Relieving Hot Spots on the World Wide Web

by

Rina Panigrahy

B.Tech, Computer Science
IIT, 1995

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1997

Author . . . .
Department of Electrical Engineering and Computer Science
May 9, 1997

Certified by . . . .
David R. Karger
Professor at Laboratory for Computer Science
Thesis Supervisor

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

Relieving Hot Spots on the World Wide Web

by

Rina Panigrahy

Submitted to the Department of Electrical Engineering and Computer Science
on May 9, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

We describe a family of caching protocols for distributed networks that can be used to decrease or eliminate the occurrence of *hot spots* in the network. Hot spots are web sites that swamped by a large number of requests for their pages. Our protocols are particularly designed for use with very large networks such as the Internet, where delays caused by hot spots can be severe, and where it is not feasible for every server to have complete information about the current state of the entire network. The protocols are easy to implement using existing network protocols such as TCP/IP, and require very little overhead. The protocols work with local control, make efficient use of existing resources, and scale gracefully as the network grows.

Our caching protocols are based on a special kind of hashing that we call *consistent hashing*. Roughly speaking, a consistent hash function is one which changes minimally as the range of the function changes. Through the development of good consistent hash functions, we are able to develop caching protocols which do not require users to have a current or even consistent view of the network. We believe that consistent hash functions may eventually prove to be useful in other applications such as distributed name servers and/or quorum systems.

Thesis Supervisor: David R. Karger
Title: Professor at Laboratory for Computer Science

Acknowledgements

I would like to thank Tom Leighton, David Karger, Eric Lehman, Matthew Levine, and Daniel Lewin for their help and cooperation.

Contents

Chapter 1

Introduction

1.1 The Problem

In this thesis, we study the problem of "hot spots" on the World Wide Web. *Hot spots* occur any time a large number of clients wish to simultaneously access data from a single server. If the site is not provisioned to deal with all of these clients simultaneously, service may be degraded or lost.

Many of us have experienced the hot spot phenomenon in the context of the Web. A Web site can suddenly become extremely popular and receive far more requests in a relatively short time than it was originally configured to handle. In fact, a site may receive so many requests that it becomes "swamped," which typically renders it unusable. It is often hard to predict such sudden changes in popularity. Besides making the one site inaccessible, heavy traffic destined to one location can congest the network near it, interfering with traffic at nearby sites.

As use of the Web has increased, so has the occurrence and impact of hot spots. Recent famous examples of hot spots on the Web include the JPL site after the Shoemaker-Levy 9 comet struck Jupiter, an IBM site during the Deep Blue-Kasparov chess tournament, and several political sites on the night of the election. In some of these cases, users were denied access to a site for hours or even days. Other examples include sites identified as "Web-site-of-the-day" and sites that release new versions of popular software.

In this work we describe a family of distributed caching protocol that can be used to decrease or eliminate the occurence of hot spots on the Web. Our protocols make use of randomization to ensure that the load of serving requests is balanced among the caches. The number of cached copies of a page adapts dynamically to its changing popularity. These protocols work with local control, makes efficient use of existing resources, and scale gracefully as the network grows. Our caching protocols are based on a special kind of hashing that we call *consistent hashing*. Roughly speaking, a consistent hash function is one which changes minimally as the range of the function changes. Through the development of good consistent hash functions, we are able to develop caching protocols which do not require users to have a current or even consistent view of the network. This allows new caches to be added to the network without every one having to know the latest set of caches. Before giving our tools, we discuss past work both practical and theoretical. Section 1.2 gives a detailed discussion of the practical work. In this thesis we ignore the issue of updates to pages, i.e., all our protocols are developed for static web pages. Dynamic pages give rise to issues of maintaining cache consistency which we have totally ignored. However, we believe that previous work done on cache consistency can be used directly in our protocols. In section 1.2 we also discuss existing methods used for maintaining cache consistency. Section 1.3 discusses the theoretical work.

1.2 Related Practical Work

Several approaches to overcoming the problem of hot spots have been proposed. Most use some kind of replication strategy to store copies of hot pages throughout the Internet; this spreads the work of serving a hot page across several servers. In one approach, already in wide use, several clients share a *proxy cache*. Proxy caches [14] have long been in use to reduce web traffic in a network. It tries to satisfy requests with a cached copy; failing this, it forwards the request to the home server. The dilemma in this scheme is that there is more benefit if more users share the same cache, but then the cache itself is liable to get swamped. Malpani et al. [7] work around this problem by making a group of caches function as one. A page request is served locally if it is cached in any of the caches. The disadvantage of this approach is that it leads to a lot of communication between the caches thus making it unsuitable for large networks. Chankhunthod et al. [1] developed the Harvest Cache, a more scalable approach using a *tree* of caches. The advantage of a cache tree is that a cache receives page requests only from its children (and siblings), ensuring that not too many requests arrive simultaneously. An independent approach called *prefetching* could be useful for reducing web latency. The idea is to send to the client along with the requested page a set of pages it is likely to request in future.

1.2.1 World Wide Web proxies

A proxy is a special server that does caching for a medium sized network. [14]. Instead of going directly to the home server for a page, the clients in the network first query the proxy. If the proxy has the page cached, it gives it to the clients; otherwise it gets a copy of the page from the home server, caches it in its cache and forwards it to the clients. So instead of receiving individual requests from each client, a popular server now receives only one request from the proxy, thus reducing network traffic. Also, once the first request for a popular document has been made, clients are able to obtain that document much faster since their requests can be served locally by the proxy. Setting up a proxy server is easy, and the most popular Web client programs already have proxy support built in.

However, the size of the network containing the proxy should not be too large; otherwise the proxy itself could get swamped with requests. This limits the benefit a popular server can derive from the use of proxy caches.

1.2.2 Co-operative Caching

Malpani et al. [7] tried to make a group of caches function as one. A user's request for a page is directed to an arbitrary cache. If the page is stored there, it is returned to the user. Otherwise, the cache forwards the request to all other caches via a special protocol called "IP Multicast" [5] . *Multicast* is a protocol for the transmission of a packets to a subset of the hosts in a network. If the page is cached nowhere, the request is forwarded to the home site of the page. The disadvantage of this technique is that as the number of participating caches grows, even with the use of multicast, the number of messages between caches can become unmanageable. A tool that we develop in this paper, *consistent hashing*, gives a way to implement such a distributed cache without requiring that the caches communicate all the time. We discuss this in chapter 4.

1.2.3 Harvest

The Harvest system [1] was the first to implement hierarchical caching on a large scale. It consists of a number of caches arranged in a hierarchy, spread over the internet. A Harvest cache can be configured with an arbitrary number of parents and siblings. When a Harvest cache needs a page it queries each of its siblings and parents to check if any of them have the page. The page is then immediately fetched from the source which responds earliest with a positive reply. This ensures that the page is obtained from the source with low latency.

From a theoretical standpoint however the Harvest approach lacks scalability as it uses the same hierarchy of caches for all pages. Since each request for a new page must pass through the top level of the hierarchy, there could be a large load on the caches near the top of the hierarchy if many distinct pages are requested simultaneously. However, the developers of the Harvest cache claim that this will not pose a problem in practice. They argue that the root level caches, located on the internet back-bone, are capable of handling the maximum request rate allowed by the bandwidth of the back-bone.

1.2.4 Push Caching

Push Caching was introduced by Gwertzman and Seltzer [17] at Harvard. The essential idea is to replicate a popular document so that the number of replicas is proportional to the popularity of the document. They envision a network infrastructure with thousands of *push-cache servers* onto which files may be *pushed*. A central *registry* service tracks available push-cache servers, helping servers decide where to replicate their objects by providing a list of available push-cache servers on demand. Network topology is also taken into account in deciding where to push a page. Since network topology is hard to obtain, Gwertzman and Seltzer use the Geographical distances as an approximation to the network latencies between machines.

Azer Bestavros [2] uses the same approach as push caching, and in addition uses a second technique that they call "speculative service". The idea is that a server responds to a client's request by sending, in addition to the document requested, a number of other documents that it speculates will be requested by that client in the near future. This is very similar to the technique of *prefetching* which we will discuss later.

1.2.5 Prefetching

Prefetching tries to make use of the locality of reference in the requests made by a client for the pages on a web site. Users usually browse the Web by following hyperlinks from one Web page to another. Hyperlinks on a page often refer to pages stored on the same server. Typically, there is a pause after each page is loaded while the user reads the displayed material. This time could be used by the client to prefetch files that are likely to be accessed soon (e.g. those pointed to by the current page), thereby avoiding retrieval latency if and when those files are actually requested. The retrieval latency has not actually been reduced; it has just been overlapped with the time the user spends reading, thereby decreasing the access time.

Padmanabhan and Mogul [13] propose a predictive prefetching scheme for the World Wide Web in which the servers tell the clients which files are likely to be requested next by the user, and the clients decide whether or not to prefetch the files based on local consideratons (such as the contents of the local cache). Each server maintains a *dependency graph* that depicts the pattern of accesses to different files stored a the server. This graph has a node for each file and a weight associated with an arc between two nodes. This weight on the arc between A and B indicates how often B is requested after A as has been accessed. This dependency graph is updated dynamically as the server receives new requests. If A is currently being requested and if the weight on the arc between A and B is higher than a *prefetch threshold* then the file B is considered as a candidate for prefetching.

1.2.6 Cache Consistency

Any caching strategy using several caches for caching a page needs to worry about dynmaic pages, i.e., pages that are modified frequently. Dynamic pages could lead to inconsistencies in the cached copies. In our work we do not study the issue of cache consistency. However existing methods used to deal with this issue apply to our protocol as well. The value of caching is greatly reduced if cached copies are not updated when the original data change. Cache consistency mechanisms ensure that cached copies of data are eventually updated to reflect changes to the original data. Gwertzman and Seltzer [22] studied different techniques for maintaining cache consistency in the context of the World Wide Web. There are several cache consistency mechanisms currently in use on the Internet: time-to-live fields, client polling, and invalidation protocols.

A time-to-live field is essentially an estimate of the time when an object expires. This is typicially implemented as the "expires" header field in the HTTP protocol. If a requested document has a TTL that has already elapsed, it is considered to have expired and a fresh copy of the document is obtained from the original server.

Client polling is a technique where cachess periodically check back with the server to determine if cached objects are still valid. The basic assumption is that young files are modified more frequently than old files and that the older the file is the less likely it is to be modified. For example, a page that is only a day old when fetched is probably modified on a daily basis and hence should'nt be kept for more than a few hours. However, a page that is a month old when fetched is probably modified less frequently and could probably be kept for for a few days without becoming stale. The update threshold is expressed as a percentage of the object's age. An object is invalidated when the time since last validation exceeds the update threshold times the object's age.

Invalidation protocols are required when weak consistency is not sufficient; many distributed file systems rely on invalidation protocols to ensure that cached copies never become stale. Invalidation protocols depend on the server keeping track of cached data; each time an item changes the server notifies caches that their copies are no longer vaid. One problem with invalidation protocols is that they are often expensive. Servers must keep track of where their objects are currently cached. If a machine with data cached cannot be notified, the server must continue trying to reach it, since the cache will not know to invalidate the object unless it is notified by the server.

Gwertzman and Seltzer perfomed trace driven simulations to compare the different mechanisms for cache consistency on the Web. They conclude that a weak cache consistency protocol reduces network bandwidth consumption and server load more than either TTL or an invalidation protocol and can be tuned to return stale data less than 5% of the time.

1.3 Related Theoretical Work

On the theoretical side, Plaxton and Rajaraman [10] introduced the idea of using randomization and hashing to balance load on a network. Their work however was done in the context of a synchronous network of nodes in a parallel machine. They assume the existence of special priority messages that reach a node even though it may be swamped with other low priority messages which is clearly not the case in the Web. R. Ravi [15] studies the problem of finding optimal broadcast trees in a graph, which could be useful for building multicast trees. A theoretical study of prefetching was done in [20].

1.3.1 Randomization and Hashing

Plaxton and Rajaraman [10] were the first to conduct a theoretical study of the problem of providing fast concurrent access to shared objects in a synchronous network of distributed computation. Their basic idea is to construct for each object a random tree of caches which can be computed using a global hash function. The use of random trees ensures good load balancing among the caches and the global hash function makes it easy to compute the structure of each tree without having to store the arrangement of the caches for each of the trees. Whenever a node needs an object it sends a request to a random node in each level of the tree. Whenever a caching node senses a large number of request for a cached object it pushes this object to its children. It is assumed that even if a node is swamped with a large number of requests, it will still be able to push an object to its children. This ensures that if the popularity of a given object exceeds the number of cached copies, the number of cached copies increases geometrically until it matches the demand for the object. Plaxton and Rajaraman do a theoretical analysis of their protocol in a model containing n nodes where each node can process at most $O(\log n)$ requests per time step, and prove that all requests are satisfied with high probability in $O(\log n)$ time steps.

