15-853: Algorithms in the Real World
Consistent Hashing

Based on the lecture by Bruce Maggs on November 14 2000 and the papers in the bibliography. Most of the figures are taken from D. Lewin's Masters Thesis [198].

November 14, 200

Contents

1 What is caching and why do we need it

Caching is a general strategy that has been employed to improve the efficiency and reliability of data delivery over the Internet. The basic idea is to replicate information from content providers at special servers called *caches*. The following example shows how caches work and why they are useful. Suppose a news site is set up in San Francisco and multiple users in Boston access this site regularly. For each access to the site, a request message and the requested information have to be routed across the country over the Internet backbone (see Figure 1(i)).

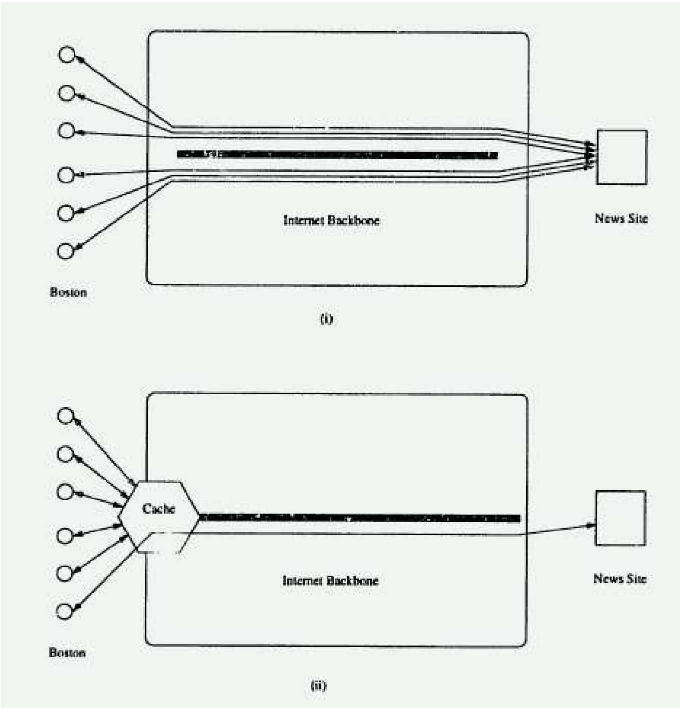

Figure 1: *(i) Multiple users in Boston access a news site across the country. For every request made by a user in Boston the information is being transmitted across the backbone. (ii) When a cache is installed in Boston, the first user retrieves the data from across the country, but all the other users in Boston get the information locally from the cache. The cache prevents redundant traffic from crossing the Internet backbone and reduces user latency.*

If the same information is requested multiple times it will have to be transmitted multiple times across the backbone. Now suppose a cache is installed in Boston. The first time a document is requested by a user in Boston it is sent across the country, but then it is kept in the cache so that later requests for this document can be served locally (see Figure 1(ii)). Installing a cache helps alleviate three problems impeding the performance on the WWW:

- Network Congestion
  As we saw in the example, replicating information in a cache prevents redundant traffic from crossing the Internet backbone. This reduces network congestion and the problems related to it (packets might be delayed or even dropped by routers if the network is congested).
- Swamped servers
  The load at the content provider is reduced. This is important for popular sites which otherwise might be swamped with requests (the "hot-spot problem").
- Distance
  Finally, the user latency is reduced since the information is closer to the user being served.

2 Different approaches to caching

There are two extreme approaches to implementing caching. In the first one we have a *monolithic caching architecture* where a single big cache is placed in a city to serve all of the users (see Figure 2). This is the approach that was used in the example in Section 1. This architecture has the disadvantage that there is a single point of failure and that the cache may be congested since all users in the city use it. The second architecture overcomes these problems by using a *distributed approach*. Each neighborhood has its own cache that serves residents in that neighborhood (see Figure 3).

