Relieving Hot Spots on the World Wide Web

David Karger Eric Lehman Tom Leighton Matt Levine Daniel Lewin
Rina Panigrahy

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

Abstract

In this paper, we consider the problems associated with hot spots on the World Wide Web. A hot spot on the web is a site that is extremely popular, i.e., one for which there are many hits (i.e., requests) in a relatively short period of time. Hot spots are troublesome because they can cause servers to become swamped and message traffic in the vicinity of the hot spot to become delayed or lost altogether. 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. And the STOC 97 submission server on the night of the deadline.

In the paper, we describe a model of the web in which the hot spot problem can be analyzed, as well as some algorithms that can be implemented using existing TCP/IP protocols that prevent the occurrence of hot spots with high probability. The algorithms that we propose are quite simple. They make use of a special hashing process (which we call consistent hashing) and a caching protocol to guarantee a high level of service to all customers with high probability, even in the face of numerous hot pages as well as service failures. The algorithms work with local control, make efficient use of existing resources, and do not require all users to have a current or even consistent view of the network. The algorithms also scale

1 Introduction

Some time ago the World Wide Web Consortium requested we develop protocols to reduce the frequency and severity of hot spots on the World Wide Web. This paper reflects our initial progress on this problem. A hot spot is a web site that is extremely popular; that is, one for which there are many hits (i.e., requests) in a relatively short period of time. In fact, there may be so many hits that the site becomes "swamped," thereby rendering it unusable.

As usage 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, several political sites on the night of the election, and the STOC 97 submission server on the night of the deadline. Other examples include sites identified as "web-site-of-the-day" and sites that provide new versions of popular software. In some of these cases, users were denied access to the sites for hours or even days. Hot spots not only affect their own avalability on the internet but can also cause traffic destined for other nearby sites to be delayed due to localized or overall network congestion.

Several approaches for overcoming the problems associated with hot spots have been proposed. Most use some kind of replication strategy to store copies of hot pages throughout the Internet in an attempt to spread out the demand for the hot pages across several servers. For example, Malpani, et al. [?] pursue this strategy with a set of cooperating caching machines using the Multicast protocol. A user's request for a page is directed to a random cache. If the page is stored there, it is returned to the user. Otherwise, the cache forwards the request to all other caches. If the page is cached nowhere, the user's request is forwarded to the home site of the page. The disadvantage of this technique is that as the number of participating caches grows, the number of requests between caches becomes unmanageable.

The Harvest approach, which was introduced by Chankhunthod et al. [?], creates far less network traffic. Rather, copies of popular pages are stored in a tree of caching machines. A user obtains a page by asking a nearby leaf cache. If neither this cache nor its siblings have the page, then 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 retains for some time a copy of any page it obtains. The advantage of a cache tree is that many requests for a page in a short period of time will only cause one request to the home site of the page. A disadvantage is that many requests for distinct, uncached pages can overload the root cache since the same tree is used for all pages.

In independent work [1], Plaxton and Rajaraman [?] show how to balance the load among all caches by using randomization and hashing. In particular, they use a different tree for each web page and they use a hash function to assign each tree node to a random cache in the network. This helps to distribute the load among caches because a machine near the root for one page may be a leaf for another page. Plaxton and Rajarman's technique is also highly fault tolerant.

The Plaxton/Rajaraman algorithm has other drawbacks, however. For example, their algorithm sends a copy of each page request to a random node on each level of the tree for the page. Although some processor that is not swamped is likely to get the request, the protocol also ensures that the root of the tree will get swamped in the event of a hot page. In fact, the algorithm uses swamping as a feature since swamped machines are supposed to send copies of the hot page to their children in the tree. (This is how the pages actually get replicated throughout the tree). As a consequence, the algorithm may not be suitable for an environment such as the Internet, where swamped machines cannot be relied upon to perform any useful tasks. Moreover, the intentional swamping of large numbers of random machines could well be viewed unfavorably by the many owners of those machines (who are really innocent bystanders on the Internet). The Plaxton/Rajaraman algorithm also requires that all communications be synchronous and that messages have priorities, which is not compatible with current Internet protocols. Another drawback

[1] Our work was developed concurrently with and independently of [?], but was not submitted to FOCS 96 as the first author was a program committee member. Since that time, we have extended the work significantly.

in the Plaxton/Rajaraman algorithm is that it does not scale with the number of machines on the web. Their hash functions assume a constant set of caches available on the web. This is unrealistic as the web is actually growing at an exponential rate.

In this paper, we describe an algorithm for caching pages which overcomes the drawbacks of the preceding approaches and which has several additional desirable properties. Our algorithm is similar in some respects to the algorithms used by Harvest and Plaxton/Rajaraman. As in Harvest, we make a copy of each page as it moves down a tree (although we only cache copies that have been requested a sufficient number of times at a node in the tree in order to economize on memory). As in Plaxton/Rajaraman, we protect each web page from the barrage of page requests with a unique tree of caches specified by a hash function.

Our algorithm differs substantially in the way that the hashing is performed, however. In fact, one of the main reasons that our algorithms scale is our application of a new hashing technique which we call consistent hashing. Consistent hashing assigns a set of items to bins so that each bin receives roughly the same number of items. (For our application this implies good load balancing.) If the number of bins were to change, then to maintain this even distribution, some items would have to be reassigned to a different bin. The key feature of consistent hashing is that as the number of bins changes, the number of items which are reassigned is the minimum necessary to maintain an even distribution. In our approach to the hotspot problem, bins correspond to caches. Our use of consistent hashing allows the set of caches to vary with minimal disruption. We believe that consistent hashing is a general tool that will find other useful applications.

The key properties of our algorithm are summarized in what follows.

- Elimination of Hot Spots. Our algorithm balances the load of page requests across the servers in the network so that no server is likely to become swamped, thereby making the occurrence of hot spots arbitrarily rare. That is, we specify a randomized protocol which ensures that no machine will receive more than a specified number of requests (even for different pages) in any fixed interval of time with probability very close to one. This performance is in stark contrast to the Plaxton/Rajaraman algorithm which is designed so as to cause hot spots at some locations and which uses the occurrence of hot spots as a feature of the algorithm.

- Low Message Latency and High Throughput. Our algorithm attains near optimal message latency and throughput for every message, even in a hierarchical model of the Internet in which the communication times and bandwidths between various pairs of processors can vary dramatically.

- Efficient Utilization of Resources. Our algorithm does not perform excessive caching of hot pages, thereby making efficient use of the available memory in the network. (In particular, the number of cached pages is a small fraction of the number of page requests.) It also does not need to automatically send multiple copies of requests, as in Malpani's approach, thereby easing the load on network bandwidth.

