Distributed Web Caching System with Consistent Hashing

by

Alexander Sherman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1999

Author ..... Signature redacted
Department of Electrical Engineering and Computer Science
February 19, 1999

Certified by ..... Signature redacted
Madhu Sudan
Associate Professor of Computer Science
Thesis Supervisor

Accepted by ..... Signature redacted
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Distributed Web Caching System with Consistent Hashing

by

Alexander Sherman

Submitted to the Department of Electrical Engineering and Computer Science
on February 19, 1999, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As the traffic on the World Wide Web increases, users experience more delays and failures in data delivery. The efficiency with which the content is served to the users is critical. Distributed Web Caching is one of the key techniques that improves the effeciency of service. This work describes a Distributed Web Caching system based on the Consistent Hashing algorithms that improve on the methods used by the existing web caching systems. Consistent Hashing as used in our system provides for a more efficient use of caches on the World Wide Web, which translates into faster service for the users. In addition, our system offers features such as load balancing and high fault tolerance, features that are absent from most web caching implementations.

Thesis Supervisor: Madhu Sudan
Title: Associate Professor of Computer Science

Acknowledgments

I would like to thank Professor Tom Leighton and Professor Madhu Sudan, my two supervisors, for their direction and help. I would especially like to thank Bill Bogstad, David Karger, Tom Leighton, Daniel Lewin and Yoav Yerushalmi, all of whom majorly contributed to the work presented here. In addition, I would like to thank undergraduate students Ken Iwamoto, Brian Kim, and Luke Matkins who also participated in this research.

Contents

List of Figures

Chapter 1

Introduction

As a solution to the problem of slow service experienced by most World Wide Web users, we offer an experimental analysis of a web caching scheme that uses consistent hashing algorithms due to [5].

1.1 Delays and Failures on the Web

As the traffic on the World Wide Web increases users experience longer delays and failures. Over the recent months the World Wide Web has become a highly popular medium for news, entertainment and commerce. With advancements in client and server technologies, many web sites can offer increasingly elaborate services such as stock trading, streaming video clips and real audio. As more users join the Internet and take advantage of these services, traffic on the World Wide Web grows steadily, servers get swamped and networks become congested. Swamped servers and network congestion in turn cause greater delays and failures for the users.

Servers for popular web sites often get swamped from too many requests. As the queue of user requests at a server grows, the average response time per request decreases. If the queue grows very large servers spend additional time on queue management and less time serving users. Many users connect to servers via modems that retrieve data at rates much slower than offered by servers. Slow modems force connections to remain open for much longer periods of time in order to retrieve all

of the data. Number of open connections is another limiting factor for the servers and additional requests are dropped. Swamped servers either shut down completely causing long timeouts on connection attempts or continue serving requests very slowly causing users to experience long delays.

Networks that serve as central exchange areas for the Internet often become congested. Multitude of packets often get held up by the routers at such networks. Just like busy web servers, routers process packets at a reduced rate as the request queue grows. At such times router's buffer could overflow and any additional packets will be dropped. Dropped packets require retransmissions and thus waste the precious bandwidth. Congested networks contribute to more delays on the Web.

1.2 Web Caching

Web caching is a common technique used to reduce the number of delays and failures experienced by the users. Caching allows users to bypass both problems mentioned in section 1.1 by storing resources closer to users. Access times are improved since most connection to busy servers are avoided and minimal packet travel over congested networks is required. A common form of web caching is present in most browsers where users can select a storage space of certain capacity to be used for caching. On request for a URL, a browser serves the resource out of the local cache if a copy that is not outdated exists.[1]

However, local browser caching is not a sufficient solution. In order for such a cache to be useful, a fresh copy of requested data must be first placed into a cache, and that requires the user to visit the original web server at least once to retrieve that data. That means at least one connection for every user visiting a given web site. There are hardly any web users today who do not use local browser caches and yet they still complain about slow connections and delays. A more comprehensive

[1] A caching HTTP client is allowed to use cached data only if the time specified in the "Expires" header has not yet arrived. If "Expires" header is not provided, a client is required to issue a conditional request to the origin server and retrieve the data if it is more recent than the client's cached copy. There are also some additional HTTP caching constraints.

solution is to provide for a larger regional cache that can serve all of the users in a region. In such a case only the first user from that region who requests a certain resource will experience a delay, but all the following requests for that resource will be served directly from the cache.

Regional cache also presents a problem. If it is serving only a small group of users it can be placed very close to those users, and retrievals from that cache are likely to be fast. At the same time a small group is unlikely to have a significant overlap in the resources they request, and such a cache would hardly do any better than a local browser cache. On the other hand, if the cache is serving a large region, a significant overlap in user requests is likely. But such a cache would be placed far from most end users, and even though hit rate may be high the data retrieval time will become considerable. In addition, such a cache may itself turn into a swamped server due to many requests. Thus, a regional cache presents a tradeoff between faster access times and higher hit rate.

1.3 Related Work - Distributed Web Caching Systems

In order to achieve both, high hit rate in a cache and proximity to the end user, many research teams have produced designs that combine individual regional caches into more complex distributed cache systems. Common approach of all such systems [8, 2, 3] is to place caches into relatively small regions (i.e. at close proximity to end users) and to improve hit rate by allowing caches to communicate in order to request data from other participating caches in case of a miss. Given that there are many caches significantly close to a region, a miss in the primary cache can still result in a win as it may be less expensive to retrieve data from other caches than the original server located further away.

1.3.1 Malpani

In one such system, Malpani [8] introduces a protocol called "IP Multicast". In case of a miss, caches send multicast packets addressed to all other caches. If one of the caches replies that it has the requested data, it is retrieved from that cache. If no cache has the data, request is forwarded to the original host for that resource. The disadvantage of this system is that communication among caches proves quite expensive when all of the caches send out multicast requests on every miss. Such requests queue up and slow down the server. If the data is not present in any cache, the cache that originated a multicast request waits until it receives all of the replies, while the user waits for the data.

1.3.2 Harvest System

To eliminate all-cache communication, the Harvest system [2] implements "A Hierarchical Internet Object Cache". As the name suggests, caches in this system form a hierarchical tree structure with leaf nodes serving as primary regional caches. In case of a miss a leaf node cache forwards its request only to its siblings and parent as opposed to all caches. This request propagates up the tree until it reaches the root node cache which in case of a miss contacts the home server for the resource. Then, data travels down to the leaf node, and its copy is stored in each cache on its path. Although this system eliminates expensive multicast, there is still significant delay associated with the inter-cache communication. In fact, several layers of communication could take place as request propagates up the tree. In addition, this hierarchical arrangement introduces new points of failure. Few caches that are at the top of the tree are likely to become overloaded if many misses are propagated simultaneously to the top. At such instances all of the users will receive very slow service, and if the top caches fail, the whole system is rendered useless. Harvest designers produced a proxy cache implementation called "Squid" that uses Internet Cache Protocol (ICP, [9, 10]) optimized for inter-cache communication. As the next generation of Squid developers noted in [3], ICP still proved too expensive and a new product, known as

Crispy Squid, was designed.

1.3.3 Crispy Squid

Crispy Squid [3] attempts to reduce inter-cache communication even further by introducing "centralized directory" into the system. On a cache miss communication is reduced to only two steps. Cache sends a lookup request to the directory to find out which other cache (if any) contains requested resource, and then it retrieves the data from a cache suggested by the directory or from origin host if no other cache is known to have the data. Mappings of resources to caches in the directory can be updated asynchronously, and do not contribute directly to user request latencies. Although the directory improves cache miss latencies, it also introduces a point of failure into the system; if the server containing the directory dies, caches stop communicating and start behaving as individual caches. Crispy Squid system allows the directory to be replicated on several servers, however such configuration requires much greater storage space as central directory can be quite large. An alternative configuration is a replicated partial directory where each directory contains only partial mappings somewhat reducing the system hit rate.

Paper written by the Crispy Squid developers [3] and the very existence of various directory configurations suggest, that the Crispy Squid project was also struggling with the same regional cache tradeoff described in section 1.2. One central directory would mean guaranteed hit if any cache in the system had the data, but the access time to one directory for the whole (possibly global) system can be quite large. Replicated partial directory can place one of its parts in every region of the world and provide very fast access times, but the hit rate in such a directory will suffer.

1.4 Need for a New Design

It appears that none of the existing distributed cache systems have been able to find a good solution to resolve the tradeoff between a good hit rate and fast access times. All of these systems found inter-cache communication to be quite expensive.

In addition, complexity of these designs has introduced many points of failure into their systems. Our work suggests a new approach that does away with all inter-cache communication, and yet allows caches to behave together in one coherent system maintaining a high hit rate. At the same time, the system we develop is free of single points of failure such as the centralized directory or a few caches at the top of the cache hierarchy.

1.5 Presentation

Chapter 2 introduces an alternative approach to distributed web caching. This approach is based on consistent hash functions that were developed by the MIT Algorithms group and will be thoroughly described in Chapter 3. Chapter 4 presents the design of the actual system that we implemented. Chapter 5 talks about some test results of our system especially in comparison with other distributed web caching systems. Additional features of our web caching system are discussed in chapter 6. Chapter 7 summarizes the practical relevance of the formal properties of consistent hashing to our web caching system. Directions for future research are outlined in chapter 8. Chapter 9 draws conclusions.

Chapter 2

Hashing

This chapter describes hashing as a possible approach to a web caching scheme. This discussion leads into the introduction of a web caching scheme based on consistent hashing that was proposed by Karger, Lehman, Leighton, Levine, Lewin and Panigrahy in [5].

2.1 A New Approach

Cache Resolver, the distributed web caching system that we developed, eliminates the necessity for any inter-cache communication on a cache miss by letting clients decide for themselves which cache has the required data. Instead of contacting a primary cache which on a miss sends requests to other caches or a directory, user's browser directly contacts the cache that contains the required resource. Browsers make their decision with help of a hash function which maps resources (or URLs) to a set of available caches. With a globally known hash function, all of the system clients (users) can agree on which caches are responsible for storing each resource. Thus, our system speeds up access time to the "right cache" in two ways. First of all, executing a hash function locally is faster than running non-trivial protocols among caches to find the right cache. Also, if there is a "right cache", no time is wasted on a possible miss in the primary cache.

2.2 Desired Properties of Hashing

In order to insure a high hit rate and fast access times, the hash function that will map URLs to caches must possess several properties. First of all, this hash function must distribute data items evenly among caches. If a large disproportionate fraction of resources is mapped to some cache, such cache is likely to become overloaded due to requests for all of the resources. It is also possible that such cache would run out of storage space for the data and the hit rate would decrease. One common hash function that can be used to achieve good balance is a modulo function:

$$f(d) = ad + b \pmod{n}$$

which can be used to map data item d to a set of n caches $\{0, 1, 2, \ldots, n-1\}$. (a and b are relatively prime). For a given distribution of data items it is possible to choose a and b for optimal balance.

Another key property for the mapping function is consistency of mappings of data items to caches. In a world wide cache system that stretches across a myriad of domains and regions, it is practically impossible to maintain a constant set of live cache machines. Some cache machines are likely to become unavailable for service due to shutdown for software re-installation, or due to network, power, or hardware failures. After some time interval, these machines will come back on line and other machines could fail. In addition, the global cache system is expected to grow steadily with new machines added every day. These fluctuations in the set of available cache machines correspond to fluctuations in range of the mapping hash function. As the range changes, if the mappings of documents to caches do not remain consistent, documents will become duplicated among many caches. The miss rate and access times will increase as many new caches will need to access the origin servers for the given documents. The cache system will require more storage for the duplicated documents.

