Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web*

David Karger[1] Eric Lehman[1] Tom Leighton[1,2] Matthew Levine[1] Daniel Lewin[1]
Rina Panigrahy[1]

Abstract

We describe a family of caching protocols for distrib-uted networks that can be used to decrease or eliminate the occurrence of hot spots in the network. 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.

1 Introduction

In this paper, we describe caching protocols for distributed networks that can be used to decrease or eliminate the occurrences of "hot spots". *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. 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 provide new versions of popular software.

Our work was originally motivated by the problem of hot spots on the World Wide Web. We believe the tools we develop may be relevant to many client-server models, because centralized servers on the Internet such as Domain Name servers, Multicast servers, and Content Label servers are also susceptible to hot spots.

1.1 Past Work

Several approaches to overcoming the 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*. All user requests are forwarded through the proxy, which tries to keep copies of frequently requested pages. 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. [6] work around this problem by making 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". 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 Section 4.

Chankhunthod et al. [1] developed the Harvest Cache, a more scalable approach using a *tree* of caches. A user obtains a page by asking a nearby leaf cache. If neither this cache nor its siblings have the page, the request is forwarded to the cache's parent. If a page is stored by no cache in the tree, the request eventually reaches the root and is forwarded to the home site of the page. A cache

*This research was supported in part by DARPA contracts N00014-95-1-1246 and DABT63-95-C-0009, Army Contract DAAH04-95-1-0607, and NSF contract CCR-9624239

[1] Laboratory for Computer Science, MIT, Cambridge, MA 02139.
email: {karger,e_lehman,danl,ftl,mslevine,danl,rinap}@theory.lcs.mit.edu
A full version of this paper is availble at:
http://theory.lcs.mit.edu/~ {karger,e_lehman,ftl,mslevine,danl,rinap}

[2] Department of Mathematics, MIT, Cambridge, MA 02139

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee
STOC '97 El Paso, Texas USA
Copyright 1997 ACM 0-89791-888-6/97/05 ..$3.50

Ex.1013 / Page 1 of 10
APPLE INC.

retains a copy of any page it obtains for some time. 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. Thus, many requests for a page in a short period of time will only cause one request to the home server of the page, and won't overload the caches either. A disadvantage, at least in theory, is that the same tree is used for all pages, meaning that the root receives at least one request for every distinct page requested of the entire cache tree. This can swamp the root if the number of distinct page requests grows too large, meaning that this scheme also suffers from potential scaling problems.

Plaxton and Rajaraman [9] show how to balance the load among all caches by using randomization and hashing. In particular, they use a hierarchy of progressively larger sets of "virtual cache sites" for each page and use a random hash function to assign responsibility for each virtual site to an actual cache in the network. Clients send a request to a random element in each set in the hierarchy. Caches assigned to a given set copy the page to some members of the next, larger set when they discover that their load is too heavy. This gives fast responses even for popular pages, because the largest set that has the page is not overloaded. It also gives good load balancing, because a machine in a small (thus loaded) set for one page is likely to be in a large (thus unloaded) set for another. Plaxton and Rajaraman's technique is also fault tolerant.

The Plaxton/Rajaraman algorithm has drawbacks, however. For example, since their algorithm sends a copy of each page request to a random element in every set, the small sets for a popular page are guaranteed to be swamped. In fact, the algorithm uses swamping as a feature since swamping is used to trigger replication. This works well in their model of a synchronous parallel system, where a swamped processor is assumed to receive a subset of the incoming messages, but otherwise continues to function normally. On the Internet, however, swamping has much more serious consequences. Swamped machines cannot be relied upon to recover quickly and may even crash. Moreover, the intentional swamping of large numbers of random machines could well be viewed unfavorably by the owners of those machines. The Plaxton/Rajaraman algorithm also requires that all communications be synchronous and/or that messages have priorities, and that the set of caches available be fixed and known to all users.

1.2 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 minimize memory requirements (without significantly increasing cache miss rates) by only caching pages that have been requested a sufficient number of times.

We believe that the extra delay introduced by a tree of caches should be quite small in practice. The time to request a page is multiplied by the tree depth. However, the page request typically takes so little time that the extra delay is not great. The return of a page can be pipelined; a cache need not wait until it receives a whole page before sending data to its child in the tree. Therefore, the return of a page also takes only slightly longer. Altogether, the added delay seen by a user is small.

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 based 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 crash or are brought into the network. 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. For example, Feeley et al [3] implement a distributed global shared memory system for a network of workstations that uses a hash table distributed among the machines to resolve references. Each time a new machine joins the network, the require a central server to redistribute a completely updated hash table to all the machines.

