Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks

by

Daniel M. Lewin

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

Master of Science in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

Author .....
Department of ~~Electrical Engineering~~ and Computer Science
May 22, 1998

Certified by .....
~~F. T. Leighton~~
Professor of Applied Mathematics
Thesis Supervisor

Certified by .....
David Karger
Associate Professor of Computer Science
~~Thesis Supervisor~~

Accepted by .....
Arthur Smith
Chairman, Departmental Committee on Graduate Students

Consistent Hashing and Random Trees: Algorithms for Caching in Distributed Networks

by

Daniel M. Lewin

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1998, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In this thesis we develop the algorithmic foundation of a large scale distributed caching system for the World Wide Web. In particular, we focus on developing hashing and replication mechanisms that are robust under rapidly changing environments such as the Internet. Two tools are developed: Consistent Hashing, a new hashing technique, and Random Trees, a new replication and load balancing technique. We focus on using rigorous mathematical methods to analyze the proposed algorithms so that they may be applied in real systems with greater confidence.

This work was supported by the US Army through grant DAAH04-95-1-0 607 and by DARPA under contract N00014-95-1-1246.

Thesis Supervisor: F. T. Leighton
Title: Professor of Applied Mathematics

Thesis Supervisor: David Karger
Title: Associate Professor of Computer Science

Acknowledgments

The author would like to thank Professor Tom Leighton for being such a supportive and enthusiastic advisor. In addition, the author would like to thank Tom Leighton, David Karger, Eric Lehman, Rina Panigrahy, and Matt Levine for allowing the contents of our joint work to appear in this thesis. Most importantly, I would like to thank my lovely wife and kids for being there.

Contents

List of Figures

Chapter 1

Introduction

In this thesis we develop and analyze algorithms for caching in large distributed networks like the Internet. In particular, we develop algorithms for distributing data over a set of caching machines and for later retrieving that data from the caches. We concentrate on using rigorous mathematical methods to analyze the proposed algorithms so that hopefully our work can be applied in the design of real world caching systems with greater confidence.

The main contribution of this thesis is the introduction of two new algorithms, one called *consistent hashing*, and another called *random trees.*

Consistent hashing is a new hashing scheme that is particularly well suited for load balancing on the Internet. Typical hashing schemes, such as the classical $ax + b \pmod{n}$ type of hashing, are a standard for assigning pieces of data (e.g. web pages) to caches. The good thing about such standard hashing schemes is that they assign roughly the same number of pages to each cache.

However, there are some serious drawbacks to using standard hashing in a dynamic environment such as the Internet where the set of caching machines may change frequently. Changing the set of caching machines amounts to changing the range of the hash function (e.g. $ax + b \pmod{n}$ changes to $ax + b \pmod{n+1}$ when a new cache is added). The crucial point is that when the range of a standard hash function changes, the assignment of data to caches is completely reshuffeled.

Such a reshuffeling is a disaster for a caching scheme since the entire contents of

each cache is invalidated, or needs to be moved from machine to machine. Even worse, is what happens when different users have different information about the current state of the system. For example, two users may know about slightly different sets of caching machines. With standard hashing techniques, these two users will not agree on the placement of *any* data. If the system is very large (imagine millions of caches) then such a situation is likely to occur for most users. That is, mostly users will have slightly differing views of the set of caching machines. Using standard hashing in this case means that nobody agrees on the placement of any data!

Consistent hashing is a new hashing technique that is designed to have the benefits of standard hashing (easy to compute, distributes items evenly over buckets) and also to react well to changes in the range of the function. Instead of every item changing place when a bucket is added, only a minimal number of elements change place. If two users have slightly different views of the set of active caches, their mapping of pages to caches only differs by a little bit, whereas with standard hashing the mappings would be totally different.

Consistent hashing is a good technique for assigning responsibility for caching pages to a set of caches. However, if there are some extremely popular pages (known as hot spots) then simply reassigning the page to a cache does not solve the whole problem since the cache will be swamped for requests for the hot page. In order to effectively cache a hot page, copies of the page need to be made and distributed to a number of different caches. Then, requests for the page can be distributed between these caches and thus eliminate the possibility of swamping any single cache. The question is how and when to make copies and how to distribute request among the copies once they have been made

Random trees, our second main algorithm, are a simple way to replicate data according to its popularity. The basic idea is to embed in the network a random replication tree with the home server for the information at the root. This replication tree then serves as a mechanism for routing requests and making copies of information. Each piece of data has a different (random) replication network. This guarantees that the load of making and storing copies of information is distributed roughly evenly over

the set of caches. An important feature of the random tree protocol is that copies of data are made *before* the data in fact become so popular that a cache could be swamped. This feature is especially important in the Internet where a swamped cache or server can be effectively cut-off from the rest of the network and thus unable to "call for help".

One of the key pieces of the random tree protocol is the use of a hash function to distribute caching machines over the nodes of a tree. In a stable and known environment, standard hashing would be a natural candidate for use in the protocol. However, since the protocol is meant to work in the Internet, we use a consistent hash function in place of the standard hashing scheme. The result is a caching algorithm that can, at least theoretically, eliminate hot spots and balance load in the rapidly changing environment of the Internet.

Other contributions of the thesis are methods for taking into account network topology in the random trees protocol so that, say, a user at MIT does not go to a cache in Japan to retrieve Harvard's web page. In addition, we show how to minimize the storage requirements of the caches in the protocol by a simple modification of the random trees protocol.

We begin with a rather lengthly discussion of caching in networks that serves as an anchor to the real world problem at hand. Note however, that the discussion highlights the practical aspects that motivate our particular theory, ignoring many other points that are relevant to designing a real life caching system. We wish to warn the reader with a systems background that the following discussion is meant to motivate our simplified model of the problem. Clearly, we do not describe all of the systems issues involved in building a real cache. We believe that our model, although lacking in many respects, still captures sufficient details of the problem so that our algorithms will have real world applicability.

1.1 The Problem

As the World Wide Web becomes a dominant medium for information distribution, mechanisms for delivering data over the Internet efficiently and reliably are needed. However, as any current user of the web can attest, today's data delivery methods are prone to unpredictable delays and frequent failures. These delays and failures have many causes, but three central ones are congested networks, swamped servers, and physical distance.

Network congestion contributes to delay because packets traversing a congested area of the network spend more time in router queues. They are also more likely to be dropped, causing an expensive retransmission. In addition, network congestion reduces the throughput that any one user experiences. Reduced throughput causes increased delays in retrieving information.

Internet protocols are designed with various congestion control mechanisms in place. However, network congestion is still commonplace because network infrastructure expansions cannot keep pace with the tremendous growth in Internet use. For example, the network and routers around the large Internet exchange points are frequently overloaded causing packets to be dropped and delays in retrieving information.

Swamped servers contribute to delay because an overloaded server does not answer requests when there is a lack of adequate resources on the server; for example memory or network bandwidth. Since an overloaded server will ignore requests, some unlucky clients get no response at all from the server. In addition, the lucky clients whose requests are chosen to be served wait longer that they normally would because a swamped server processes requests more slowly than a lightly loaded server. Thus, every user suffers when they try to access information on a swamped server.

Servers can become swamped unexpectedly and without any prior notice. For example, a site mentioned as the "cool site of the day" on the evening news may have to deal with a ten thousand fold increase in traffic during the next day. This is known as the "flash crowd" phenomenon and the server in such a case is called a "hot

spot". One solution to the hot spot problem is to maintain enough servers to handle peak loads. However, designing for peak load is not an efficient solution, especially as the web grows. Imagine every web site having to maintain enough servers to handle requests from every person in the world!

Physical distance is a cause of both delay and unpredictability while retrieving information over the Internet. When the client and server are separated by many thousands of miles, the communication latency between them can be many times human perceptible time; even when the signal is traveling at the speed of light.[1] More importantly, when the client and the server are distant, they usually communicate over a myriad of intermediate networks that neither client nor server have control over. If any of these intermediate networks malfunctions, communication is slowed and sometimes stops altogether. Thus, physical distance is a contributor to the familiar problem of unreliable and unpredictable access over the Internet.

Network congestion, swamped servers, and physical distance all contribute to the fact that the current Internet is referred to as the "World Wide Wait".

1.2 A Solution

A general strategy that has been employed to improve the efficiency and reliability of data delivery over the Internet is called *Caching*. The basic scheme is to place special servers, called caches, at various points in the network. Information is replicated to these caching machines and requests are then served by the caches instead of just by the original server storing the information.

Caching can reduce network congestion if a cached copy is close (in the network topology sense) to the requester since fewer network resources and links are used to retrieve the information. Caching can help relieve the hot spot problem since some of the requests that would normally be routed to the main server can be serviced by the caches. Finally, if the caching machines are distributed uniformly throughout the

[1]Note that the speed of light is about halved in fiber, so without taking into account any router delay, a round trip from coast to coast in the USA takes about 60_{ms} which is twice human perceptible time. Of course, routers introduce an additional delay that can be many times this figure.

network then information in the caches is much closer to the end user than before. Having information close-by in a cache removes the long trip over a congested and unpredictable network.

Figure 1-1 (i) shows on example of the basic web caching scheme. In the example there are users in Boston that every morning access a news site across the country, say in San Francisco. The figure shows only six users, but imagine that there are a thousand of them. In order to reach the site in San Francisco from Boston, packets first traverse a local network, then a regional network to an Internet exchange point, and then they cross an Internet backbone to reach San Francisco. In the morning, all the one thousand people in Boston wake up and access the news site. A thousand requests are transmitted to San Francisco, processed by the server at the news site, and then a thousand copies of the news are delivered over the network back to the readers in Boston. Notice that every user in Boston gets the news from across the country and that the backbone carries 999 redundant copies of the same data. Moreover, the server at the news site is responsible for serving all the one thousand requests.

Figure 1-1(ii) shows what happens when a cache is installed in Boston. When the first person wakes up (the early bird) and requests the news, the request is transmitted to San Francisco in the usual way and the news is returned across the network. However, this time a copy of the news is stored locally in the Boston cache. When the other 999 people wake up and request the news, they get the information from the local cache in Boston instead of from across the country. Thus, most users retrieve their data locally; the network does not carry redundant traffic, and the news server only has to serve one copy of the news for all of the readers in Boston. This simple example shows how caching can reduce network congestion, relieve server load, and reduce physical distance between users and servers.

1.3 Basic Design: Monolithic vs. Distributed

In this section we discuss the very basic architecture of the caching system that is discussed in this thesis. On a cursory level, there seems to be a tradeoff between

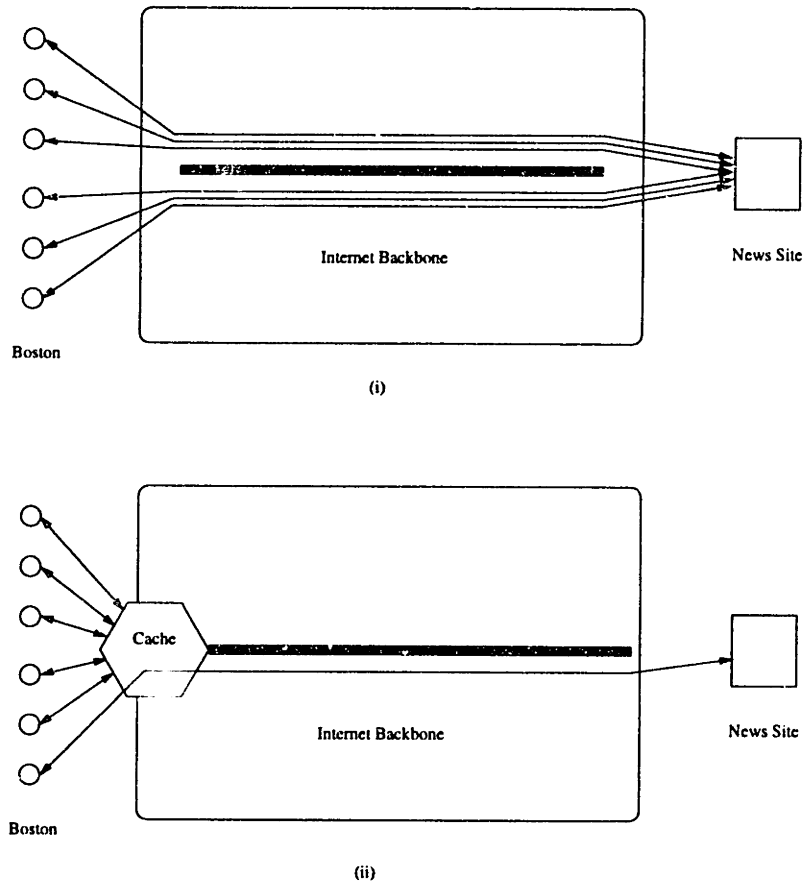


Figure 1-1: (i) Multiple users in Boston access a news site across the country. Each request is routed over the Internet to the news site server and then the news is sent back to boston. Note that the same information is being transmitted many times across the backbone, and that all of the users in Boston get their data from across the country. (ii) When a cache is installed in Boston, the first user retrieves the data from across the country, but all the other users in Boston get the information locally from the cache. The cache prevents redundant traffic from crossing the Internet backbone thus reducing network congestion. The news site server has to send data once to Boston instead of multiple times, and users in Boston get the data faster and more reliably then they did without the cache.

having a high hit rate in the cache and being in close proximity to all the users. For example, say that there is a cache placed in every neighborhood of a city and that users access the web through their local cache. Any item that is in fact stored in the cache can be retrieved very efficiently since the data is coming from a very close-by server. However, a hit only happens in the cache if two neighbors do in fact request the same item twice. This may happen for very popular items, but is not likely to occur for the vast majority of items. Therefore, the hit rate in the cache is likely to be small and the reduction in network traffic negligible. Figure 1-2 (i) shows this type of architecture.

On the other hand, if there is a single large cache for the whole city (which would have to be a very large machine), then the hit rate is likely to be high since the cache receives requests from a far greater population and the likelihood that two users request the same item goes up. Such a "monolithic" cache does eliminate more redundant traffic on the network outside the city than does the distributed approach. However, the cache is physically located far away from most of the end users. Figure 1-2 shows this type of architecture.

In summary: a distributed system keeps content very close to the user but could have a small hit rate, while a monolithic system has a high hit rate but keeps content farther away from the user and is a single point of failure.

1.3.1 Some Existing Caching Systems

In an attempt to both keep content close to users and also have a high hit rate, some groups have proposed caching systems made up of a network of distributed caches that act together as a single caching entity. In one such scheme [11], the caches can be viewed as small machines that are distributed uniformly throughout the network; like the neighborhood cache model discussed above. The difference is that when there is a miss in the local neighborhood cache, the request is forwarded to *all* of the other neighborhood caches, and if any of them have the requested data they return it to the user. This broadcast is done using a special Internet protocol called *multicast* [5] that is designed to minimize network usage.

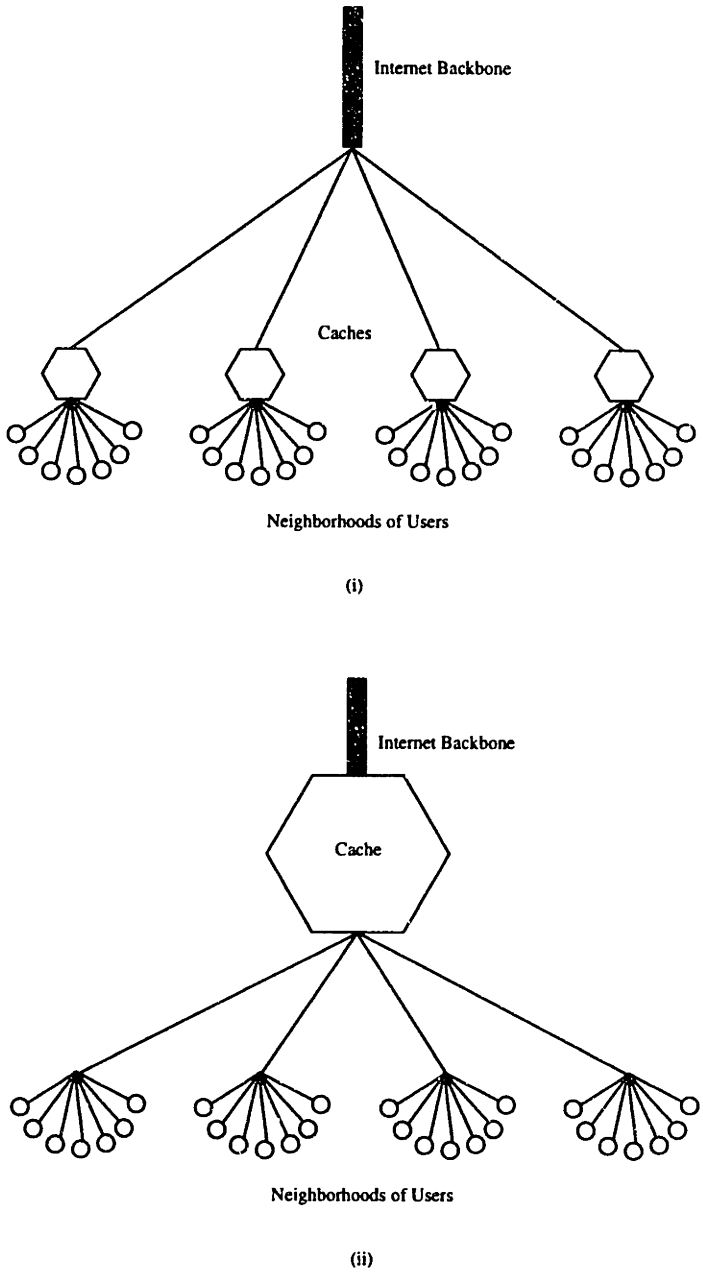


Figure 1-2: (i) A distributed caching architecture. Each neighborhood has a cache that serves residents in that neighborhood. Note that the caches are very close to the users so any content that is in fact located in the cache is retrieved very efficiently. However, since any one cache only receives requests from a small set of users, the hit rate is likely to be small. (ii) A monolithic caching architecture. A single cache is placed in the city to serve all of the users. Since the cache receives requests from a large population, the hit rate is likely to be high. However, the cache is much farther away from the end user that in the distributed scheme.

Such a system does derive the benefit of a distributed architecture since caches are close to users. In addition, as in the monolithic architecture, redundant traffic is prevented from leaving the city. However, as the network grows, the traffic generated by the broadcasts for each cache miss may become unmanageable; even if the broadcast is done using multicast.[2]

Other schemes, such as the Harvest caching system [3], propose to build a hierarchical caching system. The idea is to have both distributed neighborhood caches *and* monolithic city caches. The caches are arranged in a fixed hierarchy with the small, distributed caches at the bottom and the large, monolithic city caches at the top. Figure 1-3 depicts such a system.

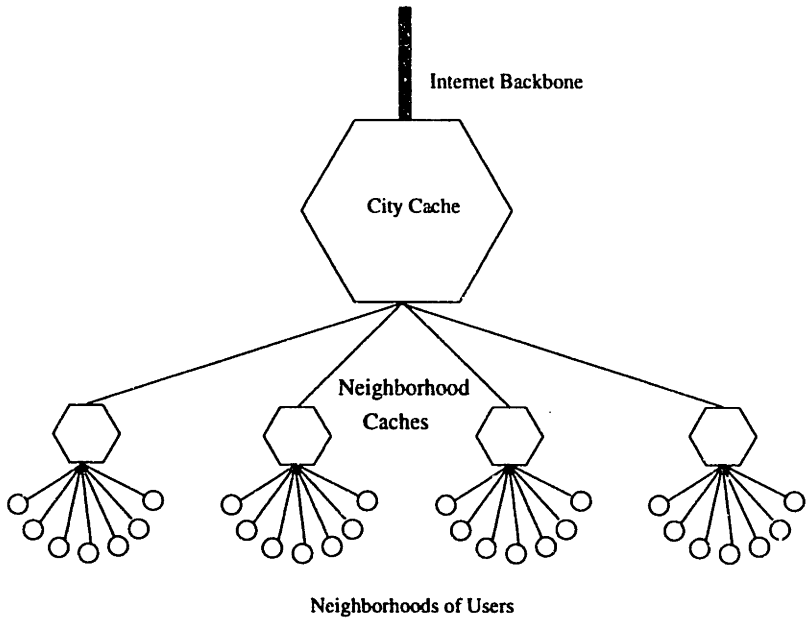
Requests are first directed to the closest neighborhood cache, and if there is a miss, the request is forwarded to the monolithic city cache. In this solution, neighborhood caches bring content close to the users, but city caches aggregate requests from many users and thus, by having a high hit rate, prevent redundant traffic from crossing the network outside the city. Thus this solution, like the previous, derives the benefits of both distributed and monolithic cache architectures.