Several of their ideas also apply to the problem of relieving hot spots on the Web. However, they assume that messages between machines can be prioritized and even though a node may be swamped with messages of low priority, it can still be reached by a high priority message. Such assumptions are not true in the case of the Internet. Their method also does not take into consideration the varying distances between the different machines on the Web. They assume a uniform distance metric between the nodes in a parallel machine.

1.3.2 Finding minimum time broadcast trees

If we model the Web as a weighted graph with nodes representing machines and weights denoting the latencies between the machine, then we can ask what is the minimum time required to broadcast a popular document residing at a node to all the other nodes in the graph. Such an optimal broadcast would correspond to sending the page along a spanning tree. Such a tree would be very suitable for use in a Multicast protocol [5]. R. Ravi [15] has studied a special case of this problem in a synchronous framework where at each time step, any processor that has received the document is allowed to communicate it to at most one of its neighbors in the network. This special case of the problem is known to be NP-complete [16], even in 3-regular planar graphs. R. Ravi gives an $O(\log^2 n/\log\log n)$-approximation algorithm for the minumum broadcast time problem on an n-node graph.

1.3.3 Prefetching Algorithms

Prefetching can be studied as a learning problem that involves predicting the page accesses of the user. In [20] Vitter and Krishnan give the first provable theoretical bounds on prefetching performance. Their approach is to use optimal data compression methods to do optimal prefetching. They model the request trace as an *mth order Markov source* (i.e., states correspond to the previous m requests) , and evaluate their prefetching algorithm relative to the best online algorithm that has complete knowledge of the structure and transition probabilities of the Markov source.

1.4 Our Contribution

Here, we describe two tools for data replication and use them to give a caching algorithm that overcomes the drawbacks of the preceding approaches and has several additional, desirable properties.

Our first tool, *random cache trees*, combines aspects of the structures used by Chankhunthod et al. and Plaxton/Rajaraman. Like Chankhunthod et al., we use a tree of caches to coalesce requests. Like Plaxton and Rajaraman, we balance load by using a different tree for each page and assigning tree nodes to caches via a random hash function. By combining the best features of Chankhunthod et al. and Plaxton/Rajaraman with our own methods, we prevent any server from becoming swamped with high probability, a property not possessed by either Chankhunthod et al. or Plaxton/Rajaraman. In addition, our protocol shows how to reduce memory requirements (without significantly increasing cache miss rates) by only caching pages that have been requested a sufficient number of times.

Our second tool is a new hashing scheme we call *consistent hashing*. This hashing scheme differs substantially from that used in Plaxton/Rajaraman and other practical systems. Typical hashing schemes do a good job of spreading load through a known, fixed collection of servers. The Internet, however, does not have a fixed collection of machines. Instead, machines come and go as they are brought into the network or crash. Even worse, the information about what machines are functional propagates slowly through the network, so that clients may have incompatible "views" of which machines are available to replicate data. This makes standard hashing useless since it relies on clients agreeing on which caches are responsible for serving a particular page. Consistent hashing may help solve such problems. Like most hashing schemes, consistent hashing assigns a set of items to buckets so that each bin receives roughly the same number of items. Unlike standard hashing schemes, a small change in the bucket set does not induce a total remapping of items to buckets. In addition, hashing items into slightly different sets of buckets gives only slightly different assignments of items to buckets. We apply consistent hashing to our tree-of-caches scheme, and show how this makes the scheme work well even if each client is aware of only a constant fraction of all the caching machines. In addition, we believe that consistent hashing will be useful in other applications where multiple machines with different views of the network must agree on a common storage location for an object without communication.

1.5 Presentation

In Chapter 2 we describe our model of the Web and the hot spot problem. Our model is necessarily simplistic, but is rich enough to develop and analyze protocols that we believe may be useful in practice. In Chapter 3, we describe our random tree method and use it in a caching protocol that effectively eliminates hot spots under a simplified model. This simplified model assumes that the latencies between all pairs of machines are the same and that each browser knows about all the caches present in the system. Independent of Chapter 3, in Chapter 4 we present our consistent hashing method and use it to solve hot spots under a model involving inconsistent views.

In Chapter 5 we combine our two techniques and show that our protocol works even if browsers have inconsistent views of the set of caches. In Chapter 6 we propose a simple delay model that captures the varying distances between machines on the Internet. We show that our protocol can be easily extended to work in this more realistic delay model. In Chapters 7 we consider fault tolerance of our protocol. In Chapter 8 we discuss some extensions and open problems.

Chapter 2

Model

This chapter presents our model of the Web and the hotspot problem.

We classify computers on the Web into three categories. All requests for Web pages are initiated by *browsers*. The permanent homes of Web pages are *servers*. *Caches* are extra machines that we use to protect servers from the barrage of browser requests. Throughout the paper, the set of caches is $\mathcal{C}$ and the number of caches is C. Note that these categories may not be disjoint. A machine could simultaneously be a cache, a server, and a browser.

Each server is home to a fixed set of pages. Caches are also able to store a number of pages, but this set may change over time as dictated by a caching protocol. We assume that the content of each page is unchanging. (However, we believe that past work done on handling dynamic pages in caching protocols, which was discussed in section 1.2.6, can be directly applied to our protocols.) The set of all pages is denoted by $\mathcal{P}$.

A machine may send a request to any other machine it is aware of. The two typical types of messages are requests for pages and the pages themselves. A machine which receives too many messages too quickly ceases to function properly and is said to be "swamped".

An adversary decides which pages are requested by browsers. However, the adversary cannot see random values generated in our protocol and cannot adapt his requests based on observed delays in obtaining pages. We consider two models. We consider a static model in which a single "batch" of requests is processed, and require that the number of page requests be at most $R = \rho C$ where C is the number of caches.

Latency measures the time for a message from machine m_1 to arrive at machine m_2. We denote this quantity $\delta(m_1, m_2)$. In practice, of course, delays on the Internet are not so simply characterized. The value of δ should be regarded as a "best guess" that we optimize on for lack of better information; the correctness of a protocol should not depend on values of δ (which could actually measure anything such as throughput, price of connection or congestion) being exactly accurate. Note that we do not make latency a function of message size; this issue is discussed in Section 3.2.1.

Objective

The "hot spot problem" is to satisfy all browser page requests while ensuring that with high probability no cache or server is swamped. While our basic requirement is to prevent swamping, we also have two additional objectives. The first is to minimize cache memory requirements. A protocol should work well without requiring any cache to store a large number of pages. A second objective is, naturally, to minimize the delay a browser experiences in obtaining a page.

Chapter 3

Random Trees

In this chapter we introduce our first tool, random trees. A standard solution to handle a large number of requests for a page p to a server s is to have a set of proxy caches to protect the server s. To obtain the page p, a browser requests the page from one of the proxy caches which might in turn make a request to the server if it does not already have the page. The problem with this scheme is choosing the size of the set of caches assigned to protect the server. If this set is small, then the caches themselves could be overwhelmed by page requests. If the set is large, then the server could be swamped by page requests from the very caches assigned to protect it.

A natural extension [1] is to introduce several layers of proxy caches where a cache in one layer makes requests only to caches in the layer above it. We must ensure that the ratio between the number of caches in a given layer to the number in the layer above it is not too large. This suggests that we arrange the caches into a balanced tree with a bounded degree d and with the server at the root of the tree. When a browser needs page p it makes a request to a cache at a random leaf node that serves the request if the page is present in its cache and otherwise delegates the request to its parent. Now if we focus our attention on a sufficiently brief interval of time (so as to ignore eviction of pages) then each cache sends out at most one request for the page p to its parent and so no internal node gets more than d requests for the page p.

If there are C caches in the tree then there are about C leaves and so the maximum demand for page p that this tree can satisfy without swamping is about C times the demand that a single machine can satisfy. As the popularity of a page increases, more copies of the page get cached down the tree. In fact, for each page's tree there will be a *threshold level* where the number of caches is roughly equal to the number of requests for the page. The caches above the threshold level are likely to see requests for the page and will tend to have cached copies and those below the threshold level are less likely to see requests and hence will tend not to have copies of the page. As the popularity of a page increases the threshold level moves down, thus increasing the number of copies of the page. In this way the number of cached copies automatically adapts to changes in the popularity of the page.

So far we have talked about handling large number of requests for one page. We need to extend our protocol to handle several potential hotspots on the web. We could try using the same tree to cache all pages. However, observe that the root machine will receive at least the first request for each page. So if many distinct pages are requested, the caches close to the root will receive too many requests and load will not be balanced evenly among all the caches. A solution is to create a distinct tree of caches for each page. In fact a good way to achieve load balancing is to have for each page a tree of caches arranged in random order, which is the technique used by Plaxton/Rajaraman [10]. This ensures that no machine is near the root for many pages, thus providing good load balancing.

We now describe our protocol. To simplify the presentation, we start with a simple caching protocol that would work well in a simpler world. In particular, we make the following simplifications to the model:

1. All machines know about all caches.

2. Distances between machines are uniform, i.e., $\delta(m_i, m_j) = 1$ for all $i \neq j$.

Under these restrictions we show a protocol that has good behavior. That is, with high probability no machine is swamped. For each request a browser may need to go through $\Theta(\log_d C)$ proxy caches and we prove that it is necessary to prevent swamping. We use total cache space which is a small fraction of the number of requests, and is evenly divided among the caches. In chapter 5 we will analyze our protocol without the assumption that all machines have full knowledge of the caches. In chapter 6 we extend our protocol to a scenario where all the latencies between pairs of machines are not uniform.

In Section 3.1 below, we define our protocol precisely. In Section 3.2, we analyze the protocol, bounding the load on any cache, the storage each cache uses, and the delay a browser experiences before getting the page.

3.1 Protocol

Just like in Harvest [1] we will use a hierarchy of caches. However, instead of using the same hierarchy for all pages we will use a different one for each page. In each hierarchy the caches are arranged in random order, similar to the approach taken by Plaxton and Rajaraman [10]. We associate with each page a rooted d-ary tree, called an *abstract tree* that will represent the tree of caches for that page. We use the term *nodes* only in reference to the nodes of these abstract trees. The number of nodes in each tree is equal to the number of caches, and the tree is as balanced as possible (so all levels but the bottom are full). We number the nodes of the tree by their rank in breadth-first search order. The protocol is described as running on these abstract trees; to support this, all requests for pages take the form of a 4-tuple consisting of the identity of the requester, the name of the desired page, a sequence of abstract nodes through which the request should be directed, and a sequence of caches that should act as those abstract nodes. To determine the latter sequence, that is, which cache actually does the work for a given node, the nodes are mapped to machines. The root of a tree is always mapped to the server for the page. All the other nodes are mapped to the caches by a hash function $h : \mathcal{P} \times [1 \ldots C] \to \mathcal{C}$, which must be distributed to all browsers and caches. In order not to create copies of pages for which there are few requests, we have another parameter, q, for how many requests a cache must see (acting as a particular abstract node) before it bothers to store a copy of the page.

Now, given a hash function h, and parameters d and q, our protocol is as follows:

Browser When a browser wants a page, it picks a random leaf to root path in the abstract tree, maps the nodes to machines with h, and asks the leaf node for the page. The request includes the name of the browser, the name of the page, the path, and the result of the mapping.

Cache When a cache receives a request, it first checks to see if it is caching a copy of the page or is in the process of getting one to cache. If so, it returns the page to the requester (after it gets its copy, if necessary). Otherwise it increments a counter for the page and the abstract node it is acting as, and asks the next machine on the path for the page. If the counter reaches q, it caches a copy of the page. In either case the cache passes the page on to the requester when it is obtained.

Server When a server receives a request, it sends the requester a copy of the page.

3.2 Analysis

In this section we will analyze the performance of our protocol. To simplify the analysis we will assume that a certain number of requests arrive simultaneously. This restricted model is "static" in the sense that there is no notion of requests arriving over time. Let the number of static requests be ρC so that ρ is the average number of requests per cache. For a static analysis we simply look at the paths taken by these ρC requests up their respective trees. Any abstract node that receives more than q requests from its children will cache the page. Each abstract node sends out at most q requests to its parent.

The analysis is broken into three parts. We begin by showing that the latency in processing a request is likely to be small, under the assumption that no machine is swamped. We then show that no machine is likely to be swamped. We conclude by showing that no cache need store too many pages for the protocol to work properly.

3.2.1 Latency

Under our protocol, the delay a browser experiences in obtaining a page is determined by the height of the tree. If a request is forwarded from a leaf to the root, the latency is twice the length of the path, $2\log_d C$. If the request is satisfied with a cached copy, the latency is only less. If a request stops at a cache that is waiting for a cache copy, the latency is still less since a request has already started up the tree. Note that d can probably be made large in practice, so this latency will be quite small. Although our protocol increases the delay in getting pages, the existence of tree schemes, like the Harvest Cache, suggests that is acceptable in practice.