Modulo hash functions, for example, will behave especially poorly under fluctuations in range. Even if only one new cache is added and the the set grows from

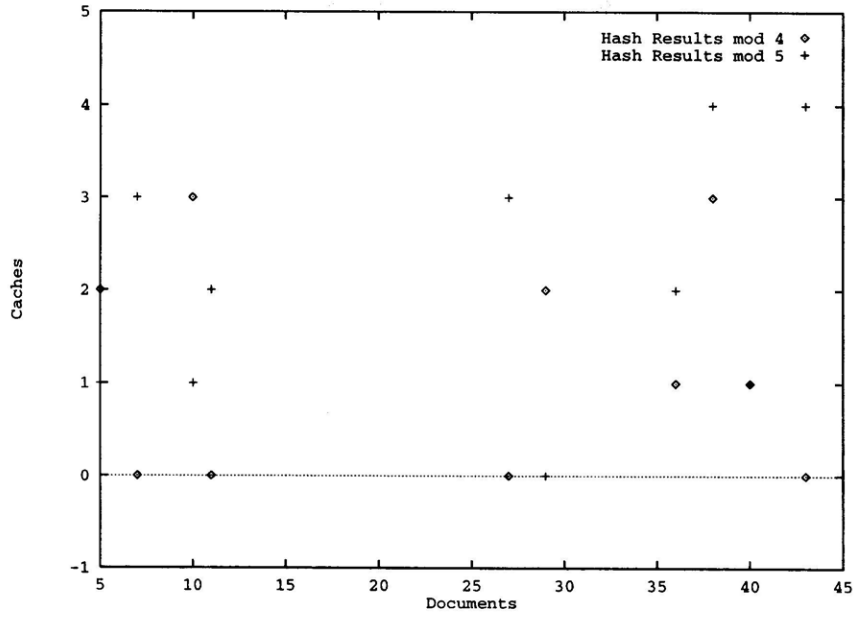

Figure 2-1: Result of adding cache with standard modulo hashing

Hash Results (mod 4) represent the assignment of 10 documents to 4 caches using the hash function $f(d) = d + 1 \pmod 4$. Hash Results (mod 5) represent the assignment after one additional cache is added. The hash function is changed to be $f(d) = d + 1 \pmod 5$. Note, that with the addition of a new cache, almost every URL is mapped to a different cache.

n to n+1, approximately only 1/n fraction of data items keep their old mappings. Example shown in figure 2.2 demonstrates that when a set of 4 caches is increased to 5, the modulo hash function changes from (mod 4) to (mod 5) to include the new cache into the range. As a result, items 5, 7, 10, 11, 27, 29, 36, 38 and 43 get reshuffled to different caches, and only item 40 remains mapped to the same cache. Because of data duplications, the total storage requirement almost doubles.

It is also important to note that in a large system all the users can not be notified instantaneously of all the changes in the set of live cache machines. As different groups of users learn about some changes sooner than others, *views* on the set of live machines are likely to differ among users. As a result, even as the cache set is evolving users could be simultaneously requesting the same data item from different caches. In such a case, if the mapping is highly inconsistent, superfluous copies of this

data item will never become outdated and forced out from caches[1]. Instead, these items will compete with other data for the critical space in the fast access memory of RAM on many machines. Using the same example as in figure 2.2, imagine one *view* consisting of one more cache than another *view*. With a modulo function these two *views* map almost all of the items to different caches, making very inefficient use of the system. Thus, the mapping hash function used in the global web cache system should map items consistently despite possible disjoint views.

2.3 Introducing Consistent Hashing

A family of *consistent hash functions* developed by the Algorithms group at MIT possesses all of the properties required for mapping documents to caches in the global cache system. As the name suggests these functions maintain consistent mappings despite fluctuations in range.

The basic construction of a consistent hash function is quite simple. Both, the data items and the caches, are mapped randomly and independently to a circle with circumference of length 1 as shown in figure 2.3(i). A data item is assigned to the closest cache in the clockwise direction along the circle. Thus, in figure 2.3(i), items 1, 2, and 3 are mapped to cache A, and items 4 and 5 are mapped to cache B. We call the arc from point B to point A in the clockwise direction a *responsibility arc* of cache A, because cache A is responsible for storing all of the data items that fall into that arc. The formal definitions and properties of consistent hash functions will be described in section 3.1.1. In this section, the intuition for the properties of consistent hash functions will be explained.

The "consistency" of consistent hash functions can be observed from the example in figure 2.3(ii). When a new Cache, C, is added to the system, only data item 1 that is now closest to C in clockwise direction change its assignment. However, items that are not closest to the new cache will not change their assignments. Thus, as the

[1]Usually caches use Least Recently Used (LRU) replacement strategy to force least recently used copies to be replaced by more recent content.

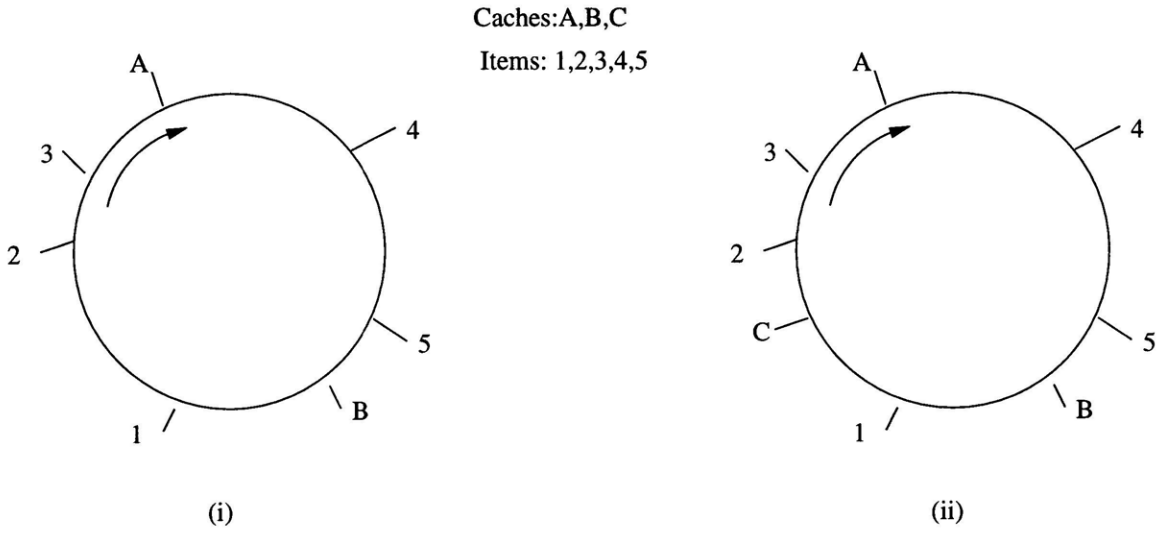

Figure 2-2: The basic construction of a consistent hash function

(i) Both, items and caches, are mapped randomly to points on a circle. An item is assigned to the closest cache going clockwise around the circle. For example items 1, 2, and 3 are mapped to cache A. items 4, and 5 are mapped to cache B. Arrow shows the direction of mapping of items to caches. (ii) When a new cache, C, is added, only items that are closest to that cache (in clockwise direction) are reassigned. In this case, only item 1 is reassigned to the new cache C. Items do not move between previously existing caches. Compare this with the results obtained from standard hashing in figure 2.2

set of caches fluctuates, mappings from URLs to caches change only slightly; only documents that are mapped closest to caches in question will be affected.

Since both, the documents and the caches, are mapped to the circle at random, documents are likely to be well balanced among caches.[2] As a result each cache will be responsible for serving, a small proportionate fraction of the data items. To further reduce the probability of imbalance each physical cache can be represented by multiple points on the circumference, collecting items from more, but smaller arcs of the circle. Figure 2.3 demonstrates an example where adding an additional copy of each cache to the circle helps balance collective lengths of responsibility arcs between caches A and B. As a result of using two copies of each cache, the sum of responsibility arcs of each cache is closer to 1/2. (Remember the total circumference of the circle is 1.) This trick can be shown to improve the distribution and will be discussed further in section 3.1.2.

Following the same intuition, it is possible to see that despite disjoint views among users, the mapping will remain consistent. Since the data items are always mapped to caches closest to them in the counterclockwise direction along the circle, they can only be hashed to caches in their vicinity and not anywhere in the circle. So if there is some disagreement with regards to which caches are alive, documents could be duplicated in more than one cache, but will be constrained to caches in their neighborhood, and the duplication degree will be small.

Next chapter formally states the properties of Consistent Hashing. It discusses practical implementation of consistent hashing and lists a few statistics on usability of these functions.

[2]The probability of balanced distribution is especially high when the data set is large. Since we are considering a set of all the web documents, it is a large set.

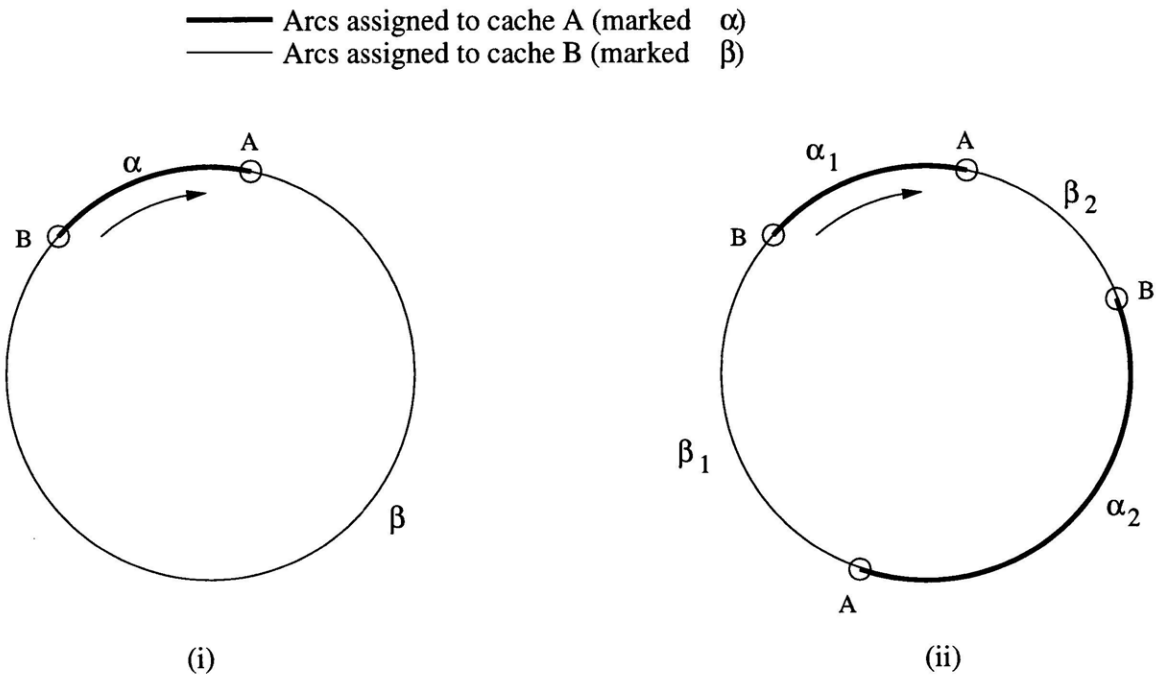

Figure 2-3: Making multiple copies of each cache

Each cache point is responsible for storing data items that fall in the arc between this point and the first cache point in counterclockwise direction from it. Each cache, is responsible for all the arcs assigned to its cache points. This figure demonstrates an example when adding an extra copy of cache A and B to the circle helps balance the responsibility arcs between the caches. (i) Here, with unlucky mapping of each cache point to the circle, cache A is responsible for a much smaller arc then cache B. (ii) With an additional copy of each cache the sum of the two responsibility arcs of A is closer to the sum of responsibility arcs of B.