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 [5] Litwin et al proposes a hash function that allows buckets to be added one at a time sequentially. However our hash function allows the buckets to be added in an arbitrary order. Another scheme that we can improve on is given by Devine [2]. In addition, we believe that consistent hashing will be useful in other applications (such as quorum systems [7] [8] or distributed name servers) where multiple machines with different views of the network must agree on a common storage location for an object without communication.

1.3 Presentation

In Section 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 Section 3, we describe our random tree method and use it in a caching protocol that effectively eliminates hot spots under a simplified model. Independent of Section 3, in Section 4 we present our consistent hashing method and use it to solve hot spots under a different simplified model involving inconsistent views.

In Section 5 we show how our two techniques can be effectively combined. In Section 6 we propose a simple delay model that captures hierarchical clustering of machines on the Internet. We show that our protocol can be easily extended to work in this more realistic delay model. In Sections 7 and 8 we consider faults and the behavior of the protocol over time, respectively. In Section 9 we discuss some extensions and open problems.

1.4 A Note on Randomization and Hashing

In several places we make use of hash functions that map objects into a range. For clarity we assume that these functions map objects in a truly random fashion, i.e. uniformly and independently. In practice, hash functions with limited independence are more plausible since they economize on space and randomness. We have proven all theorems of this paper with only limited independence using methods similar to those in [11]. However, in this extended abstract we only state the degree of independence required for results to hold. Proofs assuming limited independence will appear in

Ex.1013 / Page 2 of 10
APPLE INC.

the full version of this paper.

2 Model

This section 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 which 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.

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 generally assume that the content of each page is unchanging, though Section 9 contains a discussion of this issue. The set of all pages is denoted $\mathcal{P}$.

Any machine can send a message directly to any other with the restriction that a machine may not be aware of the existence of all caches; we require only that each machine is aware of a $1/t$ fraction of the caches for some constant t. 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".

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.

All cache and server behavior and some browser behavior is specified in our protocol. In particular, the protocol specifies how caches and servers respond to page requests and which pages are stored in a cache. The protocol also specifies the cache or server to which a browser sends each page request. All control must be local; the behavior of a machine can depend only on messages it receives.

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. First, 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 ρ is a constant and C is the number of caches. We then consider a temporal model in which the adversary may initiate new requests for pages at rate at most γ; that is, in any time interval $\tau > 1$, he may initiate at most $\gamma\tau$ requests.

Objective

The "hot spot problem" is to satisfy all browser page requests while ensuring that with high probability no cache or server is swamped. The phrase "with high probability" means "with probability at least $1 - 1/N$", where N is a confidence parameter used throughout the paper.

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.

3 Random Trees

In this section we introduce our first tool, random trees. To simplify the presentation, we give 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. $\delta(m_i, m_j) = 1$ for all $i \neq j$.
3. All requests are made at the same time.

This restricted model is "static" in the sense that there is only one batch of requests; we need not consider the long-term stability of the network.

Under these restrictions we show a protocol that has good behavior. That is, with high probability no machine is swamped. We achieve a total delay of $\Theta(\log C)$ and prove that it is optimal. We use total cache space which is a fraction of the number of requests, and evenly divided among the caches. In subsequent sections we will show how to extend the protocol so as to preserve the good behavior without the simplifying assumptions.

The basic idea of our protocol is an extension of the "tree of caches" approach discussed in the introduction. We use this tree to ensure that no cache has many "children" asking it for a particular page. As discussed in the introduction, levels near the root get many requests for a page even if the page is relatively unpopular, so being the root for many pages causes swamping. Our technique, similar to Plaxton/Rajaraman's, is to use a different, randomly generated tree for each page. This ensures that no machine is near the root for many pages, thus providing good load balancing. Note that we cannot make use of the analysis given by Plaxton/Rajaraman, because our main concern is to prevent swamping, whereas they allow machines to be swamped.

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

We associate a rooted d-ary tree, called an *abstract tree*, with each 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 refer to 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 nodes through which the request should be directed, and a sequence of caches that should act as those 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 $h : \mathcal{P} \times [1 \ldots C] \rightarrow \mathcal{C}$, 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 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, 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.

Ex.1013 / Page 3 of 10
APPLE INC.

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 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

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 server 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.

The analysis of swamping runs much the same way, except that the "weights" on our abstract nodes are now the number of requests arriving at those nodes. As above, the number of requests that hit a machine is bounded by the weight of nodes mapped to it.

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.