- Fault Tolerance. Our algorithm is tolerant to failures in the network and it degrades gracefully in the presence of such failures.

- Indefinite Stability. Over infinite execution time, unlikely as they are, hot spots and other adversarial failures will occur. Our algorithm recovers gracefully from such failures and returns to its normal state.

- Graceful Scalability. One of the most important and distinguishing features of our algorithm is that it scales well and is designed to grow gracefully with the Internet. In particular, the probability that a cataclysmic event becomes necessary (such as a large scale update of a name space) can be made arbitrarily small. In addition, our particular implementation of consistent hashing allows for

logarithmic-time address calculations as well as constant time updates for addition or removal of a cache.

- Easy Startup. The algorithm also begins to reap advantages even if only a small portion of the users employ the algorithm. In particular, those who use the algorithm will enjoy a substantial improvement in performance even if others do not use the protocol. Even those who do not use the algorithm themselves will benefit if others use the algorithm. This "win-win" property is helpful since it means that a global agreement does not need to be in place before the algorithm can be used.
- Local Control. Our algorithm works with local control. It is not even necessary for each machine to have a current or consistent view of the Internet in order for the algorithm to perform well.
- Asynchrony. Our algorithm works in an asynchronous environment such as the web. It does not make use of any timing information or prioritized messages.
- Compatibility with Existing Protocols. Our algorithm is compatible with existing protocols such as TCP/IP.

The remainder of the paper is organized as follows. Section 2 formally defines our model of the Web and of the hotspot problem. We present our basic solution in Section 3. The scalability of our solution and the ideas of consistent hashing are discussed in Section 4. We extend our results to a more realistic communication model in Section 6 and discuss fault tolerance in Section 7.

2 Model

This section proposes a formal model of the world wide web and of the hotspot problem. Naturally, this model is a compromise between simplicity and realism. Some of the concessions are discussed below. We alter the model in Section 6 to more realistically portray communications between machines.

The description of the model is divided into three sections. Section 2.1 describes the components of the web and how they interact. The next section defines the hotspot problem in terms of the web model we propose. The last section defines the degrees of freedom we have in creating a protocol to address the problem.

2.1 Infrastructure

Machines

We classify computers on the web into three categories. Browsers are machines that allow a user to request pages from the web. Servers are the permanent homes of all web pages. Caches reside between browsers and servers, serving as a buffer for servers against the barrage of page requests from browsers.

Let $\mathcal{B}$ be the set of browsers, $\mathcal{C}$ be the set of caches, $\mathcal{S}$ be the set of servers, and $M = \mathcal{B} \cup \mathcal{C} \cup \mathcal{S}$ be the set of all machines. Let $B = |\mathcal{B}|$, $C = |\mathcal{C}|$, $S = |\mathcal{S}|$, and $M = |M|$. All browsers and caches store a representation of the set $\mathcal{C}$.

In practice, one machine might fall into more than one of these categories; for example, a single machine could be not only a browser, but also a server. But for simplicity, we assume that the sets $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{S}$ are disjoint.

Pages

Let P be the universe of pages. Each server stores a fixed subset of the pages, and these subsets partition P. Any machine can quickly determine which server is home to a requested page. This can be done in practice by inspecting the host name in the URL of the requested page.

Each cache can also store a set pages, and this set may change over time as dictated by our protocol.

In this paper, we assume that the contents of web pages do not change over time. Modifications to web pages are a problem for any caching scheme, since there is no easy way to determine whether a copy of a page held by a cache is the same as the original held by the server. Worse, some pages are modified automatically each time they are accessed, and therefore can never be cached. Our protocol can be extended in a natural way to allow for page updates, but we do not explore this issue here.

Communication

A browser or cache may request a page from a cache or server. A page request has three stages. First, a browser or cache m_1 asks for page p from a cache or server m_2. Then there is some delay while m_2 obtains page p. If m_2 already has the page stored, then we assume this delay is small. Otherwise, m_2 may initiate further requests to obtain the page. Finally, the page is returned to m_1.

A cache may store any page it obtains through a page request. The protocol dictates whether a page is stored and how pages are evicted from the cache.

A small amount of additional information may be passed with each page request or return. This extra information may be used, for example, to affect routing of pages or page requests.

Dynamics

Two activities on the web are outside the control of our protocol. First, if a browser or cache requests a web page directly from a server, then (unless the server is swamped) it returns the page immediately. Second, browsers may request arbitrary web pages at arbitrary times. However, the machine that receives a browser's page request is dictated by the protocol. This allows us to define protocols in which numerous browser requests for a single page are distributed across the entire set of caches.

2.2 The Problem

Roughly, the hotspot problem is to fulfill all browser page requests quickly while maintaining a reasonable upper bound on the number of outstanding page requests to any machine.

More precisely, suppose an adversary devises a schedule indicating times and pages for all browser page requests. The adversary has complete knowledge of the network and of our protocol, but can not anticipate any random numbers we might generate. Then we run the protocol, and browsers follow the adversary's schedule. We want to guarantee two things.

First, we want to ensure that there are no hotspots. That is, we want to guarantee a specified upper bound on the number of simultaneous, outstanding page requests. Each communication request to a machine consumes some of the machine's physical resources for the duration of the request. Therefore, if a machine receives too many page requests too quickly, all of the machine's resources will be consumed, and the machine is said to be swamped.

Second, we want to ensure that all browser page requests are fulfilled quickly. Initially we take a simplistic view of this objective, trying merely to minimize the number of hops (intermediate machines) on the path of requests from a browser to a server. Later we will extend our objective to natural models of internet latency and throughput. In a sense, this second objective competes with the first. By cleverly interposing caches between browsers and servers, we may be able to protect a popular server from becoming overwhelmed by page requests. But this layer of caches increases the number of hops between browsers and servers. A request for a page not stored in any cache must now make more hops, and will therefore take more time.

2.3 What Constitutes a Protocol?

This section describes the degrees of freedom we have in designing a protocol that satisfies the goals. A protocol defines three things: which machine receives a browser's page request, how a cache satisfies a page request, and what pages a cache stores.

There is one obvious restriction: each machine may only act upon data explicitly communicated to it through page requests. For example, we can not specify that a browser's page request is directed to an unloaded cache, unless information about the load on various caches is somehow explicitly communicated to the browser.

3 Basic Solution

In this section we provide a caching protocol assuming a simplified model of the web. Subject to our constraint of not swamping machines, we optimize a simplistic distance measure, namely the number of hops on a request path from a browser to a server. In Section 6 we propose a more realistic model which takes into account the actual distances between machines. Also in this section, we consider a large set of requests are all made simultaneously. In Section 5, we extend to a model in which requests are made over time. In order to model caching, we assume that requests for a page are "combined" at caches, so that any caches sends out exactly one request for each page that is requested from it (any number of times).