Chapter 3

Consistent Hashing

This chapter states formal properties of consistent hashing, results that were first introduced in [5]. It continues with discussion of our research of implementation and experimental analysis of consistent hash functions.

3.1 Properties of

3.1.1 Previously Proven Bounds

Consistent hash functions can be shown to possess properties suitable for mapping documents to caches in a global web caching system. These properties include *monotonicity, low spread, low load, and balance. Monotonicity* implies consistency in mapping documents to caches despite fluctuations in the cache set. Spread refers to degree of duplication of each document. *Low spread* guarantees that despite disjoint views, each document will be spread to a small subset of caches. Load represents the total number of documents mapped to each cache. *Low load* says that each cache will be responsible for a proportionate fraction of documents. *Balance* property insures that for a given view, the items will be even distributed among the caches. The theoretical work described in [5, 7, 11] lists detailed analysis and proofs of each of these properties. Here, the formal definitions and the properties are only stated.

Let $U = \{V_1, V_2, \ldots, V_k\}$ be a set of views of the set of available physical caches,

B. The total number of distinct physical caches seen by any of the views is T. Each view is required to contain at least 1/t fraction of caches, so that each view contains at least T/t caches. Let I be the complete set of data items that are mapped to caches. Let $N > 1$ be a confidence factor. Each cache is replicated m times over the unit circle. For $m = \Omega(\log(T))$, the following properties hold:

1. Monotonicity: Let V_1 and V_2 be two views of live caches, such that $V_1 \subseteq V_2 \subseteq U$. Let $f_{v_j}(i)$ return a cache that item i is mapped to under view V_j. Then, $f_{v_2}(i) \in V_1$ implies that $f_{v_1}(i) = f_{v_2}(i)$. This property says, that as the caches are added to a view, all elements still mapped to one of the old caches were mapped to the same caches under the old view. Thus, mappings maintain consistency.

2. Low Spread: Since the views of the set of the available cache machines may be inconsistent among users, it is possible that a particular item will be mapped to many different physical caches by different views. The spread of an item i is the total number of caches that this item is mapped to by all of the views. *Low spread* property of consistent hash functions guaranteed that with high probability each item will be duplicated only in a small fraction of caches. Specifically, for any item $i \in I$,

$$\Pr[spread(U, i) = O(t \log(Nk))] > 1 - 1/N.$$

3. Low Load: Load of a cache is a measure of the total number of distinct items mapped to that cache by all of the views. Since the items are mapped randomly to the circle circumference, this measure can be thought of roughly as the total fraction of the circumference this cache is responsible for. Note, that for different views, a particular cache could be responsible for different arcs of the circle. The load of a cache is a union of all the responsibility arcs of that cache under all views. This means that as long as any view assigns certain items to a given cache, these items contribute to that cache's total load. Figure 3-1 demonstrates this point. Figure 3-1(i) shows responsibility arcs of cache A for a view containing only caches A and B, out of the total set, $\{A, B, C\}$. Figure 3-1(ii) shows responsibility arcs of cache A for a view containing caches A and C, excluding cache B. The total load of cache A, comprised

of the union of all its responsibility arcs under each view is indicated in figure 3-1(iii). The *low load* property of consistent hash functions limits the maximum load of any cache under all possible valid views. For any cache $b \in B$:

$$\Pr[load(b) = O((1 + I/T)t \log(Nmk))] > 1 - 1/N.$$

4. Balance: Balance property states that for each particular view, the data items are well balanced among the caches in that view. Each cache in a given view is responsible for only a small fraction of total load. Let V be a fixed view that contains $v = |V|$ physical caches. Let $m = q \log(v)$ for some constant q. (m is the number of copies, or point representations, of each physical cache, with each copy mapped randomly to the circle). Then for any cache $b \in V$:

$$\Pr[item\ i\ is\ assigned\ to\ b] = O(\frac{\log(Nv)}{v \log(v)}) + \frac{1}{N}$$

Balance property does not say anything directly about a general case, as it only fixates on one view. However, balance property for one view provides an intuition for a good balance among caches for all views. In addition, this result can be applied directly to a system that can maintain one consistent view or a small number of views. In systems where such situations are likely this property immediately implies a balanced distribution of load among caches.

3.1.2 Balancing with Multiple Cache Copies

Although it can be explained intuitively how multiple copies of each cache help reduce the probability of imbalance, this result is not obvious from the bounds proven above. More copies of each cache create more, but smaller responsibility arcs for each cache. The expected value of the sum of the responsibility arcs stays constant, regardless of the number of copies of each cache. However, greater number of responsibility arcs (i.e. cache copies) gives smaller variation in the sum of these arcs. This is the intuition gained from the Central Limit Theorem, but is not a hard proof.

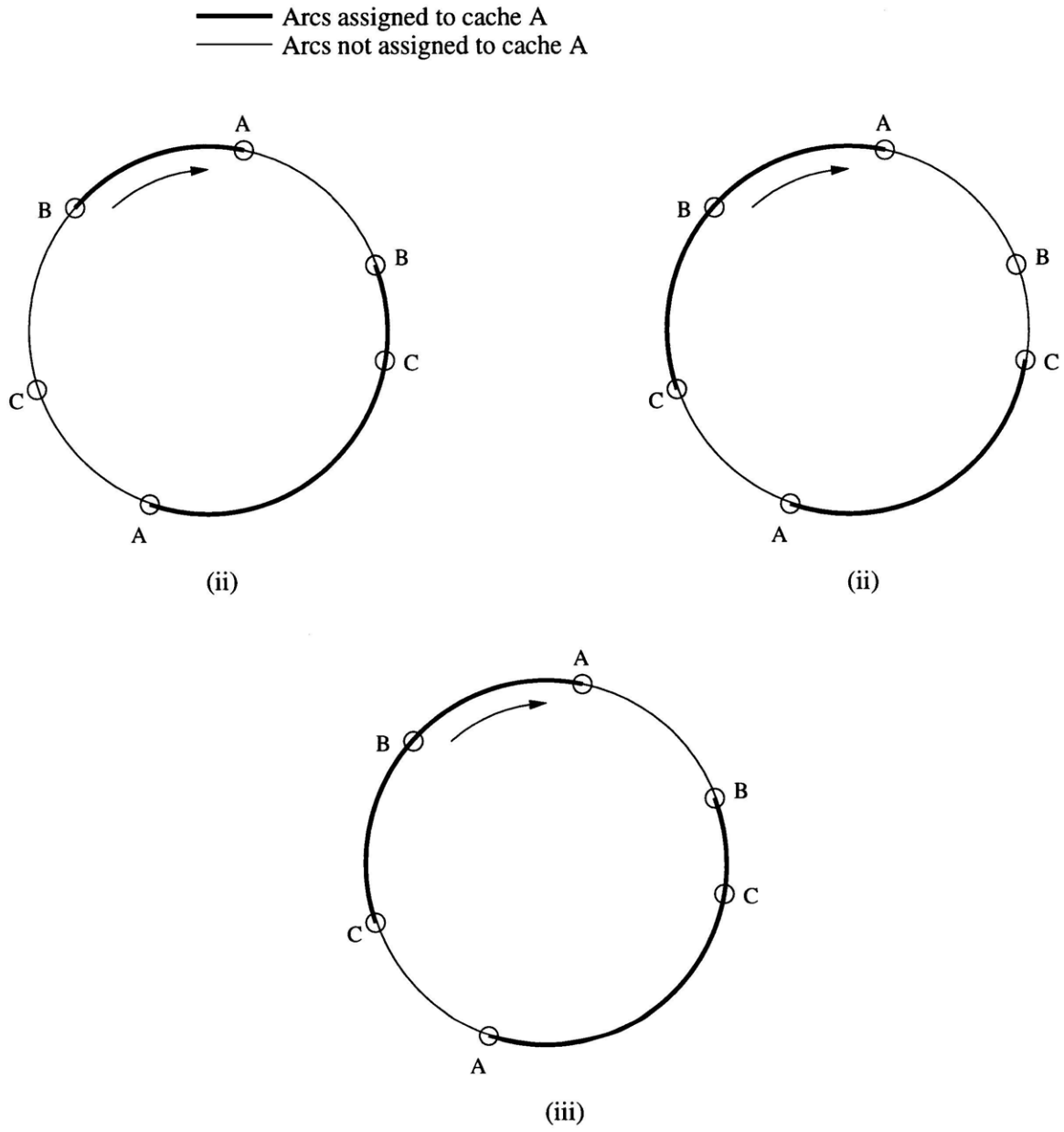

Figure 3-1: Load of cache A

Load of cache A corresponds to the union of all the responsibility arcs of cache A under all views. In this example two distinct views are possible: $\{A, B\}$ and $\{A, C\}$. Responsibility arcs of cache A under each view are shown in bold. (i) Shows responsibility arcs of cache A under view-1 that contains caches A and B, but not C. (ii) Shows responsibility arcs of cache A under view-2 that contains caches A and C, but not B. (iii) Shows total load of cache A, which is the union of its responsibility arcs under all existing views.

Bound on Balance

In case of the bound on *balance* that considers a specific view it can be shown more clearly how a greater number of copies of each cache improves the bound. Section 3.1.1 states that the probability of any item i being mapped to a specific cache b in a given view V, is at most $O(\frac{\log(N|V|)}{|V|\log(|V|)} + \frac{1}{N})$. where N is the confidence factor, and $|V|$ is number of caches in view V. Number of copies of each cache is taken to be $m = \Omega(\log(|V|))$. Lemmas 2.2.8 and 2.2.9 in [7] describe the analysis of this bound.

It is important to point out that this bound is actually composed of two components. Probability that an item i is assigned to a give cache is roughly represented by the fraction of the circle that the cache points (or copies) are responsible for, or the sum of responsibility arcs of this cache. The analysis in [7] that proves the bound on the sum of these arcs actually splits these arcs into two types, discretized arcs, and continuous arcs. Both types are represented in the bound. Discretized arcs are bound with $O(\frac{\log(N|V|)}{|V|\log(|V|)}) + \frac{1}{N}$, and continues arcs are bound with $O(1/V)$. The bound is the sum of these values. $\log(|V|)$ that appears in the denominator of the first fraction is actually exactly the measure of m that is taken to be $\Omega(\log(|V|))$. In fact the analysis in Lemma 2.2.8 can be carried out substituting m for $\log(|V|)$. So the actual bound will look as follows:

$$\Pr[\textit{item } i \textit{ is mapped to cache } b] = O(\frac{\log(N|V|)}{|V|m}) + O(\frac{1}{|V|}) + \frac{1}{N})$$

Notice that m, the number of caches, appears in the denominator of the first fraction and helps improve the bound as it increases. If N is taken to be $\Omega(|V|)$, and $m = \Omega(\log(N|V|))$, the bound on probability is reduced to $O(1/|V|)$.

Clarification of the *balance* bound demonstrates that the increase in the the number of copies helps to balance load among the caches in one view. When m is sufficiently large, the load is fully balanced with $O(1/|V|)$ bound.

Bound on Load

Just like with the *balance* property, bound on the *load* property should also reflect that the bound improves as the number of copies of each cache is increased. Although the *balance* bound provides the intuition, the *load* bound is much more difficult to prove. In fact, the bound for load stated in section 3.1.1 becomes looser as m increases. It is also possible to show a bound on *load* that does not depend on m altogether. In [7], through the analysis of π-hash functions, which are shown to be equivalent to consistent hash functions, the bound on load can be shown to be $O((1 + \frac{|I|}{T})t \log(N|I|k))$. This bound, however, does not reflect any dependence on m. It is preferable to show the same dependence on m as in the bound on *balance*, where one term decreases with m and one term is independent of m. The most likely approach that would result in such a bound is the analysis used for the *balance* property.