This system circumvents the two problems above: it will survive a failure and it spreads the work across several machines. Furthermore, it has the advantage that the caches are closer to the user and can therefore deliver content faster. However, it also has some downsides. The biggest one is that the hit rate will be lower. The first reason for that is that they receive requests from a smaller population. Another reason is that the individual caches are smaller than the big cache in the monolithic approach and therefore hold only a smaller subset of the information from the content provider.

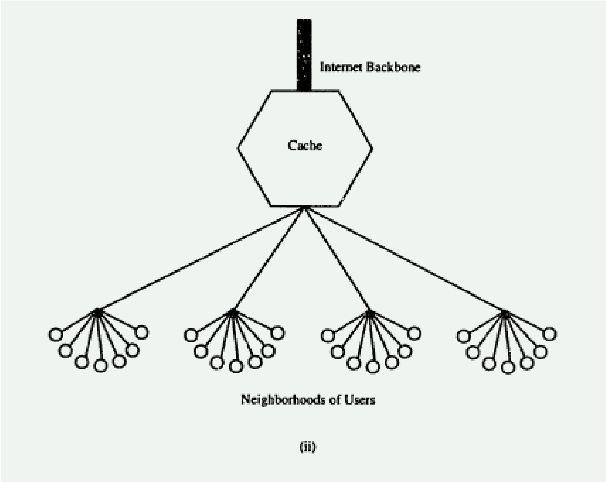

Figure 2: *A monolithic caching architecture. A single cache is placed in the city to server all of the users. Since the cache receives requests from a large population, the hit rate is likely to be high. However, it has the disadvantage that the cache must be a very large and fault-tolerant machine.*

Daniel Lewin and others developed in [K99],[L98] and [K97] a hybrid approach that strives to overcome the disadvantages of the above approaches above while keeping the advantages. The design objectives they had in mind are the following:

- Large distributed system
  The system should be built from a large number of small and cheap caches that are distributed throughout the network. This way there is a cache near every user and the total load is distributed over several machines.

- No centralized control
  The behavior of a cache should depend only on information available locally, or obtained in a non-centralized fashion without any central control that could be a critical point of failure.

- Robust under different views
  Caches can be added and removed from the network at any time and there is no central control that keeps track of the status of caching machines. Thus, different users may have different *views* of the set of caches.

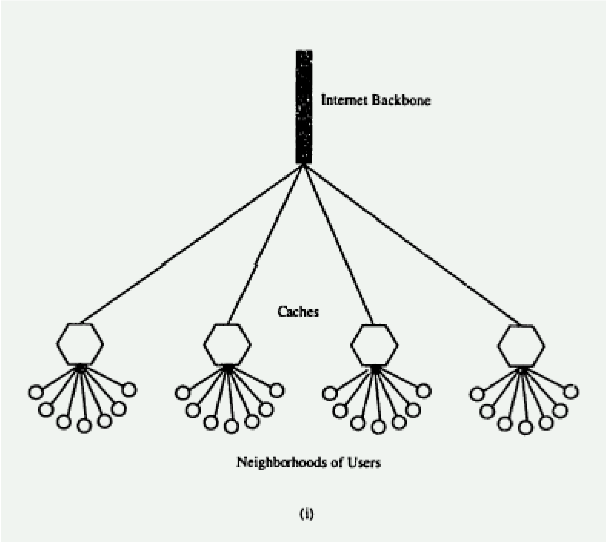

Figure 3: *A distributed caching architecture. Each neighborhood has a cache that serves residents in that neighborhood. Note that the caches are very close to the users so any content that is in fact located in the cache is retrieved very efficiently. However, since any one cache only receives requests from a small set of users, the hit rate is likely to be small.*

- Scale gracefully
  The Web grows every day, and so must the caching scheme if it is to keep up with increasing use. Therefore the system should be designed to scale gracefully as the network grows.

- Prevent swamping of hot spots
  Caching machines and servers should never be swamped. Simply re-assigning responsibility for a hot page from a server to a cache will not work since then the cache might be swamped. It will be necessary to make copies of the hot page and distribute it to many caches. A problem is that it is impossible to predict the popularity of a page and once a page becomes hot the cache it is assigned to might be swamped and become unable to communicate.