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. The existence of tree schemes, like the Harvest Cache, suggests that is acceptable in practice.

Our bound is optimal (up to constant factors) for any protocol that forbids swamping. To see this, consider making C requests for a single page. Look at the graph with nodes corresponding to machines and edges corresponding to links over which the page is sent. Small latency implies that this graph has small diameter, which implies that some node must have high degree, which implies swamping.

3.2.2 Swamping

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 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 the projected 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. We can therefore argue as in [11] that it applies 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.

We will now analyze our protocol under the simplified model. In this "static" analysis we assume for now that caches have enough space that they never have to evict pages; this means that if a cache has already made q requests for a page it will not make another request for the same page. In Theorem 3.1 we provide high probability bounds on the number of requests a cache gets, assuming that all the outputs of the function h are independent and random. Theorem 3.4 extends our high probability analysis to the case when h is a k-way independent function. In particular we show that it suffices to have k logarithmic in the system parameters to achieve the same high probability bounds as with full independence.

Analysis for Random h

Theorem 3.1 *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*

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

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

Due to space limitations, we give a proof that only applies when $N \gg R$. Its extension to small N is straightforward but long. Observe that the requests for each page are being mapped randomly onto the leaf nodes of its abstract tree. And then these leaf nodes are mapped randomly onto the set of caches. Look at collection of all the leaf nodes and the number of requests (weight) associated with each one of them. The variance among the "weights" of the leaf nodes is maximized when all the requests are made for one page. This is also the case which maximizes the number of leaf node requests on a cache.

Each page's tree has about $C(1 - 1/d)$ leaf nodes. Since a machine m has a $1/C$ chance of occurring at a particular leaf node, with probability $1-1/N$ it will occur in $O(\frac{\log N}{\log\log N})$ leaf nodes. In fact, since there are at most R requests, m will occur $O(\frac{\log N}{\log\log N})$ times in all those requested pages' trees with probability $1 - R/N$.

Given an assignment of machines to leaf nodes so that m occurs $O(\frac{\log N}{\log\log N})$ times in each tree, the expected number of requests m gets is $R\frac{1}{C}O(\frac{\log N}{\log\log N})$ which is $O(\frac{\rho\log N}{\log\log N})$. 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 $O(\frac{\rho\log N}{\log\log N} + \log N)$ requests with probability $1 - 1/N$. So we conclude that m gets $O(\frac{\rho\log N}{\log\log N} + \log N)$ with probability at least $1-(R+1)/N$. Replacing N by N^2 and assuming $N > R$ we can say that the same bound holds with probability $1 - 1/N$. It is easy to extend this proof so that the bound holds even for $N < R$.

Ex.1013 / Page 4 of 10
APPLE INC.

Requests to Internal Nodes

Again 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. 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 which 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,

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

Lemma 3.2 *The total number of internal nodes which receive at least qx requests is at most $2R/x$ if $x > 1$*

Proof (sketch): Look at the tree induced by the request paths, contract out degree 1 nodes, and count internal nodes. □

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 is 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 must have $z = \Omega(\frac{n_j}{C} + \frac{\log N}{\log(\frac{C}{n_j}\log N)})$. 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

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

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 $N\log(dq)$ and ignoring $\log\log(dq)$ in comparision with dq we get Theorem 3.1.

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

Proof: Full paper. □

Analysis for k-way Independent h We now extend our high probability analysis to functions h that are chosen at random from a k-universal hash family.

Theorem 3.4 *If h is chosen at random from a k-universal hash family then with probability at least $1 - 1/N$ a given cache receives no more than $2\rho\log_d C + O((kqN(d+\rho))^{1/k}(1 + \frac{\rho}{k} + \frac{dq}{\log(kdq/\rho)} + \frac{\rho}{\log k}))$ requests.*

Proof: The full proof is deferred to the final version of the paper. This result does not follow immediately from the results of [11], but involves a similar argument. □

Setting $k = \log N$ we get the following corollary.

Corollary 3.5 *The high probability bound proved in theorem 3.1 for the number of requests a cache gets holds even if h is selected from a $\log N$-universal hash family.*

In fact, this can be shown to be true for all the bounds that we will prove later, i.e., it suffices k to be logarithmic in the system size.

3.2.3 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.