One natural caching scheme is to use a complete d-ary tree of caches. A browser sends a request to a random leaf, and any tree node passes incoming requests to its parent. Thus each cache gets at most d requests. In practice d would be about 100 because a typical workstation can handle a few hundred connections at one time.

This scheme lets us handle a 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, a drawback of such a scheme is that if there are several distinct pages that are requested at least once then 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 map each node of each page's tree independently to one of the caches at random.

Formally speaking, for each page P we have a balanced, degree d abstract tree, G_P, with the server of the page at the root. All the other nodes are abstract nodes which are mapped to caches. Let us number the abstract nodes in a tree from 1 to c in bfs order. Let $\pi(P, j)$ represent the machine to which the j^{th} abstract node in the abstract tree for page P is mapped. We can formally state our protocol as follows.

Which Machine Receives a Browser's Page Request?

Suppose a browser B needs page P. The browser computes a random abstract leaf t in the tree G_p and makes the request for page p to the cache $C = \pi(P, t)$. We assume that along with the name of the page P the browser also tells the cache C the list of abstract tree nodes on the path up the tree, G_P, along which the request would hop.

How Does a Cache Satisfy a Page Request?

Suppose cache C receives a request for page P. It looks at the next abstract node, t', above it on the list passed to it. If c already has the page it serves the request. If the next node is the root node then it forwards the request to the server of the page. Otherwise it forwards the request to the cache $C' = \pi(P, t')$ unless it already has an outstanding request for the same page P to the same abstract node t' mapped to C'.

We will now do a static analysis of the protocol where an adversary initiates a total of rc requests over all pages. In this static analysis we will assume that caches are perfect and have sufficient space in the sense that if a cache has already made a request for a page it will not make another request for the same page. In Theorem 1 we provide high probability bounds on the number of requests an arbitrary caching machine m gets assuming that all the outputs of the function π are independent and random. However, note that in practice such functions are simulated by using pseudo-random hash functions. This can be achieved by choosing π randomly from a k-Universal Hash family which guarantees that outputs of the function π are k-way independent for some small k. Theorem 2 extends our high probability analysis to the case when π is a k-way independent function. In particular we show that it suffices to have $\log N$-way independent functions to achieve the same high probability bounds as with full independence.

3.1 Analysis for Random π

In this section we provide high probability bounds for the total number of requests received by a machine if all the tree nodes for all pages are mapped randomly and independently to the set of caches. We will assume that the total number of requests is at most rc.

Theorem 1 If π is chosen uniformly and at random from the space of functions $\mathcal{P} \times [\,] \mapsto \mathcal{C}$ then with probability at least $1 - 1/N$, where N is a parameter, an arbitrary cache gets no more than $r(2\log_d c + O(\frac{\log N}{\log\log N})) + O(\frac{d\log N}{\log(\frac{d}{r}\log N)} + \log N)$ requests.

Proof: 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 as Leaf nodes

We can think of our protocol as first running on the abstract trees and then mapping these abstract tree nodes randomly onto the set of caches. First we will bound with high probability the maximum number of requests an abstract leaf node can get. Then assuming that all the abstract nodes get less than that many requests we will bound the number of requests a cache gets when these abstract nodes are mapped onto the set of caches. Since the number of requests for a page is at most rc and since there are at least $c(1 - 1/d)$ leaves no abstract leaf gets more than $rd/(d-1)$ requests on expectation. In fact Chernoff bounds tell us that an arbitrary abstract leaf gets more than $O(r + \log N)$ with probability at least $1 - 1/N^3$. Since there are only rc requests in total we can say that no abstract leaf gets more than $y = O(r + \log N)$ requests with probability at least $O(1 - rc/N)$. Let $v_{i,j}$ denote the number of requests received by the i^{th} leaf of the j^{th} page's tree. Then with probability at least $1 - rc/N^3$ all the $v_{i,j}$'s lie between 0 and y.

Let us now map the abstract trees onto the set of caches randomly. For all i, j, an arbitrary machine m either gets all the $v_{i,j}$ requests which happens with probability $1/c$ or gets none them. We need to bound with high probability a sum of random variables each of which lies between 0 and y. So if we define random variables $U_{i,j}$ to be $v_{i,j}/y$ when the i^{th} leaf for the j^{th} page is mapped to machine m which happens with probability $1/c$ and 0 otherwise. Also if $W = \sum_{i,j} U_{i,j}$ then $E[W] = \sum_{i,j} v_{i,j}/(cy) \leq r/y \leq 1$. Since $U_{i,j}$'s lie between 0 and 1 Chernoff bounds apply and we can say that $W = \sum_{i,j} U_{i,j}$ can exceed k with probability at most $e^{-(k\ln k - k + 1)}$. To make this probability as small as $1/N$ we must have $k = \Omega(\log N/\log\log N)$. So W is $O(\log N/\log\log N)$ with probability at least $1 - 1/N$. Since the total number of requests received by m is yW we can say that with probability at least $1 - 1/N$ a machine will receive at most $O(w\log N/\log\log N) = O((r + \log N)\log N/\log\log N))$. However this was subject to the assumption that all the $v_{i,j}$'s are less than $O(r + \log N)$ which happens only with probability $1 - rc/N^3$. Assuming $rc < N$ we can say that the high probability bound just proved applies with probability at least $1 - O(1/N)$. It is easy to extend our analysis to show a bound of $O(r\log N/\log\log N)$ with probability at least $1 - 1/N$ which we do not present here.

Hits as internal nodes

Again we think of the protocol as first running on the abstract trees. Now no abstract internal node gets more than d requests because each child node gives out at most one request for a page. Consider any arbitrary arrangement of paths for all the rc requests up their respective trees. Since there are only rc requests in all we can bound the number of abstract nodes which get d requests. In fact we will bound the number of abstract nodes over all trees which receive between $d/2^{j-1}$ and $d/2^j$ requests. Let n_j denote the number of abstract nodes that receive between $d/2^{j-1}$ and $d/2^j$ requests. This divides the abstract nodes into $\log d$ categories. Since these abstract nodes are mapped randomly onto the set of caches we can bound the number of requests a cache gets due to each category of abstract nodes.

Let x_i be the number of requests for the i^{th} page. Then $\sum x_i \leq rc$. Since each of the rc requests gives rise to at most $\log_d c$ requests up the trees the total number of requests is no more than $rc \log_d c$. So,

$$\sum_{j=1}^{\log d} \frac{d}{2^j} n_j \leq rc \log_d c \tag{1}$$

The following lemma gives a tight bound for n_j.