However, there is a serious practical drawback to this solution. The large, monolithic city caches that are high up in the hierarchy are juxtaposed at critical points in the network, and handle all of the requests from a very large population. Therefore, these machines need to be fault-tolerant and able to handle huge numbers of requests at a time. Building such a machine is expensive, and our assumption is that as the web continues to grow, building such a machine may in fact become altogether unfeasable.

1.3.2 Our Model

In this thesis we design algorithms for a different basic architecture that strives to get the best of both worlds, distributed and monolithic, without requiring excessive

[2]From a theoretical standpoint, the city network would have to grow quadratically with the number of caches since the number of broadcast messages would grow this much. Since the viewpoint of this thesis is purely theoretical, we may assume that such growth is unacceptable.



(i)

Figure 1-3: (i) A hierarchy of caches with small neighborhood caches at the bottom of the hierarchy and a large monolithic city cache at the top. Requests are first sent to the local neighborhood cache and on a miss they are forwarded to the city cache. The neighborhood caches bring content closer to the user, while the city cache aggregates requests from a large population and thus prevents redundant traffic from crossing the network outside the city. The problem with this system is that the city cache has to be a large and fault tolerant machine (or cluster of machines) which is expensive to build and maintain.

network infrastructure or huge, fault-tolerant machines. The system is comprised of many small distributed caches, placed close to the users. These caches cooperate to form together a cache with large aggregate capacity and hit rate. The challenge is to show how such a large collection of caches can in fact cooperate efficiently without overloading any server, without congesting the network, and without any centralized, monolithic control.

This already difficult goal is further aggravated by the fact that the Internet is a very large and dynamic network. There are many characteristics of such networks that make algorithm design difficult, but three central ones are the lack of centralized control, the fact that the network is in a constant state of flux, and the fact that no single entity can have completely accurate data on the state of the network.

The lack of centralized control means that algorithms have to work locally and without any single point of failure. Furthermore, because machines are added to and removed from the network continually and no machine has an accurate picture of the state of the network, algorithms have to be designed to be robust under changing environments and imperfect information.

1.4 Design Objectives and Complications

In this section we summarize the proceeding discussion by outlining the general architecture of the caching system for which we design algorithms. In addition, we discuss some of the complications involved in designing the system. The section is organized as a list of objectives with accompanying explanations.

1. **Large Distributed System:** The system can be built from a very large number (imagine hundreds of thousands or even millions) of small and cheap caches that are distributed throughout the network so that there is a cache near every user. Furthermore, the caches should be distributed so that there are enough caches in any one area to satisfy all of the requests originating near that area.

   One of the reasons that the Internet has grown to its current proportions is that there is no central authority responsible for its structure. Our caching scheme

should also be designed to allow unconstrained growth so that the system will be deployed throughout the network. Therefore, the system should be designed so that anyone can add a cache to the system at any time without notifying a central authority. This will allow a large collection of users, who may not even know about each other, to cooperate to deploy the system.

2. **No Centralized Control:** The network of caching machines should operate without any centralized control that could be a critical point of failure. The caching scheme should not rely on any type of centralized service and caches should operate asynchronously. The behavior of a cache should depend only on information available locally, or obtained in a non-centralized fashion. The system should not make use of global broadcasts.

3. **Robust Under Imperfect and Inconsistent Information:** Since the network of caches will be huge, it will be infeasible for any single cache or user to have a complete view of the state of the system. Caches can be added to and removed from the network at any time, and there is no central authority that keeps track of the status of the caching machines. Thus, different users may have differing views of the set of caches. The caching system must be designed to work even under these chaotic conditions.

4. **Scale Gracefully:** The Web grows every day, and so must the caching scheme if it is to keep up with increasing use. The system should be designed to scale gracefully as the network grows. There should be no catastrophic updating or reorganizations.

5. **Prevent Swamping and Hot Spots:** Caching machines and servers should never be swamped no matter how requests are distributed between pages (e.g. many requests go to one "hot" page or many requests go to different pages). Simply reassigning responsibility for a hot page to a cache will not work since the cache will be swamped by requests. In order to prevent swamping, copies of the hot page need to be made and distributed to many caches. The requests for

the hot page need to be divided among these caches so that none are swamped.

Note however, that it is impossible to predict when a page will be hot. Moreover, when a server is swamped, it cannot communicate to request help, and therefore, copies of hot pages need to be made *before* the page becomes popular. So, for example, the decision about when to make copies cannot be based on a central counter of the number of accesses to the page since a flash crowd can swamp the counting server, thus cutting it off from the rest of the network.

6. **Minimize Network Usage:** The system should be designed so that the total traffic in the network is reduced as much as possible. In other words, the caching system should be designed so that as many requests as possible are served close (in the network topology sense) to the requester, thus minimizing network usage. For example, a user at MIT should not go to a cache located in Japan to retrieve Harvard's web page.

   Note that there is a tradeoff between load-balancing (preventing swamping) and locality. If a page is extremely popular and is always retrieved from the closest cache then that cache could be swamped if it is close to a large number of users. In order to prevent swamping, copies of the page may need to be retrieved from more distant caches.

7. **Balance Storage Requirements:** No caching machine should be required to store a disproportionate fraction of the cached pages. Responsibility for storing the current active set of pages should be equitably distributed among the caches.

8. **Low Overhead:** Users should not have to wait a long time for the system to retrieve a page because of the caching system.

1.5 Our Contribution

Designing a distributed caching system that fulfills the requirements of the previous section is a Herculean task too big to be undertaken in this thesis. Therefore, instead

of giving a complete design, we concentrate on building algorithmic tools that make up the high level architecture of a caching system that we believe can fulfill the stringent criteria set out above. In particular, we concentrate on designing high level algorithms that control how pages are distributed and replicated to caching machines, and how users find information in the caches.

We develop our algorithmic tools in a theoretical framework. Specifically, we propose a simple model of the problem that, while capturing what we believe to be the essential details of the real problem, may be severely deficient in detail. However, this abstraction will allow us to *prove* that our algorithms work - albeit in the simplified model. We believe that such an approach is justified since our goal is to develop algorithmic tools that can later be applied to the real world problem.

The two main algorithmic tools that we introduce are called Consistent Hashing and Random Trees. Consistent hashing is a new hashing scheme that is designed for load balancing in an environment where the set of caching machines changes frequently. Random trees are a new replication mechanism that can eliminate the hot spot problem. More detail is contained in the following two sections.

1.5.1 Consistent Hashing

A *hash table* is a data structure for what is known as the *dictionary problem.* In the dictionary problem there is a set of keys K, and we must build a data structure to support the operations `INSERT`, `DELETE`, and `FIND`. The hash table consists of a table of N cells $1 \ldots N$, and a hash function h that is a mapping from the set of possible keys to the table. The hash function h maps each key to a cell in the table where it is then stored, in the case of an `INSERT` operation, or deleted, in the case of a `DELETE` operation. A `FIND` query is answered by looking into the cell of the table assigned to the given key by the function h. One important intuition about hash functions that can be made theoretically sound in some sense is that they distribute the keys "randomly" into the cells of the table. Thus, each cell receives a roughly equal share of the keys.

A hash table can be used in a caching scheme to distribute work evenly over a

number of caching machines. In the case of the web, where information is in the form of web pages, the keys are page names (e.g. URLs), and the cells are caching machines (e.g. IP addresses). When a client requests a page, a `FIND` query with the page name as the key is used to compute the name of the cache where the request is sent. The hash function serves as a method for clients to agree on which machine is responsible for caching a page *without communicating.*

In a large system such as the Internet the set of active machines changes over time since machines are added to or removed from the network constantly. Having such a dynamic set of active machines amounts to having a hash table with a changing set of cells. Standard hash functions do not behave well under these conditions. The problem is that even when the set of cells is only slightly changed, most hash functions completely reshuffle the mapping of keys to cells. Thus, for each different configuration of cells in the table, most hash functions induce a completely different mapping of keys to cells (pages to machines). In the case of the Internet, it is plausible that almost every client will have a slightly different view of which caching machines are currently active. If this happens, each client has a different set of cells in its hash table, and hence a different mapping of pages to machines. This is a disaster for the caching system since clients no longer agree on where a page is cached!

Consistent Hashing is a new type of hashing which is designed to react well to changes in the set of cells in the table. Like standard hashing techniques, consistent hashing does a good job of distributing items "randomly" into cells. However, unlike standard hashing, different sets of cells do not induce completely different mappings. Instead, the mappings are "consistent" with each other.

Consistent hashing allows users to agree on the cache responsible for a page even if users have widely differing views of the set of available caching machines. This feature of consistent hashing allows us to develop caching protocols that are robust under changing environments and imperfect information; one of the more difficult criteria described in section 1.4.

We believe that consistent hashing can be applied to a variety of other problems such as distributed file systems, distributed databases, and name servers.

1.5.2 Random Trees

Consistent hashing is particularly well suited for use in a large distributed caching system. However, consistent hashing alone is not enough to build a caching system that can relieve hot spots since simply redirecting the requests to a single cache is liable to swamp the cache itself. In order to relieve hot spots without creating other hot spots, information needs to be replicated more than once in the network. We propose a novel technique called *Random Trees*, that helps replicate pages according to their changing popularity. As a page gets more popular, requests for the page are distributed over more and more caches. Moreover, copies of pages are made *before* the page is so popular that a cache is swamped.

A crucial part of the Random Trees protocol is the use of a hash function to distribute caching machines over nodes of a tree. Thus, in order to obtain a viable system for the Internet, we combine consistent hashing and random trees. The result is a protocol that can eliminate hot spots in the network even if clients have imperfect information about the state of the network.

In addition, we show how a representation of network topology we call a *Network Ultrametric* can be used in the random tree protocol to keep information close (network topology-wise) to the users that want it.

1.6 Previous Work

In this section we review some of the previous work on hashing and on web caching.

1.6.1 Previous Work on Hashing

Hashing is most commonly discussed in the context of either the static or dynamic dictionary problems. In the static dictionary problem, we are given a set of keys K ahead of time, and we need to put them into a data structure that supports `FIND` queries. In the dynamic dictionary model, the set K is not given ahead of time but is built up with a sequence of `INSERT` and `DELETE` operations that are intermingled

with the FIND queries (see [8]).

A *hash table* is a data structure for the dictionary problem that consists of a table of N cells $1 \ldots N$, and a hash function h that is a mapping from the set of possible keys to the table. The hash function h maps each key to a cell in the table where it is then stored. A FIND query is answered by looking in the cell of the table assigned to the key by h. Ideally, we would want the hash function to map different keys to different cells in the table. If two different keys are assigned to the same cell by h we say that the keys *collide.*

A hash function is called *perfect* for a set of keys K if there are no collisions between the elements of K. Much research has been devoted to building perfect hash functions (see [18], [22], [19], [6]). However none have addressed how the function behaves when the range is changed as we do.

Another important class of hash functions are called k-universal. We say that a family of functions is k-universal if any k keys are mapped independently into the table when we choose a function at random from the family. In other words a family is k-universal if for any distinct keys x_i, $1 \leq i \leq k$, and any cells y_i, $1 \leq i \leq k$:

$$\Pr[f(x_1) = y_1, f(x_2) = y_2, \ldots, f(x_k) = y_k] = \frac{1}{N^k}$$

Where the probability is over a random choice of function f from the family. Another way of looking at this is that the random variables $\{f(x)\}$ where x ranges over the keys are k-way independent.

Universal hash families were defined by Carter and Wegman in [2], and in [21]. Universal hashing has found many applications in algorithms and complexity theory (for example [13]) and will play a central role in the theory of consistent hashing.

To the best of our knowledge, no previous research has been done on hash functions that adapt well to changes in the range of the function.

1.6.2 Previous Work on Web Caching

Much practical work has been done on caching for the Web. One approach which is in wide use today is for several clients to share a single cache that handles *all* of their requests. These caches are known as *Proxies* [10]. The dilemma in this scheme is that there is a greater benefit if more clients share the same proxy, but then the proxy itself is liable to become overloaded.

One approach to solving this dilemma is to create a group of machines that function together as a cache. Malpani et al. in [11] describe a system based on IP multicast that implements this scheme. A client's request for a document is directed to an arbitrary one of the machines in the group of caching machines. If the page is currently in the cache of the machine, it it returned to the client. Otherwise, the server forwards the request to *all* of the other caching machines by means of a multicast. The main disadvantage of this scheme is that as the number of caching machines grows, the amount of communication between the machines grows very quickly. Thus, this approach is unsuitable for very large networks.

Chankhunthod at al. [3] developed the Harvest Cache, a more scalable approach that uses trees of caches. However, they use a *single* tree of caches which can, as we have already discussed, cause swamping of the caches close to the root. There are a large number of papers investigating the various systems issues involved in building web caches such as [9], [14], [16], [20], and [23].

On the theoretical side, Plaxton and Rajaraman [15] introduced the idea of using randomization and hashing to balance load on a network. Their work however was done in the context of a synchronous network of nodes in a parallel machine. They assume the existence of special priority messages that reach a node even though it may be swamped with other low priority messages which is clearly not the case for the Internet. In [7]) the random trees protocol was introduced along with consistent hashing.

1.7 Structure of the Thesis

The thesis is divided into two main chapters. In chapter 2, we introduce the notion of a consistent hash function. Consistent hashing is motivated by a simple caching scheme that ignores the hot spot problem. Most of the important technical contributions of this thesis are contained in chapter 2.

Chapter 3 introduces the random tree protocol that is designed to compliment consistent hashing by taking care of the hot spot problem. The random tree protocol is presented as an application of consistent hashing, and only a brief analysis is presented since a complete analysis appears in [7]. Chapter 3 also discusses various extensions of the random tree protocol that also appear in [7].

Chapter 2

Consistent Hashing

In this chapter we present our first algorithmic tool called Consistent Hashing. Consistent Hashing is a new hashing scheme that is motivated by a simplified version of the caching problem where we assume that there are no very popular pages. In this case, the problem becomes how to distribute load evenly over a set of caches, and hashing is a standard solution in this situation. Unfortunately, standard hashing schemes do not work well in environments where users have imperfect or inconsistent information. Therefore, we introduce consistent hashing that derives the benefits of standard hashing but is also robust under the haphazard conditions of the Internet.

Even though consistent hashing was initially developed in the context of caching systems, we believe that there are many other applications where consistent hashing should be used in place of existing standard hashing schemes. Therefore, we develop the theory of consistent hashing in a general setting and use caching as a motivating example.

This section is outlined as follows. In section 2.1 we motivate consistent hashing by describing a simple caching scheme that ignores the hot spot problem. This example is used throughout the section as motivation for the various definitions that we make. Section 2.1 also provides a brief introduction to the concept of consistent hashing. In section 2.2 we formalize the intuitive notions developed in the motivation section as definitions, and in section 2.2.3 we present our main construction of a consistent hash function. Section 2.2.4 discusses how to implement consistent hashing in practice.

Sections 2.2.6 and 2.2.7 discuss different adversarial models and some basic theory underlying consistent hashing.

2.1 Practical Motivation

In this section, a simple caching scheme that serves as a vehicle for understanding and motivating consistent hashing is presented. We assume for the moment that server swamping is caused *only* by the fact that a server is responsible for many *different* documents which are all accessed on a *regular basis*. That is, we ignore for now the case where load on a server is a result of a few very popular pages stored on the server.

Our model of the web contains two components: servers, and users. Servers contain documents which users want to retrieve. Assume that all documents are accessed by users at the same rate. However, some servers may contain far more documents than others, and therefore have more requests to service. The basic objective is to spread the load of answering user requests equitably between the servers. Note that in this sense, all the servers are also acting as caches.

Responsibility for serving documents is distributed between the servers using a hash function. The function hashes document names to servers. Users request a document from the server assigned by the hash function instead of from the original location. If the server has a copy of the document, then the request can be answered. If not, then a copy is requested by the server from the "home location" of the document (e.g. URL in document name). When a copy is received the request is answered, and subsequent requests can also be serviced. Thus, the original server does not need to be involved in all requests for documents it stores. Figure 2-1 illustrates this system. Another approach is to redirect requests from the original server to the caching server. However, serving redirect requests can in fact swamp the original server making this approach unattractive. The hash function serves as a method for users to agree on which server is responsible for a document without communicating.

A common and useful intuition about hash functions (which in fact can be made

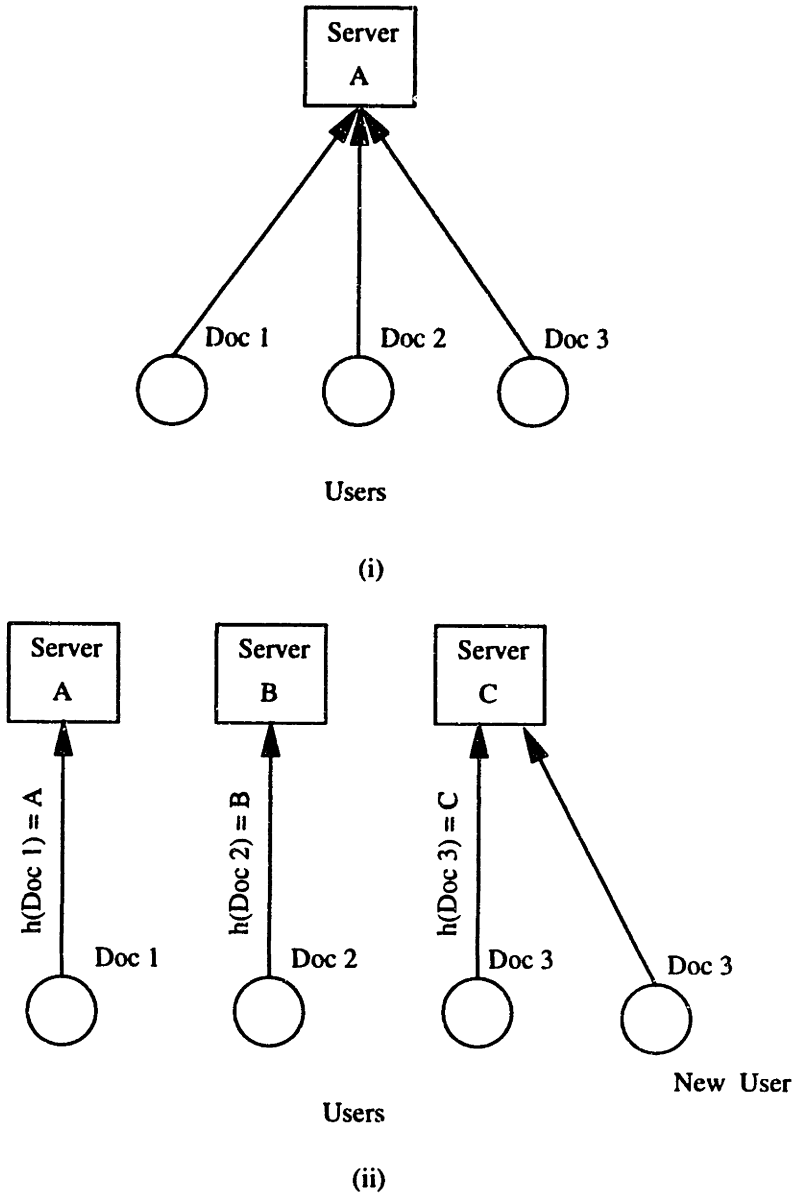


Figure 2-1: (i) Three users requesting documents Doc1, Doc2, and Doc3 from the server responsible for them. The server must answer all three of these requests. (ii) A hash function h is distributed between the users. Requests are directed to the server to which the hash function hashes a document name. Here each document is assigned to a different server. After obtaining a copy of the requested document, each server only has to answer a single request. If an additional user requests document Doc1, server A is not involved, since a copy was already obtained to serve the previous request.

theoretically sound in some sense) is that they distribute items "randomly". Following this intuition, in the caching system documents are assigned to *random* servers by the hash function. This is very desirable, since a random assignment will most likely spread the load of serving the current active set of documents roughly evenly between the servers. If the space allocated for caching on each server is large enough (e.g. an equal fraction of the total size of the average active set), then the caching will be effective and servers will have roughly the same load. A server storing many documents in the current active set supplies copies to other servers who become responsible for them.

To give an example of a hash function that can be used to distribute documents to servers assume that document names are integers, and that there are n servers $\{0, 1, 2, \ldots, n-1\}$. A very commonly used hash function is given by $f(d) = ad + b \pmod{n}$ for some fixed choice of integers a and b. Note that computing this function is very simple and efficient. In addition, note without proof (see [CLR] or [RM]) that for most choices of a and b the function distributes documents roughly evenly over the servers. Figure 2-2 illustrates how the function distributes documents to servers.