- Minimize network usage
  The system should be designed so that the total traffic in the network is reduced as much as possible. That means that the caching system should be designed so that as many requests as possible are served

close to the requester.

- Balance storage requirements
  The storage capacity of a caching machine is limited. No cache should be required to store a disproportionate fraction of the cached pages.

- Low overhead
  The caching scheme must be simple enough so that it won't increase user latency significantly.

Before going into their solution let's look at two ways of combining the two extreme approaches that have been described above.

One idea (called harvest caching) is to have a hierarchical system with both distributed neighborhood caches and a monolithic city cache (see Figure 4). A request is first sent to the closest neighborhood cache and if a miss occurs, the request is forwarded to the city cache. This solution has some of the advantages of the two extreme approaches (good locality and high hit rate) but it also has the main drawback of the monolithic system: the big cache has to be a large and fault tolerant machine, which is expensive or even infeasible to build and maintain.

The second solution tries to achieve the advantages of both extreme approaches including the good cache hit rate of one large cache without the actual need for a big fault tolerant machine. The idea is to have a network of distributed caches that *cooperate*. In such a system every client selects one primary cache that he sends his requests to. If the primary cache misses it tries to locate the document in one of the other caches (by multicasting the request to them) instead of going directly to the content provider. This technique has all the advantages of the previous ones but it introduces a new problem: as the number of participating caches grows the number of messages between messages can become unmanageable.

So the question is how to make a group of caches function together like one big cache without having the inter-cache communication overhead of the above solution.

3 The use of hashing in caching

The idea of Lewin and others is to store each object only at one (or a few) machine(s) and have the user's browser directly contact the one cache that should contain the required object. The browsers make their decision with help of a hash function that maps URL's to the set of caches. Recall that

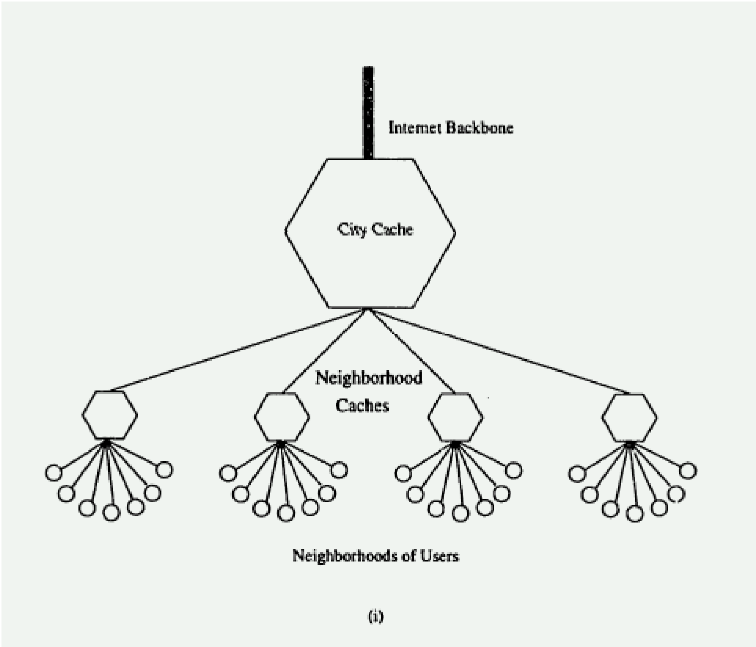

Figure 4: *A hierarchy of caches with small neighborhood caches at the bottom of the hierarchy and a large monolithic city cache at the top. Requests are first sent to the local neighborhood cache and on a miss they are forwarded to the city cache. The neighborhood caches bring content closer to the 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 or even infeasible to build and maintain.*

in classical hashing a hash function is a mapping f of a set of items I to a set of buckets B:

$$f : I \rightarrow B$$

where the goal is to spread the items evenly over the buckets. Typically, you don't have one fixed hash function but you choose a hash function randomly from a family of hash functions. This guarantees good *expected* performance. A commonly used family of hash functions is that of the linear congruential hash functions. This family consists of all functions

$$f(x) = ax + b \mod p$$

where, p is prime, $0, 1, \ldots, p-1$ is the set of buckets and a and b are in $0, 1, \ldots, p-1$. Figure 5 illustrates the use of linear congruential hash functions.