Note that in practice, the time required to obtain a large page is not multiplied by the number of steps in a path over which it travels. The reason is that the page can be transmitted along the path in a pipelined fashion. A cache in the middle of the path can start sending data to the next cache as soon as it receives some; it need not wait to receive the whole page. This means that although this protocol will increase the delay in getting small pages, the overhead for large pages is negligible.

Our bound is optimal (up to constant factors) for any protocol that forbids swamping, as shown by the following lemma.

Lemma 3.2.1 *Any protocol that can handle ρC requests for a page simultaneously, with no machine* *serving more than d requests, must have an average latency of* $\Omega(\log_d(\rho C))$ *hops per request.*

Proof: Consider making ρC requests for a single page. Look at the directed graph with nodes corresponding to machines and directed edges corresponding to links over which the page is sent. This graph has an out-degree of at most d at each node. So the number of nodes reachable from the home server of the page in x steps is at most d^x. So a constant fraction of the nodes will be $\Omega(\log_d(\rho C))$ steps away from the home server. This means that the average distance from the home server to the C requesting machines is $\Omega(\log_d(\rho C))$. $\square$

3.2.2 Swamping

We now analyze the number of requests a machine gets in our protocol under the simplified model. Note that a server can receive at most d requests per page. We will assume that a server can handle d requests for each of its page. What remains is the analysis of the number of requests received by caches. The intuition behind our analysis is the following. First we analyze the number of requests directed to the abstract tree nodes of various pages. These give "weights" to the abstract tree nodes. We then analyze the outcome when the tree nodes are mapped by a hash function onto the actual caching machines: a machine gets as many requests as the total weight of nodes mapped to it. To bound this mapped weight, we first give a bound for the case where each node is assigned to a random machine. This is a weighted version of the familiar balls-in-bins type of analysis. Our analysis gives a bound with an exponential tail. In lemma 3.2.6 we develop tools to ensure that these bounds hold even when the balls are assigned to bins only $k = O(\log N)$-way independently. This can be achieved by using a k-universal hash function to map the abstract tree nodes to machines.

Analysis for Random h

Theorem 3.2.2 *If h is chosen uniformly and at random from the space of functions $\mathcal{P} \times [1 \ldots C] \mapsto \mathcal{C}$ then with probability at least $1 - 1/N$, where N is a parameter, the number of requests a given cache gets is no more than*

$$\rho\left(2\log_d C + O\left(\frac{\log N}{\log\log N}\right)\right) + O\left(\frac{dq\log N}{\log\left(\frac{dq}{\rho}\log N\right)} + \log N\right)$$

requests

Note that $\rho \log_d C$ is the average number of requests per cache since each browser request will give rise to $\log_d C$ requests up the trees. The $\frac{\rho \log N}{\log\log N}$ term arises because at the abstract leaf nodes of a tree's page some cache could occur $\frac{\log N}{\log\log N}$ times (balls-in-bins) and if adversary chooses to devote all R requests to that page then each leaf is expected to receive ρ requests.

Corollary 3.2.3 *If h is chosen uniformly and at random from the space of functions $\mathcal{P} \times [1 \ldots C] \mapsto \mathcal{C}$ then with probability at least $1 - 1/N$, where N is a parameter,* no *no cache gets more than*

$$\rho\left(2\log_d C + O\left(\frac{\log(NC)}{\log\log(NC)}\right)\right) + O\left(\frac{dq\log(NC)}{\log\left(\frac{dq}{\rho}\log(NC)\right)} + \log(NC)\right)$$

requests

Proof: The bound given in theorem 3.2.2 holds for a given cache with probability at least $1-1/N$. Since there are C caches, the probability that the bound holds for all caches is $1 - C/N$. Since N is simply a parameter, we can replace N by NC to get the corollary. □

We prove theorem 3.2.2 in the rest of the section. We split the analysis into two parts. First we analyze the requests to a cache due to its presence in the leaf nodes of the abstract trees and then analyze the requests due to its presence at the internal nodes and then add them up.

Requests to Leaf Nodes

Each request for a page p goes to a random leaf node in that page's abstract tree. If L denotes the number of leaf nodes in each abstract tree, then L is about $C(1 - 1/d)$. We associate a weight of $\frac{r_p}{L}$ with each abstract leaf node of p's tree, which is the number of requests for p each of them is expected to receive. Then we map these weighted abstract leaf nodes over all pages onto the set of caches and bound the total weight assigned to a cache. Finally we argue that the total number of requests received by a cache is with high probability close to the total weight assigned to it. Note that we cannot simply say that the requests are mapped randomly onto the set of caches, which is different from mapping the requests to abstract nodes first and then mapping the abstract nodes to the caches.

With each abstract leaf node of page p's tree we associate a weight $w_p = r_p/L$. A machine m has $1/C$ chance of being at an arbitrary leaf node of a given page. Let V_{pj} denote the event the j^{th}

leaf node of p's tree is assigned to m. So V_{pj} is 1 with probability $1/C$ and 0 otherwise. Let us try to bound to the total weight $W = \sum w_p V_{pj}$ assigned to m. We would like to use Chernoff bounds; however, W is a weighted sum of poisson variables with weights possibly greater than 1. But note that each weight $w_p = r_p/L \leq R/L \leq \rho/d(d-1)$. So we can apply Chernoff bounds to $\frac{W}{\rho/d(d-1)}$, which is a weighted sum of poisson variables, where all weights are at most 1. This gives a bound of $O(\rho \log N/\log\log N)$ on W which holds with probability at least $1 - 1/N$.

Next we will argue that with high probability the number of leaf node requests machine m gets is close to the random variable W. For any assignment of tree nodes to machines let A denote set of leaf nodes that get assigned to m. Now observe that the random variable W is a function of A. Let f denote this function. Let the random variable H_l denote the total number of requests received by machine m due to its presence at leaf nodes. We need to provide a high probability bound for H_l.

Let α denote the high probability bound on W that we just proved. Now $Pr[H_l > \beta]$

$$\begin{aligned} &= Pr[W > \alpha] \cdot Pr[H_l > \beta | W > \alpha] \\ &\quad + Pr[W \leq \alpha] \cdot Pr[H_l > \beta | W \leq \alpha] \\ &\leq Pr[W > \alpha] + Pr[H_l > \beta | f(A) \leq \alpha] \end{aligned}$$

We know that first probability in the sum is at most $1/N$. We will choose β appropriately so that the second part of the sum is also $< 1/N$. Given the set of leaf nodes where m is present we claim that the total number of requests R can be written as a sum of independent poisson random variables. We have one poisson variable for each request of each page which is set of 1 if m gets that request and 0 otherwise. Now let μ denote the expected value of their sum Then $\mu = W \leq \alpha$. By Chernoff bounds we know that the probability that the sum $> 4 \cdot \mu + \ln N$ is $< 1/N$. So if we set β to $4\alpha + \ln N$ then the second probability is $< 1/N$. So we can claim that with probability $> 1 - 2/N$ the random variable H_l is $O(\alpha + \ln N)$

Requests to Internal Nodes

We will now bound the number of requests m gets due to its presence at internal nodes. Again we think of the protocol as first running on the abstract trees. With each abstract node we will associate a weight equal to the number of requests it receives. These weighted nodes (balls) are then randomly assigned to the set of caches (bins). Using standard balls-in-bins type analysis we will then bound the total weight falling into a bin.

Now no abstract internal node gets more than dq requests because each child node gives out at most q requests for a page. Consider any arbitrary arrangement of paths for all the R requests up their respective trees. Since there are only R requests in all we can bound the number of abstract nodes that get dq requests. In fact we will bound the number of abstract nodes over all trees that receive between 2^j and 2^{j+1} requests where $0 \leq qj \leq \log dq - 1$. Let n_j denote the number of abstract nodes that receive between 2^j and 2^{j+1} requests. Let r_p be the number of requests for page p. Then $\sum r_p \leq R$. Since each of the R requests gives rise to at most $\log_d C$ requests up the trees, the total number of requests is no more than $R \log_d C$. So,

$$\sum_{j=0}^{\log(dq)-1} 2^j n_j \leq R \log_d C \tag{3.1}$$

The following lemma gives us a bound on n_j.

Lemma 3.2.4 *The total number of internal nodes that receive at least qx requests is at most $2R/x$ if $x > 1$*

Proof: Look at the r_p requests for page p and the paths produced by these requests up the tree. Consider the tree on the internal nodes induced by these paths. Since any node can get at most q requests from each child, a node that gets at least qx requests must have downward degree of at least $x > 1$. Look at all nodes u with downward degree one. Let v and w be the parent and the child of u respectively. Replace all such downward degree one nodes u by a single edge connecting v and w. This will eliminate all nodes with downward degree equal to one but will preserve the degrees of the other nodes. Since $x > 1$, we are now left with a tree where each node has a downward degree of at least 2. In such a tree the number of leaves is at least half the total number of nodes. Also the sum of the downward degrees is equal to the total number of edges, which is the same as the number of vertices minus 1. The number of leaves in the tree is no more than the number of requests, which is r_p. So if there are y nodes with downward degree of at least x then $xy \leq 2r_p$ and so $y \leq 2r_p/x$. Thus the total number of nodes over all trees which receive at least qx requests is no more than $\sum 2r_p/x = 2R/x$. □

For $x = 1$ there can clearly be no more than $R \log_d C$ requests. The preceding lemma tells us that n_j, the number of abstract nodes that receive between 2^j and 2^{j+1} requests, is at most $\frac{2R}{2^j}$ except for $j = 0$. For $j = 0$, n_j will be at most $R \log_d C$. Now the probability that machine m assumes a given one of these n_j nodes is $1/C$. Since assignments of nodes to machines are independent the probability that a machine m receives more than z of these nodes is at most $\binom{n_j}{z}(1/C)^z \leq (en_j/Cz)^z$. In order for the right hand side to be as small as $1/N$ we can set $z = \Omega(\frac{n_j}{C} + \frac{\log N}{\log(\frac{C}{n_j}\log N)})$. Note that the latter term will be present only if $\frac{C}{n_j}\log N > 2$. So z is $O(\frac{n_j}{C} + \frac{\log N}{\log(\frac{C}{n_j}\log N)})$ with probability at least $1 - 1/N$.

So with probability at least $1 - \log(dq)/N$ the total number of requests received by m due to internal nodes will be of the order of

$$
\begin{aligned}
&\sum_{j=0}^{\log(dq)-1} 2^{j+1}\left(\frac{n_j}{C} + \frac{\log N}{\log(\frac{C}{n_j}\log N)}\right) \\
=&\sum_{j=0}^{\log(dq)-1} 2^{j+1}\frac{n_j}{C} + \sum_{j=0}^{\log(dq)-1} 2^{j+1}\frac{\log N}{\log(\frac{C}{n_j}\log N)} \\
\leq& 2\rho\log_d C + \sum_{j=1}^{\log(dq)-1} 2^{j+1}\frac{\log N}{\log(\frac{2^j}{2\rho}\log N)} \\
&+ 2\frac{\log N}{\log(\frac{C}{n_0}\log N)} \\
\leq& 2\rho\log_d C + \sum_{j=1}^{\log(dq)-1} 2^{j+1}\frac{\log N}{\log(\frac{1}{\rho}\log N) + j - 1} + 2\log N \\
=& 2\rho\log_d C + O\left(\frac{dq\log N}{\log(\frac{dq}{\rho}\log N)} + \log N\right)
\end{aligned}
$$

By combining the high probability bounds for internal and leaf nodes, we can say that a machine gets

$$
\rho\left(2\log_d C + O\left(\frac{\log N}{\log\log N}\right)\right) + O\left(\frac{dq\log N}{\log\left(\frac{dq}{\rho}\log N\right)} + \log N\right)
$$

requests with probability at least $1 - O(\frac{\log dq}{N})$. Replacing N by $N\log(dq)$ in the above expression and simplifying we get Theorem 3.2.2.

Tightness of the high probability bound In this section we show that the high probability bound we have proven for the number of requests received by a machine m is tight.

Lemma 3.2.5 *There exists a distribution of R requests to pages so that a given machine m gets $\Omega(\rho \log_d C + \rho \frac{\log N}{\log \log N} + \frac{dq \log N}{\log(\frac{dq}{\rho} \log N)})$ requests with probability at least $1/N$.*

Proof: To show that the bounds are tight up to constant factors, we need only show distributions that give rise to each of the terms.

If each of the R requests is made for a different page, then each one gives rise to $\log_d C$ requests up their respective trees. So the total number of requests generated will be $R \log_d C$ and the expected number of requests received by m is $\rho \log_d C$. This justifies the presence of the $\rho \log_d C$ term in the bound.