This analysis is conducted as follows. Responsibility arcs of a cache are discretized. It is then shown, that for a given distribution of discrete lengths of m arcs, their sum is bounded with high probability. This probability is then summed over all possible distributions, and the summation of these probabilities is still shown to be small.

The difficult step in this analysis is bounding a specific distribution of lengths of m arcs. For the *balance* property analysis, where a single view was considered, it was sufficient to show that with high probability there is one cache point that falls inside any of the arcs in the distribution. A cache point occurring anywhere, would bound the arc where it occurred, thus bounding the total lengths of the arcs. In the *load* property analysis, however, it is necessary to show that at least one point from every view falls into one of the arcs. Only then, would that arc (and therefore the distribution) be bounded under all possible views. Although the exact form for that bound has not yet been found, all the research work done so far indicates that the bound on a distribution become tighter with increase in m.

3.2 Implementation

Implementation of a consistent hash function involves two components. First component is choosing a way to map caches and data items to the circumference of a circle randomly and independently. Second component is maintaining a view of available caches. Each component is discussed in order.

3.2.1 Range

In order to map caches and data items efficiently to the unit-circumference circle, the circumference of this circle is discretized. Specifically, a large prime, P is chosen to represent the number of discrete intervals in the circumference of a circle. All points on the circle are then restricted to integer values from 0 to P-1. The values of the points themselves are not significant as long as all the points (representing caches and items) appear in the same order as they would on a continues circle. Same order guarantees same mappings of items to caches. P is chosen in such a way that all the documents and caches can be comfortably represented on the interval. Using terminology of section 3.1.1, number of points that needs to be mapped to the circle is: $m|B| + |I|$, where I is the set of all data items, B is the set of all physical caches, and m is the number of points (or replications) for each physical cache on the circle. Thus, P is chosen to be of size $O(m|B| + |I|)$. The advantage of discretizing the circumference is that only $\log(P) = \mathrm{O}(\log(m|B| + |I|))$ bits, is required to represent points on the circle.

3.2.2 Mapping Points

Although consistent hashing theory requires for all the items to be mapped randomly to the interval (or circle), in practice truly random and independent mappings are impossible. In his Master's Thesis [7], Daniel Lewin defines an independence parameter, K, that simplifies the random mapping requirement. A function is K-way independent if it maps any K elements from domain to range independently. Theorem 2.2.12 in [7] shows that if K-way independent functions are used to map caches and items to

the circle, with K = $\Omega(t \log(NTk))$, then bounds outlined in section 3.1.1 still hold.

One practical way to implement a K-way independent function is to use a K-degree polynomial with K randomly chosen coefficients. In our implementation we select coefficients in the interval [0,P-1] using UNIX pseudo-random number generation routines. Values of caches and items are plugged into these polynomials and are evaluated modulo P to determine their values on the interval.

Each cache point on the interval is responsible for the subinterval for which it is the smallest upper bound. Thus, all URLs mapped to that subinterval are assigned to this cache. As suggested by the bounds on consistent hashing in section 3.1.1, it is highly unlikely for any cache to be assigned a large fraction of items. In an unlucky scenario, however, some sub-interval that corresponds to one cache could be disproportionately large, and therefore contain more URLs. In order to reduce the likelihood of such an event, we want to deal with smaller sub-intervals. The solution is to create multiple copies of each cache and map each copy randomly to the interval [0, P-1]. As a result, each cache is represented by a multiple number of points on the interval and is responsible for many smaller subintervals instead of a larger one. The improvement due to a larger number of copies of each cache is discussed in 3.1.2.

Since each cache is represented with m points on the circumference we require m, K-way independent functions to map each copy of a given cache. So to map all the cache and data points to the interval, we generate m+1 K-degree polynomials, $\{Poly_0, Poly_1, \ldots, Poly_m\}$, with pseudo-random coefficients in the range [0, P-1]. The first polynomial is used to map data items. The other m polynomials are used to map m cache copies, such that $Poly_i$ is used to map copy i of each cache.

It should be noted that in order for the users who know about the same set of caches to have identical mappings between URLs and caches, they must use polynomials with the same coefficients to create their Cache View and to hash URLs. Therefore, exact polynomial coefficients must be included in the distribution of the hash function. We distribute coefficients via a binary file.

3.2.3 View Representation

In order to be able to actually hash URLs to the set of available caches, we must find some suitable representation of the cache view itself. Cache view must contain all the copies of the available caches with their respective values on the [0, P-1] interval. Representation of the cache view must also provide an efficient way to hash URLs to caches. Hashing a URL means finding a cache point whose value is the smallest upper bound on the mapped value of a URL and returning the physical cache associated with that point. One suitable data structure for this representation is a binary tree. Nodes in the binary tree will correspond to the cache points that will be arranged according to their values on the [0, P-1] interval. For any node A in the tree, all the non-leaf nodes in its left subtree will have values smaller than *value(A)*, and all the non-leaf nodes in its right subtree will have values greater than *value(A)*. The tree has a total of $m|B|$ nodes for all the points associated with caches. If the tree is kept balanced, the height of the tree will be $\log(m|B|)$, and it will take at most that many comparisons to hash an item. The tree representation of user's view allows the following operations:

Insert_Node(View, val, C)

Inserts a node corresponding to cache C, with given value: val, into the View

Remove_Nodes(View, C)

Removes all the nodes corresponding to cache, C, out of the View.

Find_NextNode(View, val)

Returns the cache that has a node in the View with the smallest value exceeding val. (In order for consistent hashing properties to hold strictly the sequence of nodes must correspond to the circular circumference. Therefore, if the value given to Find_NextNode exceeds the highest-value node in the tree, the lowest value node is returned).

Balance_Tree(View)

Balances binary tree representation of the View.

3.2.4 Primitives

Our implementation of Consistent Hashing provides the following primitives to maintain a view of caches and hash URLs:

Generate_Polynomials(P, K, m):

For i = 0 to m

$Poly_i$ = **Generate_Poly(P, K)**

Add_Cache(View, C):

For i = 1 to m

val = $Poly_i$**(C)**

Insert_Node(View, val, C)

Balance_Tree(View)

Remove_Cache(View, C):

Remove_Nodes(V, C)

Balance_Tree(View)

Hash_Item(View, i):

val = $Poly_0$**(i)**

node = Find_NextNode(View, val)

return Get_Cache(node)

3.2.5 Evaluating URLs and Caches

Before K-way independent polynomials are used to map caches and items onto the interval, it is important to assign numerical values to URLs and caches. Caches participating in the system can be easily assigned unique IDs by the system administrators. In our scheme we actually use numerical values of their IP addresses to plug

into the polynomials.

It is slightly more difficult to assign numerical values to string representations of URLs. Here, something simple can be performed, such as adding up string characters. However, it is preferable to preserve unique value of each URL string in its numerical representation for more random distribution of URLs across the interval. A better scheme then, is to multiply each character by its position in the URL string and take the product of these multiplications modulo some prime. The resulting value is then plugged into the first of the m+1 polynomials to be mapped to [0, P-1] interval.

3.3 Useful Statistics

We list a few brief statistics that demonstrate the usefulness of Consistent Hashing. First, it is important to realize that hashing consistently is a relatively fast operation. For our testing environment, we have set up a Cache View using 100 virtual caches, creating 1000 copies of each cache on interval [0, P-1]. We used 10-degree polynomials for mapping. We timed the dynamic step of Consistent Hashing on a PentiumII 266MHz chip. (The dynamic step includes URL string evaluation, polynomial evaluation, and traversing a binary tree). On average each call to the hash function took 20 microseconds. This is about 0.1% of the total time necessary to transmit a 20KB file from a local cache over 10Mbit Ethernet. The value of 20 microseconds can be significantly reduced if we use a balanced binary tree as the underlying Cache View representation, as opposed to a regular binary tree currently used in 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 requests through our hash function with a varying number of various caches to see how well it would distribute files among the caches.

Number of caches	Mean entries in cache	Std Dev	Deviation 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.

As described in section 3.1.1, consistent hashing possesses very convenient properties for a global web caching system. The rest of this chapter has demonstrated the triviality of the actual implementation of consistent hashing, and listed some of the usage results. Next chapter will describe the web caching system itself and how it uses consistent hashing.

Chapter 4

Our System

In order to demonstrate the practical advantage of Consistent Hashing we implemented a web caching system, called the Cache Resolver that uses our hashing algorithm. The system consists of three major components: the actual cache machines for storing the content, users' browsers that direct requests toward virtual names of the caches, and the domain name servers (also known as resolution units) that use consistent hashing to resolve virtual names requested by the users into specific physical addresses of the cache machines. Figure 4-1 depicts the relationship between various parts of the system. We describe each component in order.

4.1 Caches

Our proxy caches use a distribution of the Squid cache package. We made no modifications to the standard Squid distribution. The 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 LRU replacement strategy.

We run small programs, called "Squid monitors", on the same physical machines as the caches themselves. These monitors communicate with the Squid process using the Object Cache Protocol and monitor the status (dead or alive) and load of the Squid process. When the Squid is alive, the load represents the byte transfer rate at

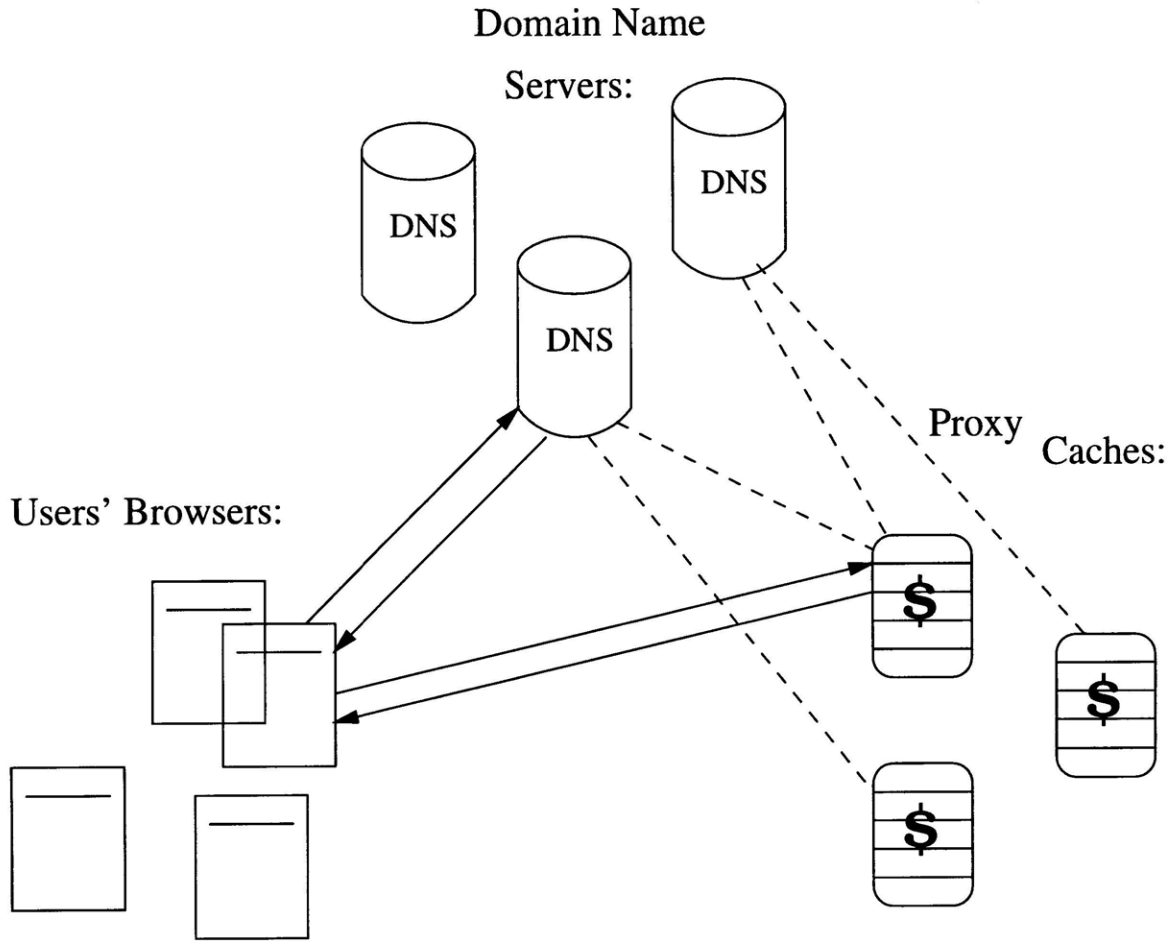

Figure 4-1: System Components

Browsers make DNS requests to resolve virtual names of proxy caches in terms of concrete IP addresses. Then, they contact a proxy cache with a given IP. Asynchronously from users' requests, DNS units monitor proxy caches as discussed in section 4.3.1. (This monitoring is indicated with dashed lines.) DNS units run consistent hashing to map all the virtual names to IP addresses of only the live cache machines.

which the cache served web requests over the last minute interval of service. Squid monitors are ready to reply with the status and load information when queried by other units in the system.

