNS directs it to a set of bottom layer DNS servers that resolve the first part of the name: '`A456`'. The bottom layer DNS resolves in terms of its geographically local servers, which are also local to the user.

5.2. Advanced load balancing: hot pages

We have mentioned in Section 2 that we achieve load balancing among our servers since each server

in a local region is assigned to the same fraction of virtual names. This is probably a good approximation when most items are requested with the same frequency. However, there is a number of items on the World Wide Web that are very popular, and others that are not. Items that are popular, such as the CNN front page, for example, will be requested with far greater frequency than most other items. Such 'hot' items will cause a high load on the cache servers that are responsible for caching these items. Such servers can easily get swamped with requests, and either die or start servicing users very slowly.

In order to handle this situation, ideally we would want to know which resources are hot and make sure that the hot resources are served by a larger set of caches. This decision would need to take place at our bottom layer DNS servers that ultimately determine the mapping between virtual names and IP addresses. The solution is to map a 'hot' virtual name to a list of IP addresses instead of just one. By default, when configured to return a list, BIND DNS round robins through the list, so only a fraction of the users will get a specific IP address when querying DNS for that virtual name. Thus the load from that 'hot' virtual name will be spread among the IP addresses on the list returned by BIND DNS.

Unfortunately, it is not trivial to establish which virtual names are hot and which are not. However, we do get some indication from the load measure on each cache. Since the DNS servers know the mapping from virtual names to IP addresses we can establish which set of virtual names is responsible for a high load on a particular cache. That set includes one or more hot virtual names. Because we want to prevent servers from being swamped, we take an aggressive step by spreading all of the virtual names mapped to a hot server to all of the caches in the region. (Meaning that immediately we map each virtual name in the set to the list of all cache IP addresses in the region so that load caused by this virtual name is spread to the entire region.) Then, we slowly reduce the size of the mapping by subtracting from it one IP address at a time. If, at any point, a reduction step causes the load on the server to increase again, we reverse the reduction and reduce from a different subset of the spread names. In this way, we soon reach a balance with some virtual names mapped to variable size lists of caches.

5.3. Fault tolerance

In addition to efficient cache management and load balancing, our system demonstrates a very high level of fault tolerance. First of all, since the physical IP addresses are abstracted from the user through virtual names, users are protected from failures of individual caches. Browsers are not required to time out on attempting to connect to those machines, as the virtual names will resolve to alternative IP addresses. In addition, our system is free of single points of failure, such as the centralized directory of the CRISP. If the centralized directory fails, the CRISP system breaks and the cache machines start acting as individual caches. In our system, there are no units solely responsible for critical tasks such as the centralized directory of CRISP, or a primary cache that serves as system entry point for a group of users. In the Cache Resolver, as long as some DNS machines function, the virtual names will be resolved, and as long as some caches in a region are alive, the virtual names can be resolved in terms of their IP addresses.

6. Conclusion

In Section 4, we compared our system to other cache systems. We demonstrated that when the user is responsible for knowing which cache has the appropriate data, significant penalties in the critical loop of a user request can be avoided. This knowledge can be provided to the user with a hash function. Since it is likely that users of a large network, such as the Internet, may have inconsistent views of live caches, we suggest the use of a Consistent Hash function which balances data quite well despite conflicting user views. Additionally, we have described implementation of our system that uses Consistent Hash function in order to show that it is quite practical to integrate such a system into the World Wide Web. Our system handles locality issues, balances load among caches, and possesses a high level of fault tolerance that is absent from other Web caching systems. In conclusion, we believe that through the efficient use of caches, consistent hashing can significantly improve the efficiency of the World Wide Web.

Acknowledgements

We would like to thank Frans Kaashoek for his help and advice.

References

[1] P. Barford and M. Crovella, Generating representative Web workloads for network and server performance evaluation, in: Sigmetrics, 1997.

[2] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz and K. Worrell, A hierarchical Internet objectcache, in: USENIX, 1996.

[3] J. Cohen, N. Phadnis, V. Valloppillil and K.W. Ross, Cache array routing protocol v. 1.0, http://www.ietf.org/internet-drafts/draft-vinod-carp-v1-03.txt, September 1997.

[4] L. Fan, P. Cao, J. Almeida and A.Z. Broder, Summary cache: a scalable wide-area Web-cache sharing protocol, Technical Report 1361, Computer Science Department, University of Wisconsin, Madison, February 1998.

[5] S.A. Gadde, J. Chase and M. Rabinovich, A taste of crispy squid, in: Workshop on Internet Server Performance, June 1998, http://www.cs.duke.edu/ari/cisi/crisp/

[6] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy, Consistent hashing and random trees: distributed cachine protocols for relieving hot spots on the World Wide Web, in: Proc. 29th Annu. ACM Symp. on Theory of Computing, 1997, pp. 654–663.

[7] R. Malpani, J. Lorch and D. Berger, Making World Wide Web caching servers cooperate, in: Proc. 4th Int. World Wide Web Conference, 1995, pp. 107–110.

[8] Microsoft Proxy Server, White paper, http://www.microsoft.com/proxy/guide/CarpWP.asp, 1998.

[9] P. Yu and E.A. MaxNair, Performance study of a collaborative method for hierarchical caching in proxy servers, Technical Report RC 21026, IBM T.J. Watson Research Center, 1997.