2.1.1 Complications

The number of servers on the web is growing at an enormous rate. Each day many machines are added to the network, and these machines should be incorporated into the caching scheme. Adding a server to the scheme amounts to changing the range of the hash function to include the new server, and notifying users of the change.

The following example exposes the complications caused by adding a server. Assume that there are n servers and that the hash function mapping documents to servers is $f(d) = ad + b \pmod{n}$. When the $n+1$'th server is added, the hash function is changed to be $f'(d) = ad + b \pmod{n+1}$. Figure 2-3 illustrates how the mapping of documents to servers changes. A common characteristic of this hash function and many other standard hash functions is that when the range of the function is modified, the mapping is completely reshuffeled. That is, most documents are mapped to different servers than they were mapped to before the change.

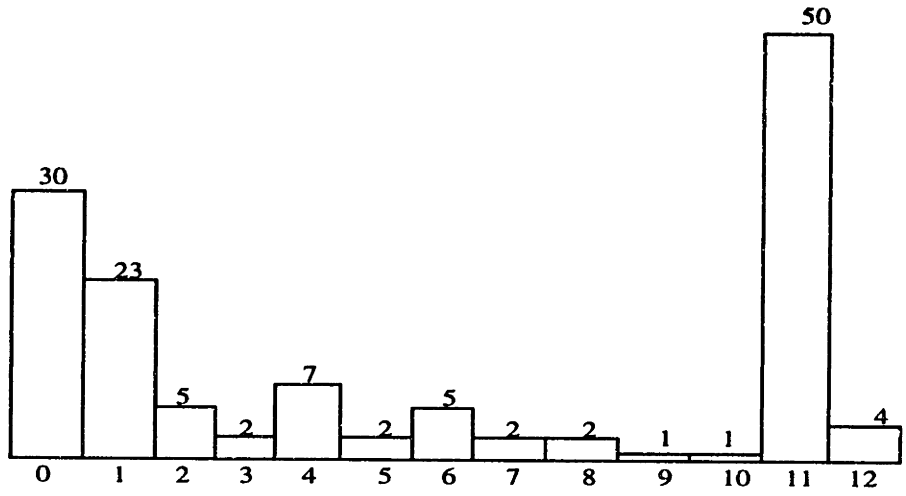


Figure 2-2: This figure illustrates how hash functions distribute documents between servers. Assume that document names are integers, and that there are 13 servers $\{1, 2, \ldots, 13\}$. Documents are hashed to servers using a common type of hash function which is $f(d) = ad + b \pmod{13}$ for some fixed integers a and b. (i) The original distribution of 134 documents to servers. Note that some servers store many more documents than others, and thus in the model of equal access frequency they are more heavily loaded. (ii) The distribution of documents to servers by the hash function. No server is responsible for a disproportionately large share of documents.

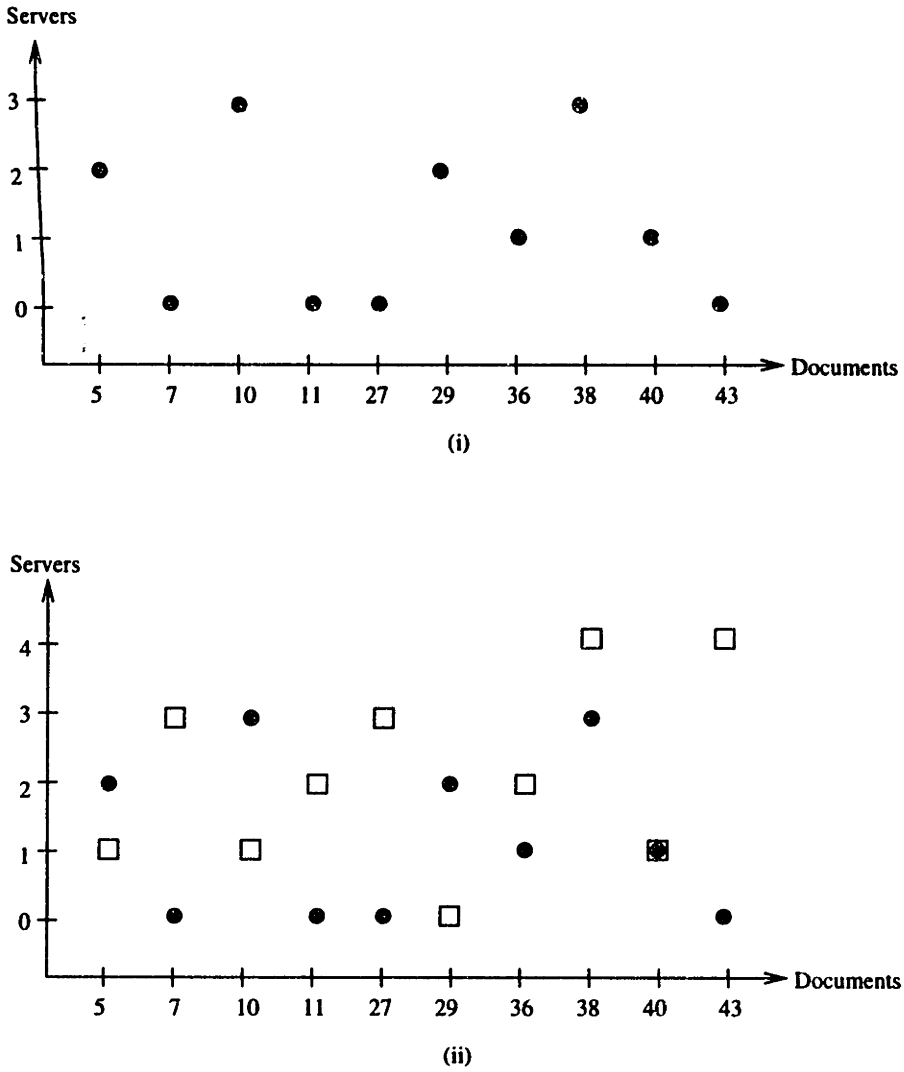


Figure 2-3: (i) The assignment of 10 documents to 4 servers using the hash function $f(d) = d + 1 \pmod 4$. (ii) The assignment after one additional server is added. The hash function is changed to be $f(d) = d + 1 \pmod 5$. Squares show the new mapping and circles show the mapping of the previous function. Note that almost every document is mapped to a different server as a result of the addition of the new server.

A complete reshuffeling of the distribution of documents to servers is catastrophic for the caching scheme. After each addition, and subsequent reshuffeling new copies of all documents need to be requested from the "home locations" since the previous copies are rendered obsolete. If servers are continuously being added to the system then the caching protocol would be worthless since a server storing a large number of documents would be constantly supplying new copies to different servers. In other words, such a server would be swamped.

In addition to the problem of reshuffeling there is another, even more serious complication. When a server is added to the system, users have to be notified of the addition so that they can start to use the new hash function. This implies that all users are notified of the addition of a new cache at the same time. In a network such as the Internet, such global synchronous notification is of course impossible. It is feasible however, to assume that notification about new servers is done by slowly propagating the information from server to server much in the same way that network news is distributed. As a consequence, the set of servers being used in the protocol may differ widely from user to user. We call the set of servers that a user is aware of the user's *View*. Since synchronous notification is impossible there are likely to be many different views being used by different users.

The example of figure 2-3 can also be used to illustrate the effects of multiple views of the servers on the caching scheme. If some of the users are not aware of the addition of server 4, then there are two views of the system held by users. One view is $0, 1, 2, 3$ and the other is $0, 1, 2, 3, 4$. Users not aware of the addition of server 4 use the old mapping of documents to servers. Those who are aware of the new server 4 use the new mapping, which is almost a complete reshuffeling of the old mapping. As a result, the number of documents tnat a server is responsible for over both views is almost doubled! For example, users with the old view use server 2 for the documents 5 and 29 however users with the new view use server 2 for documents 11 and 36. Server 2 must be prepared to handle requests for all of these documents, thus effectively doubling the requirements on the server. The total number of documents that a server is responsible for is called its *load*. For example, the load of server 2 in the

example of figure 2-3 is 4; 2 documents from each view. Clearly, the load of a server should be small in spite of having multiple views of the servers.

A further complication resulting from multiple views of the servers is that a single document may be assigned to different servers in different views. For example in figure 2-3, document D is assigned to server A in the view $0, 1, 2, 3$ and to the server B in the view $0, 1, 2, 3, 4$. The main server for document D has to supply a copy to each of these servers. A large number of views means a large number of servers responsible for a document, which in turn implies that the home server will be swamped with requests for copies. The total number of servers a document is assigned to is called the *spread* of the document. For example, the spread of document 5 in the example of figure 2-3 is 2. Clearly, the spread of a document should be small in spite of having multiple views of the servers.

Summarizing the above, the hash function should have the following general properties:

- The function should distribute documents roughly evenly over the set of servers.
- When a new sever is added, and the range of the function is changed, the mapping of documents to servers should not be completely reshuffeled. There should be a strong correlation, or consistency, between the old and new mappings.
- In the presence of multiple views of the servers, the function should not map too many documents to any server. In other words, the *load* on servers should not go up dramatically when there are multiple views.
- In the presence of multiple views of the servers, there should not be too many servers responsible for a given document. Or, in other words, the *spread* of a document should not go up drastically because of multiple views.

In this paper we present a hashing scheme called Consistent Hashing that has all of the above properties. The next section presents the basic solution, precise definitions are in section 2.2.

2.1.2 A Solution

Surprisingly, there is a very simple hashing scheme (described in this single paragraph) which fulfills all of the rather stringent requirements set out above. Documents and servers are mapped to points on a circle using standard hash functions. Assume for intuition that these functions distribute the documents and servers randomly around the circle. Now, a document is assigned to the server whose point is the first encountered when the circle is traversed clockwise from the document's point. An example is shown in figure 2-4.

Remarkably, this simple construction has all of the desired properties. Following is an intuitive discussion of the construction and in section 2.2.3 a rigorous analysis is presented.

Intuitively, if documents and servers are distributed randomly around the circle then each server should be responsible for roughly the same number of documents. Of course, there is a possibility that the points could be distributed badly so that one server receives a disproportionate fraction of the documents. While we cannot completely prevent such a misfortune, we can improve the chance of a good result with a simple trick which is described later (section 2.2.3).

Suppose that a new server is added to the system. The only documents that are reassigned are those with points that are now nearest to the new server's point; the mapping is not completely reshuffeled. Documents are only moved to the newly introduced server, and are not moved between old servers. Therefore most of the previously cached copied of documents are still valid because the newly obtained mapping is "consistent" with the previous one. Figure 2-4 illustrates this point.

This construction also behaves remarkably well in the presence of multiple views. Intuitively, if a document is assigned to a server in some view, then that document's point is likely to be relatively close to the server's point[1]. So only documents with points close to the server point are likely to be assigned to the server. Since document points are distributed randomly around the circle only a few document points fall close

[1]Since if they were far apart, then another server's point would be likely to fall in between, preventing the document from being assigned to the original server.

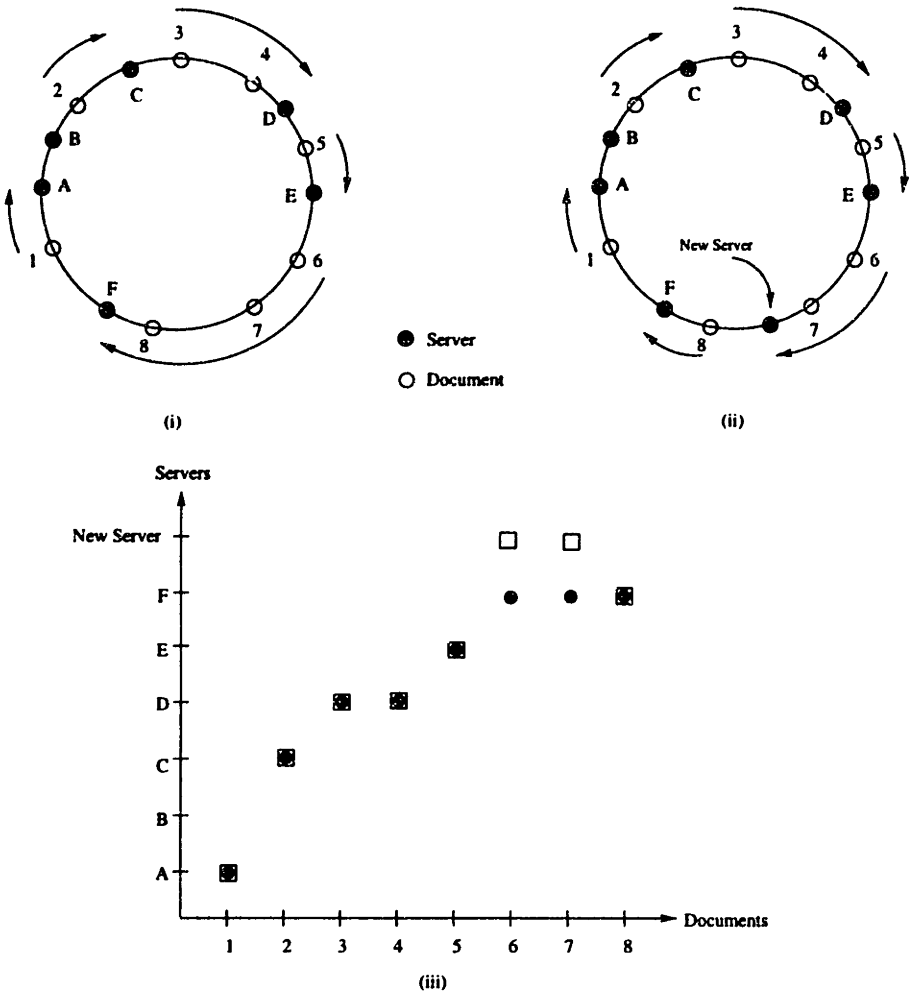


Figure 2-4: (i) Both documents and servers are mapped to points on a circle using standard hash functions. A documents is assigned to the closest server going clockwise around the circle. For example items 6, 7, and 8 are mapped to server F. Arrows show the mapping of documents to servers. (ii) When a new server is added the only documents that are reassigned are those those now closest to the new server going clockwise around the circle. In this case when we add the new server only items 6 and 7 move to the new server. Items do not move between previously existing servers. (iii) The mapping of documents to servers before and after the addition of a new server. Squares show the new mapping and circles are the previous mapping. Compare this with the results obtained from standard hashing in figure 2-3.

to a server's point. So even if there are a large number of views, only a relatively small number of documents are assigned to a server. The load of a server does not increase drastically with multiple views. In fact, we show in section 2.2.3 that the load increases only logarithmically in the number of views, while with standard hashing this dependence can be linear; clearly a substantial improvement.

Turning the above intuition inside out, if a server is responsible for a document then the server's point is likely to be relatively close to the document's point. Since server points are distributed randomly around the circle only a small number of servers points fall nearby any document point. So even if there are many views, only a few servers will have responsibility for any one document. The spread of a document does not increase dramatically with the number of views. As before, we show in section 2.2.3 that the spread increases only logarithmically in the number of views, while with standard hashing this dependence can again be linear.

Following is a summary of the important properties of the "circle hash function":

- Documents are distributd to servers "randomly".
- When a server is added, the only documents reassigned are those that are assigned to the new server. The newly obtained and old mappings are consistent with each other.
- The load of a server increases only logarithmically with the number of views in contrast to linearly with standard hashing.
- The spread of a document increases only logarithmically with the number of views in contrast to the linear dependence of standard hashing.

Clearly, using this new hashing technique in the previously described caching scheme solves the complications raised. The above "consistency properties" allow the caching scheme to grow gracefully with the growth of the network. Servers can be added to the network without disrupting the caching and multiple views of the servers can exist without degrading the performance of the system. Furthermore, the function is very simple and efficient to evaluate (see section 2.2.4).

2.2 Consistent Hash Functions

In this section we conceptualize the notion of a consistent hash function intuitively described in the previous section. Section 2.2.1 reviews the basics of hashing from a theoretical standpoint. Those familiar with the subject can skip this section. Consistent hash functions are defined in section 2.2.2. The circle construction of a consistent hash function is described in section 2.2.3, and the properties of this particular construction are derived and proved in section 2.2.3. Implementation issues are discussed in section 2.2.4.

In the following sections the motivating application of caching, is set aside as the central issue, and consistent hashing is discussed as a general hashing scheme which may have many other applications. Nonetheless, some intuitive discussions still derive from the simple caching scheme presented in the previous section.

2.2.1 Hash Families

We assume that the reader is familiar with the basic notion of a hash table data structure. In a hash table, a set of items I is mapped to a set of buckets B by a fixed *hash function.* In theoretical settings the intuitive notion of hashing is commonly modeled as follows: You are given a *family* of hash functions H whose elements are each functions that map the items I into the buckets B. The hashing is done by choosing a random function f from the family and using that function to map items to buckets.

Why do we use this seemingly more complicated model of hashing? Suppose that a malicious adversary[2] is aware of the hashing scheme being used and is attempting to devise an instance on which the scheme behaves poorly. Any scheme that uses a single fixed function to map items to buckets is vulnerable to an attack by an adversary since the input that causes the absolute worst case behavior of the scheme can be determined.

[2]ThroughoutΦΦΦThroughout this paper we assume the existence of such adversaries. In fact, we claim that such adversaries are ubiquitous, and can be found anywhere you look.

This is best explained with an example. Suppose in the caching scheme presented in section 2.1, that an adversary has the power to permanently crash a number of servers. Assume in addition, that the mapping of documents (items) and servers (buckets) to the circle is known to the adversary. This adversary can concoct a plan to crash servers which would leave a remaining server heavily loaded. Specifically, the adversary can crash all servers in a portion of the circle, creating a gap with no servers at all. All the documents previously assigned to the crashed servers are now assigned to the server whose point is on the end of the gap. This server becomes overloaded with requests.

A standard method to f l such an adversary is adding randomization to the scheme. For example, if the points associated with servers and documents are chosen randomly and the adversary is not aware of these random choices, then the above scheme for overloading some servers does not work. The adversary does not know the placement of the points in the circle and therefore does not know how to create a gap.

In this paper, randomization is added to the scheme in a particular way. Choosing a function from a family of hash functions is done by taking one *uniformly and at random.* Thus, "using a hash family H" means that first a function $f \in H$ is selected uniformly and at random and then that function is used to map items to buckets.

Hash families are classified by various properties. Using a hash family involves randomization, so these properties are probabilistic. We say that a property *holds with probability p* for a hash family H if for a function chosen from the family uniformly and at random, the property holds with probability p. Following is an example.

Example 1 *(Linear Congruential Hashing) Let both the set of items I and the set of buckets B be $\{0, 1, 2, \ldots, p-1\}$ for a prime p. The family H is defined as all the functions of the form $f(x) = ax + b \pmod{p}$ for all $a, b \in \{0, 1, 2, \ldots, p-1\}$.*

To use this hash family, a and b are chosen at random (this is œ[Dœ[equivalent to choosing a function from the family at random). Then, item i is assigned to bucket $ai + b \pmod{p}$. The following lemma gives an example of a property of the hash

family that holds with a certain probability.

Lemma 2.2.1 *Let H be the hash family defined above, and let x and y be distinct items, then* $\Pr[f(x) = i \text{ and } f(y) = j] = 1/p^2$.

In other words, the probability that a randomly chosen function f from the family H maps the item x to i and the item y to j is the same as in the case where the two items are mapped to the buckets uniformly and independently.

Proof: Since the set $\{0, 1, 2, \ldots, p-1\}$ with operations $(mod\ p)$ is a field, there is a unique pair a and b such that $ax + b = i$ and $ay + b = j$. Thus the probability that $f(x) = i$ and $f(y) = j$ is equal to the probability of selecting the pair a, b while choosing a function from H. The probability of selecting any given pair a, b is $1/p^2$. ■

Lemma 2.2.1 shows that a randomly chosen function from H behaves like a completely random function with respect to pairs of items.[3] Why would we prefer to use the family H instead of choosing a completely random function? Notice that choosing, storing, and evaluating functions from H is remarkably simple and efficient. We pick a and b at random. These values can be stored using very little memory, and evaluating the function is simple. On the other hand, selecting a truly random function requires choosing and storing a random table of the value of the function on every single item.

Hash families similar to the one presented in the above example are important elements in the practical implementation of our consistent hash functions (section 2.2.4).

2.2.2 Definitions

Unlike traditional hash functions, consistent hash functions are intended to deal with situations in which the set of buckets (the range of the function) changes. Therefore,

[3] A hash family that looks random in respect to pairs of items is called *pairwise independent.* Similarly, families that look random with respect to k elements are k-way independent. Hash families that are k-way independent are important in section 2.2.4

we introduce the notion of a *ranged hash function* that permits the set of buckets in the range of the function to change.