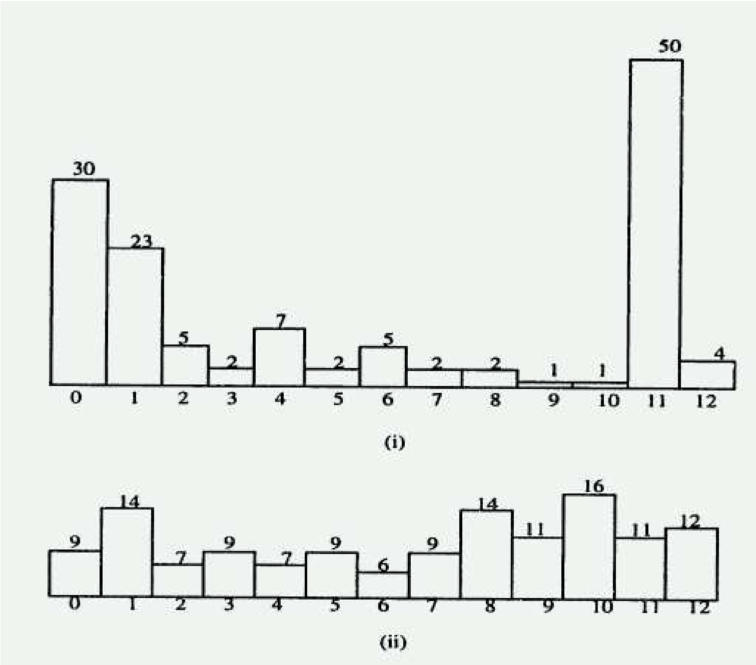

Figure 5: *This figure illustrates how hash functions distribute documents between servers. Assume that document names are integers, and that there are 13 servers* $1, \ldots, 13$. *Documents are hashed to servers using a common type of hash function which is* $f(d) = ad + b \mod 13$ *for some fixed integers* a *and* b. *(i) shows 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) shows the distribution of documents to servers by the hash function. No server is responsible for a disproportionately large share of documents.*

The question is whether these hash functions can be directly applied to the caching problem by simply associating the URL's with items and the caches with buckets and making sure that every browser knows the hash function.

It turns out that there are couple of problems with this idea because of the dynamic nature of the Internet. While traditional hashing theory assumes that the number of buckets is constant, in the Internet it happens all the time that caches go down or that new caches are added. We could fix this problem by choosing a new random hash function every time the number of caches and therefore the range of the hash function changes. This solution, however, has two major drawbacks:

1. All the users must be notified when the hash function is changed, or all of their queries will go to the wrong cache.

2. Furthermore, most items will be mapped to a different bucket under the new function which means in terms of caching that most objects will have to be shuffled to another cache. Figure 6 shows such a situation.

So what we really want is a class of hash functions that are still random (and therefore spread the items evenly over the buckets) but that don't change much when the range changes. This way only few objects have to be shuffled to another cache. It also takes care of the other problem: we can now allow different users to have different views of the system (i.e., information about which caches are up or down) and to use different hashing functions; since the hash functions don't change too much each object should be mapped to only a small number of different machines under the different views. This means we don't have to inform all users if caches go down or come up.

A hashing scheme that meets the above requirements, i.e., for most items the mapping doesn't change if the range of the hash functions changes is called *consistent hashing*. The next section will formalize the ideas from this section and describe the class of consistent hash functions that Lewin and others developed.

Please note at this point that while consistent hashing advances many of our goals it is not sufficient to solve the hot-spot problem. It is still possible that a cache that contains a very popular document becomes a hot-spot. Avoiding the hot-spot problem requires a popular page to be stored in more than one cache. [L98] describes how to do that using *random trees*.