Lemma 1 The total number of internal nodes which receive at least k requests is at most $2rc/k$ if $k > 1$

For $k = 1$ clearly there can be no more than $rc \log_d c$ internal nodes. The preceding lemma tells us that $n_j \leq \frac{2rc}{d/2^j}$ except for $j = \log d$. For $j = \log d$, n_j will be at most $rc \log_d c$. Now the probability that machine m assumes an arbitrary one of these n_j nodes is $1/c$. Also since assignments to different tree nodes are independent the probability that m is assigned to more than x of these nodes is at most $\binom{n_j}{b}(1/c)^b \leq (en_j/cb)^b$ In order for the right hand side to be as small as $1/N$ we must have $b = \Omega(\frac{n_j}{c} + \frac{\log N}{\log(\frac{c}{n_j}\log N)})$. Note that the latter term will be present only if $\frac{c}{n_j}\log N > 2$. So b is $O(\frac{n_j}{c} + \frac{\log N}{\log(\frac{c}{n_j}\log N)})$ with probability at most $1 - 1/N$

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

$$\begin{aligned}
\sum_{j=1}^{\log d} \frac{d}{2^{j-1}}\left(\frac{n_j}{c} + \frac{\log N}{\log(\frac{c}{n_j}\log N)}\right) &= \sum_{j=1}^{\log d} \frac{d}{2^{j-1}}\frac{n_j}{c} + \sum_{j=1}^{\log d} \frac{d}{2^{j-1}}\frac{\log N}{\log(\frac{c}{n_j}\log N)} \\
&\leq 2r\log_d c + \sum_{j=0}^{\log d - 1} \frac{d}{2^{j-1}}\frac{\log N}{\log(\frac{d}{2r\cdot 2^j}\log N)} \\
&\quad + \frac{d}{2^{\log d - 1}}\frac{\log N}{\log(\frac{c}{n_{\log d}}\log N)} \\
&\leq 2r\log_d c + \sum_{j=0}^{\log d - 1} \frac{d}{2^j}\frac{\log N}{\log(\frac{d}{r}\log N) - j - 1} + 2\log N \\
&= 2r\log_d c + O\left(\frac{d\log N}{\log(\frac{d}{r}\log N)} + \log N\right)
\end{aligned}$$

So we have proved that with probability at least $1 - (\log d)/N$ the number of inode requests a cache gets is no more than $2r\log_d c + O(\frac{d\log N}{\log(\frac{d}{r}\log N)}) + \log N)$

So by combining all the high probability bounds for internal and leaf nodes we can say that a machine m gets $2r\log_d c + O(\frac{d\log N}{\log(\frac{d}{r}\log N)} + \log N + \frac{r\log N}{\log\log N})$ requests with probability at least $1 - O(\log d/N)$. By replacing N by $N\log d$ we get Theorem 1. □

Proof of Lemma 1: Look at the x_i requests for the i^{th} page and the paths produced by these requests up the tree. Consider the tree on the internal nodes induced by these paths. We wish to bound the number of nodes with downward degree at most k. Since $k > 1$ we can contract out all the nodes with downward degree equal to one. We are now left with a tree where each downward degree is at least 2. In such a tree the number of leaves is at least half 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 x_i. So if there are p_i nodes with downward degree of at least k then $kp_i \leq 2x_i$ and so $p_i \leq 2x_i/k$. So the total number of nodes over all trees which receive at least k requests is no more than $\sum 2x_i/k = 2rc/k$. $\Box$.

3.2 Tightness of the high probability bound

In this section that the high probability bound we have proved for the number of requests received by a machine m is tight. We can prove the following lemma.

Lemma 2 There exists a distribution of rc requests among pages so that an arbitrary machine m gets $\Omega(r \log_d c + \frac{d \log N}{\log(\frac{d}{r} \log N)} + \frac{r \log N}{\log \log N})$ requests with probability at least $\Omega(1/N)$.

Proof: If each of the rc requests is made for a different page then the total number of requests will be $rc \log_d c$ and so the expected number of requests received by m is $r \log_d c$. This justifies the presence of the $r \log_d c$ term in the bound.

To justify the $\frac{d \log N}{\log(\frac{d}{r} \log N)}$ term we let the adversary divide the rc requests equally among rc/d^2 pages so that each page gets d^2 requests. Now for each these pages the machines at the level just below the root node will get d requests. Also the probability that machine m is present at an arbitrary second level node of an arbitrary tree is $1/c$. So the probability that m is present at x of these rc/d second level nodes is at least $\binom{rc/d}{x}(1/c)^x(1-1/c)^(rc/d-x)$. To reduce this probabilility to $1/N$, x must be $\Omega(\frac{d \log N}{\log(\frac{d}{r} \log N)})$.

Finally for the $\frac{r \log N}{\log \log N}$ term we let the adversary devotes all the rN requests to one hot page. Then since there are about c leaf positions, with high probability at least one machine will occupy $\frac{\log N}{\log \log N}$ of the positions and will receive $O(\frac{r \log N}{\log \log N})$ requests. $\Box$

3.3 Analysis for k-way Independent π

We now extend our high probability analysis to functions π which are chosen at random from a k-Universal Hash family. Note that random choices are made even in assigning even for assigning browser originated requests to leaves of the respective trees. If we use random hash functions from a k-Universal hash family to assign requests to leaves and to map abstract tree nodes to caches then we can prove the following theorem.

Theorem 2 If π is chosen at random from a k-Universal Hash family and if the assignment of requests to random leaves is k-way independent then with probability at least $1 - 1/N$ an arbitrary machine receives no more than $2r \log_d c + O((kN(d+r))^{1/k}(1 + \frac{r}{k} + \frac{d}{\log(kd/r)} + \frac{r}{\log k}))$ requests.

Now by setting $k = \log N$ we find that the bound is $2r \log_d c + O(\log N + \frac{d \log N}{\log(\frac{d}{r} \log N)} + \frac{r \log N}{\log \log N})$ which is identical to the bound with complete independence!

Corollary 1 The high probability bound proved in theorem 1 for the number of requests a cache gets holds even if π is selected from a $\log N$-Universal Hash family.

Proof of Theorem 2: The high probability bound with complete independence was derived using Chernoff bounds. For a sum of k-way independent random variables we can apply the Markov inequality to the k^{th} power of the sum. If H is a random variable denoting the number of requests received by machine m with totally independent hash functions then we know that with probability at least $1 - 1/N$, $H <$

$2r\log_d c + O(\frac{d\log N}{\log(\frac{d}{r}\log N)}) + \log N + \frac{r\log N}{\log\log N})$. Let μ denote $2r\log_d N$. If H' is the random variable denoting the number of requests received by m when the hash functions are k-way independent then by applying Markov Inequality to $(H'-\mu)^k$ we get,

$$Pr[(H'-\mu) > t] < \frac{E[(H'-\mu)^k]}{t^k}$$

So in order to reduce this probability to $1/M$ we need to have

$$t > (E[(H'-\mu)^k])^{1/k} \cdot M^{1/k} \tag{2}$$

We need a bound for $E[(H'-\mu)^k]$.