Let the set of items be I and the set of buckets be B. A *view* is a subset of the buckets B. A *ranged hash function* is a function of the form $f : 2^B \times I \rightarrow B$. Such a function specifies an assignment of items to buckets for every possible view. That is, $f(V, i)$ is the bucket to which item i is assigned in view V. (We will use the notation $f_V(i)$ in place of $f(V, i)$ from now on.) Since items should only be assigned to available buckets (buckets that are in the current view), we require $f_V(I) \subseteq V$ for every view V.

A *ranged hash family* is a family of ranged hash functions.

In section 2.1 the following characteristics of ranged hash functions were discussed informally:

- **Balance:** Items are distributed to buckets "randomly" in every view.
- **Monotonicity:** When a bucket is added to a view, the only items reassigned are those that are assigned to the new bucket.
- **Load:** The *Load* of a bucket is the number of items assigned to a bucket over a set of views. Recall that ideally, the load should be small.
- **Spread:** The *Spread* of an item is the number of buckets an item is placed in over a set of views. Ideally, spread should be small.

The remainder of this section defines formally these intuitive properties. Throughout, we use the following notational conventions: H is a ranged hash family, f is a ranged hash function, V is a view, i is an item, and b is a bucket.

Balance: A ranged hash family is *balanced* if, given any particular view V an item i, and a bucket $b \in V$, the probability that item i is mapped to bucket b in view V is $O(1/|V|)$.

The balance property is what is prized about standard hash functions: an item is equally likely to be put into any bucket. The balance property does not say anything

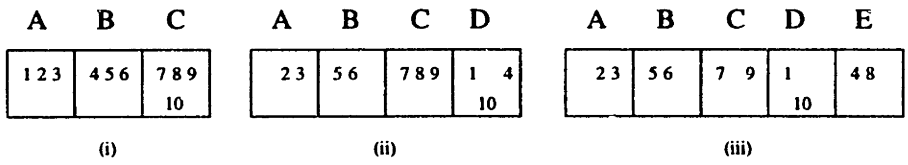


Figure 2-5: (i) The items $1, 2, \ldots, 10$ hashed into 3 buckets A, B, and C. (ii) A new, fourth bucket D is added. Since the hash function is monotone, the only items that are relocated are those that move into the new bucket D. (i.e. items 1, 4 and 10) (iii) Another new bucket E is added and some items move into E (items 4 and 8). Items do not move between old buckets.

about behavior over changing views, only that in each fixed view items are distributed with roughly equal probabilities.

Monotonicity: A ranged hash function f is *monotone* if for all views $V_1 \subseteq V_2 \subseteq B$, $f_{V_2}(i) \in V_1$ implies $f_{V_1}(i) = f_{V_2}(i)$. A ranged hash family is *monotone* if every ranged hash function in it is monotone.

The monotonicity property says that if items are initially assigned to a set of buckets V_1 and then some new buckets are added to form 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 not be completely reshuffeled. Figure 2-5 gives an example of the monotonicity property.

The following lemma is a simple consequence of these two definitions and helps to clarify them. The lemma gives the the expected number of items that remain fixed when the range of the hash function changes.

Lemma 2.2.2 *Let H be a monotonic, balanced ranged hash family. Let V_1 and V_2 be views. The expected fraction of items i for which $f_{V_1}(i) = f_{V_2}(i)$ is $\Omega\left(\frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}\right)$ where the probability is over the uniform selection of $f \in H$.*

Proof:

In this proof, we count the number of items i for which $f_{V_1}(i) \neq f_{V_2}(i)$. In other words, as the set of usable buckets changes from V_1 to V_2, we count items that "move" instead of the number of items that are "fixed".

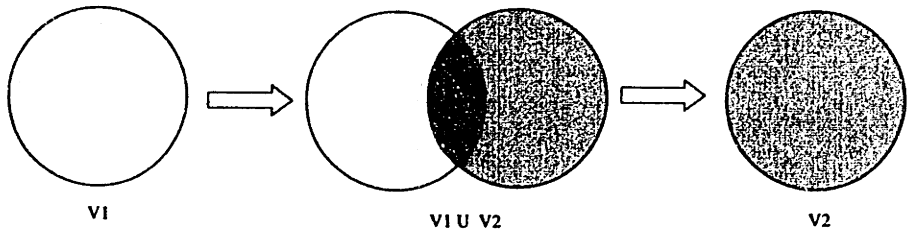


Figure 2-6: The view V_1 is transformed into the view V_2 in two steps. First V_1 is transformed into $V_1 \cup V_2$. In this step, monotonicity implies that items only move from V_1 into $V_2 - V_1$. Next, $V_1 \cup V_2$ is transformed into V_2. Here monotonicity implies that items only move from $V_1 - V_2$ into V_2. Balance tells us how many items are expected to move in each step.

We change the set of usable buckets from V_1 to V_2 in two steps. In the first step, we expand the set of buckets from V_1 to $V_1 \cup V_2$ (see figure 2-6). Balance implies that the expected fraction of items that move into $V_2 - V_1$ is $O\left(\frac{|V_2 - V_1|}{|V_1 \cup V_2|}\right)$. Monotonicity implies that no other items move.

In the second step we contract the set of buckets from $V_1 \cup V_2$ to V_2. Again, balance implies that the expected fraction of items that move into V_2 from $V_1 - V_2$ is $O\left(\frac{|V_1 - V_2|}{|V_1 \cup V_2|}\right)$ and monotonicity implies that no other items move.

In the first step, items were only moved into $V_2 - V_1$. In the second step, items were only moved out of $V_1 - V_2$. Since these sets are disjoint, no item moved in both steps. Therefore, the expected fraction of items which moved in either step is:

$$O\left(\frac{|V_2 - V_1|}{|V_1 \cup V_2|}\right) + O\left(\frac{|V_1 - V_2|}{|V_1 \cup V_2|}\right)$$

Therefore, the expected fraction of items that remain fixed is:

$$1 - O\left(\frac{|V_2 - V_1|}{|V_1 \cup V_2|} + \frac{|V_1 - V_2|}{|V_1 \cup V_2|}\right) = \Omega\left(\frac{|V_1 \cap V_2|}{|V_1 \cup V_2|}\right)$$

■

We continue to define properties of ranged hash families that capture additional aspects of the notion of "consistency".

Spread: Let $\mathcal{V} = \{V_1 \ldots V_k\}$ be a set of views. For a ranged hash function f, and a particular item i, the *spread* of the function on item i over the set of views $\mathcal{V}$ is the

quantity $|\{f_{V_1}(i), f_{V_2}(i), \ldots, f_{V_k}(i)\}|$. This quantity is denoted $spread_f(\mathcal{V}, i)$.

The spread of an item i over a set of views is the number of different buckets over all views that i is mapped to by f. Figure 2-7 illustrates spread.

In terms of the caching system described in section 2.1 spread is the number of different caches that are assigned responsibility for a document when there exist multiple views. If i is a document, then $\{f_{V_1}(i), f_{V_2}(i), \ldots, f_{V_k}(i)\}$ is the set of all the responsible caches. Recall that the central server supplies a copy of the document to each of these responsible caches, so low spread is vital if the scheme is to eliminate swamping.

Clearly, spread is very sensitive to the set of views $\mathcal{V}$. For example, if each view consisted of a single different bucket, then the spread of every item would be the total number of views! Even if views are restricted to contain at least a $1/t$ fraction of the buckets, then there exists a set of t views that force the spread of any item to be t. Simply take t *disjoint* views each containing a $1/t$ fraction of the buckets. In this case an item is assigned to a different bucket in every view. Hence, the spread of every item is t.

In general, we study spread under the assumption that each view contains at least a $1/t$ fraction of the buckets (t does not have to be a constant). As the above example shows, under this assumption spread cannot be less than t, but the question to be answered is: "How does the spread grow as a function of the number of views?" In some ranged hash families the spread grows linearly with the number of views. However, there exist families where the spread grows only logarithmically. One such family is described in section 2.2.3.

Load: Let $\mathcal{V} = \{V_1 \ldots V_k\}$ be a set of views. For a ranged hash function f, and a particular bucket b, the *load* of the function on bucket b over the set of views $\mathcal{V}$ is the quantity $\left|\bigcup_{V_j \in \mathcal{V}} f_{V_j}^{-1}(b)\right|$. This quantity is denoted $load_f(\mathcal{V}, b)$.

Note that $f_{V_j}^{-1}(b)$ is the set of items assigned to bucket b in view V_j. Thus, $load_v(\mathcal{V}, b)$ is the total number of items which in some view are assigned to the bucket b. Figure 2-8 illustrates the concept of load.

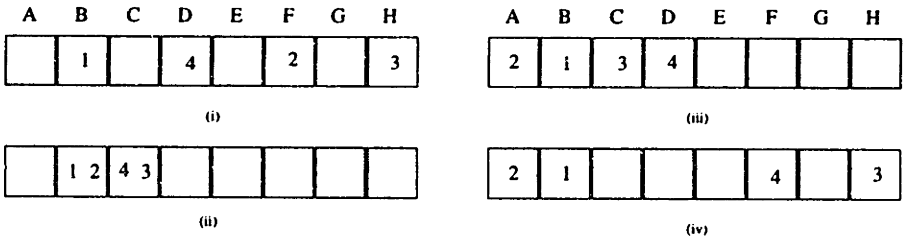


Figure 2-7: Four views of the buckets $\{A, B, C, D, E, F, G, H\}$ and the mapping of items $1, 2, 3, 4$ into the buckets for each view. Shaded buckets are not available in the given view. (i) The mapping for view $\{B, D, F, H\}$. (ii) The mapping for the view $\{B, C\}$. (iii) The mapping for the view $\{A, B, C, D\}$. (iv) The mapping for the view $\{A, B, F, H\}$. Item 1 is placed in bucket B in all the views. Thus, the spread of item 1 over these views is 1. Item 4 is placed in the buckets D,C, and F over the four views. Thus the spread of item 4 over these views is 3.

The load property measures the effect of multiple views on the number of items hashed to a given bucket. For example, in the caching scheme of section 2.1 a document i contributes to the load on a server b if there is as least one view such that i is assigned to b in that view ($f_V(i) = b$). Thus, load measures how many documents a server is responsible for in the presence of multiple views.

Load is related to spread, but they are not equivalent. For example imagine a hash function that maps all the items to a single bucket b. For any item and any set of views all containing b, the spread of i is one. On the other hand, the load of b is the total number of items. In this case the distribution of items is not balanced between the buckets.

As in the case of spread, load is generally studied under the assumption that each view contains at least a $1/t$ fraction of the buckets. (Again, t does not have to be a constant.)

We have defined a number of "consistency properties" of ranged hash families: Balance, Monotonicity, Spread, and Load. The name *consistent hash family* is used loosely to describe a ranged hash family that has good behavior with respect to these properties. When introducing a new hash family the performance relating to each of the consistency properties is explicitly stated.

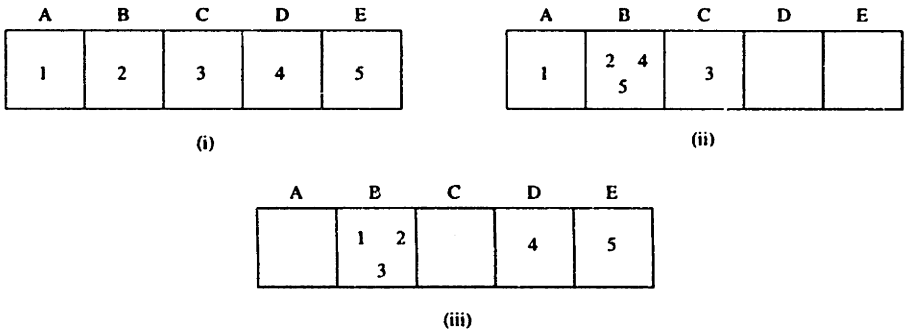


Figure 2-8: The distribution of items $1, 2, 3, 4, 5$ in buckets A, B, C, D, E for three different views. The load of bucket A over these views is one since only the item 1 is placed in it. The load of bucket B is five since all the items, in some view, are placed in it.

2.2.3 A Consistent Hash Family

This section describes the main example of a consistent hash family presented in this paper. The informal construction in section 2.1 is formalized as a ranged hash family in section 2.2.3. This formalization captures the intuition of the circle hash function. However, this simplified family requires manipulation of real numbers which is clearly a drawback. Nevertheless, in section 2.2.3 we state and prove the consistency parameters for this family. In section 2.2.4 we show how to modify the family to obtain one that is efficiently computable (does not rely on real numbers), and also retains the same consistency properties.

Construction

This section formalizes the intuition of the circle construction presented in section 2.1. In the basic construction, items and buckets are mapped to "random" points around a circle, and an item is mapped to the bucket whose point is encountered first when the circle is traversed clockwise from the item's point. Recall that there is a possibility that points could be distributed badly around the circle; one bucket may be responsible for a disproportionate section of the circle. Such an unfortunate instance is shown in figure 2-9. In order to decrease the likelihood of such an unlucky event we slightly modify the basic construction by assigning to a bucket m ($m \geq 1$) points around the circle instead of just one. As before, items are assigned to a single point

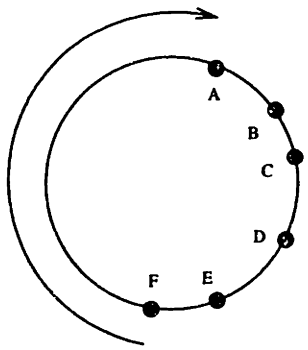


Figure 2-9: An unlucky random placement of bucket points around the unit circle. Bucket A is responsible for a disproportionately large section of the unit circle. Since items points are distributed randomly around the circle it is very likely that bucket A will have many more items assigned to it than other buckets do.

and are mapped to the bucket that has a point nearest to the item's point in the clockwise direction. Intuitively, when m is large there is less of a chance that buckets will be unequally distributed around th circle. We show that m need not be unmanageably large for there to be a very good chance of a good distribution of points. This modified version of the circle hash family is formally described below.

Let C be a circle with circumference one. We will call C the *unit circle.*[4] Let $r_I : I \mapsto C$ be a function that maps items to the unit circle. Let $r_B : B \times [m] \mapsto C$ ($[m] = \{1, 2, 3, \ldots, m\}$) be a function that maps multiple copies of buckets to the unit circle (Each pair (b, n) is considered a "copy" of the bucket b).

Each hash function in the "unit circle consistent hash family" is represented by a pair of functions: (r_B, r_I). Given such a pair, $f_V(i)$ is defined to be the bucket in the view V that has a point closest to $r_I(i)$ going clockwise around the circle. Thus, to compute the mapping of an item i in a view V we first map the bucket copies to the circle. Then, starting from the point $r_I(i)$, we sweep around the circle clockwise until we encounter a bucket point. The bucket associated with this point is the bucket that i is mapped to. Figure 2-10 depicts this situation for $m = 4$.

We say that a bucket point $r_B(b, n)$ is *responsible* for an arc in a view V if the

[4]Note that "unit circle" usually refers to a circle with radius equal to 1. We will always be referring to a circle with circumference equal to one.

bucket point is on the right end of the arc, there is no bucket point in the interior of the arc, and some other bucket point is on the left end of the arc. Any item whose point falls into this arc is assigned to the bucket b by the function. Figure 2-10 shows a mapping of bucket points to the unit circle, and the arcs that each bucket point is responsible for.

If we assume that the family is made up of all possible pairs of functions mapping buckets and items to the unit circle, then choosing a random function from the family is equivalent to choosing two completely random functions (r_B, r_I). Hence, we call this family UC_{random} (for "unit circle - random mappings").

Using the family UC_{random} requires the manipulation of real numbers. Clearly this is impractical, but in section 2.2.4 we show how to modify the construction so that real numbers are not used. For now, assume that we can compute with real numbers, that is, we can choose a function from the family and can evaluate functions in the family.

Analysis

In this section we state and prove the consistency parameters of the ranged hash family UC_{random}. These parameters hold probabilistically over the choice of function from the family. In particular, the bounds on the parameters will hold with probability at least $1 - 1/N$ where N is an arbitrarily chosen confidence factor. The confidence factor N appears in the bounds themselves so that if a high confidence factor is desired, then the bounds are degraded accordingly. This approach is a generalization of the common practice of expressing probabilities as a function of problem size. Throughout the thesis, logarithms are taken base e unless otherwise sepcified.

The following theorem states the consistency parameters of the family UC_{random}.

Theorem 2.2.3 *Let $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$ be a set of views of the set of buckets B such that: $\left|\bigcup_{j=1}^{k} V_j\right| = T$ and for all $1 \leq j \leq k$, $|V_j| \geq T/t$. Let $N > 1$ be a confidence factor. If each bucket is replicated and mapped m times then:*

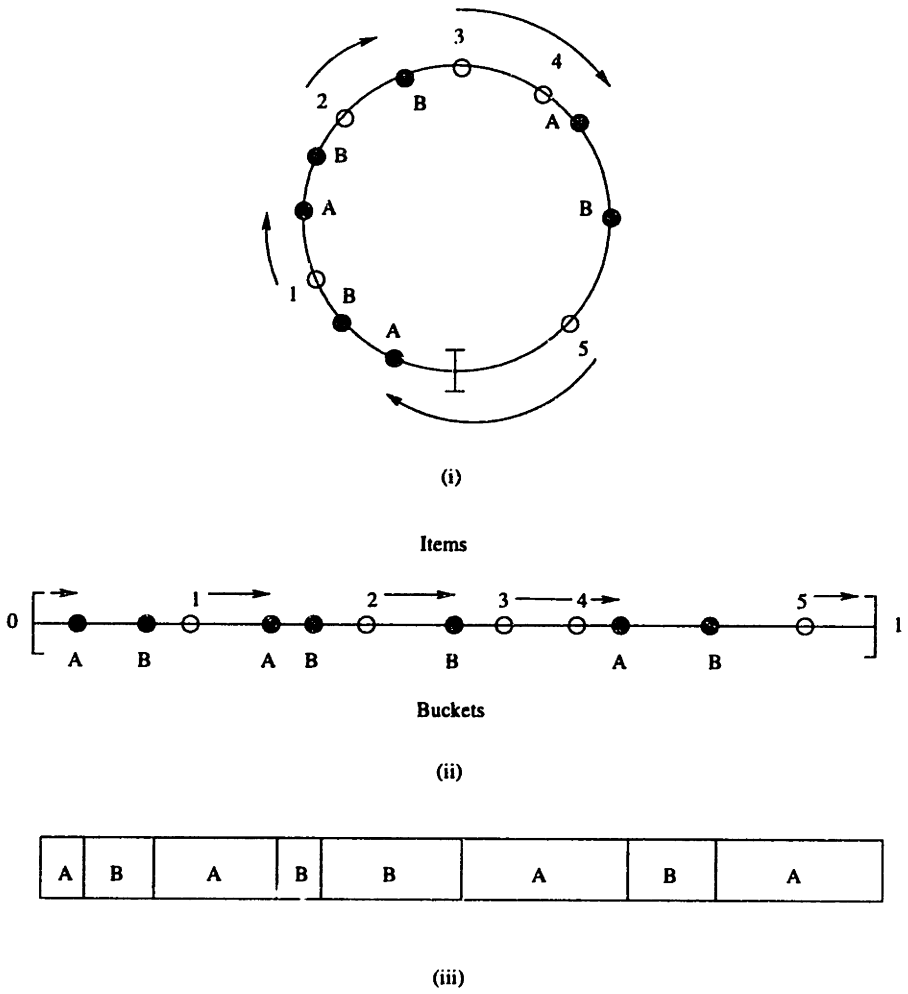


Figure 2-10: (i) A unit circle hash function with $m = 4$. Buckets A and B have 4 points associated with each of them. Items are mapped to the bucket closest to them going clockwise. Item 1 is closest to a point of bucket B and item 2 is closest to a point of bucket B. Item 5 is closest clockwise to a point of bucket A. (ii) The unit circle drawn as an interval with length one where we imagine that the endpoints of the interval are "glued together". (iii) The parts of the circle (viewed as an interval) that buckets A and B are responsible for. Bucket points are responsible for the arc directly to their left. Since there are multiple copies of each bucket, buckets are responsible for a set of arcs.