To justify the $\frac{dq \log N}{\log(\frac{dq}{\rho} \log N)}$ term, we let the adversary divide the R requests equally among $R/(d^2 q)$ pages so that each page gets $d^2 q$ requests. By Chernoff bounds, with probability $\Omega(1)$ all the second level nodes in a particular one of these pages' trees receive $\Omega(dq)$ requests. The probability that machine m is present at a given second level node of a particular abstract tree is $1/C$. The total number of second level abstract nodes over all these trees is $d \cdot R/(d^2 q) = R/(dq)$ So the probability that m is present at x of these $\frac{R}{dq}$ second level nodes is at least $\binom{R/(dq)}{x}(1/C)^x(1-1/C)^{R/dq - x}$. To reduce this probabilility to $1/N$, x must be $\Omega(\frac{\log N}{\log(\frac{dq}{\rho} \log N)})$. A given second level node of these pages is expected to receive dq requests. So with probability $\Omega(1/N)$ machine m receives $\Omega(\frac{dq \log N}{\log(\frac{dq}{\rho} \log N)})$ requests.

Finally, for the $\frac{\rho \log N}{\log \log N}$ term, we let the adversary devote all the R requests to one hot page. Then since there are about C leaf positions, each leaf node gets ρ requests in expectation. Also, with probability $1/N$, at least one machine will occupy $\frac{\log N}{\log \log N}$ of these leaf positions and will receive $O(\frac{\rho \log N}{\log \log N})$ requests in expectation. $\square$

Analysis for k-way Independent h

So far we have assumed that our hash function h is perfectly random. However, in practice, hash families are k-universal for some small k. We now extend our high probability analysis to functions h that are chosen at random from a k-universal hash family. We first prove the following general lemma which allows high probability results on a sum of fully independent random variables to be extended to the case when these random variables are k-way independent.

Lemma 3.2.6 *Let $H = \sum_{i=1}^{n} H_i$, where H_i are random varibles. Let us assume that the following high probability bound holds on H when these $H_i^{\prime}s$ are fully independent*

$$H \leq f(N) \textit{ with probability at least } 1 - 1/N$$

where N is a parameter. Further, f is an increasing function of N, and satisfies the following property: For any positive integer i,

$$f(N^i) \leq i \cdot f(N)$$

Now let J be the sum of the same random variables, H_i, which are now instead k-way independent. Then for $k = \log N$, the following high probability hound holds on J:

$$J \textit{ is } O(f(N)) \textit{ with probability at least } 1 - 1/N$$

Proof:

For a sum of k-way independent random variables we can apply the Markov inequality to the k^{th} power of the sum.

$$Pr[J > t] < \frac{E[J^k]}{t^k}$$

t is the high probability bound we wish to prove on J. We will choose t such that the probability of J exceeding t is at most $1/N$.

$$t > (E[J^k])^{1/k} \cdot N^{1/k} \tag{3.2}$$

We need a bound for $E[J^k]$.

Now, J^k can be expanded into a sum of products where each product term will consist of at most k of the H_i. But since the H_i are k-way independent we have $E[J^k] = E[H^k]$. To bound $E[H^k]$ we will make use of the fact that H exceeds $f(N)$ with probability at most $1/N$. This tells us that H lies between $f(N^i)$ and $f(N^{i+1})$ with probability at most $1/N^i$. So

$$E[H^k] \;\leq\; (f(N))^k + \sum_{i=1}^{\infty} \frac{(f(N^{i+1}))^k}{N^i}$$

Recall that $f(N^i) \leq i \cdot f(N)$ and $k = \log N$. So,

$$\begin{aligned}
E[H^k] &\leq (f(N))^k + \sum_{i=1}^{\infty} \frac{((i+1)\cdot f(N))^k}{N^i} \\
&\leq (f(N))^k (1 + \sum_{i=1}^{\infty} \frac{(i+1)^{\log N}}{N^i}) \\
&\leq (f(N))^k (1 + \sum_{i=1}^{\infty} \frac{N^{\log(i+1)}}{N^i}) \\
&\leq (f(N))^k (1 + \sum_{i=1}^{\infty} N^{\log(i+1)-i})
\end{aligned}$$

The sum of the terms $N^{\log(i+1)-i}$ decreases by a factor of at least $N^{1/2}$ after $i \geq 4$. So after $i = 4$ the sum starts behaving like a geometric sum. Also the largest term in the sum is no more than N. So the sum is $O(N)$.

Thus we have, $(E[H^k])^{1/k}$ is $f(N)(O(N))^{1/k}) = O(f(N))$. Substituting in equation 3.2 we get

$$t \leq O(f(N))$$

So we have shown that J is $O(f(N))$ with probability at least $1 - 1/N$. □

The above lemma can be used to extend the high probability bound proved in Theorem 3.2.2 to the case when $\log N$-way independent hash functions are used.

Corollary 3.2.7 *The high probability bound proved in theorem 3.2.2 for the number of requests a cache gets holds up to constant factors if h is selected from a $\log N$-universal hash family.*

Proof: The number of requests received by machine m can be expressed as a weighted sum of Bernoulli random variables. We associate a bernoulli random variable $U_{i,p}$ with the event that the i^{th} abstract node in the tree for page p is mapped to m. We attach a weight $w_{i,p}$ with $U_{i,p}$ that is the number of requests received by this node. If h is fully independent then these $U_{i,p}$'s also are fully independent. However, if h is k-way independent then so are the $U_{i,p}$'s. We will now apply lemma 3.2.6. Let $f(N)$ denote the bound proved in theorem 3.2.2. Since $f(N)$ grows logarithmically in N, it satisfies the condition $f(N^i) \leq i \cdot f(N)$. So we deduce that the same bound holds up to constant factors if we h is chosen from a $\log N$-universal hash family. $\square$

We will use this idea on all the bounds that we will prove hence forth.

3.2.3 An Improvement in the protocol for better bounds

Observe that the $\frac{\rho \log N}{\log \log N}$ term in the bound proved in Theorem 3.2.2 occurs due to the requests arising at leaf nodes. Since there are about C leaves, some machine m occurs $O(\frac{\log N}{\log \log N})$ times in the leaf nodes of a tree, with probability at least $1/N$. What if we modify the protocol slightly so that instead of having about C leaf nodes there are about $C \log(MC)$ leaf nodes, where M is a parameter to be set later? It turns out that instead of having C abstract nodes in each tree, if we have $A = C \log(MC)$ abstract nodes we can get rid of the $\frac{\rho \log N}{\log \log N}$ term in the bound by choosing $M = \rho N$. This definitely has an impact on the latency of getting a page, but this impact is not significant because the latency will now be $\log_d(C \log(\rho N C))$, which is $O(\log_d C)$ if N is $C^{O(1)}$

Theorem 3.2.8 *If each abstract tree has $A = C \log(MC)$ abstract nodes and if h is chosen uniformly at random from the space of function $\mathcal{P} \times [1 \ldots A] \mapsto \mathcal{C}$ then for $M = \rho N$ with probability at least $1 - 1/N$, where N is a parameter, the number of requests a given cache gets is less than*

$$\rho \cdot 2 \log_d A + O\left(\frac{dq \log N}{\log\left(\frac{dq}{\rho} \log N\right)} + \log N \right)$$

Proof: The proof is similar to that of Theorem 3.2.2. Again we can divide the analysis into two parts. First let us analyse the number of requests a cache gets due to its presence at at leaf nodes.

Requests to Leaf Nodes

Each page's tree has about $A(1-1/d)$ leaf nodes. Since a machine m has a $1/C$ chance of occurring at a particular leaf node, it will occur at $O(\log(MC))$ leaf nodes on an expectation. and by Chernoff bounds, it will occur at $O(\log(MC) + \log N)$ leaf positions with a high probability of $1 - 1/N$. Further, since there are at most R requests, m will occur in $O(\log(MC)) + \log N)$ leaf nodes in all those requested pages' trees with probability $1 - R/N$.

Given an assignment of machines to leaf nodes so that m occurs $O(\log(MC)+\log N)$ times in each tree, the expected number of requests m gets is $R \cdot \frac{1}{A} \cdot O(\log(MC)+\log N)$ which is $O(\rho(1+\frac{\log N}{\log(MC)}))$. Also, once the assignment of machine to leaf nodes is fixed, the number of requests m gets is a sum of independent Bernoulli variables. So by Chernoff bounds m gets $O(\rho(1+\frac{\log N}{\log(MC)})+\log N)$ requests with probability $1-1/N$. So we conclude that m gets $O(\rho(1+\frac{\log N}{\log(MC)})+\log N)$ requests with probability at least $1-(R+1)/N$. Replacing N by $N(R+1)$, we get a bound of $O(\rho(1+\frac{\log(NR)}{\log(MC)})+\log N)$ which holds with probability $1-1/N$.

As for the number of m gets due to its presence at internal nodes, by doing exactly the same analysis as in proof of Theorem 3.2.2 we get a bound of $2\rho \log_d A + O((\frac{dq \log N}{\log(\frac{dq}{\rho}\log N)})+\log N)$ which holds with probability at least $1-1/N$.

Combining the bounds on requests due to leaf and internal nodes we get a bound of $2\rho\log_d A + O((\frac{dq\log N}{\log(\frac{dq}{\rho}\log N)})+\log N)+O(\rho(1+\frac{\log(NR)}{\log(MC)})+\log N)$. By setting $M=\rho N$ we get the desired result. $\square$

3.2.4 Storage

In this section, we discuss the amount of storage each cache must have in order to make our protocol work. The amount of storage required at a cache is simply the number of pages for which it receives more than q requests acting as a particular abstract node. Besides, m needs to keep a counter for each page that is requested at each abstract node that gets mapped to m. The total number of counters that will be maintained over all caches is at most the number of requests R. Since the space required to maintain a counter is negligible in comparison to the space required for a typical page, we expect the latter to dominate the storage requirements of a cache. The following lemma bounds the total number of cached pages in the system and per cache.

Lemma 3.2.9 *The total number of cached pages, over all machines, is* $O(\log N + \frac{R}{q})$ *with probability at least* $1-1/N$. *A given cache* m *has* $O(\frac{\rho}{q}+\log N)$ *cached pages with probability at least* $1-1/N$.

Thus the number of cached copies is proportional to the number of requests R and the constant of proportionality can be made small by choosing q large. Also $\rho = R/C$ is the "average" number of requests per cache. So the ρ/q term in the expression for the number of cached pages makes sense.

The rest of the section is devoted to proving this lemma. As before, we perform the analysis by assuming that abstract tree nodes are randomly mapped to caches, and deduce from lemma 3.2.6

that the same bound holds when the mapping is done via a k-wise independent hash function, for $k = \log N$. We show that the total number of cached pages, over all abstract nodes, is $O(\ln N + \frac{R}{q})$ with a high probability of $1 - 1/N$. It follows from a standard balls-in-bins argument that with probability $1-1/N$, the number of cached pages at a machine is $O(R/(qC)+\log N) = O(\rho/q+\log N)$.

We begin by studying the distribution of weights (storage counts) at the nodes of a particular abstract tree. Consider a certain abstract tree T_p for a given page p. Suppose there are r_p requests for page p. For an abstract node at level ℓ, the *expected* number of requests the node receives is r_p/d^{ℓ}. This quantity drops by d at every level. Thus, there is a certain level at which the expectation is at most q/e and at least $q/(ed)$. We call this the *threshold level* of the given page. We bound the number of cache copies in two parts: the number above the threshold level, and the number below.

Above the threshold level, we make the pessimistic assumption that every abstract node receives q requests and therefore caches the page. Since the number of nodes per level is decreasing geometrically, the total number of nodes anywhere above the threshold level is at most $d/(d-1)$ times the number of nodes at the level above the threshold. By definition, the number of nodes at the threshold level for page p is at most $r_p e/q$. Thus, the number of nodes above the threshold level is at most $r_p e/(qd)$ and the total in all levels above is at most $r_p e/(q(d-1))$. Thus, the total number of copies of *all* pages cached above their own thresholds is only

$$\sum_p \frac{r_p e}{q(d-1)} = \frac{Re}{q(d-1)}$$

Our remaining task is to bound the number of cache copies at and below the threshold level. To do so we use a generating function argument for each level separately. We begin with the threshold level. Note that we are throwing $r = r_p$ balls (requests) at $b = d^{\ell}$ bins (abstract nodes). A bin "counts" (causes a cache copy) if it receives at least q requests. Now the probability that j bins receives at least q balls is at most the probability that some set of j bins receives a total of qj balls, which is at most

$$\begin{aligned}
\binom{b}{j}\binom{r}{qj}(j/b)^{qj} &\leq \frac{b^j}{j!}(\frac{er}{qb})^{qj} \\
&\leq \frac{b^j}{j!}(\frac{e}{q}\frac{r}{b})^{qj} \\
&\leq \frac{b^j}{j!}(\frac{e}{q}\frac{q}{e})^{qj} \\
&= \frac{b^j}{j!}
\end{aligned}$$

Let us consider the generating function $\sum \phi_j x^j$, where ϕ_j is the probability that exactly j bins get more than q balls. We upper bounded ϕ_j above, so we deduce that this generating function is upper bounded (term by term) by the generating function for the above sequence, namely $\exp(bx)$. From the fact that at the threshold level, $r/b \leq q/e$, we deduce that this function is upper bounded term-by-term by $\exp((re/q)x)$. Now let us consider the probability generating function for the number of threshold nodes over *all* pages that receive more than q requests. This is simply the product of the PGFs for all the pages, and is therefore upper bounded by the product of the generating functions given above, namely

$$\begin{aligned}
\prod_p \exp((re/q)x) &= \exp(\textstyle\sum_r (re/q)x) \\
&= \exp((Re/q)x)
\end{aligned}$$

Next let's consider the level below threshold. This has d times as many nodes, so the expected number of requests per machine divides by a factor of d. What impact does this have on the above analysis? Well, we used the expectation in only one place, where we replaced r/b by q/e. At one level below the threshold, we can replace r/b by $q/(ed)$. We then continue the same analysis as before and get a bound of $\exp((Re/(qd))x)$ on the probability generating function provided $q \geq 2$. Similarly, at two levels below threshold the probability generating function is bounded by $\exp((Re/(qd^2))x)$, and so on. So the probability generating function for the total number of abstract nodes over all pages, below threshold levels, that receive more than q requests is bounded by

$$\prod_{i\geq 0}\exp((Re/(d^i q))x) = \exp(\sum_{i\geq 0}(Re/(d^i q))x) \leq \exp(\frac{Red}{((d-1)q)}x)$$

It follows that the probability there are more than j cache copies at threshold levels is at most

$$\frac{(Red/((d-1)q))^j}{j!}$$

This quantity is $1/N$ when

$$j = O(\ln N + R/q)$$

So we have shown that with probability $1-1/N$ number of cached pages at threshold levels and below is $O(\ln N + \frac{R}{q})$.