4.2 Users' browsers

We take advantage of the autoconfiguration option for proxy cache configuration offered by the most commonly used browsers (Netscape Navigator 2.0x and above and Internet Explorer 3.01 and above). In such browsers, 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. A browser contacts proxy caches in the order listed until it contact one that can respond with the data.

We wrote an example of such an autoconfiguration function that users of the web caching system could down-load and install in their browsers. This function takes requested URLs as input and performs a modulo hash (the hash function here is not related to Consistent Hashing that takes place elsewhere in the system) and returns a list of virtual names of the cache machines. To resolve virtual names into actual IP addresses of the caches, a browser will have to contact the DNS system.
To generate virtual names, we use a function of the form

$$f(< url >) = a* < url > +b \pmod{p}.$$

Here url is the numerical value of the URL string, a and b are a pair of small but relatively prime integers, and p is a prime number larger than a and b (here, we chose 997). The range of possible return values of this hash function is from 0 to 996. We use a few pairs (a, b) in order to return a list of values. These numeric values returned by the hash function are included in the virtual names (e.g., A123.ProxyCache.com, where 123 is the numeric value corresponding to a hash of a URL string).

We return a list of five names instead of just one to make up for the problem

of the "broken" browsers. In our testing, we discovered that some versions of the browsers under some operating systems will not reinvoke DNS even if they contain an outdated resolution of a virtual name (i.e. a virtual name resolves to an IP address that no longer responds). By returning a list, we give the "broken" browsers several chances to achieve a correct resolution of a physical name as they try each element on the list in order. The last value on the list that is returned is "DIRECT" which is understood by the browser as an instruction to connect directly to the content server if all the other names fail.

4.3 Domain Name Servers

4.3.1 Functionality

The primary function of our DNS system is to resolve the virtual names generated by the users' Java Script function to the actual physical IP addresses of the caches. We use a number of DNS servers each running a copy of unmodified BIND 8.0 distribution. (BIND is an implementation of DNS protocol.) In addition to BIND, each DNS machine runs one of our programs, called "dnshelper". The "dnshelper" program periodically polls all of the caches, by communicating with the Squid monitors (that run on the cache machines), to determine the dead/alive status of each machine. It then uses consistent hashing to map all of the virtual names to physical addresses of only the live machines. (As noted in section 4.2 our virtual name space consists of 997 possible names.) If any of the mappings change, it writes the new mappings to a "records" file and signals BIND to update itself by reloading that file. BIND updates itself "dynamically" which allows it to respond to DNS queries even during the update stage. It is important to note that all of the "dnshelper" activity and the BIND updates take place independently of user requests. When a user request arrives, DNS immediately replies with the mappings that it currently contains. The DNS units represent the key component of our system since these units ultimately decide which physical cache stores each resource.

4.3.2 Efficient Cache Use with Consistent Hashing

Properties of consistent hashing allow us to use the caches very effectively. First, we eliminate duplication of resources among caches since each DNS server with the same view of caches will compute the same mapping from virtual names to IPs. Thus, each cache is only responsible for storing a small fraction of content that it serves to all of the users who request this content. As a result, the number of misses is reduced, since only one cache responsible for the content can miss only once, and fewer users see delays associated with cache misses. Moreover, since the total set of documents stored by each of our caches is small, our caches are more likely to keep the popular content in RAM and thus serve it faster. In a large web caching system it is possible that the views of live caches will differ slightly from one DNS server to another perhaps due to some lost connections over congested Internet links. In addition, if the set of caches is large, that view is likely to change with the shutdown of some servers and other servers (new or old) coming back up. Consistent Hashing guarantees that even when the view is changing, only a small fraction of mappings from virtual names to IPs will change, and the duplication of resources could be contained to a very small fraction of caches.

The load property of Consistent Hashing guarantees that the number of virtual names assigned to each cache will be evenly distributed among the caches. This property helps to balance load and prevent any one cache from serving too much content which would make the service time for that content longer. We describe more advanced load balancing strategies with consistent hashing in section 6.2.

4.3.3 Advantages

By placing a significant portion of functionality of our system into the DNS units, we simplify modifications that need to be done to users' browsers and also allow for a more flexible web caching system. Since consistent hashing and communication protocols with proxy caches are carried out asynchronously by the DNS units (i.e. outside user's request loop), the processing time for these operations do not con-

tribute to request latencies. We do not need to place this functionality inside the users' browsers which would require significant modifications to the browser code and would render the system impractical for most users. Even if the browser software could easily be modified to carry out communication protocols with the cache machines, network congestion at the cache machines would simply build up from all the browsers running these protocols, and would slow down service. Traffic is insignificant when the protocol runs only by the designated DNS machines. In addition, since it is much easier for us to maintain control over the DNS units than the users' browsers, the system can behave much more flexibly. If we ever decide to augment consistent hashing algorithms with some heuristics or to make changes to communication protocols, such changes would be trivial to perform on our own DNS machines. Thus, the DNS units play a key role in our web caching scheme by providing for a simple use and smooth evolution of our system.

4.3.4 Disadvantages

Of course, there are also disadvantages associated with adding additional components to any system. Adding DNS servers creates another point of possible failure in our system. However, our system is designed in such a way that no single failing DNS machine could bring down the system. As long as some DNS machines continue to respond to DNS queries, the system will continue to function properly, without even slowing down. In this way, our DNS system is strikingly different from the directories in the Crispy Squid or individual caches in the Harvest System that could bring down the effectiveness of the web caching system through failure. The fault tolerance aspects of our system are further discussed in section 6.3. Another downside of designating special DNS units to handle some of the functionality is the additional DNS lookup. Browsers need to resolve virtual cache names into IP addresses before they can contact the caches. However, we believe that DNS lookups should be fairly quick on average. DNS lookups will not actually travel directly from the browsers to our DNS machines, but rather will travel through users' Internet Service Provider's (ISP) DNS machines that will cache responses of our DNS machines. Responses from

our DNS machines will be cached upto the Time-To-Live (TTL) value specified by our DNS machines. All of the ISP clients (except for the first few) that will request a given virtual name within the TTL period will have very fast DNS lookups as ISP's own DNS will reply. So, on average, virtual name resolutions should not contribute significantly to user latencies.

4.4 Monitor

We also implemented a tool to monitor our system. This tool, from time to time, communicates with the "dnshelper" programs on some of the DNS machines, and generates very low traffic. From a "dnshelper" it could retrieve all of the information about the status and load of all the cache machines (since they are being monitored by the "dnshelper"). The tool displays status and load information graphically on the screen. This monitor could be used by an administrator of the web caching system.

4.5 Overview

The main innovation in our cache system design is allowing users to contact directly the caches that are responsible for storing desired data. In this way, we achieve a high hit rate and fast access times as no inter-cache communication takes place during requests. All heavy-bandwidth inter-cache communication for exchange of URL pointers and actual data is avoided. A very low-bandwidth communication is conducted by the DNS servers that simply poll the caches for status and load information, and even that is performed asynchronously, not contributing to user latencies. Consistent Hashing allows all of the DNS servers to agree on the mappings from virtual names to IP addresses. This agreement provides for more efficient use of caches and reduces latencies for the users. Additional side effects, including high fault tolerance of the system, are further described in chapter 6.

Chapter 5

Comparison with Other Systems

We believe that our design based on consistent hashing algorithms warrants better performance than the existing web caching systems we studied. We compare our Cache Resolver against two such systems: the Harvest Project and the Crispy Squid. We demonstrate results that corroborate our hypothesis. (Results listed here, first appeared in [6]).

5.1 Overview of Systems

5.1.1 Harvest System

The Harvest System [2] arranges its caches in a hierarchical tree. First, a user contacts his/her primary cache. On a miss, that cache forwards user's request to its siblings and the parent in the cache tree. If one of these caches has the data, it is transferred back to the user's primary cache, stored there, and returned to the user. Otherwise, the request propagates further up the tree until finally the root node of the tree gets the content from the origin content server.

Harvest System operated under the assumption that on average it is more expensive to fetch the data from the origin server than traverse a hierarchical subtree of caches. However, in case of a miss in the primary cache, there is a large cost associated

with inter-cache communication between tree nodes, and inter-cache data transfer. In addition, since data will travel from cache to cache, it will become duplicated in a large number of caches that will cause very inefficient space management. Poor space management may cause many disk swaps and even forcing some data out of caches. Disk swaps translate into slower data retrievals, and frequent data replacement results in lower hit rate.

5.1.2 Crispy Squid

The Crispy Squid project [3] came as an improvement to the Harvest System. The analysis conducted by the Crispy Squid developers has shown that inter-cache communication of the Harvest System can be extremely expensive, especially if it is propagated to higher levels of the hierarchy. Instead of a tree, Crispy Squid introduces an additional system unit, the directory, that helps locate missing data. The directory maintains a lookup table with mappings of URLs to caches that store these files. On a miss, a primary cache contacts the directory, which replies with the location of the cache that has the data or states that no cache has it. The primary cache then, retrieves the data from the suggested cache. The directory can be maintained asynchronously without directly contributing to user request latencies. Crispy Squid significantly reduces inter-cache communication; on a miss only one query to the directory is made, followed by data transfer from the suggested cache.

However, Crispy Squid still manages space very inefficiently as there are no determined assignments of data to caches. Any piece of data can still be requested from any primary cache, which will cause that cache to store a copy of that data upon request completion.

In addition, the directory location presents a non-trivial problem. A centralized directory located on one machine will be far from most caches in a global cache system. From most locations, a query to such a directory will be slow, as it will travel a long network distance, possibly over many congested links. Crispy Squid allows directories to be replicated. However, as mentioned in [3] each replica of the directory would require a lot of resources. Moreover, communication will become more

complex as each cache will need to send updates to each of the replicas. Crispy Squid allows yet another, compromising configuration of a distributed partial directory. Partial directories are limited in size and store only some of the pointers. The latter configuration solves the problem of proximity to a directory, but the hit rate suffers as only some of the URL pointers are stored.

5.1.3 Our System

Cache Resolver solves many of Harvest's and Crispy Squid's problems. By assigning each piece of data to a specific cache or a small subset of caches massive data duplication is avoided. In addition, Cache Resolver entails no inter-cache communication; on a miss the content provider is contacted by the cache. The logic is simple: if the cache is contacted it means it is the one that's responsible for storing this data so it's worth fetching a copy to store locally. Why not fetch from other caches? The *low spread* property of Consistent Hashing guarantees that only one cache or a small subset of caches will contain a specific data item. So on a cache miss, it is likely that no other cache has the data, or else very few caches have it, and it is expensive to search for the few that do.