- *Monotonicity: The family UC_{random} is monotone (regardless of $\mathcal{V}$ and N).*
- *Spread: For any item $i \in I$, $spread_f(\mathcal{V}, i) = O(t \log(Nk))$ with probability greater than $1 - 1/N$ over the choice of $f \in UC_{random}$.*
- *Load: For any bucket $b \in B$, $load_f(\mathcal{V}, b) = O\left(\left(\frac{|I|}{T} + 1\right) t \log(Nmk)\right)$ with probability greater than $1 - 1/N$ over the choice of $f \in UC_{random}$.*
- *Balance: For any fixed view V and item i, the probability that item i is mapped to bucket b in view V is $O\left(\frac{1}{|V|}\left(\frac{\log(N|V|)}{m} + 1\right)\right) + \frac{1}{N}$.*

The balance claim of theorem 2.2.3 requires clarification. Note that if we choose $m = \Omega(\log(|V|))$, and $N = poly(|V|)$, then the bound simplifies into $O(1/|V|)$ which gives the definition of the balance property.

The proof of theorem 2.2.3 is presented in the remainder of this section as a series of lemmas.

In a number of the proofs, the Chernoff bound (See appendix A) is used where a more direct method could be applied. The reason for this is that in section 2.2.4 the family UC_{random} is modified so that the mapping of points to the circle is not completely random. The modification will be such that the Chernoff bounds will still hold; thus the proofs presented in this section remain valid. Cases in which superfluous use of Chernoff bounds are made are highlighted, and should not make reading the proofs any harder.

Intuitively, the family is monotone since when a new bucket is added, the only items that move are those that are now closest clockwise to points associated with the new bucket. The proof of monotonicity is simple and is given in the following Lemma.

Lemma 2.2.4 *The family UC_{random} is monotone.*

Proof:

Let $V_1 \subseteq V_2 \subseteq B$ be two views of the buckets. Let f be any function in UC_{random}. We need to show that $f_{V_2}(i) \in V_1$ implies $f_{V_1}(i) = f_{V_2}(i)$. Now, $f_{V_2}(i) \in V_1$ implies

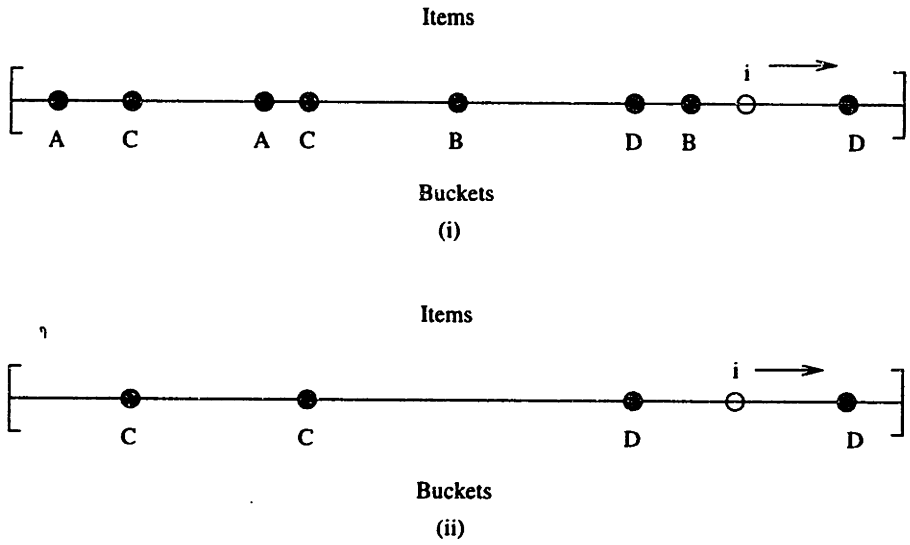


Figure 2-11: Monotonicity for the family UC_{random}. In this figure the unit circle is depicted by an interval of length one, which is obtained by cutting the unit circle at an arbitrary point. (i) The mapping of points to the circle for a view $V_2 = \{A, B, C, D\}$ ($m = 2$ in this example). The closest bucket point clockwise of i's point is one associated with the bucket D. (ii) For any view $V_1 \subseteq V_2$ containing the bucket D (here $V_1 = \{C, D\}$), the point closest to i's point will still be D.

that when adding buckets to V_1 to get V_2, none of the points that we add to the unit circle fall in the arc between i's point and the bucket point that it was previously closest to in V_1. Thus, the item i must be mapped to to the same bucket in both V_1 and V_2 (see figure 2-11).

■

Before showing the bound on spread, a technical lemma is derived which is used in both the spread and load bound proofs. The lemma shows that an arc does not have to be very long if we want there to be high probability that at least one bucket point from *every* view falls into the arc.

Lemma 2.2.5 *Any fixed set of measure at least* $\frac{4t\log(2Nk)}{Tm}$ *in the unit circle contains at least one bucket point from every view with probability greater than* $1 - \frac{1}{2N}$

Proof:

Note that the probability is over the choice of the function r_B. Since in the family UC_{random} all functions are equally likely, we can assume in the proof that points for the buckets are distributed uniformly and at random around the unit circle.

Let X_j be a random variable denoting the number of points associated with buckets in view V_j that fall into a set of measure l. There are at least $M = \frac{mT}{t}$ bucket points associated with each view (since every view contains at least T/t buckets). However, we can assume that there are exactly M points in each view since more points would imply that a set with smaller measure would suffice (more precisely, the distribution of X_j when there are more than M points stochastically dominates the distribution when there are M points). Thus, we have $E[X_j] = Ml$. We will choose the value of l so that the probability of X_j being 0 is at most $1/2Nk$.

Following is a use of the Chernoff bound where we could have made a more direct argument. [5]

$$\begin{aligned} \Pr[X_j = 0] &\leq \Pr[|X_j - Ml| \geq Ml] \\ &\leq 2e^{-\frac{Ml}{4}} \end{aligned}$$

So we choose l so that $2e^{-\frac{Ml}{4}} = \frac{1}{2Nk}$. We obtain $l = \frac{4\log(4Nk)}{M} = \frac{4t\log(4Nk)}{Tm}$.

From the union bound we have:

$$\Pr[\ \textit{Some}\ X_j = 0] \leq \sum_{j=1}^{k} \Pr[X_j = 0] \leq k\frac{1}{2Nk} = \frac{1}{2N}$$

Thus, any set of measure $\frac{4t\log(4Nk)}{Tm}$ contains a bucket point from every single view with probability at least $1 - 1/2N$.

■

We now show the bound on spread.

Lemma 2.2.6 *For any item $i \in I$, $spread_f(\mathcal{V}, i) = O(t\log(Nk))$ with probability greater than $1 - 1/N$ over the choice of $f \in UC_{random}$.*

[5]The direct argument is $\Pr[X_j = 0] = (1 - l)^M$ since each point is mapped independently to the circle.

Proof:

The proof of the lemma uses a technique that reoccurs many times in this paper. The basic idea is simple. If we want to show that some event A has low probability, then we can proceed as follows:

1. Find a set of events $\mathcal{B}$ so that the occurrence of event A implies that at least one of the events in $\mathcal{B}$ occurs.

2. Show that there is only a small probability that *any* event in $\mathcal{B}$ occurs.

3. Deduce that the probability of A must also be small.

More formally stated, the technique is to find a set of events $\mathcal{B}$ so that $A \subset \bigcup_{B_i \in \mathcal{B}} B_i$. This implies that:

$$\Pr[A] \leq \Pr\left[\bigcup_{B_i \in \mathcal{B}} B_i\right] \leq \sum_{B_i \in \mathcal{B}} \Pr[B_i]$$

So, if the sum of the probabilities of the events in $\mathcal{B}$ is small then the probability of A must also be small.

The first step is to define the event A and the family of events $\mathcal{B}$. In our case, A is the event that the bound on the spread does *not* hold. The family $\mathcal{B}$ contains two events: B_1 and B_2 which are defined as follows:

1. B_1: Fix an arc α of length $\frac{4t\log(4Nk)}{Tm}$ to the right of the point $r_I(i)$ associated with the item i. Denote by B_1 the event that there is *some* view that has *no* bucket point in the arc α. Note that this is, in some sense, the complement of the event that we considered in the previous lemma (lemma 2.2.5).

2. B_2: Let X be the random variable denoting the total number of bucket points in the arc α. Let B_2 denote the event that X is *more* than $8t\log(4Nk)$; that is, more than $8t\log(4Nk)$ bucket points fall into the arc α.

The next step is to show that the event A implies at least one of the events B_1 and B_2. In this case, it is easier to show the contrapositive, or: $\overline{B_1} \cap \overline{B_2} \Rightarrow \overline{A}$.

If event $\overline{B_1}$ occurs, we know that *every* view has at least one bucket point in the arc α. Now, for each view V, the item i is mapped to the first bucket point from V encountered in a clockwise traversal of the circle. The event $\overline{B_1}$ implies that this bucket point will be found somewhere in the arc α. So, it must be that in every view, i is mapped to some bucket with a point in α. Therefore, the spread of i cannot be larger than the total number of bucket points that fall into the arc α! If in addition to $\overline{B_1}$, event $\overline{B_2}$ occurs, then we know that the number of buckets in α is *less* than $8t\log(4Nk)$, and hence, the spread of i must be less than $8t\log(4Nk) = O(t\log(4Nk))$. This is precisely the event $\overline{A}$! This proves that $\overline{B_1} \cap \overline{B_2} \Rightarrow \overline{A}$.

The last and final step is to bound the probabilities of the events B_1 and B_2.

Lemma 2.2.5 shows that $\Pr[\overline{B_1}] \geq 1 - 1/2N$, and thus $\Pr[B_1] \leq 1/2N$. It remains to bound the probability of event B_2. Note that there are a total of Tm points coming from all the views and thus $E[X] = Tm\frac{4t\log(2Nk)}{Tm} = 4t\log(2Nk)$. The Chernoff bound implies that:

$$\begin{aligned}\Pr[\ |X - 4t\log(4Nk)| \geq 4t\log(4Nk)] \\ \leq 2e^{-t\log(4Nk)} = \frac{2}{(4Nk)^t} \leq \frac{1}{2N}\end{aligned}$$

(Assuming that $t \geq 1$.)

Consequently, $\Pr[B_2] \leq 1/2N$. To wrap up the proof we have:

$$\Pr[A] \leq \Pr[B_1 \cup B_2] \leq \Pr[B_1] + \Pr[B_2] \leq 1/2N + 1/2N = 1/N$$

This proves that the probability that the bound on spread does *not* hold is less than $1/N$, and thus, the probability that the bound *does* hold is at least $1 - 1/N$. This concludes the proof of the lemma.

■

The next Lemma shows the load bound.

Lemma 2.2.7 *For any bucket $b \in B$, $load_f(\mathcal{V}, b) = O\left(\left(\frac{|I|t}{T} + 1\right)\log(Nmk)\right)$ with probability greater that $1 - 1/N$ over the choice of $f \in UC_{random}$.*

Proof:

The intuition for the proof is simple: An item that is assigned to a bucket must fall into one of the arcs that the bucket's points are responsible for. Lemma 2.2.5 implies that the length of the arc that a single bucket point is responsible for over all views is not too big. Hence the total fraction of the circle that a bucket is responsible for over all views is relatively small. Thus, only a small fraction of the items are assigned to a bucket even if there are many views. More precisely, lemma 2.2.5 is used to bound the length of the arc that a single bucket point is responsible for over all views. Multiplying this length by m we get a bound on the total measure of the set in which items could fall and be assigned to the bucket in question. We then bound the total number of items that fall into this set, getting an upper bound on the load.

Since in the family UC_{random} all functions are equally likely to be chosen, we can assume in the proof that points for the buckets and items are distributed uniformly and at random in the unit circle. In addition, we note that item points are distributed independently of bucket points.

The bucket b has m points associated with it in the circle. An item is assigned to the bucket b if in some view it is closest clockwise to one of these m points among all other bucket points.

Fix one of the m points associated with the bucket b. We examine an arc starting at this point and going counter-clockwise around the circle that is long enough so that in *every* view, there is another bucket point in the arc with probability at least $1 - \frac{1}{2Nm}$. Invoking lemma 2.2.5 we get that the length of such an arc is given by $\frac{4t\log(8Nmk)}{Tm}$.

Now, by the union bound we have that with probability at least $\frac{1}{2N}$, the length of the arc to the left of *every one* of the m points associated with b is $\frac{4t\log(8Nmk)}{Tm}$. We denote by A this event.

Now we have:

$$\begin{aligned}
\Pr[\ load_f(\mathcal{V}, b) \geq z\ ] &= \Pr[\ load_f(\mathcal{V}, b) \geq z \mid A]\Pr[A] \\
&+ \Pr[\ load_f(\mathcal{V}, b) \geq z \mid \overline{A}]\Pr[\overline{A}] \\
&\leq \Pr[\ load_f(\mathcal{V}, b) \geq z \mid A] + \Pr[\overline{A}] \\
&\leq \Pr[\ load_f(\mathcal{V}, b) \geq z \mid A] + 1/2N
\end{aligned}$$

It remains to bound the load on bucket b given that event A has occurred. If event A occurs then we know that any item assigned to bucket b falls within distance $\frac{4t\log(8Nmk)}{Tm}$ of one of the m points associated with b. Thus, if an item is assigned to bucket b it must fall in a set of total measure at most $m\frac{4t\log(8Nmk)}{Tm} = \frac{4t\log(8Nmk)}{T}$. If there is overlap of arcs then the total measure may be smaller, but this will only lead to a better bound on the load. Therefore, all we need to do is bound the number of item points that fall into a set of measure $\frac{4t\log(8Nmk)}{T}$.

Let X denote the number of item points in the set. The expected number of item points is $E[X] = \frac{4t|I|\log(8Nmk)}{T}$. Since item points are mapped independently of bucket points we can use Chernoff bounds on X. This unfortunately turns out to be a bit technical. There are two cases to be considered according to the value of $\frac{T}{|I|}$:

Case 1: $0 < \frac{T}{|I|} \leq 2e - 1$

In this case we use a Chernoff bound with $\delta = \sqrt{\frac{T}{|I|}}$. This gives us:

$$\Pr\left[X > \left(1 + \sqrt{\frac{T}{|I|}}\right)\frac{4t|I|\log(8Nmk)}{T}\right] \leq \frac{1}{(8Nmk)^t} \leq \frac{1}{2N}$$

(Since $t, k, m \geq 1$.)

Case 2: $2e - 1 \leq \frac{T}{|I|}$

In this case we use a Chernoff bound with $\delta = \frac{T}{|I|}$. This gives us:

$$\begin{aligned}\Pr\left[X > \left(1+\frac{T}{|I|}\right)\frac{4t|I|\log(8Nmk)}{T}\right] &\leq \frac{1}{(8Nmk)^{t\frac{|I|}{T}(1+\frac{T}{|I|})}} \\ &\leq \frac{1}{(8Nmk)^{t}} \\ &\leq \frac{1}{2N}\end{aligned}$$

(Since $t, k, m \geq 1$.)

Since both $\left(1+\sqrt{\frac{T}{|I|}}\right)\frac{4t|I|\log(8Nmk)}{T}$ and $\left(1+\frac{T}{|I|}\right)\frac{4t|I|\log(8Nmk)}{T}$ are $O\left(\left(\frac{|I|t}{T}+1\right)\log(Nmk)\right)$ we have shown that for $z = O\left(\left(\frac{|I|t}{T}+1\right)\log(Nmk)\right)$:

$$\begin{aligned}\Pr[\ load_f(\mathcal{V}, b) \geq z\ ] &\leq \Pr[\ load_f(\mathcal{V}, b) \geq z \mid A] + 1/2N \\ &\leq 1/2N + 1/2N = 1/N\end{aligned}$$

This concludes the proof of the lemma.

■

It remains to show the balance property. If we fix a particular view, then the probability that an item is assigned to a particular bucket is exactly the total length of the unit circle that the bucket is "responsible" for. The following lemma bounds the total length of the set that each bucket is responsible for, and thus will be the main step in showing the balance property (which is proved in lemma 2.2.9).

For a bucket b, denote by $length(b)$ the measure of the set of points in the unit circle that b is responsible for.

Lemma 2.2.8 *Let V be a fixed view containing $v = |V|$ buckets. Then with probability at least $1 - 1/N$ for all $b \in V$ $length(b) = O\left(\frac{1}{v}\left(\frac{\log(Nv)}{m}+1\right)\right)$.*

Proof:

The proof of lemma 2.2.8 is based on the same idea as the proof of lemma 2.2.6. If we want to show that some event A has low probability, then we can find a set of

events $\mathcal{B}$ so that the occurrence of event A implies that at least one of the events in $\mathcal{B}$ occurs. Then, if we show that there is only a small probability that *any* event in $\mathcal{B}$ occurs, then the probability of A must also be small.

In our case the event A is that some bucket b has $length(b)$ greater than the value stated in the lemma. Recall that the part of the unit circle that the bucket b is "responsible" for is broken up into m non-overlapping arcs. One (not good) set of events $\mathcal{B}$ is obtained by observing that if b is responsible for a set of measure p, then there is a collection of m arcs of total length p (with right ends at the m points associated with b) in which no other bucket point falls. Thus each event in $\mathcal{B}$ is described by a set of m arcs of total length p, into which no other bucket point falls. The problem is that there are uncountable many ways to divide length p among m arcs. Since we want to use a union bound to bound the probability that any event from $\mathcal{B}$ occurs, the set needs to be finite!

We make $\mathcal{B}$ smaller by discretizing the circle and counting the number of ways to distribute total length p by discrete units. This is of course finite, and the error introduced by the discretization turns out to be small. We now formalize the above argument.

Recall that a bucket point is *responsible* for an arc if the bucket point is on the right end of the arc, there is no bucket point in the interior of the arc, and some other bucket point is on the left end of the arc. A bucket b is responsible for the union of the m arcs that the points associated with b are responsible for.

The main result is that the probability a single bucket b is responsible for more than an $O\left(\frac{1}{v}\left(\frac{\log(Nv)}{m}+1\right)\right)$ fraction of the unit circle is at most $1/Nv$. The union bound then implies that none of the v buckets are responsible for more than an $O\left(\frac{1}{v}\left(\frac{\log(Nv)}{m}+1\right)\right)$ fraction of the circle with probability $1-1/N$.

To show the main result, we begin by fixing a bucket b. The portion of the unit circle for which b is responsible must consist of m non-overlapping arcs in the circle, each bounded on the right by one of b's points. Suppose we shrink all these arcs by moving the left endpoints rightward, until the length of every arc is a multiple of $\Delta = \frac{4\alpha}{mv}$ (α will be set later). Since there are m of these arcs and each shrinks by at

most Δ, the decrease in the total length of all arcs is at most $4\frac{\alpha}{v}$. So if the total length of these arcs after shrinking is $4\frac{\alpha}{v}$, then the total length before shrinking is at most $4\frac{\alpha}{v}+4\frac{\alpha}{v}=8\frac{\alpha}{v}$. This implies that if bucket b is responsible for a $8\frac{\alpha}{v}$ fraction of the unit circle, then b must be responsible for every point in a collection of non-overlapping arcs, each bounded on the right with one of b's points, each a multiple of Δ in length, and with total length $4\frac{\alpha}{v}$.

Now, given a collection of arcs of total length $4\frac{\alpha}{v}$ we will bound the probability that in these arcs there is no point associated with some other bucket. The expected number of the $mv-m$ points falling in this collection of arcs is $4\frac{\alpha m(v-1)}{v}$. So we have from a superfluous use of the Chernoff bounds:

$$\begin{aligned} Pr[X=0] &\leq Pr[(X-4\alpha m\frac{v-1}{v})\geq 4\alpha m] \\ &\leq e^{-\alpha m\frac{v-1}{v}} \end{aligned}$$

The number of collections of m arcs with total length $\frac{4\alpha}{v}$ and with all lengths multiples of Δ is exactly the number of ways to partition $\frac{4\alpha}{v}/\Delta=m$ into m integral parts, which is:

$$\binom{2m-1}{m-1}\leq 2^{2m-1}\leq e^{2m}$$

By the union bound, the probability that any of the above collections of arcs contains no point associated with other buckets is at most:

$$e^{-\alpha m\frac{v-1}{v}}e^{2m}=e^{-(\alpha m\frac{v-1}{v}-2m)}$$

We will choose α so that this probability is at most $1/(Nv)$. which gives:

$$\alpha=O\left(\frac{\log(Nv)}{m}+1\right)$$

This proves that with probability at least $1 - 1/(Nv)$ the total length assigned to a bucket is at most $\frac{8\alpha}{v} = O(\frac{1}{v}\left(\frac{\log(Nv)}{m} + 1\right))$. Now since there are v buckets the same bound holds for all buckets with probability $1 - 1/N$ by the union bound.

■

We now prove the balance bound given in theorem 2.2.3.