Let V_{ij} be a Bernoulli random variable denoting that event that the j^{th} node in the i^{th} page is assigned to machine m. Let U_{ijk} be a Bernoulli random variable denoting the event that the j^{th} request for the i^{th} page starts off at the k^{th} leaf in the tree for that page.

Now both H' and H can be written as a the same sum of product pairs of V_{ij}'s and U_{ijk}'s. So $(H'-\mu)^k$ can be expanded into a sum of products where each product term will consist of at most k of the V_{ij}'s and atmost k of U_{ijk}'s. But since these Bernoulli variables are k-way independent we have $E[(H'-\mu)^k] = E[(H-\mu)^k]$. We can rewrite the high probability bound on H as follows: The probability that $R-\mu > n$ is $e^{-\Omega(n)} + e^{-\Omega(\frac{n}{d}\ln\frac{n}{r})} + e^{-\Omega(\frac{n}{r}\ln\frac{n}{r})}$. Let $\rho(n)$ denote this probability. Then note that for any positive integer n_0, Let V_{ij} be a Bernoulli random variable denoting that event that the j^{th} node in the i^{th} page is assigned to machine m. Let U_{ijk} be a Bernoulli random variable denoting the event that the j^{th} request for the i^{th} page starts off at the k^{th} leaf in the tree for that page.

Now both H' and H can be written as a the same sum of product pairs of V_{ij}'s and U_{ijk}'s. So $(H'-\mu)^k$ can be expanded into a sum of products where each product term will consist of at most k of the V_{ij}'s and atmost k of U_{ijk}'s. But since these Bernoulli variables are k-way independent we have $E[(H'-\mu)^k] = E[(H-\mu)^k]$. We can rewrite the high probability bound on H as follows: The probability that $R-\mu > n$ is $e^{-\Omega(n)} + e^{-\Omega(\frac{n}{d}\ln\frac{n}{r})} + e^{-\Omega(\frac{n}{r}\ln\frac{n}{r})}$. Let $\rho(n)$ denote this probability. Then note that for any positive integer n_0,

$E[(H-\mu)^k]$ is less than

$$n_0{}^k + \sum_{n=n_0}^{\infty} n^k\rho(n) < n_0{}^k + \sum_{n=n_0}^{\infty} e^{-\Omega(n-k\ln n)} + \sum_{n=n_0}^{\infty} e^{-\Omega(\frac{n}{d}\ln\frac{n}{r}-k\ln n)} + \sum_{n=n_0}^{\infty} e^{-\Omega(\frac{n}{r}\ln\frac{n}{r}-k\ln n)}$$

We need to set n_0 optimally. Observe that all the exponents are $\Omega(n)$. So after a certain number of terms we can treat the sums as geometric sums. By finding the value of n where the terms in the sums start decreasing we find the optimal value of n_0 to be $O(k + r + kd/\log(kd/r) + kr/\log k)$. At that value of n_0, $(E[(H-\mu)^k])^{1/k}$ is $O((k + r + \frac{kd}{\log(kd/r)} + \frac{kr}{\log k})(d+r)^{1/k})$. Equation 2 completes the proof. □

4 Consistent Hashing

In this section we introduce a new kind of hashing called consistent hashing. The difference between a consistent hashing scheme and regular hashing is that when a number of elements are hashed into a table of size n and then the size of the table is incremented to $n+1$, most of the elements in the new table are hashed to the same place as before. In fact, we require that only the minimal fraction of elements needed to retain an even distribution in the table be moved. Most well known hashing schemes do not have this property. That is, when the size of the table changes, items are rehashed so that the positions of almost all elements in the table are changed.

Consistent hash functions are vital to our web caching scheme in a number of places. First, we assume that new caching machines will be added to, and removed from the web every day. In order to avoid a central control service that notifies other caches about new/removed machines so that they may adjust their protocols, we assume that notification will be passed from machine to machine in much the same way as network news. Thus, not all caches in the web will know about the exact same set of machines at any given moment. The effect of this is that not all caches are hashing tree nodes into the same size "table" of caches. Using a regular hash function, the trees that these caches would build for a page would be totally different, and our protocol would not work at all. Using a consistent hash function solves this problem. Additional sections where consistent hashing plays a central role are fault tolerance (Section 7) and the realistic communications model (Section 6). In all of these cases the essential property that we require is that small changes in the hash function be made incrementally. Hash functions that require a large change when the range size crosses some boundary do not suffice since when this happens, the whole caching scheme will be disrupted.

We define below an consistent hash family, and then show how we can implement such hashing.
Definition A consistent hash family H on a universe U is a family of functions of the form $f : 2^U \times U \mapsto U$ such that for all $R \subseteq U$ and $x \in U$, $f(R, x) \in R$, and the following properties hold:

1. For all $R \subseteq U$, $\{f(R, \cdot)\}$ is a k-universal hash family. In other words, if we fix an R we get a k-universal hash family into R.
2. Range Consistency: For all $R, S \subseteq U$ the following holds with high probability: $\frac{1}{|D|} \sum_{x \in R \cup S} |f^{-1}(R, x) \oplus f^{-1}(S, x)| \leq 1 - O(\frac{|R \cap S|}{|R \cup S|})$. $\oplus$ is the symmetric difference between sets. The probability is over the choice of the hash function.
3. Monotone Property: $R \subseteq S \subseteq U \Rightarrow f^{-1}(S, r) \subseteq f^{-1}(R, r)$ for all $r \in R$.

Range consistency insures that changes are small, and incremental. The sum on the left is the fraction of elements that change position when changing the range from R to S. Consistent hashing is optimal in the sense that the quantity on the right is the minimal fraction of elements that must change positions in order for the distribution of elements into R to remain balanced. The monotone property states that when the range is changed, the only elements that move are those that get reassigned to the new spots.

Theorem 3 There exists a consistent hash family such that all range updates and hash function computations can be performed in $O(\log n)$ time per operation.

The remainder of this section proves the above Theorem. Our basic scheme is as follows. We assume for simplicity that U is the natural numbers. In the following, we assume that we can hash integers into the real interval $[0, 1]$. In fact this can be done by seeding a pseudo-random number generator with the integer and generating bits of the real number, or by using scaled standard hashing techniques.

We build the function as follows: given a range S ($|S| = i$) we first associate with each integer $j \in S$ a random real number $r(j)$ in the unit interval $[0, 1]$. This is done by a hash function h_1 mapping integers into reals. This is called the setup stage, and is only done once.