5.2 Testing

5.2.1 Network Setup

To test these systems, we used a network of seven machines connected to one 100Mbit switch. Three of these machines ran Squid, the proxy cache program. One machine was designated as the origin web server and was placed on a 10Mb link in order to make data transfer with the web server more costly. Another machine ran a copy of BIND and was the designated domain name server that used Consistent Hashing to resolve virtual names to IP addresses of the three caches. One more machine was used to run the test driver. The seventh machine was used for some additional cache

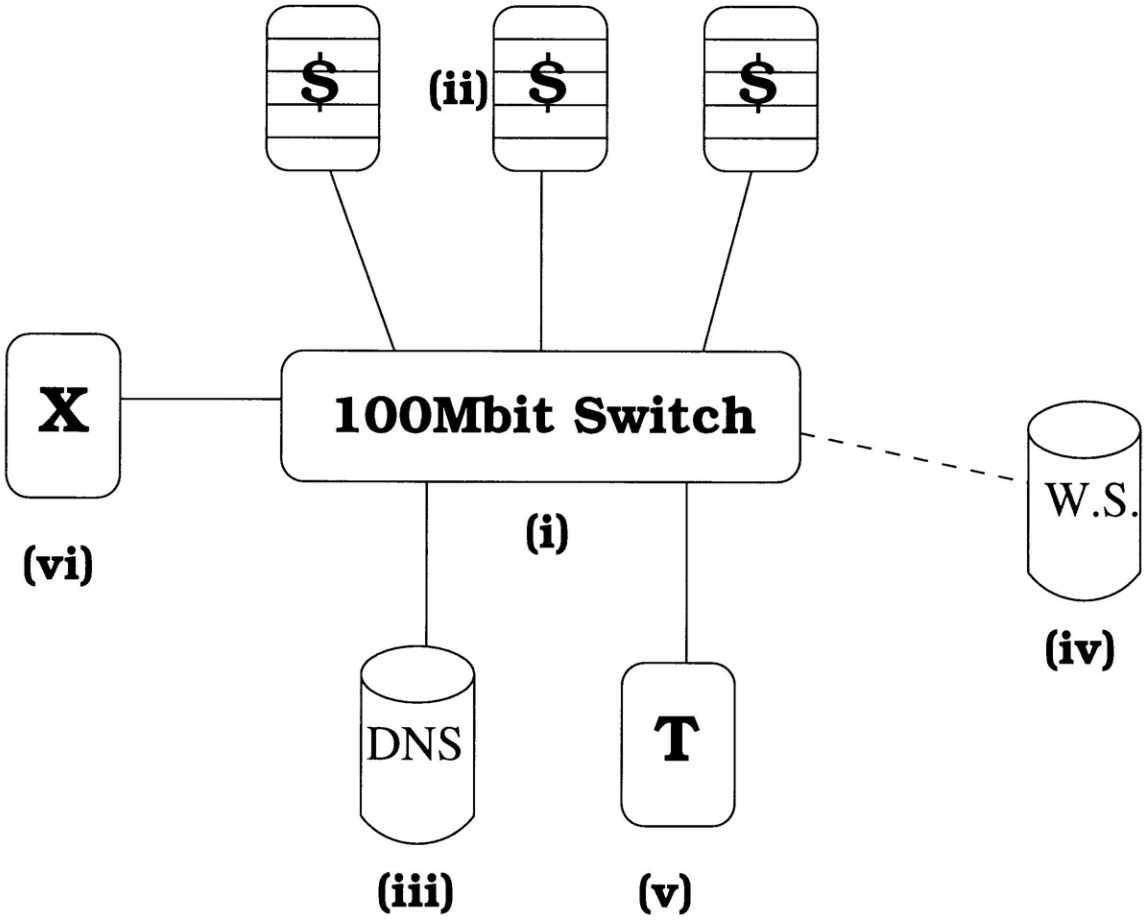

Figure 5-1: Test Network Setup
The test network consists of several components. (i) 100 Mbit Ethernet network switch, (ii) 3 proxy caches, (iii) DNS unit, (iv) web server hosting the content(connected to the switch over a slower, 10Mbit link, represented with the dashed line), (v) test driver running Surge, and (vi) an extra machine used in some of the tests

system configurations. Figure 5-1 demonstrates this setup.

5.2.2 Test Driver - Surge

For the test driver, we used Surge [1], a web load generating tool developed at Boston University. Prior to the test, Surge generates a web server database. It also generates a sequence in which files from the database are to be requested. Each file may be requested a variable number of times. The designers of Surge were among the first researchers to realize that sizes of web requests form a Pareto distribution, a heavy

tailed decaying function. In order to generate the database and the query sequence, Surge takes in two parameters: number of files and the total number of requests for the most popular file. These parameters define the Pareto distribution of the web requests. Surge also differentiates among image files, plain HTML files, and HTML files with embedded objects. Surge developers found Pareto distributions for these type of files to differ slightly and they account for these distributions in the model. Surge makes requests for objects rather than files. Each object may result in requests for many additional files to simulate embedded objects. The detailed mathematical distribution model used by Surge to simulate web server request sequence is described in [1].

For our testing we generated a request sequence using 1500 distinct objects, with the most popular one requested 5000 times. Using this parameters, Surge generated a 34MB database that we copied to our web-server.

To simulate user-equivalents, Surge is run with a specified number of clients (processes), and threads per clients. The total number of threads represents the user-equivalents. During one Surge test, all of the threads together run through the global object request sequence. Each thread reads the next object to be requested out of the globally available sequence. It requests all the files associated with a given web object, and then sleeps for a specified amount of time before reading the next object. Sleep time is also chosen from an upper-bounded Pareto distribution. Surge program stops when the global request sequence is exhausted. For our tests, we always run Surge with 3 clients, 75 threads each.

5.2.3 Modifications

We modified Surge to run in two distinct modes: *Common Mode* and *Cache Resolver Mode*. Common Mode represents a common cache setup where users always contact a primary local cache first. When a miss occurs, that cache either fetches the data from another cache or from the origin content server. In Common Mode, each of the three Surge clients is assigned one of the three proxy caches as its primary cache. Those proxy caches run various protocols, based on the configuration we specify, to

fetch the data for their clients. (A client in Surge runs many threads and represents a whole set of users from one region.)

Cache Resolver Mode is designed to test our system. In this mode, each of the three clients execute a simple hash function similar to the autoconfiguration function that would be utilized by our users' browsers described in section 4.2. That function takes the URL (or object) requested as input and returns a set of virtual names. Surge clients proceed to resolve the names in that set to IP addresses of the three proxy caches through our DNS unit.

5.2.4 Results

We ran tests to compare our system, the Cache Resolver, with the Common Mode under various proxy cache configurations. We then analyzed proxy cache logs to compare miss rates from different test runs. We expected to see lower miss rate for the Cache Resolver for a couple of reasons. First, the Cache Resolver mode, assigns each piece of data to a particular cache, so the hit rate for each file should be higher. Also, we vary cache sizes between tests from 9MB to 36MB per cache. Since the total database we are testing has a size of 34MB, caches with smaller sizes will have higher miss rate because data could be forced out of caches with LRU. With smaller size caches, we expect the Cache Resolver to behave significantly better than other systems. Since other systems duplicate data among caches, they will cause a lot more data to be forced out of smaller size caches. 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 Mbytes for different tests. The largest capacity tested, 36MB, exceeded the total web server database of 34MB, and therefor could be completely stored in a single cache.

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, figure 5-2 shows higher miss rate displayed by the Common Mode than the Cache Resolver Mode. The difference in the miss rates is even higher with

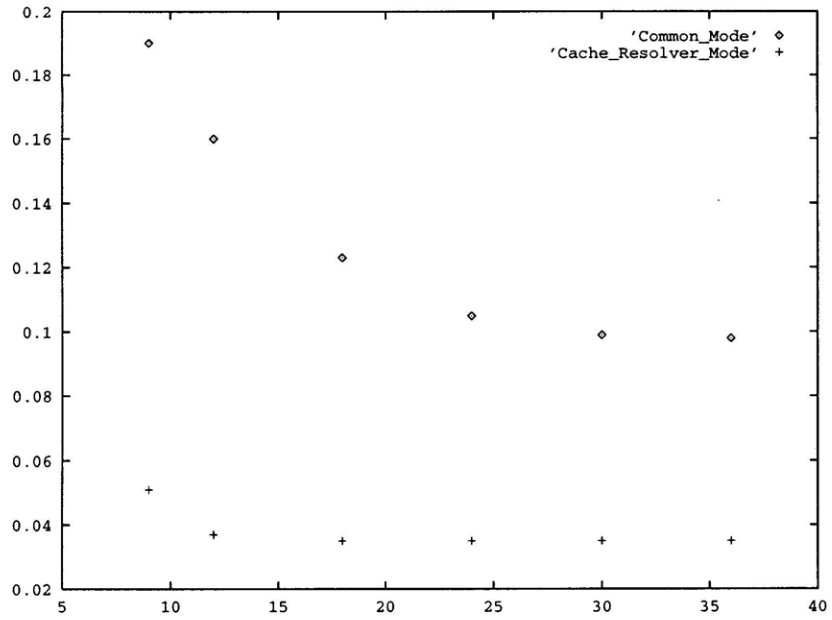

Figure 5-2: Cache Miss rates of Common Mode vs Cache Resolver Mode
X-axis represents cache sizes in MBytes of all the caches used in the test and Y-axis is the miss rate.

smaller cache capacities, where the data duplication of the Common Mode has even worse effect.

The average file size in our database is 20KB. Given the difference in miss rates and the fact that the link to the web server is 10 times slower (10Mbits per sec), we can perform some trivial calculations to compute the difference in latencies that the users of each cache system experience.

Figure 5-3 shows that average users of the Common Cache Mode system observe latencies of 1 to 2.2 milliseconds higher on every request than the users of the Cache Resolver Mode. In reality, these differences will be even greater since longer queues will form at the caches of the slower system. Since the users of the Common Mode system will have to compete with more users for service, their request latencies will be even longer.