Lemma 2.2.9 *For any fixed view V and item i, the probability that item i is mapped to bucket b in view V is $O\left(\frac{1}{|V|}\left(\frac{\log(N|V|)}{m} + 1\right)\right) + \frac{1}{N}$.*

Proof:

Denote by A the event that for every bucket b in the view V we have $length(b) = O\left(\frac{1}{|V|}\left(\frac{\log(N|V|)}{m} + 1\right)\right)$. Lemma 2.2.8 says that the probability of A is at least $1-1/N$. Now for any $b \in V$:

$$\begin{aligned}
\Pr[\, f_V(i) = b] &= \Pr[\, f_V(i) = b |\, A] \Pr[A] \\
&+ \Pr[\, f_V(i) = b |\, \overline{A}] \Pr[\overline{A}] \\
&\leq \Pr[\, f_V(i) = b |\, A] + \Pr[\overline{A}] \\
&\leq \Pr[\, f_V(i) = b |\, A] + 1/N \\
&= O\left(\frac{1}{|V|}\left(\frac{\log(N|V|)}{m} + 1\right)\right) + \frac{1}{N}
\end{aligned}$$

Since, given that event A has occurred, the probability that item i is mapped to any particular bucket b is exactly $length(b)$ (the mapping of items is independent of the mapping of buckets) which is $O\left(\frac{1}{|V|}\left(\frac{\log(N|V|)}{m} + 1\right)\right)$.

■

This concludes the proof of theorem 2.2.3. We state and prove one corollary of theorem 2.2.3 that is useful for various applications.

Corollary 2.2.10 *With the same conditions as theorem 2.2.3, the probability that an item i is mapped to a bucket b in at least one of the views is $O\left(\frac{t\log(Nmk)}{T}\right) + \frac{1}{N}$.*

Proof:

We saw in the proof of lemma 2.2.7 that with probability at least $1 - 1/N$, the bucket b is not responsible for more than a $O\left(\frac{t \log(Nmk)}{T}\right)$ fraction of the circle over all views. Using the same argument as in the proof of lemma 2.2.9, the corollary follows.

■

2.2.4 A Practical Consistent Hash Family

In the previous section, we showed that the family UC_{random} has good consistency properties: the family is monotone, spread and load grow logarithmically with the number of views, and the family has the balance property.

However, there is a drawback to the family UC_{random}; it requires manipulation of real numbers. More specifically, storing a function requires infinite space, and furthermore, choosing a function from the family requires an infinite number of random bits.

In this section we remedy these problems by modifying the basic construction in two simple ways. We show that limited independence in the mapping of points to the circle suffices for the family to have the same consistency properties as UC_{random}. Using limited independence reduces the number of random bits required to choose a function from the family, and reduces the space required to store a function from the family. Furthermore, we show how to use limited precision in the real numbers used in the basic construction, thus eliminating the need to manipulate arbitrary real numbers. In section 2.2.4 these two modifications are combined to construct a new, more practical hash family. An implementation of this family, which is remarkably simple and efficient, is presented in section 2.2.5.

Using Limited Independence

We say that a family of functions is k-way independent if any k elements from the domain are mapped independently into the range when we choose a function at random from the family. In other words a family is k-way independent if for any distinct

x_i, $1 \leq i \leq k$ from the domain of the family, and any y_i, $1 \leq i \leq k$ from the range of the family:

$$\Pr[f(x_1) = y_1, f(x_2) = y_2, \ldots, f(x_k) = y_k] = \prod_{i=1}^{k} Pr[f(x_i) = y_i]$$

Where the probability is over a random choice of function f from the family. Another way of looking at this is that the random variables $\{f(i)\}$ where i ranges over the domain of the family are k-way independent.

As an example, consider the linear congruential hash family introduced in example 1. This is a 2-way independent family. We showed that the number of random bits required to choose a function from the family and the space required to store a function from the family are very small compared to a completely random function. These are exactly the reasons that we are interested in using limited independence mappings.

We show that if UC_{random} is modified so that items and bucket copies are mapped to the circle using limited independence, then the consistency properties of the family remain unchanged.

The basic tool used is the following theorem from[17] that shows that Chernoff bounds apply to cases with certain limited amounts of independence.

Theorem 2.2.11 *If X is the sum of k-wise independent random variables each of which is in the range $[0, 1]$ with $\mu = E[X]$, then:*

- *For any $\delta \geq 1$ and $k \geq \lceil \delta\mu \rceil$:*

$$\Pr[\mid X - \mu \mid \geq \delta\mu] \leq e^{\frac{-\delta\mu}{3}}$$

- *For any $\delta \leq 1$ and $k \geq \lfloor \delta^2 \mu e^{-1/3} \rfloor$:*

$$\Pr[\mid X - \mu \mid \geq \delta\mu] \leq e^{\lfloor \frac{-\delta^2\mu}{3} \rfloor}$$

The only probabilistic tool used in the proof of theorem 2.2.3, which states the

consistent properties of the family UC_{random}, is the Chernoff bound on sums of indicator variables (other than the union bound which is true regardless of independence of mappings). Theorem 2.2.11 implies that the claims of theorem 2.2.3 are still valid even if item and bucket points are mapped to the circle with only limited independence and not completely randomly as assumed in the proof. In fact, we show that if the bucket and item points are each mapped $\Omega(t\log(NTk))$-way independently then all the bounds of theorem 2.2.3 still hold.

Theorem 2.2.12 *If bucket copies and items are mapped to the unit circle using $\Omega(t\log(NTk))$-way independent families, then, as long as item points are mapped independently of bucket points, theorem 2.2.3 still holds.*

Proof: Monotonicity is not affected by the independence of the mappings.

We need to check that each of the Chernoff bounds used in the proof of spread, load, and balance are still valid with only $\Omega(t\log(NTk))$-way independence. Recall that the proof of theorem 2.2.3 was divided into lemmas 2.2.6, 2.2.7, and 2.2.9.

For each of lemma 2.2.5, lemmas 2.2.6 and 2.2.7, and also lemma 2.2.8 we will show that using a function with $\Omega(t\log(NTk))$-way independence is sufficient.

- Lemma 2.2.5: The proof uses a Chernoff bound with $\delta = 1$ and $\mu = \frac{4t\log(2Nk)}{Tm}$. Invoking the first case of theorem 2.2.11 we see that we need $\left\lceil \frac{4t\log(2Nk)}{Tm} \right\rceil$-way independence. Since this is $O(t\log(NTk))$, lemma 2.2.5 holds.

- Lemma 2.2.6 **(spread):** The proof uses lemma 2.2.5 that we showed above holds. The proof of the spread bound (lemma 2.2.6) contains one more Chernoff bound with $\delta = 1$ and $\mu = 4t\log(2Nk)$. From the first case of theorem 2.2.11 we see that $\Omega(t\log(NTk))$-way independence suffices.

- Lemma 2.2.7 **(load):** The first part of the proof relies on lemma 2.2.5 which we have shown still holds. Item points are mapped independently of bucket points so the remaining Chernoff bound is still valid as long as we have enough independence: There are two cases to consider depending on whether $T/|I|$ is larger or smaller than 1.

Case 1: $T/|I| < 1$

In this case we use a Chernoff bound with parameters $\delta = \sqrt{T/|I|} < 1$ and $\mu = \frac{4t|I|\log(4Nmk)}{T}$. We apply the second case of theorem 2.2.11 and observe that $\Omega(t\log(NTk))$-way independence suffices. This case then parallels the case of $T/|I| < 2e-1$ in the proof of lemma 2.2.7.

Case 2: $T/|I| \geq 1$

In this case we use a Chernoff bound with parameters $\delta = T/|I| \geq 1$ and $\mu = \frac{4t|I|\log(4Nmk)}{T}$. We invoke the first case of theorem 2.2.11. This case then parallels the case of $T/|I| \geq 2e-1$ in the proof of lemma 2.2.7.

- Lemma 2.2.8 **(balance):** The proof relies on a Chernoff bound with $\delta = 1$ and $\mu = O(\log(NT) + m)$. Invoking the first case of theorem 2.2.11 we see that $\Omega(t\log(NTk) + m)$-way independence suffices, however if we choose $q = O(\log(T))$, then $\Omega(t\log(NTk))$-way independence suffices. The proof also relies on the fact the item points are mapped independently of bucket points.

■

Using Limited Precision

In this section we show how to use limited precision for representing real numbers for the unit circle hash function. The next section defines the final practical hash family and proves its properties.

The basic idea is simple. Each function in the family UC_{random} is defined by a total of $|I|+m|B|$ random points on the real unit circle. The important observation is that what really matters about these points is their clockwise *ordering* around the circle. Given just the clockwise ordering of all the points, we can reconstruct the mapping of every item in every view. An item i is mapped to a bucket b with a point closest to the point of i in the clockwise ordering. The following lemma shows that the ordering on a set of $|I| + m|B|$ random points is with high probability, already completely

defined by the $O(\log(|I| + m|B|))$ most significant bits of the binary expansions of the points.

Lemma 2.2.13 *With probability at least* $1 - 1/N'$, *the clockwise ordering on* n *random points in the unit circle is determined by the* $2\log(N'n)$ *most significant bits of the points.* ($N' \geq 1$ *is an arbitrary confidence factor.*)

Proof:

The probability that any two of the numbers can not be distinguished by their $2\log(N'n)$ most significant bits is $1/(N'n)^2$ (since the probability of an infinite sequence of 0's or 1's is zero). By the union bound, the probability that any of the $\binom{n}{2} \leq n^2$ pairs of points are not distinguished by their $2\log(N'n)$ most significant bits is less than $(1/N')^2 \leq 1/N'$.

Thus, for each point we need no more than $2\log(N'n)$ bits to determine the ordering with probability at least $1 - 1/N'$.

■

The following simple corollary shows that lemma 2.2.13 holds even if the points are distributed only k-way independently for $k \geq 2$.

Corollary 2.2.14 *Lemma 2.2.13 holds if the points are distributed uniformly and* k*-way independently for any* $k \geq 2$.

Proof: The corollary follows from two observations. The first observation is that if the points are k-way independent, then the bits at a fixed place in the binary expansion of the points are k-way independent. The second observation is that in the proof of Lemma 2.2.13 we only used 2-way independence of these bits. ■

Putting it all Together

This section describes the hash family obtained by modifying UC_{random} to use finite precision, and limited independence mappings. This family is called UC.

Let $c = O(\log(N'(m|B| + |I|)))$ for an arbitrary confidence factor $N' \geq 1$. The family UC is defined in the same way as UC_{random} except in the following aspects:

- Item and bucket copies are only mapped to points on the unit circle that can be represented by no more than c bits of precision.

- Item and bucket points are mapped to to the unit circle $\Omega(t\log(NTk))$-way independently, but items points are mapped independently of bucket points. In other words, the items and bucket copies are mapped by $\Omega(t\log(NTk))$-way independent families of functions, and that the mapping of items is independent of the mapping of buckets.

Note that now, unlike the continuous case, when we choose a random function from the family there is a positive probability that two bucket copies, are mapped to the same point. If this happens we do not have a well defined hash function (since we do not have a well defined order on the points), so in this case we use some arbitrary fixed order of the buckets. Corollary 2.2.14 implies that the probability that we choose a "bad" function from the family is less than $1/N'$, and this probability can be made arbitrarily small by making N' larger (using more bits of precision).

Let A denote the event that we choose a "bad" function from the family. Putting together theorem 2.2.12 and the fact that $\Pr[A] \leq 1/N'$, we obtain the following theorem:

Theorem 2.2.15 *If the the mappings of items and bucket copies are $\Omega(t\log(NTk))$-way independent (N is the confidence factor in theorem 2.2.3), items are mapped independently of bucket points, and // $c = O(\log(N'(m|B| + |I|)))$ bits of precision are used then theorem 2.2.3 holds with probability at least $1 - 1/N'$.*

In the next section we show that UC is remarkably simple to implement, and that the implementation is very efficient.

2.2.5 Implementation

In this section, we describe how to practically implement the consistent hash family UC. Three parameters are used throughout the description of the implementation. The first is a prime p used for discretization by restricting attention to p evenly spaced

points around the circle. The parameter p can be chosen to be $(N'(m|B| + |I|))^{O(1)}$ since the previous section shows that only numbers represented by $O(\log(N'(m|B| + |I|)))$ bits need be considered for the hash family.[6] The second parameter is m, the number of points on the unit circle associated with each bucket. The parameter m is $O(\log(|B|))$. Finally, items and buckets are associated with points on the unit interval k-way independently. The previous section shows that we may choose k to be $\Omega(t \log(NTk))$.[7]

We assume that items and buckets are both numbered 0 to $p-1$. (A bucket and an item may have the same number.)

Our implementation relies on two standard tools, a balanced binary tree data structure and a way of obtaining k-way independent random variables. These tools are described in the first subsection. Following this, we show how to implement four basic hashing operations: choosing a random hash function from a family, adding a bucket, removing a bucket, and hashing an item to a bucket. First a simple implementation is presented, and then a slightly more complicated but faster version is briefly outlined.

Standard Tools

We will need to maintain a set of ordered keys. For this purpose, any standard balanced binary tree structure will suffice. We assume the following operations:

`CreateTree()` Return a new, empty binary tree.

`AddKey(T, K)` Add key K to tree T.

`DeleteKey(T, K)` Delete key K from tree T.

`NextKey(T, K)` Return the smallest key in T greater than or equal to K. If there is no such key, return the smallest key in T.

[6] N' is the parameter that controls the probability of choosing a "bad" function from the family. The larger N' is, the more bits of precision need to be used in the representation of the family. In this implementation more bits of precision implies a larger p.

[7] N is the confidence parameter in the statement of theorem 2.2.3.

The last three operations run in time $O(\log n)$ where n is the number of keys in the tree.

We will also need to generate k-way independent, random points on the unit circle. This is equivalent to generating k-way independent integers in the range $0, \ldots, p-1$; that is, elements of the field $\mathbf{Z}_p$. (Recall that we are restricting attention to points in only p positions, evenly spaced around the circle.) A standard solution is to choose k random elements of $\mathbf{Z}_p$ and to regard these as coefficients of a degree $k-1$ polynomial. Evaluating this polynomial at $0, \ldots, p-1$ over $\mathbf{Z}_p$ gives a set of p values that are k-way independent and lie in the range $0, \ldots, p-1$. The following lemma proves this.

Lemma 2.2.16 *If Q is a random degree $k-1$ polynomial over the field Z_p for a prime p, then for any distinct $x_1, x_2, \ldots, x_k \in Z_p$ and any $y_1, y_2, \ldots, y_k \in Z_p$ we have:*

$$\Pr[Q(x_1) = y_1, Q(x_2) = y_2, \ldots, Q(x_k) = y_k] = \prod_{i=1}^{k} \Pr[Q(x_i) = y_i]$$

Proof: Since determining the values of a degree $k-1$ polynomial at the k points $x_1, \ldots, x_k$ determines the coefficients of the polynomial uniquely, the left hand side is equal to $1/p^k$ (the probability of choosing these uniquely determined k coefficients). Since the value of random polynomial at any fixed point is random, the right hand side is also equal to $1/p^k$. ∎

This method for generating k-way independent random variables still assumes the existence of some random bits (used to choose the random polynomial). However, the number of bits needed is far less that what would be required to generate p completely random values in Z_p.

Hashing Operations

In this section, we show how to implement four basic hashing operations. These are: choosing a random hash function from the family, adding a bucket, removing a bucket, and hashing an item to a bucket. Pseudo-code for these four operations is shown below.

```
ChooseHash()

  Choose random polynomials P and Q_1, ..., Q_m of degree k-1.

AddBucket(T, b)

  for i = 1 to m
    AddKey(T, (Q_i(b), b))

RemoveBucket(T, b)

  for i = 1 to m
    DeleteKey(T, (Q_i(b), b))

Hash(T, i)

  (f, b) = NextKey(T, (P(i), 0))
  return b
```

The first step is to choose a hash function at random from a family. This is done with a call to `ChooseHash()`. This function picks random polynomials P and $Q_1, \ldots, Q_m$ of degree $k-1$. The polynomial P will be used to map items to points on the circle k-way independently. The Q polynomials will be used to map each bucket to m points on the circle k-way independently. If several people plan to use the same hash function, then one person should generate these random polynomials and distribute them to the other people.

A binary tree is used to keep track of the current view. One must initially create an empty binary tree T with `CreateTree()` and supply this tree as an argument to all subsequent functions. The keys that will be stored in the tree are pairs (f, b) where f is an element of the field $\mathbf{Z}_p$ and b is a bucket. That is, each key is a point on the unit

circle and the bucket associated with it. The pairs are kept in the dictionary order, i.e. primarily based on the value of f, but based on the value of b when necessary to break ties. In effect, the tree records all points associated with available buckets in the order these points appear clockwise around the circle.

The tree T must be updated when a new bucket becomes available or when a previously available bucket is discarded. In other words, the tree T must reflect the current view. These updates are done with the `AddBucket(T, b)` and `RemoveBucket(T, b)` operations. These functions maintain the invariant that for each available bucket b, the tree stores pairs $(Q_1(b), b), \ldots, (Q_m(b), b)$. Intuitively, this just means that the tree records all of the points associated with available buckets. Both `AddBucket` and `DeleteBucket` run in time $O(mk + m \log mv)$ where v is the number of buckets in the current view.

The actual work of hashing is done with the `Hash(T, i)` operation. This function returns the bucket b to which item i is assigned in the view described by the tree T. Recall that an item should be assigned to the bucket whose point is the first encountered when the circle is traversed clockwise from the item's point. This is precisely the effect of the call to `NextKey`.

There is one technicality in the implementation of `Hash`. Suppose two buckets, b_1 and b_2, are both associated with the same point f. Then there may be some ambiguity about whether an item is assigned to b_1 or b_2. In the implementation, the item is always assigned to the lower-numbered bucket. This follows from the dictionary ordering of keys and from choosing zero as the second argument to `NextKey`. Recall however that the probability of this happening can be made arbitrarily small be choosing a larger confidence factor (using a larger field by making the prime p larger).

The `Hash` operation runs in time $O(k + \log mv)$ where v is the number of buckets in the current view.

A Faster Method

The Hash operation of the previous section runs in time $O(k + \log mv)$ where v is the number of buckets in the current view. In this section a faster method, with hash time $O(1)$, is described. The idea is simple: The unit circle is divided into roughly mv arcs of equal length, and a separate search tree for each arc is maintained. Thus, the Hash operation boils down to determining in which arc the item point falls, and then searching for the corresponding bucket in that arc's search tree. Determining in which arc an item point falls is a simple task that requires constant time (in a RAM model). If bucket points are distributed randomly around the circle, then the expected number of points in each segment is $O(1)$, and thus the time to determine the bucket is also $O(1)$ in expectation (the expectation is over the choice of function from the family UC).

One caveat to the above is that as the set of buckets in a view grows, the size of the arcs needs to shrink. The following trick solves this problem: only arc lengths of $1/2^x$ for some x are used. At first, we choose the smallest x such that $1/2^x \leq 1/mv$. Then as points as added, we bisect the arcs gradually so that when we reach the next power of two, all of the arcs have already been divided. 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 arcs 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. We do not include a detailed analysis of this method here.

The Independence Pitfall

In this section we describe a pitfall that could be encountered while implementing our hash function. There is a strong temptation to say, "Well.. we probably don't need all that much independence for our mappings - lets try and get by with less!". In this section we show that Theorem 2.2.3 does *not* hold if substantially less independence is used in the mappings of item and bucket points to the circle. That is, if we try and

economize too much on randomness, then there exists a set of views that will cause an item to have large spread, or a bucket to have large load; no matter which function from the family we choose! Note however, that for applications where the full power of theorem 2.2.3 is not required, less independence (and thus less randomness) may be perfectly adequate.

Assume that there are p items and p buckets, and that items and buckets are mapped into a circular array of length $\Theta(p)$ by degree k random polynomials. An item is mapped by sweeping clockwise around the array until a bucket point is encountered. If multiple bucket points are in the same cell of the array, then we use some arbitrary but fixed order on the buckets to break ties. Assume that there is an adversary who is trying to foil the hashing scheme by concocting a devious set of v views that somehow force the family to "malfunction". Of course, the adversary still has to play by the rules so he has to present views that contain at least p/t buckets. How well can the adversary do? We proved that if k is a constant times $t \log pv$, then the adversary can't make an item have spread more than roughly $\log v$ or a bucket have load more than roughly $\log v$ with high probability. The question is what happens for smaller values of k? We show that for smaller values of k, the adversary *can* do better, and we exhibit good strategies for the adversary. For clarity assume that $t = 3$.