Chapter 4

Consistent Hashing

In this chapter we define a new hashing technique called *consistent hashing*. We motivate this technique by reference to a simple scheme for data replication on the Internet. Consider a single server that has a large number of objects that other clients might want to access. It is natural to introduce a layer of caches between the clients and the server in order to reduce the load on the server. In such a scheme, the objects should be distributed across the caches, so that each is responsible for a roughly equal share. In addition, clients need to know which cache to query for a specific object. The obvious approach is hashing. The server can use a hash function that evenly distributes the objects across the caches. Clients can use the hash function to discover which cache stores a object. Consider now what happens when the set of active caching machines changes, or when each client is aware of a *different* set of caches. (Such situations are very plausible on the Internet.) If the distribution was done with a classical hash function (for example, the linear congruential function $x \rightarrow ax + b \pmod{p}$), such inconsistencies would be catastrophic. When the range of the hash function (p in the example) changed, almost every item would be hashed to a new location. Suddenly, all previously cached data is useless because clients are looking for it in a different location.

Consistent hashing solves this problem of different "views." We define a *view* to be the set of caches of which a particular client is aware. We assume that while views can be inconsistent, they are substantial: each machine is aware of a constant fraction of the currently operating caches. A client uses a consistent hash function to map a object to one of the caches in its view. We analyze and construct hash functions with the following consistency properties. First, there is a "smoothness" property. When a machine is added to or removed from the set of caches, the expected fraction of objects that must be moved to a new cache is the minimum needed to maintain a balanced load across the caches. Second, over all the client views, the total number of different caches to which an object is assigned, which we call "spread", is small. Similarly, over all the client views, the number of distinct objects assigned to a particular cache, which we call "load", is small.

Consistent hashing therefore solves the problems discussed above. The "spread" property implies that even in the presence of inconsistent views of the world, references for a given object are directed only to a small number of caching machines. Distributing a object to this small set of caches will insure access for all clients, without using a lot of storage. The "load" property implies that no one cache is assigned an unreasonable number of objects. The "smoothness" property implies that smooth changes in the set of caching machines are matched by a smooth evolution in the location of cached objects.

In section 4.1 we define a "ranged hash function" and then precisely define several quantities that capture different aspects of "consistency". In section 4.2 we construct practical hash functions which exhibit all four to some extent.

4.1 Definitions

In this section, we formalize and relate four notions of consistency.

Let $\mathcal{D}$ be a domain of items and $\mathcal{B}$ be a set of buckets. A *view* is any subset of the buckets $\mathcal{B}$.

A *ranged hash function* is a function of the form $f : 2^{\mathcal{B}} \times \mathcal{D} \to \mathcal{B}$. Such a function specifies an assignment of items to buckets for every possible view. That is, $f(\mathcal{V}, i)$ is the bucket to which item i is assigned in view $\mathcal{V}$. (We will use the notation $f_{\mathcal{V}}(i)$ in place $f(\mathcal{V}, i)$ from now on.) Since items should only be assigned to usable buckets, we require $f_{\mathcal{V}}(\mathcal{D}) \subseteq \mathcal{V}$ for every view $\mathcal{V}$.

A *ranged hash family* is a family of ranged hash functions. A *random ranged hash function* is a function drawn at random from a particular implicitly specified ranged hash family.

In the remainder of this section, we state and relate some reasonable notions of consistency regarding ranged hash families. Throughout, we use the following notational conventions: $\mathcal{F}$ is a ranged hash family, f is a ranged hash function, $\mathcal{V}$ is a view, i is an item, and b is a bucket. In the following discussion we will focus on a particular subset of items $\mathcal{I} \subseteq \mathcal{D}$. Let $I = |\mathcal{I}|$ be the number of items in the subset. In this chapter we will assume that C denotes the total number of buckets in $\mathcal{B}$. This is because in chapter 5 the set of caches will be the set of buckets.

Load: Define a set of V views as before. For a ranged hash function f and bucket b, the *load* $\lambda(b)$ is the quantity $\left|\bigcup_{\mathcal{V}} f_{\mathcal{V}}^{-1}(b)\right|$, that is, the number of distinct items that get mapped to b, over the different views.(Note that $f_{\mathcal{V}}^{-1}(b)$ is the set of items from $\mathcal{I}$ assigned to bucket b in view $\mathcal{V}$.)

The load of a hash function $\lambda(f)$ is the maximum load of a bucket. The *load* of a hash family, $\lambda(\mathcal{F})$, is a random variable that is the load of a randomly chosen hash function from the family. The property says that there are at most $\lambda(b)$ distinct items that at least one person thinks belongs in the bucket b. Since we would like the mapping of items to buckets to be uniform, a good consistent hash function should also have low load.

Spread: Let $\mathcal{V}_1 \ldots \mathcal{V}_V$ be a set of views, altogether containing C distinct buckets and each individually containing at least C/t buckets. For a ranged hash function and a particular item i, the *spread* of i, $\sigma(i)$ is the quantity $\left|\{f_{\mathcal{V}_j}(i)\}_{j=1}^{V}\right|$, that is the number of distinct buckets i gets mapped to, over the different views.The *spread* of a ranged hash function $\sigma(f)$ is the maximum spread of an item. The *spread* of a hash family, $\sigma(\mathcal{F})$, is a random variable that is the spread of a randomly chosen hash function from the family.

The idea behind spread is that there are V people, with different views of a set of buckets. Each person tries to assign an item i to a bucket using a consistent hash function. The property says that across the entire group, there are at most $\sigma(i)$ different opinions about which bucket should contain the item. Clearly, a good consistent hash function should have low spread over all items.

Balance: A ranged hash family is *balanced* if, given a view containing all the C buckets and a given item i, and a randomly chosen function selected from the hash family, the probability that i is mapped to a given bucket in the view is $O(1/C)$.

Monotonicity: A ranged hash function f is *monotone* if for all views $\mathcal{V}_1 \subseteq \mathcal{V}_2 \subseteq \mathcal{B}$, $f_{\mathcal{V}_2}(i) \in \mathcal{V}_1$ implies $f_{\mathcal{V}_1}(i) = f_{\mathcal{V}_2}(i)$. A ranged hash family is *monotone* if every ranged hash function in it is.

This property says that if items are initially assigned to a set of buckets $\mathcal{V}_1$ and then some new buckets are added to form $\mathcal{V}_2$, then an item may move from an old bucket to a new bucket, but not from one old bucket to another.

Our main result for consistent hashing is Theorem 4.4.1 which shows the existence of an efficiently computable balanced monotonic ranged hash family with logarithmic spread and load.

4.2 Construction

We give a construction of a ranged hash family with good properties. We will use two random functions $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$ that map items and buckets to real numbers in the range $[0,1]$ respectively. Since it is hard to implement completely random functions in practice, we only demand that the functions $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$ map points $\Omega(\log(NCV))$-way independently where N is the high probability parameter. The function $r_{\mathcal{B}}$ maps buckets randomly to the unit interval, and $r_{\mathcal{I}}$ does the same for items. $f_{\mathcal{V}}(i)$ is defined to be the bucket $b \in \mathcal{V}$ that minimizes $|r_{\mathcal{B}}(b) - r_{\mathcal{I}}(i)|$. In other words, i is mapped to the bucket "closest" to i. This will create an interval around each bucket so that it is responsible for all the items falling in that interval. Since the items and buckets are mapped randomly, the items will tend to be uniformly distributed among the buckets. Let us consider what happens when a new bucket is added. Some of the items around this new buckets that were earlier assigned to other buckets will now move to this new bucket. For reasons that will become apparent, we actually need to have more than one point in the unit interval associated with each bucket. Assuming that the number of buckets in the range is always less than C, we will need $\kappa \log(C)$ points for each bucket for some constant κ. The easiest way to view this is that each bucket is replicated $\kappa \log(C)$ times, and then $r_{\mathcal{B}}$ maps each replicated bucket randomly. Denote the above described hash family as $\mathcal{F}$. Clearly if there is no lower bound on the size of each view then the spread and load can be made as large as I by making view sizes equal to one. So we will also assume that each view consists of at least a certain fraction of the buckets.

4.3 Implementation

In this section we show how the hash family just described can be implemented efficiently. Specifically, the expected running time for a single hash computation will be $O(1)$. The expectation is over the choice of hash function. The expected running time for adding or deleting a bucket will be $O(\log C)$ where C is the total number of buckets in all views.

A simple implementation uses a balanced binary search tree to store the points on the unit interval corresponding to the buckets. To find the bucket to which item i is mapped, one simply needs search for $r_I(i)$ in this search tree. This will give the desired bucket that is closest to i on the real interval. If there are C buckets, then there will be $\kappa C \log(C)$ intervals, so the search tree will have depth $O(\log(C))$. Thus, a single hash computation takes $O(\log(C))$ time. The time for an addition or removal of a bucket is $O(\log^2(C))$ since we insert or delete $\kappa \log(C)$ points for each bucket.

The following trick reduces the expected running time of a hash computation to $O(1)$. The idea is to divide the interval into roughly $\kappa C \log(C)$ equal length segments, and to keep a separate search tree for each segment. Thus, the time to compute the hash function is the time to determine which interval $r_{\mathcal{I}}(i)$ is in, plus the time to lookup the bucket in the corresponding search tree and its two neighbors. The first time is always $O(1)$. Since the expected number of points in each segment is $O(1)$, the second time is $O(1)$ in expectation.

Another practical limitation is that hash functions do not hash to real numbers, which we have assumed of r_I and r_B. However we can think of these random real numbers as a sequence of random bits and only pick enough random bits to distinguish the point from all other points. It turns out that with high probability it suffices to compute only the first $O(\log(I+B)$ bits of each real number to distinguish them.

4.4 Analysis

The following theorem proves that the hash family described above has the desired properties of a consistent hash family.

Theorem 4.4.1 *If the functions $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$ are $\Omega(\log(NVC))$-way independent and if each view contains at least a $1/t$ fraction of the buckets then a random function f chosen from the the ranged hash family $\mathcal{F}$ described above has the following properties:*

1. *$\mathcal{F}$ is monotone.*
2. *Spread: For any item $i \in \mathcal{I}$, $\sigma(i)$ is $O(t\log(NV))$ with probability greater than $1 - 1/N$.*
3. *Load: For any bucket $b \in \mathcal{B}$, $\lambda(b)$ is $O((1 + I/C)t\log(NV))$ with probability greater than $1 - 1/N$.*
4. *Balance: With probability at least $1 - 1/C^{\Omega(1)}$, for a fixed view $\mathcal{V}$ containing C buckets, $\Pr[f_{\mathcal{V}}(i) = b] \le O(\frac{1}{C})$ for $i \in \mathcal{I}$ and $b \in \mathcal{V}$*

We will prove the above theorem in the rest of this section. Note that the monotonicity is immediate. When a new bucket is added, the only items that move are those that are now closest to that bucket's associated points. No items move between old buckets

The proof of theorem 4.4.1 requires the following technical lemma from [12] that gives upper bounds on a sum of Bernoulli variables when these variables are only k-way independent.