$f(R, d)$ is computed as follows. A second hash function h_2 from integers to reals is used to map d into a real number $r(d)$ in $[0, 1]$. $f(R, d)$ is then defined to be the $j \in S$ such that $r(j)$ is nearest to $r(d)$. This hash computation can be implemented in $O(\log n)$ time using a standard balanced binary tree data structure.

The hashing scheme is defined with two hash functions since the first, h_1 is used only once in the setup stage, thus h_1 may be an expensive (in time) hash function. h_2 is used every time we compute a hash value so h_2 should be efficient. Note however that we only demand that the numbers generated by h_2 are k-wise independent.

Intuitively, the hash function h_1 spreads the elements S randomly over the unit interval such that each point "covers" a roughly equal fraction of the unit interval. Thus, whenever we hash an element of U into

the unit interval, it is equally likely to land in the coverage area of any point. When we add a new element of the range via h_2, it "captures" only those elements of U that are closer to it than any other element of R; this will be roughly a $1/|R|$ fraction of the elements of U. Any other elements will be left in their original places.

In the scheme it is likely that there exist two $r(j)$'s that have a large gap between them in the interval. The size of this gap is likely to be $O((\ln|R|)/|R|)$. Thus, these two hash values will be assigned a disproportionate amount of values, and the resulting hashing will not be k-universal. To overcome this problem we map $O(\ln|R|)$ copies of each j in the setup stage onto the interval. We prove that in this scheme, with high probability the hashing is k-universal.

Notice that in the scheme, each hash value $j \in S$ has associated with it a set of intervals made up of the points closest to the $O(\ln i)$ $r(j)$'s. The probability that a hashed element assumes the value j is the total length of all these intervals .

Lemma 3 If $q \ln i$ copies of each hash value are mapped onto the real unit interval then with probability $\geq 1 - O(q \ln i \cdot i^{1-q\epsilon^2/3})$ the total length of the subintervals corresponding to a hash value is $\leq (1+\epsilon)/i$.

Proof: We will try to bound the sum of the intervals to the left of the $q \ln i$ copies of a hash value. Due to symmetry the same bound will apply to the sum of the right intervals. This gives us a bound on the total interval assigned to the copies of a hash value which is simply the average of the two sums. Now could have easily analysed the sum of the left intervals if their lenghts were independent which unfortunately is not the case. However if we modify our experiment slightly and think of it as a Poisson process with rate $qi \ln i$ then all the inter-arrival times will be independent. We will show that in such a Poisson process the probability of ending up with exactly $qi \ln i$ arrivals in unit time is $\Omega(1/q \ln i)$. Using this fact we derive a bound on the sum of the left intervals in our original experiment from the bound on their sum in the Poisson process.

Let B denote the event that this sum exceeds $(1+\epsilon)/i$. Instead of throwing $qi \ln i$ hash value copies onto a unit length interval let us consider a Poisson process with rate $qi \ln i$. The Poisson process stops after $qi \ln i$ arrivals. These arrivals correspond to placement of the $qi \ln i$ copies of the hash value in random order. Let B denote the event that the sum of the time intervals immediately to the left of the points of arrivals of $q \ln i$ copies of an arbitrary hash value exceeds $(1+\epsilon)/i$. Now in a Poisson process all the inter-arrival times are independent. So the sum of the $q \ln i$ intervals has the same distribution as the time for $q \ln i$ arrivals. So $Pr[B]$ is upper bounded by the probability of $q \ln i$ arrivals in $(1+\epsilon)/i$ time. Since the expected number of arrivals in that time is $(1+\epsilon)\ln i$, by Chernoff bounds $Pr[B] \leq e^{-\epsilon^2 q \ln i/3} = i^{-q\epsilon^2/3}$. Let U be the random variable denoting the number of arrivals at $t = 1$.

$Pr[B] = \sum_k Pr[B|U=k]Pr[U=k]$

In particular it follows that

$Pr[B] \geq Pr[B|U = qi \ln C]Pr[U = qi \ln i]$

Suppose we condition on the event $U = qi \ln i$. Since in a Poisson process each infinitesimal time interval is equally likely to be an arrival point, constructing the outcome corresponds to picking $qi \ln i$ random points on the unit real interval. In other words, if we let D be the event that in our original experiment the sum of the intervals immediately to the left of the $\ln i$ copies of a hash value exceeds $(1+\epsilon)i$ then,

$Pr[B] \geq Pr[D]Pr[U = qi \ln i]$

and therefore

$$\begin{aligned} Pr[D] &\leq Pr[B]/Pr[U = qi \ln i] \\ &\leq i^{-q\epsilon^2/3}/Pr[U = qi \ln i] \end{aligned}$$

Since arrival rate, $\lambda = qi \ln i$, $Pr[U = qi \ln i] = e^{-\lambda}\lambda^\lambda/\lambda!$. This turns out to be $\Omega(1/(q \ln i))$. So $Pr[D] = O(q \ln i \cdot i^{1-q\epsilon^2/3})$

So we have bounded the sum of the intervals to the left of the $q \ln i$ copies of a hash value. Similarly we can bound the right intervals and since the total length of the intervals corresponding to the hash interval is the average of the two the same high probability bound applies. □

Proof of Lemma 3 Now we sketch the proof that this hashing scheme is indeed consistent hashing. Assume that $|S| = i$. We have three properties to verify:

Fix S, now according to the previous Lemma, any hash value $j \in S$ assumes responsibility for less than $\frac{1+\epsilon}{i}$ of the interval with high probability. Since the numbers that h_2 generates are k-wise independent, we have that $Pr[f^t(S, d_1) = f^t(S, d_2) = \ldots = f^t(S, d_k)] \leq \frac{(1+\epsilon)^k}{i^k}$ conditioned on the event that we have a "good" function (one that spreads evenly the hash values in the interval). Since the probability that we have a such a good function is large, we have very close to k-uniform hashing.

We now show range consistency. We first show range consistency for $R \subseteq S$. Assume that $S \backslash R = V$ and that $|V| = a$. According to the previous theorem, the a elements in V are with high probability responsible for less than $a\frac{1+\epsilon}{i}$ of the interval. So we have by Chernoff bounds that the probability that more than a $O(a\frac{1+\epsilon}{i})$ fraction of the hashed elements move when V is added to R is small (Chernoff bounds) Thus, we have shown that conditioned on the event that we have a "good" function, we have range consistency for $R \subseteq S$. Range consistency for arbitrary R and S follows from this case in the same fashion.