Lemma 3.6 *The total number of cached pages, over all machines, is $O(\log R \log_d C + \frac{R}{q})$ with probability at least $1 - 1/R^{\Omega(1)}$. A given cache m has $O(\frac{\rho}{q} + \log R)$ cached copies with high probability.*

Proof (sketch): The analysis is very similar to that in proof of Theorem 3.1. We again play the protocol on the abstract trees. Since a page is cached only if it requested q times, we assign each abstract node a weight of one if it gets more than q requests and zero otherwise. These abstract nodes are then mapped randomly onto the set of caches. We can bound the total weight received by a particular cache, which is exactly the number of pages it caches. □

4 Consistent Hashing

In this section 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 $x \mapsto ax + b \pmod p$), such inconsistencies would be catastrophic. When the range of the hash function (p in the example) changed, almost every item would be

Ex.1013 / Page 5 of 10
APPLE INC.

hashed to a new location. Suddenly, all 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 a object is assigned is small. We call this property "spread". Similarly, over all the client views, the number of distinct objects assigned to a particular cache is small. We call this property "load".

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.

Since there are many ways to formalize the notion of consistency as described above, we will not commit to a precise definition. Rather, in Section 4.4 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. In Section 4.4, we discuss other aspects of consistent hashing whihc, though not germane to this paper, indicate some of the richness underlying the theory.

4.1 Definitions

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

Let $\mathcal{I}$ be the set of items and $\mathcal{B}$ be the set of buckets. Let $I = |\mathcal{I}|$ be the number of items. A *view* is any subset of the buckets $\mathcal{B}$.

A *ranged hash function* is a function of the form $f : 2^{\mathcal{B}} \times \mathcal{I} \mapsto \mathcal{B}$. 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 $f_{\mathcal{V}}(\mathcal{I}) \subseteq \mathcal{V}$ 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 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.

Balance: A ranged hash family is *balanced* if, given a particular view $\mathcal{V}$ a set of items, and a randomly chosen function selected from the hash family, with high probability the fraction of items mapped to each bucket is $O(1/|V|)$.

The balance property is what is prized about standard hash functions: they distribute items among buckets in a balanced fasion.

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. This reflects one intuition about consistency: when the set of usable buckets changes, items should only move if necessary to preserve an even distribution.

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* $\sigma(i)$ is the quantity $\left|\{f_{\mathcal{V}_j}(i)\}_{j=1}^{V}\right|$. The *spread* of a hash function $\sigma(f)$ is the maximum spread of an item. The *spread* of a hash family is σ if with high probability, the spread of a random hash function from the family is σ.

The idea behind spread is that there are V people, each of whom can see at least a constant fraction $(1/t)$ of the buckets that are visible to anyone. 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.

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 $\left|\bigcup_{\mathcal{V}} f_{\mathcal{V}}^{-1}(b)\right|$. The *load* of a hash function is the maximum load of a bucket. The *load* of a hash family is λ if with high probability, a randomly chosen hash function has load λ. (Note that $f_{\mathcal{V}}^{-1}(b)$ is the set of items assigned to bucket b in view $\mathcal{V}$.) The load property is similar to spread. The same V people are back, but this time we consider a particular bucket b instead of an item. The property says that there are at most $\lambda(b)$ distinct items that at least one person thinks belongs in the bucket. A good consistent hash function should also have low load.

Our main result for consistent hashing is Theorem 4.1 which shows the existence of an efficiently computable monotonic ranged hash family with logarithmic spread and balance.

4.2 Construction

We now give a construction of a ranged hash family with good properties. Suppose that we have two random functions $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$. 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. 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. In order to economize on the space to represent a function in the family, and on the use of random bits, we only demand that the functions $r_{\mathcal{B}}$ and $r_{\mathcal{I}}$ map points $\log(C)$-way independently and uniformly to $[0, 1]$. Note that for each point we pick in the unit interval, we need only pick enough random bits to distinguish the point from all other points. Thus it is unlikely that we need more than log(number of points) bits for each point. Denote the above described hash family as $\mathcal{F}$.

Theorem 4.1 *The ranged hash family $\mathcal{F}$ described above has the following properties:*

1. *$\mathcal{F}$ is monotone.*

Ex.1013 / Page 6 of 10
APPLE INC.