Lemma 4.4.2 *If X is the sum of k-wise independent binary random variables, with $\mu = E[X]$, then*

(I) for $\delta \leq 1$ and $k \leq \lfloor \delta^2 \mu e^{-1/3} \rfloor$, $Pr(\mid X - \mu \mid \geq \delta\mu) \leq e^{-\lfloor k/2 \rfloor}$

(II) for $\delta \geq 1$ and $k = \lfloor \delta \mu e^{-1/3} \rfloor$, $Pr(\mid X - \mu \mid \geq \delta\mu) \leq e^{-\lfloor \delta\mu/3 \rfloor}$

We now proceed to prove the claims of theorem 4.4.1 as a series of Lemmas. We first show the spread and load properties. We begin by looking at a large enough interval on the unit real interval so that with high probability every view has at least one bucket point in that interval. The following lemma specifies the length of such an interval.

Lemma 4.4.3 *If we consider at an interval of length $\frac{3t\log(NV)}{C\kappa\log(C)}$ on the unit real interval then with probability at least $1 - 1/N$ every view has at least one bucket point in that interval.*

Proof: Let X_i be the random variable denoting the number of points from buckets in view $\mathcal{V}_i$ in an interval of length l. There are at least $M = C\kappa\log(C)/t$ bucket points associated with each view. We can assume that there are exactly M points for each view since this is the worst case, so we have $E[X_i] = Ml$. We will choose the value of l such that the probability of X_i being 0 is at most $1/NV$. If the points are $k = 2\log(NV)$-way independent and if $Ml = 3t\log(NV)$, then there exists a $\delta < 1$ such that conditions of lemma 4.4.2 apply. So we have

$$
\begin{aligned}
\Pr[X_i = 0] &\leq \Pr[\,|X_i - Ml| > \delta Ml] \\
&\leq e^{-k/2} \\
&= \frac{1}{NV}
\end{aligned}
$$

So we choose

$$l = \frac{3t\log(NV)}{C\kappa\log(C)}$$

From the union bound we get:

$$\Pr[\ \textit{Some}\ X_i = 0] \leq \sum_{i=1}^{V} \Pr[X_i = 0] = V\frac{1}{NV} = \frac{1}{N}$$

So we have shown that in every view, some bucket falls within an interval of length l. $\square$

Lemma 4.4.4 *Spread: For any item $i \in \mathcal{I}$, $\sigma(i)$ is $O(t\log(NV))$ with probability* $1 - 1/N$.

Proof:

Fix an $i \in \mathcal{I}$. We know from lemma 4.4.3 that if we consider an interval of length $l = \frac{3t\log(NV)}{C\kappa\log(C)}$ centered at itm then with probability at least $1-1/N$ every view will have at least one bucket point in that interval. So in any view i gets mapped only to a bucket in that interval. We can get a bound on $\sigma(i)$ by bounding the number of bucket points that fall in that interval.Let X be a random variable denoting the number of points in this interval of length l around $r_I(i)$ coming from the union of all the views. Since there are $C\kappa\log(C)$ points in total, we have $E[X] = Cl\kappa\log(C) = 3t\log(NV)$. Also if the mapping is $2\log(NV)$-way independent, we can find a $\delta \leq 1$ so that lemma 4.4.2 applies. It follows that.

$$\begin{aligned} &\Pr[\ |\sigma(i) - 3t\log(NV)| > 3t\log(NV)] \\ &\leq \ e^{-k/2} \\ &= \ \frac{1}{NV} \end{aligned}$$

So we have proved that $\sigma(i)$ is $O(t\log(NV))$ with probability at least $1 - 1/N - 1/NV$. Since this bound holds for all N, we can replace N by $N(1+1/V)$, which gives us the desired result. $\square$

Lemma 4.4.5 *Load: For a bucket $b \in \mathcal{B}$, $\lambda(b)$ is $O((1 + I/C)t\log(NV))$ with probability* $1 - 1/N$.

Proof: For each bucket b there are $\kappa\log(C)$ points on the unit interval. An item is assigned to b only if it is closest to one of these points among all the bucket points. Around each of the $\kappa\log(C)$ points associated with b we will consider an interval containing at least one other point associated with some other bucket, on either side. This gives as a collection of $\kappa\log(C)$ intervals and any item that maps to b must fall in this collection of intervals. We will first bound the total length of this collection of intervals and then bound the number of items that fall in any of them. Thus we get a bound on the load of b.

Fix one of the $\kappa \log(C)$ points associated with b, call this point y. We know from lemma 4.4.3 that if we look at an interval of length $l = \frac{3t\log(NV)}{C\kappa\log(C)}$, with its left end on y, we have with probability at least $1 - 1/N$ that in every view there is a bucket in that interval. Similarly, if we look at an interval of the same length with its right end on y, we have with probability at least $1 - 1/N$ that in every view there is a bucket in that interval. Thus, with probability at least $1 - 2/N$ we have a bucket within distance l on the right of y, and to the left of y. Assuming this event occurs, y is "responsible" for no more than a segment of length l of the interval. That is, items mapped into the interval of length l centered at y will be closest to the point y and therefore will be placed in the bucket b. If y is closer than l to one of the endpoints of the unit interval, then y can only be responsible for *less* than length l of the interval. It follows from the union bound that in all views the $\kappa \log(C)$ points associated with b are responsible together for an interval of length no more than $\kappa \log(C) l = 3t\log(NV)/C$ with probability at least $1 - 2\kappa \log C/N$. Call this event A.

Assuming the event A has occurred, then the load of b can be bounded by the number of *items* that are mapped into a set of measure $3t\log(NV)/C$ in the interval. As in the proof of spread, we define X to be a random variable equal to the number of items in the interval. Since there are I items we have $E[X] = I\frac{3t\log(NV)}{C}$. We will now apply lemma 4.4.2. We will consider two two cases, $I \leq C$ and, $I \geq C$, applying part (I) of lemma 4.4.2 in the former and part (II) in the latter case. In either case it follows that if $k = \Omega(\log(NV))$ then X is $O((1 + I/C)t\log(NV))$ with probability at least $1 - 1/(NV)$.

So with probability at least $1 - 2\kappa \log C/N - 1/NV$, the load of b is $O((1+I/C)t\log(NV)$. Since this bound holds for all N we can replace N by $O(N\kappa \log C)$, thus giving the desired result. $\square$

The proof of the above lemma implies the following corrollory which is useful in the rest of the paper.

Corollary 4.4.6 *With probability at least* $1-1/N$ *the mapping of buckets will be such that* $\Pr[f_{\mathcal{V}}(i) = b$ *in* any *view* $V] = \frac{O(t\log(NV))}{C}$ *for* $i \in \mathcal{I}$ *and* $b \in \mathcal{B}$.

Proof: We saw in the proof of lemma 4.4.5 that the total length of intervals for which a bucket b is responsible is $3t\log(NV)/C$ with probability at least $1 - 2\kappa \log C/N$. Since a given item i is mapped to a random point on the unit interval, it has a $3t\log(NV)/C$ chance of being assigned to bucket b. Replacing N by $N2\kappa \log C$ proves the corrollary. $\square$

It remains to show the balance property of the hash family. Note that the probability of an item getting assigned to a particular bucket is exactly the total length of the parts of the unit interval that bucket is responsible for. The following lemma bounds the section of the unit interval that each bucket is assigned responsibility for.

Lemma 4.4.7 *Assume $\kappa \log(C)$ points are mapped to the unit interval $\Omega(\log(NVC))$-way independently for each of C buckets. For a bucket b denote by $length(b)$ the measure of the set of points that are closer to one of the y points than any other bucket point. Then with probability at least $1 - 1/N$, $length(b)$ is $O(\frac{1}{C}\frac{\log(NC)}{\log C})$.*

Proof:

Let b be a bucket and x be a point on the unit interval. If x lies to the right of a point associated with b and is closer to that point than to any point of any other bucket, then say that b is *right-responsible* for x.

The main result is that the probability a single bucket b is right-responsible for more than a $O(\frac{1}{C}\frac{\log(NC)}{\log C})$ fraction of the unit interval is at most $1/(NC)$. A union bound then implies that none of the C buckets is right-responsible for more than $O(\frac{1}{C}\frac{\log(NC)}{\log C})$ fraction of the unit interval, with a high probability of $1 - 1/N$. By symmetry, no bucket is "two-sided responsible" for more than twice of what it is right-responsible, which is still $O(\frac{1}{C}\frac{\log(NC)}{\log C})$.

To show the main result, begin by fixing a bucket b. The portion of the unit interval for which b is right-responsible must consist of $\kappa logC$ non-overlapping intervals in $[0, 1]$, each bounded on the left by one of b's points. Suppose we shrink all intervals by moving the right endpoints leftward, until the length of every interval is a multiple of $\Delta = 4/(C \log C)$. Since there are $\kappa \log C$ of these intervals and each shrinks by at most Δ, the decrease in the total length of all intervals is at most $\frac{4\kappa}{C}$. So if the total length of these intervals after shrinking is $a \log C \cdot \frac{1}{\kappa C \log C}$ (a will be set later), then the total length before shrinking is at most $\frac{4(a+\kappa)}{C}$. This implies that if bucket b is right-responsible for a $\frac{4(a+\kappa)}{C}$ fraction of the unit interval, then b must be right-responsible for every point in a collection of non-overlapping intervals, each bounded on the left with one of b's points, each a multiple of Δ in length, and with total length $\frac{4a}{C}$. Now given a collection of intervals of total length $\frac{4a}{C}$ we will bound the probability that these intervals do not get any point associated with some other bucket. The expected number of the $\kappa C \log C$ points falling in this collection of intervals is $4a\kappa \log C$. So, if r_B is $k = 4a\kappa \log C$-way independent, and if X denotes the number of points falling in this collection of intervals, then from Lemma 4.4.2

$$\begin{aligned} Pr[X = 0] &\leq Pr[|X - 4a\kappa \log C| \geq 4a\kappa \log C] \\ &\leq e^{-k/2} \\ &\leq e^{-4a\kappa \log C} \end{aligned}$$

The number of collections of $\kappa \log C$ intervals with total length $\frac{4a}{C}$ and with all lengths multiples of Δ is exactly the number of ways to partition $a \log C$ into $\kappa \log C$ integral parts, which is:

$$\begin{aligned} \binom{a \log C + \kappa \log C}{\kappa \log C} &\leq \left(\frac{e(a+\kappa)\log C}{\kappa \log C}\right)^{\kappa \log C} \\ &\leq (e(1+\kappa/a))^{\kappa \log C} \\ &\leq e^{(\kappa+a)\log C} \end{aligned}$$

By the union bound, the probability that any of the above collections of intervals contains no point associated with other buckets is at most $e^{-2a\kappa \log C} e^{(\kappa+a)\log C} = e^{-(2a\kappa-\kappa-a)\log C}$. We will choose a so that it is at most $1/(NC)$. which gives

$$2a\kappa - \kappa - a = \frac{\log(NC)}{\log C}$$

Also we had assumed that $k = 4a\kappa \log C$ and we wish this to be $O(\log(NC))$. This gives

$$4a\kappa = O(\frac{\log(NC)}{\log C})$$

Since κ is a constant, we can set b so that both the above condition are satisfied. Simply set $a = O(\frac{\log(NC)}{\log C})$. This proves that with probability at least $1 - 1/(NC)$ the total length assigned to a bucket is at most $\frac{4(a+\kappa)}{C} = O(\frac{1}{C}\frac{\log(NC)}{\log C})$ if r_B is $\Omega(\log(NC))$-way independent. Now since there are C buckets the same bound holds for all buckets with probability $1 - 1/N$.

□

Lemma 4.4.8 *For a fixed set of buckets $\mathcal{V}$, with probability at least $1 - 1/N$,* $\Pr[f_{\mathcal{V}}(i) = b] \leq O(\frac{1}{C}\frac{\log(NC)}{\log C})$ *for $i \in \mathcal{I}$ and $b \in \mathcal{V}$, and, conditioned on the choice of r_B, the assignments of items to buckets are $\Omega(log(NC))$-way independent.*

Proof: Fix $i \in \mathcal{I}$ and $b \in \mathcal{V}$. Lemma 4.4.7 says that with probability $1 - 1/N$, the $\kappa \log(C)$ points associated with b are responsible for no more than $O\left(\frac{1}{C}\frac{\log(NC)}{\log C}\right)$ of the interval. The probability that i is mapped to b is equal to this interval length. The first statement of the lemma follows.

Since the function r_I is assumed to map items to points $\log(NC)$-way independently, once we have chosen r_B, the items are assigned $\log(C)$-way independently to intervals, and thus to buckets.

□

Setting $N = C^{O(1)}$ in the previous lemma gives us the following desired balance property.