We also used our setup to test three other common cache system configurations for the Common Mode. 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

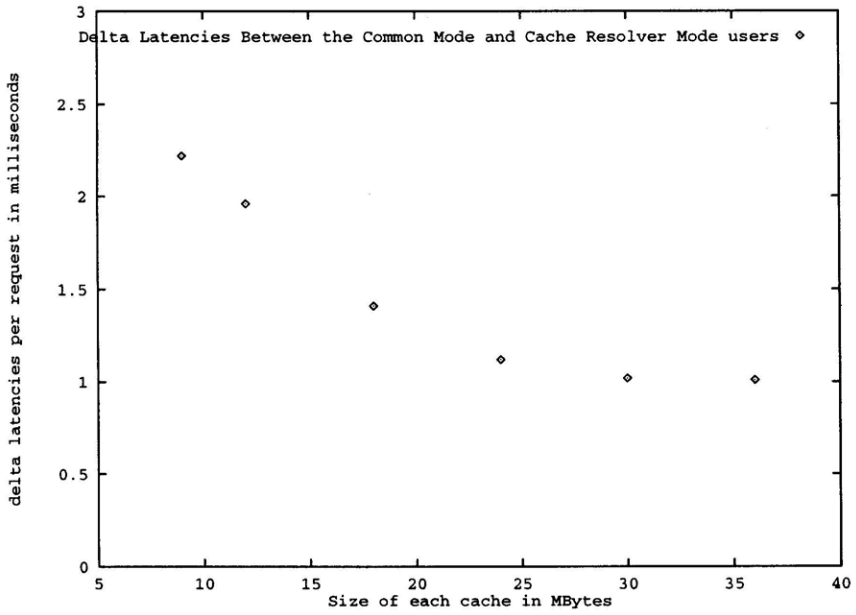

Figure 5-3: Difference in Latencies

Difference in latencies per request observed by the user of Common Mode cache and that of the Cache Resolver. The result is expressed in milliseconds

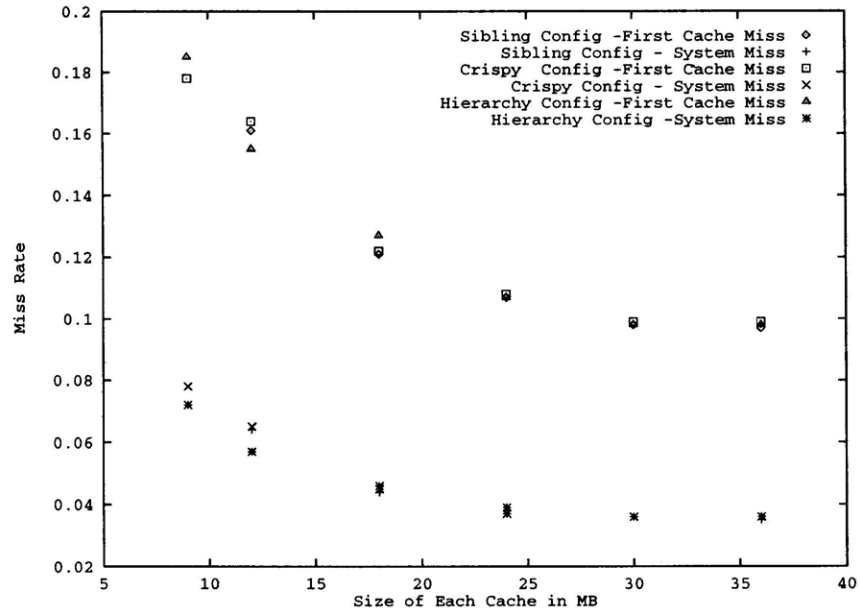

Figure 5-4: Miss rates of three additional configurations

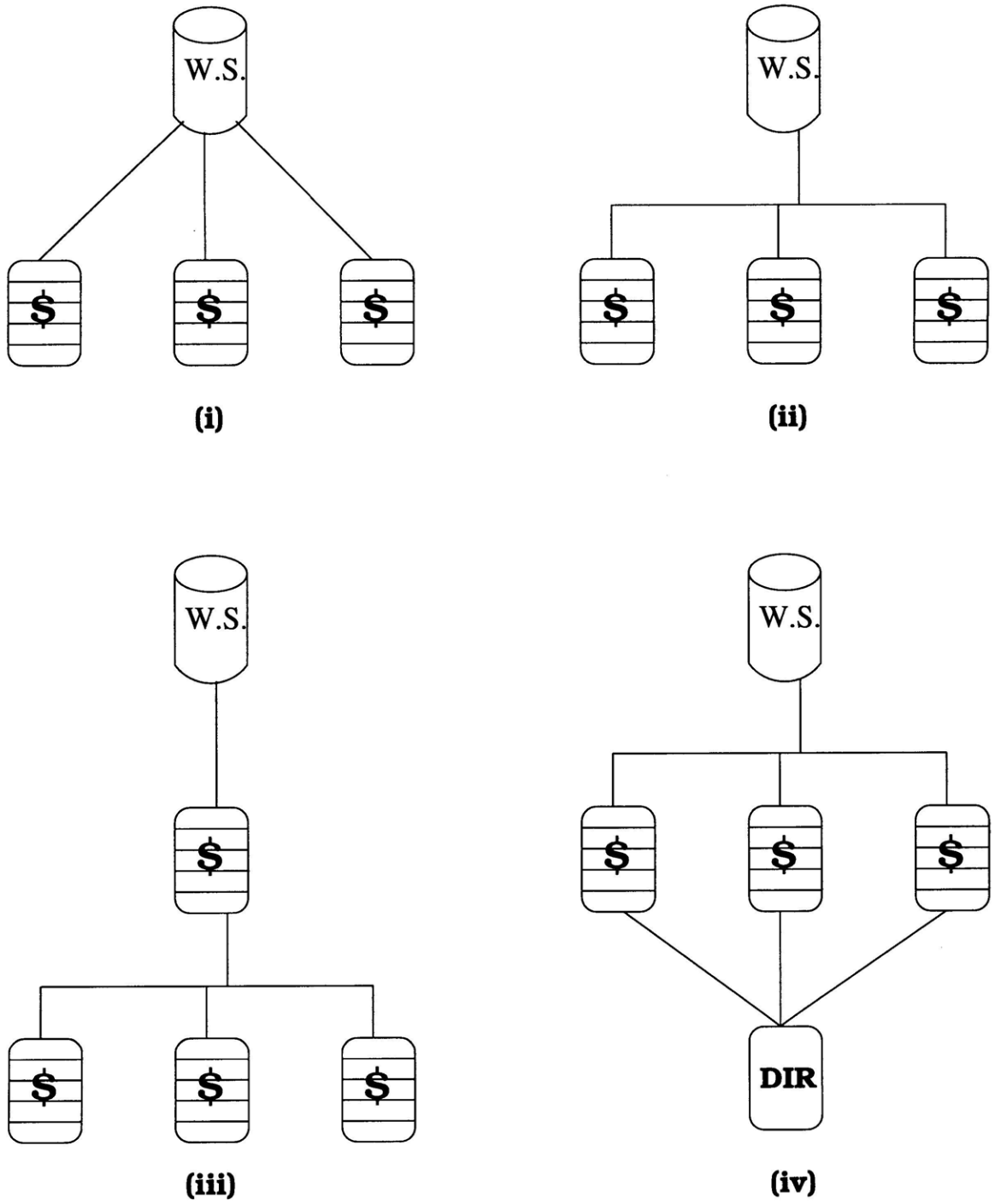

Figure 5-5: Various Cache System Configurations
(i) Regular setup with each of the three caches contacting the origin web server on a miss. (ii) Sibling Configuration: on a miss each cache sends a multicast request to its siblings, before resorting to the web server. (iii) Harvest hierarchical setup of 3 siblings and a parent. If neither the siblings nor the parent has the data, parent contacts the origin web server and returns the data to the requesting cache. (iv) Crispy Squid configuration. On a miss each cache asks the directory where the content could be found and contacts that cache. The directory is updated asynchronously, outside user request loop.

to the siblings in the style of the Harvest approach. Finally, we tested a Crispy Squid configuration where a central asynchronous directory was set up to be queried by all the caches on a miss. All of these setups are shown in figure 5-5. For each system, we measured the primary cache miss rate (the miss rate at the first cache queried by users) and system miss rate (when no cache in the system has the data as determined by the inter-cache communication protocol) Figure 5-4 shows that the first cache miss rate for all three configurations resembles the miss rate of the Common Cache Mode displayed in Figure 5-2, and the system miss rate for all of the configurations is almost as good as the Cache Resolver Mode's miss rate. (The system miss rate is still not quite as low as the Cache Resolver's miss rate because of data duplication.) For these three systems, there is a penalty associated with the first 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. In our system, 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 are avoided.

Chapter 6

Side Effects

In this section, we describe additional features of our web caching system.

6.1 Locality

In addition to efficient cache management, 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 user's geographical region inside the JavaScript function in the users' browsers. The JavaScript function can be made customizable. When users down-load it, they are given a choice of regions. 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 machines serve all of the users. The bottom-layer DNS machines as well as the cache machines correspond to specific regions. The top layer DNS servers resolve for the part of the name that contains geographical information (e.g. "ProxyCache3") 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 resolve for the next

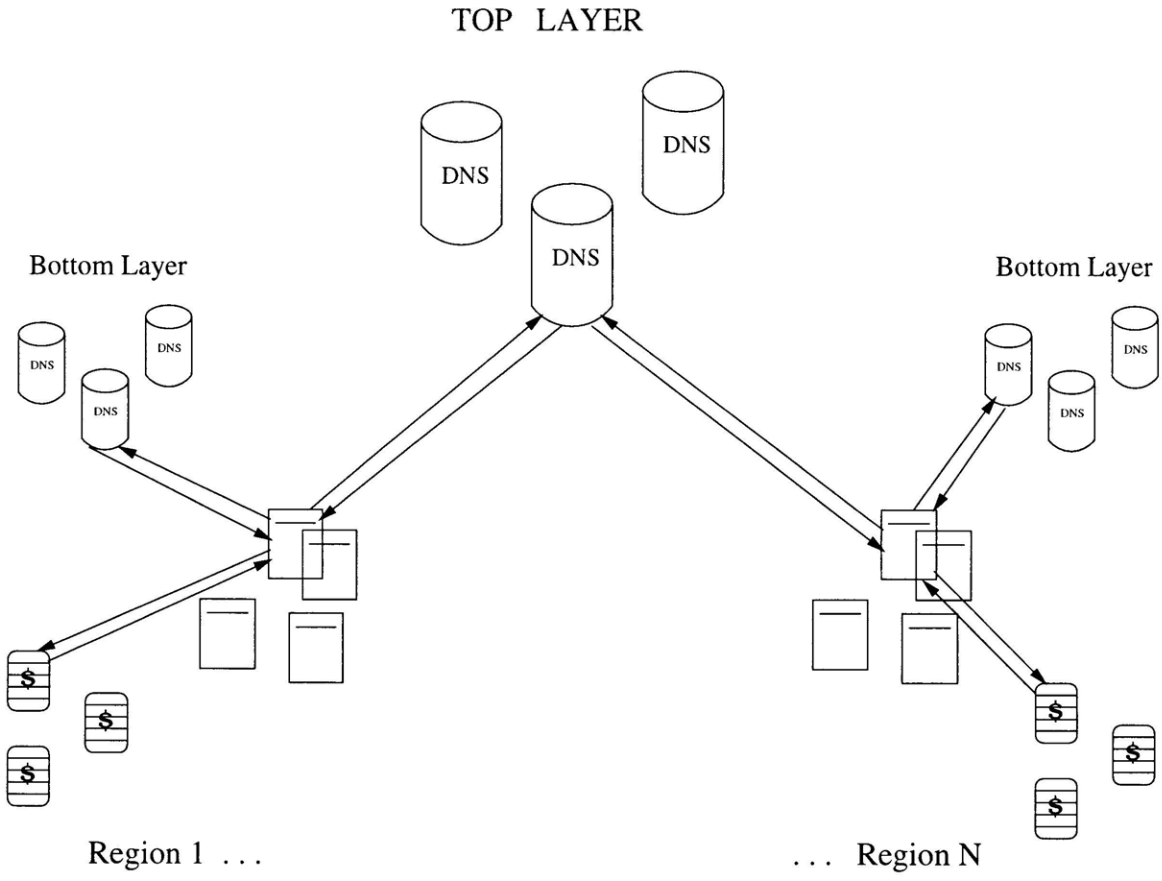

Figure 6-1: Splitting the System into Locality Regions
Users from different regions are first directed to the same top layer Domain Name Servers to resolve one part of the virtual names. Top layer DNS replies with the set of bottom layer DNS machines in the user's region. Bottom layer DNS machines resolve the next part of the virtual names only in terms of IP addresses of the local cache machines.

part of the name (e.g "A456") in terms of IP addresses of the cache machines in the same region. Figure 6-1 shows how the two layers of DNS machines are used to split the system into locality regions.

When resolving a name such as A456.ProxyCache3.com, user's DNS resolver is first directed to a top layer DNS that can resolve the second part of the name: 'ProxyCache3'. The top layer DNS directs the resolution to a set of bottom layer DNS servers in a given geographical region numbered '3', as specified by the name. One of the bottom layer DNS servers is then picked from the set returned by a top layer

DNS.[1] That DNS will 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.