2. *Balance: For a fixed view $\mathcal{V}$, $\Pr[f_{\mathcal{V}}(i) = b] \leq \frac{O(1)}{|\mathcal{V}|}$ for $i \in \mathcal{I}$ and $b \in \mathcal{V}$, and, conditioned on the choice of $r_{\mathcal{B}}$, the assignments of items to buckets are $\log(C)$-way independent.*

3. *Spread: If the number of views $V = \rho C$ for some constant ρ, and the number of items $I = C$, then for $i \in \mathcal{I}$, $\sigma(i)$ is $O(t\log(C))$ with probability greater than $1 - 1/C^{\Omega(1)}$.*

4. *Load: If V and I are as above, then for $b \in \mathcal{B}$, $\lambda(b)$ is $O(t\log(C))$ with probability greater than $1 - 1/C^{\Omega(1)}$.*

Proof (sketch): Monotonicity is immediate. When a new bucket is added, the only items that move are those that are now closest to one of the new bucket's associated points. No items move between old buckets. The spread and load properties follow from the observation that with high probability, a point from *every* view falls into an interval of length $O(t/C)$. Spread follows by observing that the number of bucket points that fall in this size interval around an item point is an upper bound on the spread of that item, since no other bucket can be closer in any view. Standard Chernoff arguments apply to this case. Load follows by a similar argument where we count the number of item points that fall in the region "owned" by a bucket's associated points. Balance follows from the fact that when $\kappa\log(C)$ points are randomly mapped to the unit interval, each bucket is with highu probability responsible for no more than a $\frac{O(1)}{|\mathcal{V}|}$ fraction of the interval. The key here is to count the number of combinatroially distinct ways of assigning this large a fraction to the $\kappa\log(C)$ points associated with a bucket. This turns out to be polynomial in C. We then argue that with high probability none of these possibilities could actually occur by showing that in each one an additional bucket point is likely to fall. We deduce that the actual length must be smaller than $O(1/|\mathcal{V}|)$. All of the above proofs can be done with only $\log(C)$-way independent mappings. □

The following corollary is immediate and is useful in the rest of the paper.

Corollary 4.2 *With the same conditions of the previous theorem, $\Pr[f_{\mathcal{V}}(i) = b$ in any view$] \leq \frac{O(t\log(C))}{|\mathcal{V}|}$ for $i \in \mathcal{I}$ and $b \in \mathcal{B}$.*

4.3 Implementation

In this section we show how the hash family just dexcrobed 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 an upper bound on the total number of buckets in all views.

A simple implementation uses a balanced binary search tree to store the correspondence between segments of the unit interval and buckets. 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. 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.

One caveat to the above is that as the number of buckets grows, the size of the subintervals needs to shrink. In order to deal with this issue, we will use intervals only of length $1/2^x$ for some x. At first we choose the largest x such that $1/2^x \leq 1/\kappa C\log(C)$. Then, as points are added, we bisect segments gradually so that when we reach the next power of 2, we have already divided all the segments. In this way we amortize the work of dividing search trees over all of the additions and removals. Another point is that the search trees in adjacent empty intervals may all need to be updated when a bucket is added since they may all now be closest to that bucket. Since the expected length of a run of empty intervals is small, the additional cost is negligible. For a more complete analysis of the running time we refer to the complete version of the paper.

4.4 Some Theorems on Consistent Hashing

In this section, we discuss some additional features of consistent hashing which, though unneccessary for the remainder of the paper, demonstrate some of its interesting properties.

To give insight into the monotone property, we will define a new class of hash functions and then show that this is equivalent to the class of monotone ranged hash functions.

A *π-hash function* is a hash function of the familiar form $f : 2^{\mathcal{B}} \times \mathcal{I} \mapsto \mathcal{B}$ constructed as follows. With each item $i \in \mathcal{I}$, associate a permutation $\pi(i)$ of all the buckets $\mathcal{B}$. Define $f_{\mathcal{V}}(i)$ to be the first bucket in the permutation $\pi(i)$ that is contained in the view $\mathcal{V}$. Note that the permutations need not be chosen uniformly or independently.

Theorem 4.3 *Every monotone ranged hash function is a π-hash function and vice versa.*

Proof (sketch): For a ranged hash function f, associate item i with the permutation $b_1 = f_{\mathcal{B}}(i), \ldots, b_{j+1} = f_{\mathcal{B}-\{b_1\ldots b_j\}}(i), \ldots$. Suppose b_j is the first element of an arbitrary view $\mathcal{V}_1$ in this permutation. Then $\mathcal{V}_1 \subseteq \mathcal{V}_2 = \mathcal{B} - \{b_1 \ldots b_{j-1}\}$. Since $f_{\mathcal{V}_2}(i) = b_j \in \mathcal{V}_1$, monotonicity implies $f_{\mathcal{V}_1}(i) = b_j$. □

The equivalence stated in Theorem 4.3 allows us to reason about monotonic ranged hash functions in terms of permutations associated with items.

Universality: A ranged hash family is *universal* if restricting every function in the family to a single view creates a universal hash family.