Corollary 4.4.9 *For a fixed set of buckets $\mathcal{V}$, with probability at least $1 - 1/C^{O(1)}$,* $\Pr[f_{\mathcal{V}}(i) = b] \leq O(\frac{1}{C})$ *for $i \in \mathcal{I}$ and $b \in \mathcal{V}$, and, conditioned on the choice of r_B, the assignments of items to buckets are $\Omega(log(NC))$-way independent.*

Chapter 5

Basic Solution in an Inconsistent World

In this chapter we apply the techniques developed in the last section to the simple hot spot protocol developed in Chapter 3. We now relax the assumption that clients know about all of the caches. We assume only that each machine knows about a $1/t$ fraction of the caches chosen by an adversary. There is no difference in the protocol, except that the mapping h is a consistent hash function. This change will not affect latency. Therefore, we only analyze the effects on swamping and storage. The basic properties of consistent hashing are crucial in showing that the protocol still works well. In particular, the blowup in the number of requests and storage is proportional to the spread and load of the hash function.

5.0.1 Swamping

Theorem 5.0.10 *If h is implemented using the $\Omega(\log(NRC))$-way independent consistent hash function of Theorem 4.4.1 and if each view consists of $C' = C/t$ caches then with probability at least $1 - 1/N$ a given cache gets no more than $O((2\rho t^2 \log_d C' \log(NR)) + (dqt \log(NR) + \rho t) \log N)$ requests.*

Proof: We look at the different trees of caches for different views for one page, p. The number of different views is at most the total number of requests R. Let $C' = C/t$ denote the number of caches in each tree. Under each view the nodes of the abstract tree get mapped to caches differently. We overlay these different trees, obtained by mapping abstract nodes to caches in each view, on one another to get a new tree where in each node, there is a *set* of caches. Due to the *spread* property of the consistent hash function at most $\sigma = O(t \log(NR))$ caches appear at any node in this combined tree with high probability. In fact since there are only R requests, this will be true for the nodes of all the R trees for the requested pages. If $E_{p,j}$ denotes the event that m appears in the j^{th} node of the combined tree for page p then we know from Corollary 4.4.6 that the probability of this event is $O(\lambda/C)$, where λ is the *load* which is $O(t\log(NR))$ with high probability. We condition on the event that σ and λ are $O(t\log(NR))$ which happens with high probability.

Since a cache in a node sends out at most q requests, each node in the combined tree sends out at most $q' = q\sigma$ requests. We now adapt the proof of Theorem 3.2.2 to this case. In Theorem 3.2.2 where every machine was aware of all the C caches, an abstract node was assigned to any given machine with probability $1/C$. We now assign an abstract node to a given machine with probability $O(\lambda/C)$. So we have a scenario with $C' = C/t$ caches where each abstract node sends out up to q' requests to its parent and m occurs at each abstract node independently and with probability $O(\lambda/C)$. The rest of the proof is very similar to that of Theorem 3.2.2.

We analyze the number of requests m gets in the new scenario. As before we split this analysis into two parts. First we analyze the hits on a cache due to its presence in the leaf nodes of the trees and then analyze the hits due to its presence at the internal nodes and then add them up.

Hits at Leaf Nodes

Each tree has $C'(1 - 1/d)$ leaves. Since m is assigned to a leaf with probability $O(\lambda/C)$, it is expected to occur in $O(t\lambda) = O(\log C)$ leaf nodes per tree. In fact by Chernoff bounds it occurs in $O(\log C + \log N)$ leaf nodes in all trees with probability at least $1 - 1/N$. Now given such an assignment of m to leaf nodes, m will get $O(\rho t(\log C + \log N)$ requests out of the total of R requests, on an expectation and in fact with a high probability of $1 - 1/N$. So we can conclude that m gets $O(\rho\lambda + \rho t \log N)$ requests with a probability of at least $1 - 1/N$.

Hits at Internal Nodes

Again as proof of Theorem 3.2.2 we think of the protocol as first running on the abstract trees. Now no abstract internal node gets more than dq' requests because each child node gives out at most q' requests for a page. Consider any arbitrary arrangement of paths for all the R requests up their respective trees. An abstract node can receive at most dq' requests. Let n_j denote the number of abstract nodes that receive between 2^j and 2^{j+1} requests where $0 \le j \le \log(dq') - 1$. Let

$\rho' = \rho t$ denote the ratio of number of browsers to the number of caches per view. Since each of the $R = \rho' C'$ requests gives rise to at most $\log_d C'$ requests up the trees the total number of requests is no more than $\rho' C' \log_d C'$. So,

$$\sum_{j=0}^{\log(dq')-1} 2^j n_j \leq \rho' C' \log_d C' \tag{5.1}$$

As in lemma 3.2.4 we prove that $n_j \leq \frac{2\rho' C'}{2^j}$ for $j > 0$ and for $j = 0$, n_j is at most $\rho' C' \log_d C'$. Now each of these n_j abstract nodes is assigned to m with probability of $\Theta(\lambda/C)$. So as before we show that with probability at least $1 - 1/N$, m is assigned $O(\frac{n_j \lambda}{C'} + \frac{\log N}{\log(\frac{C'}{n_j \lambda} \log N)})$ of these n_j abstract nodes.

So with probability at least $1 - (\log(dq'))/N$ the total number of requests received by m due to internal nodes will be of the order of

$$\sum_{j=0}^{\log(dq')-1} 2^{j+1} (\frac{n_j \lambda}{C'} + \frac{\log N}{\log(\frac{C'}{n_j \lambda} \log N)})$$

Using Equation 5.1 and the bound for n_j, the above expression simplifies to be at most $O(\lambda(2\rho' \log_d C') + dq' \log N)$. So we have proved that with probability at least $1 - (\log(dq'))/N$ the number of requests m gets acting as internal nodes is no more than $O(\lambda(2\rho' \log_d C') + dq' \log N))$ requests. Replacing N by $N \log(dq')$ we get that the same bound holds with probability $1 - 1/N$.

Combining the high probability bounds for internal and leaf nodes tells us that with probability $1 - 1/N$, m gets $O(\lambda(2\rho' \log_d C') + (dq' + \rho') \log N))$ requests.

However so far we had assumed that the $E_{p,j}$'s are fully independent whereas actually they are $\Omega(\log NR)$-way independent. To take this into account we will use lemma 4.4.2, which tells that the same bound holds even if the $E_{p,j}$'s are $\Omega(\log NR)$-way independent. Now since λ and σ are also $O(t \log NR)$ with probability $1 - 1/N$ theorem 5.0.10 follows.

□

5.0.2 Storage

Using techniques similar to those in proof of theorem 5.0.10 we get the following lemma.

Lemma 5.0.11 *The total number of cached pages, over all machines is $O(\sigma(\log R \log_d C + \frac{R}{q}))$ with probability of $1 - 1/N$. A given cache m has $O(\lambda(\rho/q + \log N))$ cached copies with a high probability of $1 - 1/N$.*

Proof:

Again as in the proof of Theorem 5.0.10 we overlay the trees of caches for different views of one page p. Due to the *spread* property of the consistent hash function at most $\sigma = O(t \log(NR))$ caches appear at any node in this combined tree with high probability. Again if $E_{p,j}$ denotes the event that m appears in the j^{th} node of the combined tree for page p then we know from Corollary 4.4.6 that the probability of this event is $O(\lambda/C)$, where λ is the load. So we have a scenario with $C' = C/t$ caches and where m occurs at each abstract node independently and with probability $\Theta(\lambda/C)$. The rest of the proof is very similar to that of Theorem 3.2.9.

As before we define the *threshold level* of a page as the level in the combined tree where each abstract node is expected to receive between q/e and $q/(ed)$. We bound the number of cached copies over all pages above the threshold level and use the generating function argument to bound number of copies at and below the threshold level. Using exactly the same analysis as before we get that the total number of abstract nodes over all pages receiving more than q requests is $O(\log N \log_d C + \frac{R}{q}$ with a probability at least $1 - 1/N$. However, since in in this combined tree each abstract node contains σ machines, the total number of cached copies of pages will get multiplied by a factor of σ, giving a bound of $O(\sigma(\log N \log_d C + \frac{R}{q}))$.

Now let us bound the number of cached copies in a machine m. These $O(\log N \log_d C + \frac{R}{q})$ abstract nodes are assigned to m with a probability of $O(\lambda/C)$. So by Chernoff bounds, with high probability of $1 - 1/N$, m gets $O(\lambda(\log N + \frac{\rho}{q}))$ of these abstract nodes. Again we can use lemma 4.4.2 to get the same bound when h is $\log N$-way independent.

□

Chapter 6

Ultrametric Distances

The assumption that every pair of machines can communicate with equal ease is obviously unrealistic; we adapt our protocol to a more realistic model in this section.

Recall that a request by machine m_1 for page p from machine m_2 has three stages: m_1 asks for the page from m_2, m_2 obtains the page, and m_2 returns the page to m_1. The latency of the page request is defined to be the duration of all three stages. The duration of the first and third stages is a function of the ease of communication between m_1 and m_2.

Modeling the ease of communication between machines on the Internet is tricky. The Internet communications protocol, TCP/IP, gives no formal guarantee on the time to pass a message between two machines. Empirically, this time can vary considerably due to network congestion and changes in routing hardware. However, by compiling statistics on past communications, one may obtain a reasonably accurate "typical" time to pass a packet between machines.

We assume that such typical communication times are available. In particular, if machine m_1 requests a page from machine m_2, then let the duration of the first and third stages of the page request be given by $\delta(m_1, m_2)$. Furthermore, we assume that machine m_1 knows $\delta(m_1, m_2)$ for any machine m_2. But it may not know the distance $\delta(m_2, m_3)$ between two other machines, say m_2 and m_3. Thus the storage required for this information is linear in the number of machines.

The latency of a page request can now be expressed in terms of δ. For example, if a browser b requests a page from a cache c and the cache forwards the request to the server s, then the latency of the page request is $\delta(b, c) + \delta(c, s)$.

We extend our protocol to a restricted class of functions δ. In particular, we assume that δ is an *ultrametric*. Formally, an ultrametric is a metric which obeys a more strict form of the triangle inequality: $\delta(a,c) \leq \max(\delta(a,b),\delta(b,c))$.

The ultrametric is a natural model of internet distances, since it essentially captures the hierarchical nature of the internet topology, under which, for example, all machines in a given university are equidistant, but all of them are farther away from another university, and stil farther from another continent. The logical point-to-point connectivity is established atop a physical network, and it is generally the case that the latency between two sites is determined by the "highest level" physical communication link that must be traversed on the path between them. Indeed, another definition of an ultrametric is as a hierarchical clustering of the points. The distance between two points depends only on which is the smallest cluster containing both. Thus, for example, the distance between any two machines at the same university is less than the distance between any two machines at different universities in the same country.

In addition to modeling communication latency, ultrametrics are also good models of the *throughput* between two machines. For large pages, maximizing throughput is more important than minimizing latency. Throughput is typically determined by the maximum-congestion (physical) communication link on the path implementing the virtual point-to-point connection between two machines and is therefore an ultrametric.

6.1 Protocol

The only modification we make to the protocol is the following: When a browser needs a page p it only uses the caches that are no further away than the server for the page. The size of the abstract tree is now equal the the number of caches within the distance to the server. By doing this, we insure that our path to the server does not contain any caches that are unnecessarily far away in the metric. The mapping is done using a consistent hash function, which is the vital element of the solution.

Clearly, requiring that browsers use "nearby" caches can cause swamping if there is only one cache and server near many browsers. Thus, in order to avoid cases of degenerate ultrametrics where there are browsers that are not close to any cache, and where there are clusters in the ultrametric without any caches in them, we restrict the set of ultrametrics that may be presented to the protocol. The restriction is that in any cluster the ratio of the number of caches to the number of browsers may not fall below $1/\rho$ (recall that $R = \rho C$).For the sake of analysis this restriction is equivalent to imagining that the requests originate at the caches where each cache is allowed to make at most ρ requests. This restriction makes sense in the real world where caches are likely to be evenly spread out over the Internet. It is also necessary, as it is clear that a large number of browsers clustered around one cache can be forced to swamp that cache if we use our modified protocol.

6.2 Analysis

It is clear from the protocol and the definition of an ultrametric that the latency will be no more than the depth of the tree, $\log_d C$, times the latency between the browser and the server. So once again we need only look at swamping and storage. The intuition is that inside each cluster the bounds we proved for the unit distance model apply. The monotone property on consistent hashing will allow us to restrict our analysis to $\log(C)$ clusters. Thus, summing over these clusters we have only a $\log(C)$ blowup in the bound.

6.2.1 Swamping

Theorem 6.2.1 *Let δ be an ultrametric. Suppose that each browser makes at most one request. Then in the protocol above, an arbitrary cache gets no more than* $\log C(\rho(8\log_d C + O(\frac{\log N}{\log\log N})) + O(\frac{dq\log N}{\log(\frac{dq}{\rho}\log N)}))$ *requests with probability at least* $1 - 1/N$ *where N is a parameter.*

Proof: We fix an arbitrary cache m and prove the theorem for m. We consider the clustering of machines according to their distance from m. Denote the resulting clusters by $\mathcal{C}_1 \subseteq \mathcal{C}_2 \subseteq \ldots \subseteq \mathcal{C}_S$.