Spread:

Say that the adversary is trying to cause an item i to have large spread. The item i can be mapped to one of p possible starting points (the point from which the clockwise sweep starts). Fix one possible starting point. Each of the p^k possible mapping of buckets to the array induces an ordering of the buckets with respect to the starting position of i: The order in which bucket points are encountered starting from i's point and sweeping clockwise around the circle. Fix a possible ordering. The adversary uses the set of $2p/3$ views that are missing all of the prefixes of length less than $2p/3$ of this ordering. Assuming that i is mapped to the chosen starting point, and that the buckets are mapped according to this fixed ordering, this set of views forces the spread of i to be $2p/3$. Doing the same for all p possible starting points for i and all p^k possible orderings of the buckets gives a set of $p \times p^k \times 2p/3 = \frac{2}{3}p^{2+k}$ views

that force the spread of i to be $2p/3$. However, recall that the adversary can only give us v views. The strategy is to evenly distribute these v views over all possible starting points and bucket orderings. This strategy guarantees that the spread of i will be roughly v/p^{1+k}. So, for a fixed k, the spread can be made to grow roughly linearly with the number of views v.

Note that setting $k = \log v$, as required in the statement of theorem 2.2.3, foils the above scheme!

Load:

The situation for load is similar to that of spread. Say that the omnipresent adversary is trying to make bucket b have large load. The adversaries strategy is for every possible starting point, and every ordering of the buckets, to use the view missing the prefix of the ordering so that the bucket b is the first encountered point. Using these $p \times p^k$ views, no matter what function we choose from the family, every item will be placed at least once into b. Note however, that if b appears in the last $p/3$ buckets in the ordering, then the resulting view has size less than $p/3$. These small views are not included, but they will only keep about a third of the items from being placed in b. Hence, using about p^{k+1} views the adversary can guarantee that the load on a bucket will be roughly p (up to a multiplicative constant). Again, since the adversary is restricted to only use v views, we distribute them evenly over all starting points and all bucket orderings. This gives load of roughly v/p^k on b, so for fixed k the load can be made to grow linearly with v.

As before, setting $k = \log v$ as required by theorem 2.2.3, foils the adversary's scheme.

2.2.6 A Different Adversarial Model

In this section we introduce into our already hostile world a new type of adversarial figure: the "bucket killer" adversary. The bucket killer adversary has the power, and need, to permanently remove up to d buckets from the total set of buckets $\mathcal{B}$. As a motivation for this new type of evil, consider an adversary that can crash up to d

servers in our simple caching scheme.

The goal of the bucket killer adversary is to overload some of the remaining buckets (i.e. servers). Thus, one strategy of the adversary could be to attempt to destroy some of the buckets so that one of the remaining buckets is responsible for much more that its fair share of the unit circle. If the adversary is successful in doing this, then the unlucky bucket targeted by the adversary gets more than its fair share of items. Note that we are assuming that the bucket killer adversary *is aware* of the random choice of hash function, Thus, the standard solution of putting up a smoke screen by adding randomness to the scheme is not effective gainst the bucket killer adversary.

In this section, we show that with a trick, we can protect ourselves against such an adversary - even if he/she is aware of our random choices. The trick is very simple: just use more points per bucket in the standard scheme. In fact, to tolerate a bucket killer adversary that has the power to crash d buckets, we only need to multiply the number of points-per-bucket by a factor of about d.

More precisely, we prove the following theorem:

Theorem 1 *Let B be a set of buckets, and let $\{V_1, \ldots, V_r\}$ be the set of views obtained from B by deleting at most d buckets. Then, if we use m' points per bucket where $m' = \Theta(dm)$, then with probability greater than $1 - 1/N$, for every $V \in \{V_1, \ldots, V_r\}$, and $b \in V$, length$(b) = O\left(\frac{1}{|V|}\left(\frac{\log(N|B|)}{m} + 1\right)\right)$*

Note that if we choose $m = \Omega(\log(|B|)$, and $N = poly(|B|)$, we get that the bucket killer adversary cannot harm the balance property of the hash family.

In other words, using the trick, no matter what buckets the bucket killer adversary decides to remove, the remaining buckets still divide the responsibility for the circle roughly evenly, with high probability.

Proof:

The proof is as simple as the trick itself. We invoke 2.2.8, replacing the confidence factor N with $Nc|B|^d$ (c is an arbitrary constant), to show that for any particular view in the set $\{V_1, \ldots, V_r\}$, the probability that any bucket is responsible for more than the stated fraction of the circle is less than $\frac{1}{Nc|B|^d}$. Now, since clearly $r = O(|B|^d)$,

we obtain the result from a union bound.

■

Theorem 1 shows that if we want to tolerate a bucket killing adversary that can delete d buckets, we should use a factor of d more points than before.

2.2.7 Theory of Consistent Hashing

In this section we introduce a new type of ranged hash family we call a π-hash families. This new class of functions may seem very restricted at first, however, we prove that in fact, every monotone ranged hash function is actually a π-hash function, and visa versa. This correspondence allows us to study the entire class of monotone ranged hash families by studying π-hash families in particular.

Definition 1 *A* π-hash function *is a ranged 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 an arbitrary list* $\pi(i)$ *of all the buckets* B*. (We will call such a list a* permutation*.) Define* $f_V(i)$ *to be the first bucket in the permutation* $\pi(i)$ *which is contained in the view* V*.*

Theorem 2.2.17 *Every monotone ranged hash function is a* π*-hash function and vice versa.*

Proof: First, we show that every π-hash function f is a ranged, monotone hash function.

By the definition of a π-hash function, items are always assigned to usable bins; that is, $f_V(\mathcal{I}) \subseteq V$ for every view V. Therefore, f is a valid ranged hash family. Now suppose all the items were initially assigned to a subset of the buckets. If some new buckets are now added, then an item i is reassigned only if a new bucket appears earlier than the item's current bucket in $\pi(i)$. Therefore, an item may move from an old bucket to a new bucket, but not from one old bucket to another. This implies that f has the monotone property.

Next we show that every monotone ranged hash function is a π-hash function. Let g denote a ranged hash function with the monotone property. We must show that

g associates each item i with a permutation $\pi(i)$. Furthermore, we must show that $g_V(i)$ is the first bucket in the permutation $\pi(i)$ which is in the view V.

We associate item i with the permutation $\pi(i)$ constructed as follows. Let the first bucket be $b_1 = g_B(i)$, let the second bucket be $b_2 = g_{B-\{b_1\}}(i)$, and generally let the $k+1$ bucket be $b_{k+1} = g_{B-\{b_1 \ldots b_k\}}(i)$. Now let V_1 be an arbitrary view, and let b_k be the first bucket in $\pi(i)$ which is in the view V_1. Consider the view $V_2 = B - \{b_1 \ldots b_{k-1}\}$. Since $V_1 \subseteq V_2$ and $g_{V_2}(i) = b_k \in V_1$, monotonicity implies $g_{V_1}(i) = b_k$ as desired. ■

The equivalence stated in Theorem 2.2.17 allows us to reason about monotonic ranged hash functions in terms of permutations associated with items.

One property of ranged hash families that is not captured well by the notion of balance is what we call uniformity. A hash family is uniform when items are assigned to buckets with absolute uniform probabilities. That is, when the probability that an item i is assigned to a bucket $b \in V$ is *exactly* $1||V|$.

Uniformity: A ranged hash family is *uniform* if, given any particular view V an item i, and a bucket $b \in V$, the probability that item i is mapped to bucket b in view V is *exactally* $1/|V|$.

Given theorem 2.2.17, one of the most natural monotone ranged hash functions is obtained by choosing the permutations independently and uniformly at random for every item. We denote the resulting family by function by $\mathcal{F}_\pi$. One nice property of this family is that in addition to having good spread and load characteristics, it is also uniform.

Theorem 2.2.18 *Let $\mathcal{V} = \{V_1, V_2, \ldots, V_k\}$ be a set of views of the set of buckets B such that: $\left|\bigcup_{j=1}^{k} V_j\right| = T$ and for all $1 \leq j \leq k$, $|V_j| \geq T/t$. Let $N > 1$ be a confidence factor. Then the following propoerties hold:*

- *Monotonicity: The family $\mathcal{F}_\pi$ is monotone (regardless of $\mathcal{V}$ and N).*
- *Spread: For any item $i \in I$, $spread_f(\mathcal{V}, i) = O(t \log(Nk))$ with probability greater than $1 - 1/N$ over the choice of $f \in \mathcal{F}_\pi$.*

- *Load: For any bucket* $b \in B$, $load_f(\mathcal{V}, b) = O\left(\left(\frac{|I|t}{T} + 1\right)\log(N|I|k)\right)$ *with probability greater than* $1 - 1/N$ *over the choice of* $f \in \mathcal{F}_\pi$.

- *Uniformity: The family* $\mathcal{F}_\pi$ *is uniform.*

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

We 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 s buckets in $\pi(i)$ then in every view item i will be assigned to one of the first s buckets in $\pi(i)$. This implies that item i is assigned to at most s distinct buckets over all the views.

We have to show that with high probability every view contains one of the first s buckets in $\pi(i)$. We do this by showing that for $s = t\log(Nk)$, the complement has low probability; that is, the probability that some view contains none of the first s 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 s buckets is at most $(1 - 1/t)^s = (1 - 1/t)^{(t\log(Nk))} < 1/(N)$. By the union bound, the probability that even one of the k views contains none of the first s buckets is at most $1/N$.

We now turn to load. By similar reasoning, every item i in every view is assigned to one of the first $t\log(NkI)$ buckets in $\pi(i)$ with probability at least $1 - 1/(2N)$. We show below that a fixed bucket b appears among the first $t\log(2NkI)$ buckets in $\pi(i)$ for at most $l = O\left(\left(\frac{|I|t}{T} + 1\right)\log(N|I|k)\right)$ items i with probability at least $1 - 1/(2N)$. By the union bound, both events occur with high probability. This implies that at most l 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 $t\log(2NkI)$ buckets in $\pi(i)$ is $It\log(2NkI)/C$. The result then follows from a use of the Chernoff bound similar to

that in lemma 2.2.7.

■

There are some problems with the family $\mathcal{F}_\pi$. For example, to choose a random function from the family requires about $|I||B|\log(|B|)$ random bits. Moreover, it is unclear how to efficiently compute the function.

In order to better understand how to construct efficient (small) families of π-hash functions we study the following model: Assume that there is some fixed set P of permutations of the buckets. Each function in the family is obtained by randomly selecting for each $i \in I$, one of the permutations in P (this selection may or may not be uniform).

For example, the family $\mathcal{F}_\pi$ is obtained by taking P to be the set of all permutations on the buckets. We ask the following questions:

- Does there exist a set P of permutations much smaller than $|B|!$ that gives a uniform ranged hash family?
- Say that we choose the set P to be a set of k-way independent permutations (see for example [12]). Does this give rise to a uniform family? Is the resulting hash family k-way independent for a fixed view?

Chapter 3

Random Trees

In this chapter, we introduce our second main algorithmic tool we call *random trees.* The random tree protocol is a replication mechanism that is designed to make copies of pages according to their relative popularity, and then to distribute requests between the copies. The objective is to eliminate hot spots, that is, prevent servers from becoming swamped by a huge number of requests for the same data. A key feature of the protocol is that copies are made *before* the data become so popular that servers could be swamped. In addition, the protocol works without any centralized control. Each cache runs the same simple, local protocol and the combined behavior of all the caches gives rise to the global replication and load balancing properties of the algorithm.

We begin the chapter in section 3.1 with an intuitive presentation of the main ideas behind the protocol. Next, in section 3.2 we discuss our theoretical model of the problem. This model is very simplistic, but allows us to *prove* that the random tree protocol works well, albeit in the simplified model. In section 3.3 we describe the random trees protocol formally, and in section 3.4 we prove that the protocol prevents swamping. In section 3.4.3 we prove that the storage requirements of the algorithm are not very large.

A crucial component to the random tree protocol is a hash function that is used to distribute caches over the nodes of a tree. Standard hashing could be used in this context, however, as is explained in the previous chapter, the resulting system would

not function well in a changing environment. Therefore, in section 3.5 we discuss how a consistent hash function can be used to implement the random tree protocol to produce a system that eliminates hot spots without requiring huge storage space and also tolerates incomplete and inconsistent information on the status of the network. The proof that this construction works is contained in [7].

In section 3.6 we discuss how to add to the model of the problem a way of representing network topology we call a network ultrametric. We then show how to modify the basic random trees protocol to take the network ultrametric into account.

3.1 Random Trees

The simple caching algorithm described in section 2.1 does a good job of distributing the load of serving a group of pages that are all accessed at the same low frequency between a network of caches. The scheme, however, does not perform well for swamping caused by a few very popular pages that are stored on a single server. We call this type of load *concentrated load.* An example of concentrated load is the web site of a software company when a popular software upgrade is made available. Simply reassigning the very popular page to another server does not solve the problem since the new server will become overloaded just like the old one. In order to reduce the load *without* causing high load on other servers, the hot page needs to be replicated more than once in the network. Balancing requests for the hot page over these multiple replicas can relieve load on the original server without overloading any other server. The more popular a page is, the more copies need to be made in the network. However, it is difficult to predict how popular a page is going to be, so the replication mechanism must adapt to changing popularity of pages. In the following sections, we introduce a new algorithmic tool called *Random Trees* that can be used in a caching algorithm to deal with concentrated load.

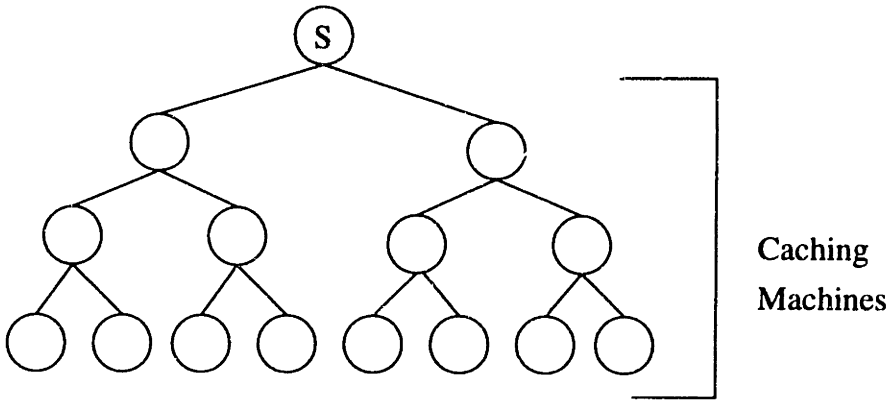


Figure 3-1: The server S is "protected" from swamping by a hierarchy of caching machines arranged in a tree with S as the root.

3.1.1 Trees

Say that p is a hot page that is stored on a server S. The basic scheme is to "protect" S from being swamped by putting a hierarchy of caches between S and the clients requesting the page p. The hierarchy is organized as a tree with S as the root (See figure 3-1). When a client wants to retrieve the page p, it chooses a *random* leaf of the tree of caches and directs its request to that cache. The request is then propagated up the tree until a copy of the page p is encountered. If none of the caches along the way have a copy of p, the request my have to travel all the way up to the server S. However, if one of the caches along the way does have a copy, then the request stops there and does not continue along up the tree to the server S. Once a copy of the page has been found, the page is returned to the client by going down the tree along the same path it came up. While the page is being returned to the client, all the caches along the path up the tree also store a copy of p to be used to answer subsequent requests.

This simple protocol effectively balances the load of requests for the hot page p between the caches in the tree. Figure 3-2 illustrates how the protocol works. The server S is the root of the tree, and initially is the only location where p is stored (a dark circle denotes places where a copy of p is stored). When the first request for p is made, a random leaf of the tree is selected by the client, and the request propagates

from the leaf all the way to the server S at the root since no caches contain a copy of p (See figure 3-2 (i)). After the first request has been serviced, all the caches along the path from the leaf to the root receive a copy of p (figure 3-2 (ii)). The next request starts again at a random leaf of the tree. Again, the request goes all the way up the tree to the server S, and subsequently, p is copied all the way down the path (figure 3-2 (iii)). At this point, the server S has received two requests for p. The important feature of the system is that from this point on, S will not receive *any more requests* for p as long as the caches at the second level of the tree do not evict p from their storage space. After a few more requests for p are made, all the caches on the third level of the tree have a copy of p (see figure 3-2 (iv)), and so on.

The more requests there are for p, the lower in the tree this "cutoff" level is. Since lower levels of the tree are wider (i.e. they contain more caches), requests for p are distributed among more caches as more requests for p are made. Randomly selecting a leaf insures that requests are distributed relatively evenly between the caches that are on the cutoff level. To summarize: the algorithm adapts automatically to changing popularity of pages. As a page is requested more often, it is spread into more caches, and the requests are distributed evenly between these caches.

3.1.2 Complications

The tree scheme works well to balance the load caused by a single page p between the caches. However, if there are requests being made for many *different* pages, the scheme can swamp some caches.

Suppose that there are 500 servers each with a single page. Say that each server has the same tree of caches protecting it from being swamped with requests for its page. Now, if the caches are all empty, and at the same time a request for each page is issued, then the caches on the second level of the tree receive 500 requests! One for each different page! Since there are far more that 500 pages on the web, this scheme would swamp the caches on high levels of the tree. Of course one solution to this problem is to install more powerful machines at these top levels so that they can tolerate the higher load. However, this solution lacks scalability if for example

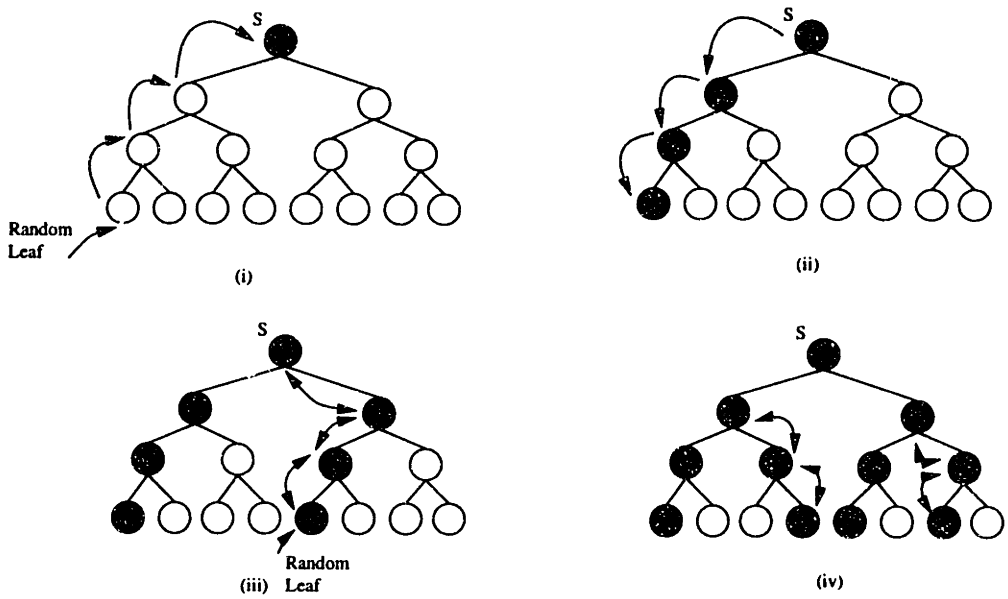


Figure 3-2: (i) Initially, only the server S has a copy of the page. Machines that have a copy of the page are shaded. The first request chooses a random leaf of the tree, and is propagated to the root, where it encounters the copy of the page. (ii) The page is then propagated down the tree, and the machines on the way save a copy. (iii) The second requests again starts at a random leaf. After the request has gone up and down the tree, more machines store copies of the page. Note that at this stage, the server S will not get any more requests for the page since the two caches below it already have copies. (iv) After a few more requests, more machines have cached copies of the page. At this stage, no machines in the top two levels of the tree will receive any more requests for the page; the cutoff level is already on the third level of the tree.

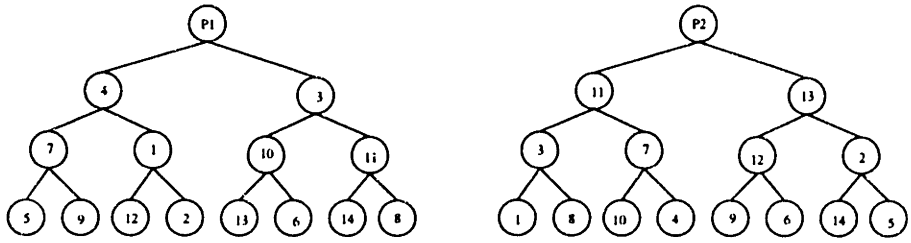


Figure 3-3: Two random placement of 14 caches in trees for two different pages $P1$ and $P2$ (cache names here are numbers). Since the tops of the trees are small, the probability that a specific cache is placed there is small. On the other hand, the bottoms of the trees are large. Hence most caches are placed towards the bottom of each tree.

the number of page requests grows more quickly than the power of available caching machines; a scenario quite likely to occur.