4 Consistent hashing

Let's first summarize and formalize the properties we strive for in a consistent hash function:

- **Balance:** Items are distributed to buckets "randomly".

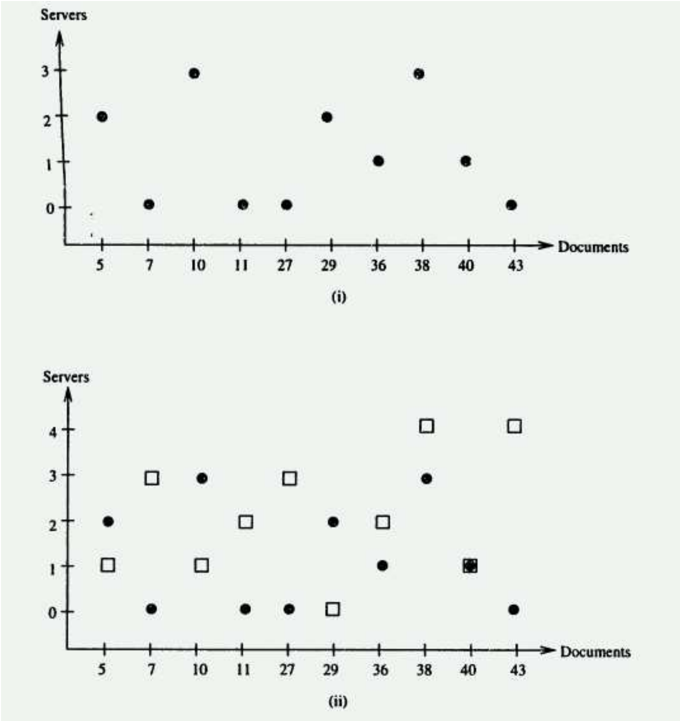

Figure 6: *The top figure shows the assignment of 10 documents to 4 servers using the hash function* $f(d) = d + 1 \mod 4$*. The bottom part of the figure shows the new assignment after one additional server is added and the hash function changed to* $f(d) = d + 1 \mod 5$*. Squares show the new mapping and circles show the mapping of the previous function. Note that almost every document is mapped to a different server as a results of the addition of the new server.*

- **Monotonicity:** When a bucket is added, 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. 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, the spread should be small.

A *consistent hash family* is one that has all these properties. Before we describe the consistent hash family given by Lewin and others we need a few definitions:

Definition: Let I be the set of items and B the set of buckets. A *view* $V \subset B$ is a subset of buckets. A *ranged hash function* is a function that maps (view, item) pairs to buckets:

$$f : 2^B \times I \to B$$

$f_V(i)$ gives the bucket item i is mapped to under view V.

The hash family given by Lewin is called UC_{random} which stands for *Unit Circle Random.* Let C be the circle of unit circumference. UC_{random} maps *both* items *and* buckets to points on the unit circle using two standard hash functions r_I and r_B:

$$r_I : I \to C$$

$$r_B : B \to C$$

Given r_I and r_B, $f_V(i)$ is defined to be the first bucket in V that we come to when traversing the circle clockwise from $r_I(i)$. The following two examples illustrate this concept.

Example 1: Figure 7 gives an example for a hash function from the UC_{random} family with 6 buckets and 8 items. Both documents and servers are mapped to points on a circle using standard hash functions. A document 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. When a new server is added the only documents that are reassigned are 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. The squares in the lower part of the figure show the new mapping and circles are the previous mapping. As you can see that fewer items move than under the standard hash function.

Example 2: Figure 8 gives another example of a hash function from the UC_{random} family. Note the unlucky placement of bucket points around the unit circle. Bucket A is responsible for a disproportionately large section of the unit circle. Since items 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!

To avoid situations like in the second example we add another little tweak: instead of mapping each bucket to one point on the unit circle we map it to several points. Formally, we use a function $r_B : B \times [m] \to C$ to map m copies of each bucket to the circle. This makes a poor distribution of buckets on the circle, where most of the items map to the same bucket, less likely. An example for two buckets and $m = 4$ is shown in Figure 9.