Consider a request machine m receives. Since m is on the request path, it must be in the smallest cluster, $\mathcal{C}_i$, containing both the browser that made the request and the server of the requested page. Thus for every request that m receives there is an associated cluster on which the caching protocol was run. Let r_i denote the number of requests that m receives that are associated with $\mathcal{C}_i$. Let us now find the maximum number of browsers that could contribute to r_i. A browser that is outside $\mathcal{C}_i$ cannot possibly contribute to because it would use a cluster bigger than $\mathcal{C}_i$. So by definition of r_i any request made by such a browser cannot contribute to r_i.

Let us now try to bound r_i. Let C_i denote the number of caches in the cluster $\mathcal{C}_i$. Since we lower bounded the density of the caches, C_i is at least $|\mathcal{C}_i|/\rho$. We note that r_i is simply the number of requests m receives due to at most C_i browsers playing the caching protocol on at least $\mathcal{C}_i$ caches. Therefore from theorem 3.2.2 we deduce that r_i is less than $\rho(2\log_d C + O(\frac{\log N}{\log\log N})) + O(\frac{dq\log N}{\log(\frac{dq}{\rho}\log N)}) = \beta$ requests with probability at least $1/N^{\Omega(1)}$.

Thus, we conclude that the probability that m receives more than $S\beta$ requests (where S is the number of clusters around m) is less than $1/N^{\Omega(1)}$ since

$$
\begin{aligned}
& Pr[m \text{ receives more than } S\beta \text{ requests from } \bigcup \mathcal{C}_i] \\
& \leq \sum_{i=1}^{S} Pr[x \text{ receives more than } \beta \text{ requests from } \mathcal{C}_i] \\
& \leq S/N^{\Omega(1)} \leq 1/N^{\Omega(1)}.
\end{aligned}
$$

Unfortunately, the number of clusters around m could be as large as as the number of caches, C, so the above bound is not very good. The following Lemma will improve the bound by showing that consecutive clusters that do not grow too fast can be counted as a single cluster.

Lemma 6.2.2 *If there are a set of consecutive clusters $\mathcal{C}_i, \mathcal{C}_{i+1}, \ldots, \mathcal{C}_{i+j}$ such that $\frac{C_{i+j}}{C_i} \leq 2$, then $\sum_{j=i}^{i+j} r_i$ is less than 4β with probability at least $1 - 1/N^{\Omega(1)}$.*

Proof: Let r_j be the requests received by m as part of cluster $\mathcal{C}_j$ in the protocol with C_j nodes. Since $\frac{C_{i+j}}{C_i} \leq 2$, observe that for a given page the difference between the structure of the smallest tree (built using caches in $\mathcal{C}_i$) and the largest tree (built using caches in $\mathcal{C}_{i+j}$) is only in adding leaf positions to the tree that do not overlap. The basic idea of the proof is to modify the protocol for the given clusters to obtain a protocol that is *worse* for m than the real protocol, and to apply Theorem 3.2.2 to the modified protocol. Specifically we show that if we use the smallest set of caches (those in C_i for all of the page requests using any of the clusters $\mathcal{C}_i$ through $\mathcal{C}_{i+j}$ for the caching protocol then m will only get *more* requests than in the real protocol.

Observe that due to the *monotone* property of the consistent hashing scheme described earlier, when we decrease the number of caches filling a tree, only the positions previously filled by deleted caches change, and they are filled by one of the remaining caches. Since m is not removed from the set of caches, the number of times that m appears as an internal node of the tree can only go up. The original places that m appeared remain unchanged. Thus by reducing the number of caches m can only receive more requests. Let us now look at the number of requests a cache gets due to its presence in internal nodes and due to its presence in the leaf nodes. Due to its presence in the internal nodes a cache can get at most the number of requests it would have got if everyone had used trees of size C_i, which is at most 2β by Theorem 3.2.2. Due to its presence in leaf nodes a cache could get at most the number of requests it would have got if everyone had used trees of size C_{i+j} which is at most 2β again. So the total number of requests a cache gets is at most 4β. □

The previous lemma implies that when summing over clusters to compute the bound on the number of requests made to m we need only sum over clusters that are at least twice the size of the previous one. Thus the sum is only made over $\log C$ terms and the bound that we achieve is $\log C$ times the previous bound. □

6.2.2 Storage

Using techniques similar to those in proof of Theorem 6.2.1 we get the following lemma.

Lemma 6.2.3 *The total number of cached pages, over all machines is* $\frac{R \log_d C}{q}$. *A given cache* m *has* $O(\log C(\frac{\rho}{q} \log_d C + d \log N))$ *cached pages with probability* $1 - 1/N$.

Proof: It is easy to prove the bound of $\frac{R \log_d C}{q}$ on the total number of cached pages over all machines. Since each request gives rise to at most $\log_d C$ up its tree, the total number of requests received by the caches is at most $R \log_d C$. Now an abstract nodes caches a page only if it receives at least q requests for it. So clearly there can be no more than $\frac{R \log_d C}{q}$ cached copies over all caches.

To bound the number of cached pages in single machine we use the bound on number of requests proved in theorem 6.2.1. The bound stated in theorem 6.2.1 is less than $O(\log C(\rho \log_d C + dq \log N))$. Since m must receive at least q requests for a page before it is cached, the number of cached pages is at most the number of requests divided by q, which gives us the desired result. □

Chapter 7

Fault Tolerance

Basically, as in Plaxton/Rajaraman, the fact that our protocol uses random short paths to the server makes it fault tolerant. We consider a model in which an adversary designates that some of the caching machines may be *down*, that is, ignore all attempts at communication. Remember that our adversary does not get to see our random bits, and thus cannot simply designate all machines at the top of a tree to be down. The only restriction is that a specified fraction s of the machines in every view must be up. Under our protocol, no preemptive caching of pages is done. Thus, if a server goes down, all pages that it has not distributed become inaccessible to any algorithm. This problem can be eliminated using standard techniques, such as Rabin's Information Dispersal Algorithm [11]. So we ignore server faults.

In this chapter, we analyze a minor modification of the protocol and show that it ensures that any page request is satisfied with high probability.

Observe that the analysis of whether a request is satisfied is quite simple to make: we look at that path of machines that the request travels through, and check if any of them is down. If none are, then the request gets through.

We say that a path up the tree is *clean* if it does not encounter any dead caches. The following lemma ensures that a random request path has a good chance of being clean.

Lemma 7.0.4 *If $d \geq 2 \log C$ and $s \geq 1 - 1/\log_d C$ then with probability at least $1 - O(\log_d C/C)$ more than a $1/2e^2$ fraction of the paths up the tree are clean.*

Proof: We will start from the server and count the fraction of paths that remain clean all the way up to the leaves. Let $f = 1 - s$ denote the fraction of dead machines. We will say that a *node* is *clean* if the path from the node to the root is clean. The level just below the server contains $d \geq 2\log C$ caches. Also the probability of an arbitrary node being dead is $f \leq 1/\log_d C$. The expected number of dead caches in the second level is df. So 1/2 the nodes at depth 1 are clean. Also by Chernoff bounds the probability that more than $d/2$ caches are dead is less than $1/C$. Next we inductively argue that with probability at least $1 - i/C$ more than $\frac{1}{2}(1-2f)^{i-1}$ fraction of the nodes at depth i are clean. We have already proved it for $i = 1$. Now at depth $i+1$ the number of clean paths up to depth i extend to give at least $\frac{1}{2}d^{i+1}(1-2f)^{i-1}$ paths. Since i is less than the depth of the tree, which is less than $1/f$, the fraction of nodes at depth $i+1$ that have clean paths up to depth i is at least $d^{i+1}/2e^2$. Of these an f fraction is expected to have dead caches at depth i. For $i > 1$ by Chernoff bounds the probability that more than $2f$ fraction encounter dead caches at depth i is at most $1/C$. This completes the inductive step. Now by setting $i = \log_d C$ we get that with probability at least $(\log_d C)/C$ more than $1/2e^2$ fraction of the paths are clean. $\square$

The modification to the protocol is therefore quite simple. Choose a parameter t, and simultaneously send t requests for the page. A logarithmic number of requests is sufficient to give a high probability that one of the requests goes through. This will clearly increase the total load on the network by only an $O(\log n)$ factor. In practice, instead of sending all these requests simultaneously a browser could wait for a certain time interval before sending the next request.

Note that since communication is a chancy thing on the Internet, failure to get a quick response from a machine is not a particularly good indication that it is down. Thus, we focused on the tolerance of faults, and not on their detection. However, given some way to decide that a machine is down, our consistent hash functions make it trivial to reassign the work to other machines. If a you decide a machine is down, remove it from your view.

Chapter 8

Conclusion

This work has focused on one particular caching problem—that of handling read requests on the Web. We believe the ideas have broader applicability. In particular, consistent hashing may be a useful tool in a network where different users have different views of the network and need to agree on the location of a resource without having to communicate with each other.

It remains open how to deal with time when modeling the Internet, because the communication protocols have no guarantees regarding time of delivery. Indeed, at the packet level, there are not even guarantees regarding eventual delivery. This suggests modeling the Internet as some kind of distributed system. Clearly, in a model in which there are no guarantees regarding delivery times, the best one can hope to prove is some of the classical *liveness* and *safety* properties underlying distributed algorithms. It is not clear what one can prove about caching and swamping in such a model. We think that there is significant research to be done on the proper way to model this aspect of the Internet.

We also believe that interesting open questions remain regarding the method of consistent hashing that we present in this paper. Among them are the following. Is there a k-universal consistent hash function that can be evaluated efficiently?? What tradeoffs can be achieved between spread and load? Are there some kind of "perfect" consistent hash functions that can be constructed deterministically with the same spread and load bounds we give? On what other theoretical problems can consistent hashing give us a handle?

Bibliography

[1] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael Schwartz and Kurt Worrell. A Hierarchical Internet Object Cache. In *USENIX Proceedings*, 1996.

[2] Azer Bestavros. Speculative Data Dissemination and Service.

[3] Robert Devine. Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm. In *Proceedings of 4th International Conference on Foundations of Data Organizations and Algorithms*, 1993.

[4] M. J. Feeley, W. E. Morgan, F. P. Pighin, A. R. Karlin, H. M. Levy and C. A. Thekkath. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[5] Sally Floyd, Van Jacobson, Steen McCanne, Ching-Gung Liu and Lixia Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing, *SIGCOMM'* 95

[6] Witold Litwin, Marie-Anne Neimat and Donovan A. Schneider. LH^*-A Scalable, Distributed Data Structure. ACM Transactions on Database Systems, Dec. 1996

[7] Radhika Malpani, Jacob Lorch and David Berger. Making World Wide Web Caching Servers Cooperate. In *Proceedings of World Wide Web Conference*, 1996.

[8] M. Naor and A. Wool. The load, capacity, and availability of quorum systems. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 214-225, November 1994.

[9] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation* 123(2):210-233, 1995.

[10] Greg Plaxton and Rajmohan Rajaraman. Fast Fault-Tolerant Concurrent Access to Shared Objects. In *Proceedings of 37th IEEE Symposium on Foundations of Computer Science*, 1996.

[11] M. O. Rabin. Efficient dispersal of Information for Security, Load Balancing, and Fault Tolerance. *Journal of the ACM* 36:335–348, 1989.

[12] Jeanette Schmidt, Alan Siegel and Aravind Srinivasan. Chernoff-Hoeffding Bounds for Applications with Limited Independence. In *Proc. 4th ACS-SIAM Symposium on Discrete Algorithms*, 1993.

[13] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. In *ACM SIGCOMM*'95.

[14] Ari Luitonen and Kevin Altis. World-wide web proxies. In *Computer Networks and ISDN systems.* First International Conference on the World-Wide Web, Elsevier Science BV, 1994. available from 'http://www.cern.ch/PapersWWW94/luotonen.ps'

[15] R. Ravi. Approximating the minimum broadcast time. In *FOCS*'94.

[16] M. Grigni and D. Peleg. Tight bounds on minimun broadcast networks. In *SIAM Journal on Discrete Math.*, May 1991, pp. 207-222.

[17] James S. Gwetzman and Margo Seltzer. The Case for Geographical Push-Caching. Personal Communication.

[18] M. Palmer and S. Zdonik, Fido: A Cache that Learns to Fetch. In*Proceedings of the 1991 International Conference on Very Large Databases*, September 1991.

[19] K. Salem. Adaptive Prefetching for Disk Buffers. *CESDIS, Goddard Space Flight Center, TR-91-64*, January 1991.

[20] Jeffrey Scott Vitter and P. Krishnan. Optimal Prefetching via Data Compression. *FOCS 91.*

[21] Stephen E. Deering and David R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. In *ACM Transactions on Computer Systems*, May 1990.

[22] James Gwetzman and Margo Seltzer. World-Wide Web Cache Consistency. *Personal Communication.*