This property is one way of requiring that a ranged hash function be well-behaved in every view. The above condition is rather stringent; it says that if a view is fixed, items are assigned randomly to the bins in that view. This implies that in any view $\mathcal{V}$, the expected fraction of items assigned to j of the buckets is $j/|\mathcal{V}|$. Using only monotonicity and this fact about the uniformity of the assignment, we can determine the expected number of items reassigned when the set of usable buckets changes. This relates to the informal notion of "smoothness".

Theorem 4.4 *Let f be a monotonic, universal ranged hash function. Let $\mathcal{V}_1$ and $\mathcal{V}_2$ be views. The expected fraction of items i for which $f_{\mathcal{V}_1}(i) = f_{\mathcal{V}_2}(i)$ is $\frac{|\mathcal{V}_1 \cap \mathcal{V}_2|}{|\mathcal{V}_1 \cup \mathcal{V}_2|}$.*

Proof (sketch): Count the number of items that move as we add buckets from $\mathcal{V}_1$ until the view is $\mathcal{V}_1 \cup \mathcal{V}_2$, and then delete buckets down to $\mathcal{V}_2$ □

Note that monotonicity is used only to show an upper bound on the number of items reassigned to a new bucket; this implies that one can not obtain a "more consistent" universal hash function by relaxing the monotone condition.

We have shown that every monotone ranged hash function can be obtained by associating each item with a random permutation

Ex.1013 / Page 7 of 10
APPLE INC.

of buckets. The most natural monotone consistent hash function is obtained by choosing these permutations independently and uniformly at random. We denote this function by $\tilde{f}$.

Theorem 4.5 *The function $\tilde{f}$ is monotonic and universal. For item i and bucket b each of the following hold with probability at least $1 - 1/N$: $\sigma(i) \leq t\log(NV)$ and $\lambda(b) \leq (1 + \sqrt{\frac{4C}{Tt}})tI\log(2NVI)/C2$.*

Proof: Monotonicity and universality are immediate; this leaves spread and load. Define:

$$\sigma = t\log(NV)$$
$$\lambda = \left(1 + \sqrt{\frac{4C}{tI}}\right)\frac{tI\log(2NVI)}{C}$$

We use $\pi'(i)$ to denote a list of the buckets in $\mathcal{V}_1 \cup \ldots \cup \mathcal{V}_V$ which are ordered as in $\pi(i)$.

First, consider spread. Recall that in a particular view, item i is assigned to the first bucket in $\pi(i)$ which is also in the view. Therefore, if every view contains one of the first σ buckets in $\pi'(i)$ then in every view item i will be assigned to one of the first σ buckets in $\pi'(i)$. This implies that item i is assigned to at most σ distinct buckets over all the views.

We have to show that with high probability every view contains one of the first σ buckets in $\pi'(i)$. We do this by showing that the complement has low probability; that is, the probability that some view contains none of the first σ buckets is at most $1/N$.

The probability that a particular view does not contain the first bucket in $\pi'(i)$ is at most $1 - 1/t$, since each view contains at least a $1/t$ fraction of all buckets. The fact that the first bucket is not in a view only reduces the probability that subsequent buckets are not in the view. Therefore, the probability that a particular view contains none of the first σ buckets is at most $(1 - 1/t)^\sigma = (1 - 1/t)^{(t\log(NV))} < 1/(NV)$. By the union bound, the probability that even one of the V views contains none of the first σ buckets is at most $1/N$.

Now consider load. By similar reasoning, every item i in every view is assigned to one of the first $t\log(NVI)$ buckets in $\pi'(i)$ with probability at least $1 - 1/(2N)$. We show below that a fixed bucket b appears among the first $t\log(2NVI)$ buckets in $\pi'(i)$ for at most λ items i with probability at least $1-1/(2N)$. By the union bound, both events occur with high probability. This implies that at most λ items are assigned to bucket b over all the views.

All that remains is to prove the second statement. The expected number of items i for which the bucket b appears among the first $t\log(2NVI)$ buckets in $\pi'(i)$ is $tI\log(2NVI)/C$. Using Chernoff bounds, we find that bucket b appears among the first $t\log(2NVI)$ buckets in $\pi'(i)$ for at most λ items i with probability at least $1 - 1/(2NVI) \geq 1 - 1/(2N)$.

□

A simple approach to constructing a consistent hash function is to assign random scores to buckets, independently for each item. Sorting the scores defines a random permutation, and therefore has the good properties proved in the this section. However, finding the bucket an item belongs in requires computing all the scores. This could be restrictivly slow for large bucket sets.