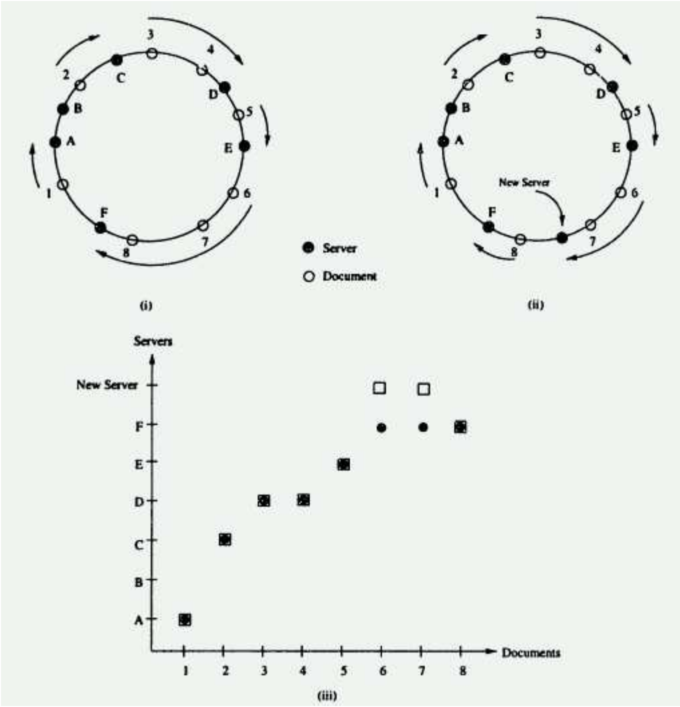

Figure 7: *An example for a hash function from the UC_{random} family with 6 buckets and 8 items. When a bucket is added only two documents are mapped to a different server than before.*

It is quite easy to show the monotonicity of UC_{random}.

Theorem 1: The family of hash functions UC_{random} is monotone.

Proof: We have to show that if buckets are removed the mapping changes only for items that were originally in one of the removed buckets. Similarly, we have to show that if we add buckets then all the items whose mapping changes are now mapped to the new bucket. Let $V_1 \subset V_2 \subset B$ 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 that $f_{V_1}(i) = f_{V_2}(i)$. From $f_{V_2}(i) \in V_1$ follows that none of the buckets in $V_2 \setminus V_1$ lies between i and $f_{V_2}(i)$ on the unit circle. Therefore, adding or removing buckets in $V_2 \setminus V_1$ doesn't change the mapping of i. Figure 10 illustrates this proof.

With respect to load, spread and balance the following can be proven:

Theorem 2: Let $V = V_1, V_2, \ldots, V_k$ be a set of views of the set of buckets B such that: $|\bigcup_{j=1}^{k} V_j| = T$ 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

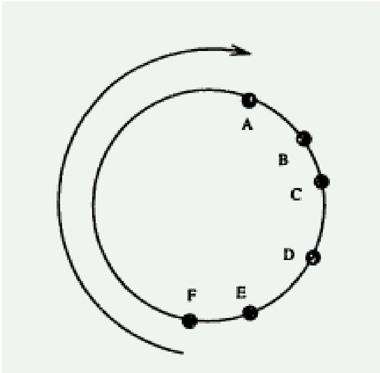

Figure 8: *Another example of a hash function from the UC_{random} family. Note that this function has a very poor distribution of buckets and most items will be mapped to server A.*

times then:

- Spread: For any item $i \in I$, $spread_f(V, i) = O(t \log(Nk))$ with probability greater than $1 - 1/N$ over the choice of $f \in UC_{random}$.
- Load: For any bucket $b \in B$, $load_f(V, b) = O\left(\left(\frac{|I|}{T} + 1\right) t \log(Nmk)\right)$ 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 $O\left(\frac{1}{|V|}\left(\frac{\log(N|V|)}{m} + 1\right)\right) + \frac{1}{N}$.