The monotone condition follows immediately from the description of the hashing scheme. Since when we add new points to the real line, the only hashed points that change are those moving to the new points. □

In terms of our caching scheme, the hash function is as follows. Each machine in the protocol is aware of a set of caching machines which are playing the protocol. Each machine maps each caches multiple times into the real interval. Then when we want to compute the cache that is assigned to a tree node, we hash the tree node into the line, and take the closest machine. The three properties of consistent caching are vital to our caching scheme. K-universality is required to assure no machine is swamped. The range consistency property is needed to insure that even if caches have slightly different sets of machines they know about, the scheme still works. The monotone property is needed for the extension of the protocol to the realistic communication model.

5 Adding Time to the Model

In Section 3, we considered a static model in which all requests happened simultaneously, and showed that no machine received too many requests. In this section, we add time to the model. For a given rate of browser requests for pages, we bound the rate at which any one machine receives requests. Our goal, of course, is to determine how fast browsers can request pages before caches begin to get swamped by requests faster than they can handle them.

Clearly if there are rc browsers they can generate an average of r requests per cache under any protocol simply by all simultaneously sending a request. Thus, if we wish to upper bound the load on caches, we must upper bound the number of browsers in the system by some bound rc. Similarly, if browsers can generate requests infinitely often, then they can overload any set of caches that can serve requests at a finite rate. Thus we also assume that each browser issues at most one request in any interval of length T_b.

We now consider an adversarial model. Suppose that each browser chooses, in advance, a set of pages they will request together with the times they will request them. The adversary decides (in advance) how long each page takes to serve once it is requested (the reflects the fact that some pages might be very large and take a long time to transmit to the browser). We assume that when a browser requests a new page, he gives up on the old one if it has not yet arrived. Clearly, since the adversary wishes to maximize the communication load on the network, he will keep any brower's request path active until the browser makes its next request. Our goal is for the protocol to run well with high probability under any such sequence.

Let us consider the set of (at most rc) requests outstanding at any given time t. Since the adversary has chosen the request schedule in advance, this is just, for each browser, the last page it was scheduled to request before time t. We would like to apply our static analysis to argue that the number of communication requests incident on any machine at this time is, with high probability, $O(r \log_d c + d \log c)$.

Our static analysis assumed that all requests arrived simultaneously and "merged" multiple requests for a page at a cache into a single request for that page from the parent of that cache. It was under this model that we bounded the number of requests. In order to make this work in the temporal model, we need a mechanism for combining requests that arrive at different times. This method is, of course, the caching of pages. When a machine receives a request for a page, it serves it immediately if the page is already cached. If the page is not cached but has already been requested from a parent, the cache waits for the original request to be satisfied and then serves all requests that have arrived in the meantime. If the page is not in cache and has not been requested, it sends the request on to a parent.

Suppose for now that each cache is large enough to hold all pages requested in a time interval of length T_b. It follows that over any interval of length T_b, at most one request for a page is generated by any machine (cache or browser). Thus our static analysis applies to show that in time T_b, at most $O(r \log_d c + d \log c)$ requests are received by any cache with high probability. Turning this around, we deduce that our protocol will work with high probability if each cache is able to store $O(r \log_d c + d \log c)$ pages and uses an LRU protocol to evict them.

Lemma 4 If every cache has room for $O(r \log_d c + d \log c)$ pages, then with high probability, in any time interval of length T_b, no machine will receive more than $O(r \log_d c + d \log c)$ communication requests.

Observe that d is lower bounded by the above analysis in terms of the number of simultaneous requests that a cache is willing to accept.

Corollary 2 Over any polynomial length of time, using the space described above, with high probability no machine will ever become swamped.

By changing the constants in the O-notation, we can make the exponent in the polynomially small failure probability arbitrarily large.

Of course, we would like for this protocol to run indefinitely. After a sufficiently long time, even extremely unlikely events (machines swamping) become likely. Even worse, if use used a fixed amount of randomization (the initial choices of universal hash functions) then the adversary can generate a sequence that has a tuny (but nonzero) small probability of permanently swamping certain machines. We would like to ensure that with high probability, even if some machines are swamped at a given time, the do not remain swamped. One way to model this is to allow the adversary to start the machine in an arbitrarily bad state (with swamped machines, particular cache trees, etc.) and prove that with high probability, a short time later, the system restabilizes without swamped machines. We can achieve this goal by having each machine, at large randomly chosen intervals, "rerandomize" its location on the real line used to generate the consistent hash function. This essentially guarantees that after a polynomial amount of time, the state of the system is independent of its starting state, and allows us to derive the same results as were stated above. We prove the following in the full paper:

Theorem 4 A variant of our protocol achieves the following: with probability one as the length of protocol execution grows unboundedly, the fraction of time any machine spends swamped is bounded by one over an (arbitrarily large) polynomial. The amount of additional communication required (over page requests) is negligible.

6 Improved Communication Model

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 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 can 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 can compute $\delta(m_1, m_2)$ for any machine m_2.

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.

6.1 Protocol

For $M \in \mathcal{M}$ and $l \geq 0$, let $L(m; l) = \{C \in \mathcal{C} | \delta(\mathcal{M}, \mathcal{C}) \leq \updownarrow\}$. (That is, $L(M; l)$ is a radius l ball around M in the ultrametric space.) Assume Z_C and z are defined as before.

The basic idea of the protocol is the same as in the unit metric case: use consistent hashing to map an abstract tree of caches onto the actual caches. The ultrametric information is incorporated into the protocol by restricting the set of caches used to construct a tree to those caches that are not farther away from the host with the page than the requesting browser is. By doing this, we insure that our path to the host does not contain any caches that are unnecessarily far away in the metric. We also assume that when a browser sends a request to a cache c it also sends the list of caches on the path up the tree.

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 $1/r$. 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.

Theorem 5 Let δ be an ultrametric. Suppose that the number of requests made is at most rc. Then in the protocol above, an arbitrary cache gets no more than $2r \log_d c \log rc + O(\frac{\log rcd \log N}{\log(\frac{d}{r} \log N)} + \log N + \frac{r \log N}{\log \log N})$ requests with probability at least $1 - 1/N$ where N is a parameter. The latency of a page request by browser B to a page on host H is at most $\lceil \log_d c \rceil \delta(B, H)$.

Proof: We fix an arbitrary cache x and prove the theorem for x. We consider the clustering of machines according to their distance from x. Denote the resulting clusters by $k_1 \subseteq k_2 \subseteq \ldots \subseteq k_a$. According to the protocol, if a request from a browser b is going to have any chance of loading x, b must request a page p such that $\delta(x, p) \leq \delta(b, p)$, or in other words that the minimal cluster containing b and p contains also x. From now on we assume that all requests are to such pages since this is the worst case.