5 Random Trees in an Inconsistent World

In this section we apply the techniques developed in the last section to the simple hot spot protocol developed in section 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 maximum σ and λ of the hash function.

5.1 Swamping

Theorem 5.1 *If h is implemented using the $\log(C)$-way independent consistent hash function of Theorem 4.1 and if each view consists of $C' = C/t$ caches then with probability at least $1 - 1/C^{\Omega(1)}$ an arbitrary cache gets no more than $O((2\rho t^2 \log_d C' \log C) + (dqt\log C + \rho t)\log C)$ requests.*

Proof (sketch): We look at the different trees of caches for different views for one page, p. Let $C' = C/t$ denote the number of caches in each tree. We overlay these different trees 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 $\sigma = O(t\log C)$ 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 jth node of the combined tree for page p then we know from Corrollary 4.2 that the probability of this event is $O(\lambda/C)$, where λ is the *load* which is $O(t\log C)$ with high probability. We condition on the event that σ and λ are $O(t\log C)$ 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 $q\sigma$ requests. We now adapt the proof of Theorem 3.1 to this case. In Theorem 3.1 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 and abstract node to a given machine with probability $\Theta(\lambda/C)$. 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 $\Theta(\lambda/C)$. The rest of the proof is very similar to that of Theorem 3.1.

□

5.2 Storage

Using techniques similar to those in proof of Theorem 5.1 we get the following lemma. The proof is deferred to the final version of the paper.

Lemma 5.2 *The total number of cached pages, over all machines is $O(\sigma\lambda(\log R\log_d C + \frac{R}{q}))$ with probability of $1 - 1/R^{\Omega(1)}$. A given cache m has $O(\sigma\lambda(\rho/q + \log R))$ cached copies with high probability.*

6 Nonuniform Communication Costs

So far we assumed that every pair of machines can communicate with equal ease. In this section we extend our protocol to take the latency between machines, δ, into account. The latency of the whole request will be the sum of the latencies of the machine-machine links crossed by the request. For simplicity, we assume in this section that all clients are aware of all caches.

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: $\delta(x,z) \leq \max(\delta(x,y), \delta(y,z))$.

Ex.1013 / Page 8 of 10
APPLE INC.

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 still 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 points. The distance in the ultrametric between two points is completely determined by the smallest cluster containing both of the points.

6.1 Protocol

The only modification we make to the protocol is that when a browser maps the tree nodes to caches, it only uses caches that are as close to it as the server of the desired page. 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 $R = \rho C$). 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 we can prove that a large number of browsers clustered around one cache can be forced to swamp that cache in some circumstances.

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 6.1 *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* $\log C(\rho(8\log_d C + O(\frac{\log N}{\log\log N})) + O(\frac{dq\log N}{\log(\frac{dq}{\rho}\log N)} + \log N))$ *requests with probability at least* $1 - 1/N$ *where N is a parameter.*

Proof (sketch): The intuition behind are proof is the following. We bound the load on a machine m. Consider the ranking of machines $m_1, , m_2, \ldots$ according to their distance from m. Suppose m_i asks for a page from a machine closer to itself than m. Then according to our modified protocol, it will never involve m in the request. So we need only consider machine m_i if it asks for a page at least as far away from itself as m. It follows from the definition of ultrametrics that every $m_j, j \leq i$, is also used in the revised protocol by m_i.

Intuitively, our original protocol spread load among the machines so that the probability a machine got on the path for a particular page requests was $O((\log_d C)/C)$. In our ultrametric protocol, m_i plays the protocol on a set of at least i machines. So m is on the path of the request from m_i with probability $O((\log_d i)/i)$. Summing over i, the expected load on m is $O(\log C)$.

Stating things slightly more formally, we consider a set of $\log C$ nested "virtual" clusters $\mathcal{C}_i = \{m_1, \ldots, m_{2^i}\}$. Note that any browser in $\mathcal{C}_{i+1} - \mathcal{C}_i$ will use all machines in $\mathcal{C}_i$ in the protocol. We modify the protocol so that such a machine uses *only* the machines in $\mathcal{C}_i$. This only reduces the number of machines it uses. According to the monotonicity property of our consistent hash functions, this only increases the load on machine m.

Now we can consider each $\mathcal{C}_i$ separately and apply the static analysis. The total number of requests arriving in one of the clusters under the modified protocol is proportional to the number of caches in the cluster, so our static analysis applies to the cluster. This gives us a bound of $O(\log_d C)$ on the load induced on m by $\mathcal{C}_i$. Sumnming over the $\log C$ clusters proves the theorem. □

6.2.2 Storage

Using techniques similar to those in proof of Theorem 6.1 we get the following lemma.