6.2 Advanced Load Balancing - Hot Pages

Another problem to consider in a web caching scheme is the problem of "Hot Pages". *Low load* and *balance* properties of consistent hashing guarantee the total load to be balanced among caches in one region. These properties are a good approximation of balanced load among the caches when all of the documents 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 CNN front page, for example, will be requested with far greater frequency than most other items. "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. Standard DNS resolvers usually use the top IP address in the list returned by BIND. Since BIND will round robin through the list of IP addresses on every DNS request, only a fraction of 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

[1] A standard browser's resolver usually stores a full set of DNS machines returned to it, (a set of bottom layer DNS machines in this case). On each subsequent DNS query, it chooses the next machine from the set and measure latency of that query. When it goes through the full set, it pick a DNS machine with the shortest DNS query latency and subsequently always contacts that one.

addresses on the list returned by BIND DNS.

Unfortunately, it is not trivial to establish which virtual names are hot and which are not. One way would be to modify BIND DNS to keep statistics on how frequently each virtual name is requested. However, the frequency of DNS requests for virtual names may not directly correspond to the frequency of Web requests for the documents represented by those virtual names. The reason is that DNS replies are cached for some periods of time by various DNS resolvers, browsers and intermediate Domain Name Servers, complicating the analysis.

Another way to find hot virtual names is by analyzing the load generated by the URL requests on the cache machines. As mentioned in section 4.1, our DNS servers frequently query cache machines in their region for status and load information. Load is measured in bytes of web requests served by a given cache per second over the last short interval (usually one minute). If some cache indicates a high load, it means that one or more virtual names that resolve to the IP address of that cache are hot. Even though this method does not allow unique identification of a hot virtual name, it identifies a set of virtual names that contain one or more hot ones.

Once some cache is discovered to be overloaded, the DNS servers take aggressive steps to prevent that cache from becoming further swamped. The set of virtual names mapped to that cache is identified, and all of the virtual names on the set are spread 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 IPs in the region so that load caused by this virtual name is spread to the entire region.) Then, we slowly back-off from the aggressive mapping, by reducing the size of these cache lists. We select a small subset of "spread" names and decrement their corresponding cache lists. We wait several minutes after each "back off" step. If some cache becomes overloaded again, we retract the last "back off" step. If the situation persists, we "spread out" a new set of virtual names corresponding to the overloaded cache, then we proceed with "back-off" again. In this way, we soon reach a balance with some virtual names mapped to variable size lists of caches.

6.3 Fail-over Features

In addition to efficient cache management and load balancing, Cache Resolver possesses a number of fail-over features which are absent in both, the Harvest Cache and the Crispy Squid.

Consider what happens when a regional proxy cache of a Harvest System dies. Since a regional cache in the Harvest System serves as the entry point into the Harvest hierarchy for its users, once it dies none of its users can connect to the cache system. The same problem exists in the Crispy Squid system, where users are assigned to one regional cache which they contact on every request. The Cache Resolver does not have static assignments of groups of users to caches. If a cache machine fails, the efficiency of the system is reduced only by the capacity of that one cache, but it still continues to function as a system, with all users able to connect. Some users may experience a timeout due to a stale caches resolution of a virtual name to an IP address that just died, but DNS servers quickly find a dead machine and stop mapping virtual names to its address.

The hierarchical structure of the Harvest System also presents a problem. In case of misses, requests propagate to few cache machines on top of the hierarchy. If many misses occur simultaneously in many regions, requests from many leaf nodes will hit the top nodes. The nodes at the top could easily become swamped in such a situation, slowing down the whole system. Such occurrence is possible, for example, at the start of a work day when all the people will hit their favorite sites for news, sports, or business information.

Crispy Squid introduces another point of failure: the directory unit. If the centralized, Crispy Squid directory dies, no cache would be able to communicate with any other caches on a miss and would go directly to the content server. In that case, the system breaks down; all caches begin acting as independent regional caches. As a result the hit rate is reduced.

The Cache Resolver has no single points of failure and will still continue to service all of its users even when a number of its units fail. The reason for this, is that the

Cache Resolver does not have any especially designated units, but rather has many copies of equivalent units: a number of caches in each region, a number of bottom layer DNS servers in each region, and a number of top layer DNS servers. (See figure 6-1.) It would take all of the caches or all of the bottom layer DNS units to stop functioning in order for a region to become invalidated. If some cache machines fail, the DNS units will simply stop resolving virtual names in terms of their IP addresses. The top layer DNS machines always return a list of bottom-layer DNS machines. So if some bottom-layer DNS machine fails, the user's DNS resolver library will wait and query the next DNS machine on the list. Similarly, it would also take all of the top layer DNS servers to fail in order for the system to stop functioning.

Chapter 7

From Theory to Practice

Properties of consistent hashing stated in section 3.1.1 were proven for a very general case. In reality, more assumptions about a global caching system and existence of multiple views can be made. With these specific assumptions, effects of consistent hashing can be understood more clearly.

One main assumption is that the number of possible views, k, is usually quite small. As described in section 6.1 our system is actually subdivided into locality regions, so that each user draws only on his/her regional caches. Since all of the users of a region will only use caches in that region the maximum number of views is the number of possible views of caches in one region alone. In a local region, it is much easier to keep track of all the caches than in the global world wide system. We expect, that in a local region constrained to several networks there would be only one or two machines in question at a time. Thus, we expect k not to exceed 4, a small integer. Unfortunately for a small value of k the bounds on *load* and *spread* properties of consistent hashing are quite loose. Without consistent hashing for $k = 2$, for example, the spread of each item should at most double. Where as the worst bound on consistent hashing is $O(t\log(Nk))$, where $N > 1$ is a confidence factor. This bound is much greater than 2.[1].

However, for a small value of k we can take advantage of the *balance* property of

[1]The exact bound proven in [7] is $8t(\log(Nk))$

consistent hashing. The *balance* property of consistent hashing deals with one view, ($k = 1$), and states that each cache in a particular view is assigned only a small fraction of items. Specifically, probability that for a view, V, any item is assigned to a given cache is $O(1/|V|)$, where $|V|$ is the number of caches in view V. (see sections 3.1.1 and 3.1.2) For $k > 1$, this fraction is at most multiplied by k. If k can be approximated by a small constant, this bound still remains $O(1/|V|)$ when multiplied by that constant. From our testing on some web server database mentioned in section 3.3 it can be seen that consistent hashing balances load quite well for a single view. We saw the same results from our testing described in chapter 5

So in practice, when the number of possible views is usually small we can rely on the *balance* property of consistent hashing for tight bounds on balanced data distribution. At the same time when k is large, *spread* and *load* properties will give tight bounds on data duplication.

Chapter 8

Future Research

The research described in this work opens a number of questions for further work. Further research should be conducted in both, the theoretical side of consistent hashing, and the system developed based on the principle. This chapter outlines possible questions for investigation.

8.1 Bounds on Consistent Hashing

Our current research indicates that it is possible to establish a tighter bound on the *low load* property of consistent hashing. Specifically, it is important to show that increasing the number of copies of each cache mapped to the circle circumference helps balance the load and tighten the bound. The intuition for a tighter bound comes from the *balance* property of consistent hashing that considers a single *view* of available caches. Section 3.1.2 shows, that as the number of copies of each cache increases, the probability with which an item is mapped to a specific cache decreases. The probability decreases until it reaches the true balance of $O(\frac{1}{|V|})$, where $|V|$ is the number of caches in a given *view*.

Section 3.1.2 suggests the analysis of the *load* bound to follow the *balance* bound proof. (The approach is outlined in that section). The difficulty is that in the *load* bound analysis responsibility arcs of a cache must be considered and bounded individually, and can not be considered collectively. Considering each arc individually

results in a very complex closed form expression, and simplification, still needs to be obtained.

8.2 Load Balancing among Caches

Even though, Consistent Hashing provides for an even distribution of documents among caches, other parameters that affect balance should be considered. One parameter, for example, is the frequency of requests for each document. Section 6.2 addresses the issue of "hot pages", documents that are very popular and are requested with very high frequency. Such documents can easily swamp the cache machine they are mapped to. The approach suggested in section 6.2, is to map more popular documents to more than one cache machine. We adopt some heuristics for "spreading" documents among a number of caches such that at a steady state, the number of machines a document is mapped to is roughly proportional to its popularity. These heuristics were tested, but the methods were not fully analyzed. An analysis in this area may show a better algorithm for dealing with popular web content.

In addition to popularity, size of the documents should also be considered. Just because documents are evenly spread among the caches does not imply that the number of bytes is evenly spread. One possible way of improvement is mapping documents of specific size range to specific caches. A paper on task assignment by Mor Harchol-Balter and Mark Crovella [4], for example, shows that when each server in a system serves jobs (e.g. documents) of little variance in size, the request latency over all jobs in the system (e.g. documents) drops.

8.3 URL Hashing

Another area of examination should be the scheme used for hashing URLs to virtual cache names that takes place inside the browsers. We use a modulo hash function described in sections 3.2.5 and 4.2. However, we did not conduct an analysis on how well this hashing distributes a global set of URLs among all the virtual cache names.

Obviously, a more uniform distribution among the virtual names will help a more uniform mapping of load to caches. A global set of URLs should be studied more closely to see what hash function could give a more uniform mapping. One idea, is to use consistent hashing inside the user browsers as well.

Additional heuristics may be added to URL hashing that could help the system performance overall. One heuristic is to consider how deep a URL file is in a web server's directory tree structure. It may be a good idea to consider locality of user request patterns and map files from the same subdirectory to the same cache.

8.4 General Goal

As can be observed from all the suggestions for further research, the general goal for future work is to find and analyze methods for uniform load distribution in a web caching system. First of all tighter bounds on the existing algorithms should be proved. Then, ways of mapping URLs to virtual names and virtual names to caches should be investigated. The goal is to find methods of most efficient cache system utilization, optimizing for request latencies.

Chapter 9

Conclusion

In chapter 5, we compared our system to other cache systems. We demonstrated that when users are know 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 users via a hash function. Since it is likely that users of a large network, may have inconsistent views of live caches, we suggest the use of consistent hashing which balances data quite well despite conflicting user views. Additionally, we have described implementation of our system, that uses consistent hashing, 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 performance of the World Wide Web.

Bibliography

[1] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *Sigmetrics*, 1997.

[2] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael Schwartz, and Kurt Worrell. A hierarchical internet objectcache. In *USENIX*, 1996.

[3] Syam A. Gadde, Jeff Chase, and Michael Rabinovich. A taste of crispy squid. In *WISP98*, 1998.

[4] Mor Harchol-Balter, Mark Crovella, and Cristina Murta. Task assignment in a distributed server. In *10th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Lecture Notes in Computer Science, No. 1469*, September 1998.

[5] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed cachine protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pages 654-663*, 1997.

[6] David Karger, Tom Leighton, Daniel Lewin, and Alex Sherman. Web caching with consistent hashing. In *WWW8 Conference*, 1999.

[7] Daniel Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's project, Massachusetts Institute of Technology, Department of Computer Science and Electrical Engineering, May 1998.

[8] Radhika Maplani, Jacob Lorch, and David Berger. Making world wide web caching servers cooperate. In *World Wide Web Conference*, 1996.

[9] D. Wessels and K. Claffy. Internet Cache Protocol (ICP) version 2. RFC 2186. National Laboratory for Applied Network Research/UCSD. September 1997. http://squid.bilkent.edu.tr/rfc2186.txt.

[10] D. Wessels and K. Claffy. Application of Internet Cache Protocol (ICP) version 2. RFC 2187. National Laboratory for Applied Network Research/UCSD. September 1997. http://squid.bilkent.edu.tr/rfc2187.txt.

[11] Rina Panigrahy. Relieving hot spots on the world wide web. Master's project, Massachusetts Institute of Technology, Department of Computer Science and Electrical Engineering, 1997.