Proof: The proof of this theorem is beyond the scope of this document. Please see [L98] for a full proof.

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.

5 Consistent hashing in practice

In practice, there would be two problems if we tried to implement consistent hashing in exactly the way we described it above. The first one is that storing a hash function from our family of consistent hash functions would require infinite space, since we are using real numbers. Second, choosing a function from the family would require an infinite number of random bits.

Fortunately, it is possible to change the basic scheme to remedy these problems. It turns out that it is sufficient to have *limited* independence in the mapping of points to the circle and also to use *limited* precision in the real numbers.

5.1 Using limited independence

A family of functions is *k-way independent* if any k elements from the domain are mapped independently into the range, i.e., let $x_1, x_2, \ldots, x_k$ be elements of the domain and $y_1, \ldots, y_k$ be elements of the range. Then

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

It can be proven that it is sufficient to use a k-way independent mapping of the buckets and items:

Lemma 1: Theorem 1 and Theorem 2 still hold if the bucket and item points are each mapped $\Omega(t \log(NTk))$-way independently, where t, N, and T are the same as in Theorem 2.

5.2 Using limited precision numbers

Each function in the family UC_{random} is defined by the mapping of $|I|+m|B|$ random points on the real unit circle. The important observation is that it is not the exact position of these points but only their clockwise ordering around the circle matters. Therefore, we need only enough precision bits so that the ordering on a set of $|I| + m|B|$ random points is with high probability completely defined if we use only this number of bits in the representation of the points. Luckily, it turns out that this is already the case for $O(\log |I| + m|B|)$ bits of precision. More precisely:

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

And even better, this is still the case if the points are k-way independently distributed for $k \geq 2$, instead of completely randomly distributed.

Putting the above results together gives us the following theorem.

Theorem 3: If the mapping of items and buckets are $\Omega(t \log(NTk))$-way independent (N is a confidence factor), items are mapped independently of bucket points and $O(\log(N(m|B| + |I|)))$ bits of precision are used then Theorem 1 and Theorem 2 hold with probability at least $1 - 1/N$.

That means we can map items and buckets

- $\Omega(t \log(NTk))$-way independently instead of completely randomly

- and only to those points on the unit circle that can be represented by no more than $O(\log(N(m|B| + |I|)))$ bits of precision

and with high probability we will still have all the nice properties of the original UC_{random} function.

5.3 An implementation of consistent hashing

The authors of [K98] founded a company called Akamai that actually employs consistent hashing. This company maintains caches all over the world and offers content providers to cache data for them. Akamai caches mainly pictures and other embedded files. The reasons are that these are typically the big files (while the frame document itself is usually small) which make up most of the load at the content provider and are also mainly responsible for the high latency in retrieving a web page. Another reason for caching only pictures is that the content provider is still able to monitor the traffic at their site.

Akamai uses distributed caches that employ consistent hashing. The interesting question is where the hashing happens. It turns out that Akamai uses a nameserver hack to allow you to get the files from the nearest cache. Figure 11 illustrates how this works.

The hashing is done in two steps:

1. The content provider hashes the URL to a serial number, e.g. 212 in the example. If this document is requested the local nameserver first asks the Akamai high-level nameserver for the IP-address of the low-level nameserver.

2. The low-level name server evaluates the consistent hash function for the current view at the given serial number. It then returns the IP-addresses of the caches that the document is mapped to under the consistent hash function.

Akamai provides the content provider with a program that does the first step, i.e., the mapping of URLs to serial numbers. In general, it tries to map all the URLs in an HTML-document to the same serial number so as to minimize the number of DNS lookups. However, if a certain limit on the total number of object bytes on a serial number is reached it will start using a new serial number. The goal is to minimize DNS lookups without making it difficult to perform load balancing.