Lemma 6.2 *The total number of cached pages, over all machines is $O(R/q \log R \log_d C \log C)$ with probability of $1 - 1/R^{\Omega(1)}$. A given cache m has $O(\log C(\rho/q + \log R))$ cached copies with high probability.*

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 [10]. So we ignore server faults.

Observe that a request is satisfied if and only if all the caches serving for the nodes of the tree path are not down. Since each node is mapped to a machine (k-wise) independently, it is trivial (using standard Chernoff bounds) to lower bound the number of abstract nodes that have working paths to the root. This leads to the following lemma:

Lemma 7.1 *Suppose that $d = \Omega(\log N)$. With high probability, the fraction of abstract-tree leaves that have a working path to the root is $\Omega(s^{\log_d C})$. In particular, if $s = 1 - O(1/\log_d C)$, this fraction is a constant.*

The modification to the protocol is therefore quite simple. Choose a parameter t, and simultaneously send t requests for the page. A logarithmic number if requests is sufficient to give a high probability of one of the requests goes through. This change in the protocol will of course have an impact on the system. This impact is described in the full paper.

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.

Ex.1013 / Page 9 of 10
APPLE INC.

8 Adding Time to the Model

So far, we have omitted any real mention of time from our analyses. We have instead considered and analyzed a single "batch" of R requests, and argued that this batch causes a limited amount of caching (storage usage) at every machine, while simultaneously arguing that no machine gets swamped by the batch. In this section, we show how this static analysis carries implications for a temporal model in which requests arrive over time. Recall that our temporal model says that browsers issues requests at a certain rate γ.

Time is a problematic issue when modeling the Internet, because the communication protocols for it have no guarantees regarding time of delivery. Thus any one request could take arbitrarily long. However, we can consider the rate at which servers receive requests. This seems like an overly simplistic measure, but the rate at which a machine can receive requests is in fact the statistic that hardware manufacturers advertise. We consider an interval of time τ, and apply our "requests all come at once" analysis to the requests that come in this interval.

We can write the bounds from the static analysis on R requests as follows:

$$\text{cache size } = a_s R + b_s \qquad \text{cache load } = a_l R + b_l$$

Suppose machines have cache size m. Consider a time interval small enough to make $R = \gamma\tau$ small enough so that $m > a_s R + b_s$. In other words, the number of requests that arrive in this interval is insufficient, according to our static analysis, to use storage exceeding m per machine. Thus once a machine caches a page during this interval, it keeps it for the remainder of the interval. Thus our static analysis will apply over this interval. This gives us a bound on how many requests can arrive in the interval. Dividing by the interval length, we get the rate at which caches see requests: $\gamma(a_l + \frac{b_l a_s}{m - b_s})$. Plugging in the bounds from Section 3, we get the following:

Theorem 8.1 *If our machines have $m = \Omega(\log N)$ storage, for some constant N, then with probability $1/N$, the bound on the rate of new requests per cache when we have C machines of size m is $\gamma\left(\frac{2\log_d C}{C} + O\left(\frac{dq}{m}\right)\right)$.*

Observe the tradeoffs implicit in this theorem. Increasing m causes the load to decrease proportionately, but never below $\Omega(\gamma \log C/C)$. Increasing d increases the load linearly (but reduces the number of hops on a request path). Increasing q seems only to hurt, suggesting that we should always take $q = 2$.

The above analysis used the rate at which requests were issued to measure the rate at which connections are established to machines. If we also assume that each connection lasts for a finite duration, this immediately translates into a bound on the number of connections open at a machine at any given time.

9 Conclusion

This paper 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 for distributing information from name servers such as DNS and label servers such as PICS in a load-balanced and fault-tolerant fashion. Our two schemes may together provide an interesting method for constructing multicast trees [4].

Another important way in which our ideas could be extended is in handling pages whose information changes over time, due to either server or client activity. If we augment our protocol to let the server know which machines are currently caching its page, then the server can notify such caches whenever the data on its pages changes. This might work particularly well in conjunction with the currently under development *multicast* protocols [4] that broadcast information from a server to all the client members of a multicast "group." Our protocol can be mapped into this model if we assume that every machine "caching" a page joins a multicast group for that page. Even without multicast, if each cache keeps track, for each page it caches, of the at most d other caches it has given the page to, then notification of changes can be sent down the tree to only the caches that have copies.

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?

References

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

[2] 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.

[3] 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.

[4] 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*

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

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

[7] 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.

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

[9] 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.

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

[11] 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.