3.1.3 A Solution - Random Trees

There is a simple modification to the tree scheme that solves the complication described above. The basic idea is to use a *different* arrangement of the caches in the tree for each different page. To gather intuition, assume that for each page we select a *random* arrangement of caches in the tree. Figure 3-3 shows two arrangements of 14 caches for two different pages.

The problem with having a single arrangement of caches was that the caches at the top of the tree get at least one request for every page. But, there are only a few caches at the top of the tree. Therefore, if we arrange the caches randomly, a given cache will most likely be placed low down in the tree (close to a leaf). So for most pages a cache is found low in the tree, and only for a few pages the cache is located in the top of the tree.

To understand how the new scheme prevents swamping, we give an intuitive argument that a given cache c cannot be swamped in the extreme cases where swamping could happen. First, if there are many requests for different pages for which c is in the top of the trees, then c would get a request for each such page. However, we argued earlier that c will be found in the top of only a small number of trees, so c cannot be

swamped by such requests. The other extreme is that c is swamped by requests for pages for which c is close to the bottom of the tree. If this is true, then there must be a huge number of requests for such pages, since on the bottom levels of the tree there are alot of caches, and requests are spread roughly equally between them by the random choice of leaf in the protocol. So, if the number of total requests is not outrageously high, c cannot be swamped by such requests (if we make the request rate high enough, *any* scheme will cause swamping).

In section 3.4 we *prove* that no matter how requests are distributed between pages no cache will be swamped in the random trees protocol with high probability.

In the above discussion we have assumed that caches are arranged randomly in the trees. However, it is clearly infeasible to distribute to all clients a randomly chosen tree for *every* page on the Internet. In practice, we suggest to use a hash function to assign caches to nodes in the tree.

As was discussed in chapter 2, standard hash functions are not suited for use in caching applications on the Internet. Thus, in section 3.5 we show how to use consistent hashing to distribute caches "randomly" in the trees. Even if clients have different views of the set of caches, they will construct trees that are almost the same for a given page. This "consistency" allows the random trees protocol to be used in an environment such as the Internet.

The details of the random tree protocol are given in the following few sections.

3.2 Our Model

In this section we describe the simplified model of the caching problem that we use to design and analyze our algorithms. The objective is to define a model that both captures some important details of the real model and also allows tractable analysis or protocols. There is a tradeoff between details and tractable analysis and our model falls clearly onto the tractable analysis side of the playing field. We hope, however, that our model is sufficient to make the analysis relevant to a real world implementation of the protocol.

3.2.1 Components

We classify computers on the Web into three categories. All requests for Web pages are initiated by *browsers*. The permanent homes of Web pages are *servers*. *Caches* are extra machines that we use to protect servers from the barrage of browser requests. Throughout the remainder of the thesis, 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 assume that the content of each page is unchanging. The set of all pages is denoted by $\mathcal{P}$.

3.2.2 Types of Communication

There are two types of messages:

1. **Requests for pages:** A browser or cache can send a request for a page to a cache or a server.

2. **Page responses:** Responses containing the data of a page can be sent by caches or servers to browsers or caches. The machine sending the response must have a copy of the page. At the start of the protocol, only the home server has a copy of any page. All other copies need to be made from the home server by requesting the page from the server, and receiving a page response.

3.2.3 Objective

We work with a static model of requests. That is defined as follows. At some time T, there are a batch of requests that are issued by the browsers. We assume that the number of requests is at most $R = \rho C$ where C is the number of caches.

These requests are answered by some protocol that the browsers, caches and servers run together. Say that at time $T + t$ all of the browsers that made requests

receive the data that they requested. At this time we count the total number of requests that every cache and server have received.

The objective of the protocol is to answer all of the queries that the browsers initiate while minimizing the total number of requests that any one cache or server receives.

We assume that 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. The idea is then to show that the protocol, which will be randomized, prevents any machine from getting too many requests with high probability.

While the basic requirement is to prevent swamping, we also have two additional objectives. The first is to minimize cache memory requirements. That is, 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.

3.3 The Basic Random Trees Protocol

We associate with each page a rooted d-ary tree, called an *abstract tree* that will represent the tree of caches for that page. We use the term *nodes* only in reference to the nodes of these abstract trees. The number of nodes in each tree is equal to the number of caches, and the tree is as balanced as possible (so all levels except possibly the bottom level are full). We number the nodes of the tree by their rank in breadth-first search order.

A request consists of 4 things packaged together:

- The identity of the requester
- The name of the desired page (i.e. the URl of the page)
- The numbers of the abstract nodes on a leaf-root path in the abstract tree

- The sequence of caches associated with the above leaf-root path for the particular page.

All of the above are easy to generate except for the last. In order to decide which cache is responsible for a given abstract node n in the tree of page p, we use a hash function of the form $h : \mathcal{P} \times [1 \ldots C] \rightarrow \mathcal{C}$, and compute $h(p, n)$. The hash function h needs to be distributed to all browsers and caches. The root of a tree is alway mapped to the server for the page, that is we require that $h(p, 1)$ be the server for the page p.

The actions of the three components of the system are as follows:

Browsers: When a browser requests a page, it picks a random leaf to root path in the abstract tree, maps the nodes on the path to caching machines with the hash function h and sends the request to the leaf node on the path. The request includes the identity of the browser, the name of the page, the path nd the result of the mapping from abstract nodes to actual caches.

Cache: When a cache receives a request, it first checks to see if it is caching a copy of the page or is already waiting for a copy of the page. If so, it returns the page to the requester (after it gets its copy if necessary). Otherwise, the cache increments a local counter associated with the page and the abstract node the cache is acting as, and forwards the request to the next machine on the path for the page. If the counter reaches a threshold of q (where $q \geq 1$ is set ahead of time), then it caches a copy of the page when its request is answered. In any case, when the cache's request is answered, it forwards the data on to the requester.

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

The parameter q controls how much storage each cache must have in order to make the protocol work correctly. There is, however, a tradeoff between storage requirements and preventing swamping. The larger q is, the less storage each cache needs to have. However, if q is large then caches and servers can recieve more requests in the protocol. This is quantified in the following analysis of the protocol.

3.4 Analysis of the Random Tree Protocol

In this section we give a basic analysis of the performance of the protocol in our simplified model of the problem. That is, there are a set of ρC requests that are made in a batch, and we measure the total number of requests that a server or cache receives until all requests are answered. Many more details can be found in [7].

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 assumptions that no machine is swamped, and that every machine can communicate with every other machine with equal ease. We then show that the protocol prevents, with high probability, any machine from being swamped. We then show that no cache needs to store too many pages for the protocol to work properly.

In section 3.6 we explain how to take into account network topology and communication times. However, until then we ignore this issue. More details of various modification in the protocol are described in [7].

3.4.1 Latency

Since we are assuming that every cache can communicate with every other cache with equal latency (different communication latencies are addressed in section 3.6), the latency that any browser experiences when retrieving a page is determined by the height of the tree. A request forwarded from a leaf node experiences latency no more $2 \log_d cC$ times the communication time between two machines. Note that if a request stops at a cache that is waiting for a copy of the page, then another request for the page has already been sent up the tree by that cache so the total latency is no more than the stated amount.

The parameter d could be made large so that the tree is short. However, as the analysis for swamping will show, there is a tradeoff between having a shallow tree and preventing swamping.

One semi-practical point worth mentioning is that the time required to obtain a large page need not be multiplied by the number of steps in the path if the caches

pipeline the data going through them. In other words, the hops through additional caches increase the latency, but do not necessarily decrease the throughput of the system.

The following lemma shows that if you want to prevent swamping from occurring, then you must introduce some additional latency into the system.

Lemma 1 *Any protocol that can handle ρC requests for a page simultaneously, with no machine serving more than d requests, must have an average latency of at least $\Omega(\log_d(\rho C))$.*

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

3.4.2 Swamping

We now analyze the number of requests a machine gets in our protocol in the simplified static model. First, note that a server can receive at most d requests for any given page. We assume that the server can handle this load for each of its pages.[1] What remains is the analysis of the number of requests received by caches.

The intuition behind the analysis is the following. Note that there are two "phases" of the randomness in the protocol. There is the random choice of leaf node made by the browser, and then there is the random placement of caches in abstract tree nodes that is done at the beginning by the hash function. Because of this two stage process, we break the analysis into two stages. First we analyze the

[1] If a server cannot cope with this load because it is the permanent site of a very large number of documents, then we can modify the protocol in the following way. The root of the abstract tree is mapped to a cache in the standard way and there is a second root above the actual root of the tree. The permanent server takes the place of this new root node.

number of requests for each node in the abstract tree for each page. This involves the randomness of the browser choosing the leaf node to start a request at. Then, the number of requests for each abstract node is translated into a "weight" for each abstract node. We then analyze how the caches are mapped into the abstract tree nodes, this involves th randomness introduced by the hash function. We count the total weight that is associated with each cache, and then show that each cache is likely to actually receive a number of requests that is close to its associated weight.

We analyze the case that the hash function is a completely random function. In [7] we show that the hash function can actually have limited independence.

Theorem 3.4.1 *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*

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

requests

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

Corollary 3.4.2 *If h is chosen uniformly and at random from the space of functions $\mathcal{P} \times [1 \ldots C] \mapsto \mathcal{C}$ then with probability at least $1 - 1/N$, where N is a parameter,* no *no cache gets more than*

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

requests

Proof: The bound given in theorem 3.4.1 holds for a given cache with probability at least $1 - 1/N$. Since there are C caches, the probability that the bound holds for all caches is $1 - C/N$. Since N is simply a parameter, we can replace N by NC to get the corollary. ■

We prove theorem 3.4.1 in the rest of the section. We split the analysis into two parts. First we analyze the requests to a cache due to its presence in the leaf nodes of the abstract trees and then analyze the requests due to its presence at the internal nodes and then add them up.

Requests to Leaf Nodes

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

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

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

Let α denote the high probability bound on W that we just proved. Now we have:

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

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

Requests to Internal Nodes

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

Now no abstract internal node gets more than dq requests because each child node

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

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

The following lemma gives us a bound on n_j.

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

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

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

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

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

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

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

requests with probability at least $1 - O(\frac{\log dq}{N})$. Replacing N by $N \log(dq)$ in the above expression and simplifying we get Theorem 3.4.1.

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

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

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

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

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

In [7] we show that the bound on the number of requests each cache receives also holds if the hash function is chosen from a $\log N$-way independent family. This makes choosing and distributing the function much easier.

3.4.3 Storage

In this section, we discuss the amount of storage that each cache must have in order to make the protocol work. Each cache needs to have space to store all the pages for which it receives more than q requests. Each cache needs to maintain a counter for each page that it receives a request for, however the storage required to maintain a counter is small compared to the space required to actually store a copy of a whole page. Thus, we ignore the space used by the counters and concentrate on the number of actual pages that each cache has to store.

The following lemma bounds the total number of cached pages in the system and in each cache.

Lemma 4 *The total number of cached pgespages over all machines is* $O(\log N + \frac{R}{q})$ *with probability at least* $1 - 1/N$. *A given cache has* $O(\frac{reqrate}{q} + \log N)$ *cached pages with probabiliiy at least* $1 - \frac{1}{N}$.

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

The rest of this section is devoted to proving this lemma.

Proof:

We show that the total number of cached pages, over all abstract nodes is $O(\ln N + \frac{R}{q})$ with probability at leat $1-1/N$. From a standard balls-in-bins argument it follows that with probability at least $1 - 1/N$, the number of cached pages at a machine is $O(\frac{R}{qC} + \log N) = O\frac{reqrate}{q} + \log N)$.

We begin by studying the distribution of weights (storage counts) at the nodes of a particular abstract tree. Consider the abstract tree T_p for a given page p. Suppose

that there are r_p requests for page p. For an abstract node at level l, the *expected* number of requests that the node receives isr_p/d^l. This quantity droips by a factor of d at every level. Thus, there is a certien level at which the expectation is at most q/e and at least $q/(ed)$. e call this the *threshold level* of the given page. We bound the number of cached copies in two parts: the number above the threshold level and the number below the threshold level.

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

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

Our remaining task to is bound the number of cache copies at and below the threshold level. To do this, we use a generating function argument for each level separately. We begin with the threshold level. We look at the requests coming up the tree at the threshold level as throwing $r = r_p$ balls (requests) into $b = d^l$ bins (abstract nodes). A bin "matters" (caches a copy of the page) if it receives at least q requests. Now, the probability that j bins receive at least q balls is at most the probability that some set of j bins receives a total of qj balls, which is at most:

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

We used the fact that at the threshold level, $r/b \leq q/e$.

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

$$\begin{aligned}
\prod_p e^{\frac{Re}{q}x} &= e^{\sum_r \frac{re}{q}x} \\
&= e^{\frac{Re}{q}x}
\end{aligned}$$

Next, we consider the level below the threshold. This level has d times as many nodes, so the expected number of requests per machine drops by a factor of d. What impact does this have on the above analysis? We use the expectation in only one place - where we replaced r/b by q/e. At one level below the threshold, we can replace r/b by $q/(ed)$. We then continue with the same analysis as before and get a bound of $e^{\frac{Re}{qd}x}$ on the probability generating function provided that $q \geq 2$. Similarly, at two levels

below the threshold, the probability generating function is bounded by $e^{\frac{Re}{qd^2}x}$, and so on. So, the probability generating function for the total number of abstract nodes over all pages below threshold levels that receive more than q requests is bounded by:

$$\begin{aligned}\prod_{i\geq 0} e^{\frac{Re}{d^i q}x} &= e^{\sum_{i\geq 0}\frac{Re}{d^i q}x} \\ &\leq e^{\frac{Red}{(d-1)q}x}\end{aligned}$$

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

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

This quantity is $1/N$ when:

$$j = O(\ln N + \frac{R}{q})$$

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

■

3.5 Using Consistent Hashing

So far, we have described the random tree protocol in a situation where every machine knows about the existence of every other machine and that machines never malfunction. This is clearly not the case on the Internet. The component of the protocol that is sensitive to having changing and incomplete information about the set of caches is the hash function h.

Clearly, it is important that two browsers agree on how caches are distributed in a tree for a given page. However if we use standard hashing techniques and two users have different views of the caches, the trees that they build for the same page are completely different. In order to overcome this difficulty, we can use the consistent

hash functions that we develop in the previous chapter. We assume that each browser knows about at least a $1/t$ fraction of the caches, however this set can be chosen by an adversary. There is no difference in the protocol, except that the mapping h is a consistent hash function.

This change will not affect latency. Therefore, we only analyze the effects on swamping and storage. The basic properties of consistent hashing are crucial in showing that the protocol still works well. In particular, the blowup in the number of requests and storage is proportional to the spread and load of the hash function.

A complete analysis of the resulting protocol is presented in [7].

3.6 Ultrametric

The assumption that every pair of machines can communicate with equal ease is obviously unrealistic, and we show how to 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. For example, a browser sends a page request to a cache, the cache obtaines the page by forwarding the request to another cache or server, and then the cache returns the page to the user. The latency of the page request is defined to be the duration of all three stages. The duration of the first and third stages is a function of the ease of communication between m_1 and m_2.

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

We assume that such typical communication times are available. In particular, if

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

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

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

The ultrametric is a natural model of internet distances, since it essentially captures the hierarchical nature of the internet topology, under which, for example, all machines in a given university are equidistant, but all of them are farther away from another university, and 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 the points. The distance between two points depends only on which is the smallest cluster containing both. Thus, for example, the distance between any two machines at the same university is less than the distance between any two machines at different universities in the same country.

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

3.6.1 Protocol

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

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

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 allows us to restrict our analysis to $\log(C)$ clusters. Thus, summing over these clusters we have only a $\log(C)$ blowup in the bound. Details are given in [7].

Chapter 4

Conclusion

We have introduced two algorithmic tools for caching in distributed networks; consistent hashing and random trees. Consistent hashing has been developed and analyzed in a general setting because we believe that there are many other applications that can benefit from the construction. Random trees were introduced and motivated and some basic analysis was presented. More detailed analysis appear in [7].

There are both practical and theoretical questions that are left open. We have presented a basic architecture of a caching scheme for the web. On the practical side, this leaves open the possibility of designing real systems based on our algorithms. In particular, it remains to see how well the random tree protocol behaves in a setting where requests are not sent in a batch, but are a continuous stream. In addition, it remains to see how well the ultrametric model fits what is really happening on the Internet, and whether the modified tree protocol works well in practice.

On the theoretical side, there are a number of questions that remain open. Can one prove lower bounds or better upper bounds on the number of permutations that need to be used to construct uniform monotone hash function? Are there other efficient implementations of consistent hash functions other than the circle hash function. Are there other models of network topology that admit tractable analysis of the random tree protocol?

Bibliography

[1] *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 4–6 May 1997.

[2] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, April 1979.

[3] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael Schwartz, and Kurt Worrell. A hierarchical internet object cache. *USENIX Proceedings*, July 1996.

[4] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. In *Annals of Mathematical Statistics (23)*, pages 493–509, 1952.

[5] Sally Floyd, Van Jacobson, Steen McCanne, Ching-Gung Liu, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM 95*, 1995.

[6] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[7] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In ACM [1], pages 654–663.

[8] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.

[9] Ulana Legedza and John Guttag. Using network level support to improve cache routing. In *3rd International WWW Caching Workshop*, 1998.

[10] Ari Luitonen and Kevin Altis. World-wide web proxies. *Computer Networks and ISDN systems, First International Conference on the World-Wide Web*, July 1994.

[11] Radhika Malpani, Jacob Lorch, and David Berger. Making world wide web caching servers cooperate. *Proceeings of the World Wide Web Conference*, July 1996.

[12] Moni Naor and Omer Reingold. On the construction of pseudo-random permutations: Luby-Rackoff revisited (extended abstract). In ACM [1], pages 189–199.

[13] Noam Nisan. Psuedorandom generators for space-bounded computation. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing*, pages 204–212, Baltimore, Maryland, 14–16 May 1990.

[14] Katia Obraczka, Peter Danzig, Solos Arthachinda, and Muhammad Yousuf. Scalable, highly availiable web caching. In *NLANR Web cache workshop*, 1997.

[15] C. Greg Plaxton and Rajmohan Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *37th Annual Symposium on Foundations of Computer Science*, pages 570–579, Burlington, Vermont, 14–16 October 1996. IEEE.

[16] Dean Povey and John Harrison. A distributed internet cache. In *Proceedings of the 20th Australian Computer Science Conference*, 1997.

[17] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–340, Austin, Texas, 25–27 January 1993.

[18] R. Sprugnoli. Perfect hashing functions: A single probe retrieving method for static sets. *Communications of the ACM*, 21(11):606–611, December 1979.

[19] R.E. Tarjan and A. Yao. Storing a sparse table. *Communications of the ACM*, 22:606–611', July 19879.

[20] Zheng Wang and Jon Crowcroft. Cachemesh: A distributed cache system for the world wide web. In *NLANR Web cache workshop*, 1997.

[21] Mark N. Wegman and J. Lawrence Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265–279, June 1981.

[22] Andrew Chi-Chih Yao. Should tables be sorted? *Journal of the ACM*, 28(3):615–628, July 1981.

[23] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive web caching. In *NLANR Web cache workshop*, 1997.

Chapter 5

Appendix A

5.0.2 Chernoff Bounds

Chernoff bounds are bounds on large deviations that are very useful in many situations involving sums of indicator variables. Most of the results in this appendix can be found, or immediately derived from, the seminal paper of H. Chernoff [4].

The basic result is:

Theorem 5.0.1 *Let $X_1, X_2, \ldots, X_n$ be independent Bernoulli variables such that for $1 \leq i \leq n$, $\Pr[X_i = 1] = p_i$ where $0 < p_i < 1$ then for $X = \sum_{i=1}^{n} X_i$, $\mu = E[X] = \sum_{i=1}^{n} p_i$ we have:*

1. *For $\delta > 0$, $\Pr[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{\mu}$*
2. *For $0 < \delta \leq 1$, $\Pr[X < (1-\delta)\mu] < e^{-\frac{\mu\delta^2}{2}}$*

Following are a number of simplifications of the first case that come in handy.

Corollary 5.0.2 *With the same conditions as theorem 5.0.1:*

1. *For $0 < \delta \leq 2e-1$, $\Pr[X > (1+\delta)\mu] \leq e^{-\frac{\mu\delta^2}{4}}$*
2. *For $\delta > 2e-1$, $\Pr[X > (1+\delta)\mu] < 2^{-(1+\delta)\mu} \leq e^{-\frac{(1+\delta)\mu}{3}} \leq e^{-\frac{(1+\delta)\mu}{4}}$*

In addition we have:

Corollary 5.0.3 *With the same conditions as theorem 5.0.1:*

$$\Pr[|X - \mu| > \mu] \leq 2e^{-\frac{\mu}{4}}$$