First, notice that the set of caches used in the protocol for a browser b requesting p will be the set of caches in the minimal cluster around x containing b and p. This is because any cache that is as close to p as b is, will be in the minimal cluster containing b and p which also contains x by assumption. Thus, if the protocol is being played on a set of caches inside a given cluster k_i, the request must have been initiated by a browser inside that cluster. Furthermore, if the request is being made to a page p in cluster k_i, then the set of caches used by the protocol for that page request is exactly the set of caches in cluster k_i.

Assume that the number of requests a single browser may initiate is bounded by r. Then, the number of page requests being serviced by the protocol on a given clusters caches is at most r times the number of elements in that cluster since the request must have come from a browser inside the cluster.

We now look at the requests made to x where the original page request was to a page in cluster k_i. From the above we see that there are at most $r|k_i|$ such requests and they are all serviced by the protocol using the set of caches in k_i. Since we lower bounded the density of caches, the number of caches in k_i is at least $|k_i|/r$. Therefore from theorem 1 we deduce that the probability that x receives more than $2r \log_d |k_i|/r + O(\frac{d \log N}{\log(\frac{d}{r} \log N)} + \log N + \frac{r \log N}{\log \log N}) \leq 2r \log_d c + O(\frac{d \log N}{\log(\frac{d}{r} \log N)} + \log N + \frac{r \log N}{\log \log N}) = B$ requests is less than $1/N^2$.

Thus, we conclude that the probability that x receives more than aB requests (where a is the number of clusters around x) is less than $1/N$ since $Pr[x\text{receives more than } aB \text{ requests from } \bigcup k_i] \leq \sum_{i=1}^{a} Pr[x\text{receives more than } B \text{ requests from } k_i] \leq a/N^2 \leq 1/N$.

Unfortunately, the number of clusters around x could be as large as rC, 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 5 If there are a set of consecutive clusters $k_i, k_{i+1}, \ldots, k_{i+b}$ such that $\frac{|k_{i+b}|}{|k_i|} \leq 2$, then $\Pr[x$ receives more than B requests from $\bigcup_{j=i}^{i+b} k_j] \leq 1/N^2$.

The previous lemma implies that when summing over clusters to compute the bound on the number of requests made to x we need only sum over clusters that are at least twice the size of the previous one. Thus the sum is only made over $\log rc$ terms and the bound that we achieve is thus $\log rc$ times the previous bound. The bound on path length is clear from the definition of the protocol. □

Proof of Lemma 5 For the requests to pages in cluster k_j, a tree is built in the protocol with $|k_j|$ nodes. Since $\frac{|k_{i+b}|}{|k_i|} \leq 2$, observe that the difference between the structure of the smallest tree (built for pages in k_i) and the largest tree (built for pages in k_{i+_b}) 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 x than the real protocol, and to apply Theorem 1 to the modified protocol. Specifically we show that if we use the largest tree (of size $|k_{i+b}|$) filled with the smallest set of caches (those in k_i) for all of the page requests to pages in $\bigcup_{j=i}^{i+b} k_j$, that x will only get more requests than in the real protocol.

The analysis is divided again into two cases: internal nodes, and leaf nodes. We show that for both cases, when we add nodes to trees, and reduce the number of caches filling them, then this can only be worse for x.

Assume that x appeared as an internal node in a tree. Observe that with the real line 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. This is clear since the points that the tree nodes map onto do not change, and the closest cache mapped onto the line will change only if the closest cache is being removed. Since x is not removed from the set of caches, the number of times that x appears as an internal node of the tree can only go up. The original places that x appeared remain unchanged. Thus x can only receive more requests in internal nodes by reducing the number of machines. Note that the number of internal nodes may also go up since new nodes are being added at leaf positions. Let I_x be the random variable denoting the number of requests x receives in internal nodes in the regular protocol, and let ${I_x}'$ be the number of requests x receives in the modified protocol. We have shown that ${I_x}' \geq I_x$.

New tree positions are being added as leaf nodes, and some nodes that were previously leaf nodes are now covered up, and become internal nodes. In the resulting tree there will be at least as many leaf nodes as in the original tree. We observe that the number of requests that x receives in leaf node positions in a given tree is proportional to the number of times that x appears in a leaf node. Let E_x be the random variable denoting the number of requests x receives in leaf nodes in the regular protocol, and let ${E_x}'$ be the number of requests x receives in the modified protocol. ${E_x}'$ stochastically dominates E_x. Thus, high probability bounds for ${E_x}'$ imply high probability bounds for E_x

We now apply Theorem 1 to the modified protocol in the same manner as we did for the single largest cluster $k+_{i+b}$, and derive the same high probability bound. This bound holds for all of the requests made to pages in clusters $k_i, k_{i+1}, \ldots, k_{i+b}$ since they are all contained in k_{i+b}, and by running the real protocol (smaller tree, and more caches) they can only do better for x. □

7 Fault Tolerance

Our protocol needs to be robust against failure of caches. We need ensure that if a small fraction of the caches die then browsers still get their pages and also that no hotspots are formed. So let us assume that the adversary kills a small fraction f of the caches.

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

Theorem 6 If $d \geq 2 \log C$ and $f \leq 1/\log_d C$ then with probability at least $1 - O(\log_d C/C)$ more than $1/2e^2$ 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 upto the leaves. 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. 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 paths upto depth i are clean. We have already proved it for $i = 1$. Now at depth $i+1$ the number of clean paths upto 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 paths at depth $i+1$ clean upto 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 $\leq 1/C$. 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. □

The fact that a path is clean with large probability suggests the following scheme for theoretical fault tolerance. Send out $O(\log N)$ random request paths. With high probability one of them gets through. This will clearly inrcease the total load on the network by only an $O(\log n)$ factor.

In practice, sending many simultaneous requests would be fround upon as a waste of bandwidth. We might therefore wait some length of time α after sending each request. We might choose a length of time (say 10 seconds) such that each request has a resonable chance of getting through if it will get through at all. Using such a scheme, we would have a geometrically distributed waiting time for the page without generating a great deal of additional traffic.

8 Conclusion

We have presented an effective caching strategy to eliminate hotspots on the World Wide Web. Many of our results rely on the new technique of consistent hashing. In particular, this gives us fault tolerance and scalability. Furthermore, we obtain a protocol which minimizes latency for a realistic model of the Web.

References

[1] Andreesen M., NCSA Mosaic home page, May, 1974.

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

[3] Berners-Lee T., Masinter L. and McCahill K., RFC 1738: uniform resource locators(URL), December, 1974.

[4] Bestavros A., Demand-based document dissemination for the World Wide Web, Technical Report BU-CS-95-003, Boston University Computer Science Department, Boston, MA, February, 1995.

[5] O'Callaghan D., A central caching proxy server for WWW users at the University of Melbourne, Proceedings of AusWeb95, the First Australian World Wide Web Conference, March, 1995.