Suppose a content provider hashes a document in step 1 to “a1.g.akamai.net”. Below we use some useful tools to show how names resolution works for this "akamaized" document and to track the way that a request for this document takes.

dig (domain information groper) sends domain name query packets to name servers and can be used to gather information from the Domain Name System servers. In Figure 12 we ran `dig a1.g.akamai.net` to find out which akamai server our document is mapped to. The output tells us that "a1.g.akamai.net" resolves to the IP-addresses 206.245.157.79 and 206.245.157.71 and that the query time was 1 msec.

The second tool is *traceroute*. It prints the route packets take to a network host. Figure 13 shows the results of a `traceroute a1.g.akamai.net`. We see that it takes a packet 10 hops until it finally reaches the akamai server with IP-address 206.245.157.79 (recall from above that this is one of the two IP-addresses that a1.g.akamai.net is mapped to). Furthermore, it takes a packet on average around 16 ms to get there.

6 Bibliography

[K99] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, Y. Yerushalmi. "Web Caching with Consistent Hashing." *Proceedings of the 8th International WWW Conference*, May 1999.

[L98] D. Lewin "Consistent hashing and random trees : algorithms for caching in distributed networks." *MIT Master Thesis*, May 1998.

[K97] D. Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy, "Consistent Hashing and Random Trees : Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web." *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, May 1997.

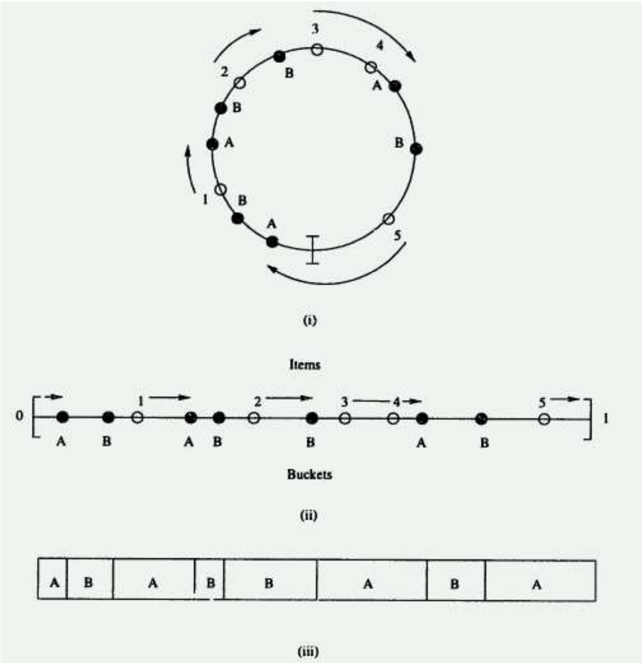

Figure 9: *(i) A unit circle hash function with* $m = 4$*. Buckets A and B have 4 points associated with each of them. Items are mapped to the buckets closest to them going clockwise. Item 1 is closest to a point of bucket A and item 2 is closest to a point of bucket B. Item 5 is closest clockwise to 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.*

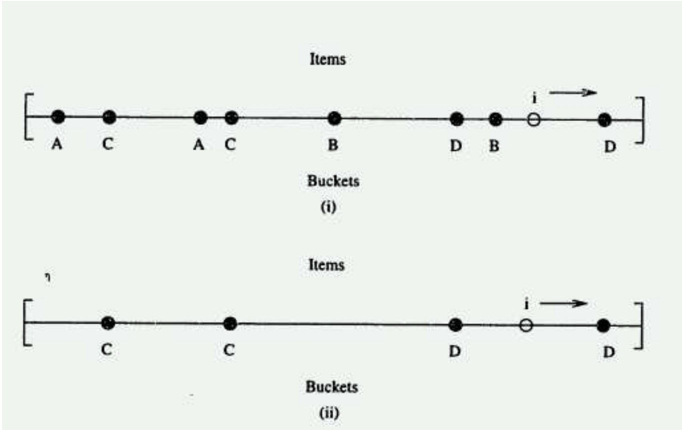

Figure 10: *Monotonicity for the family UC_{random}. In this figure the unit circle is depicted by an interval of length one, which is obtained by cutting the unit circle at an arbitrary point. (i) The mapping of points to the circle for a view $V_2 = A, B, C, D$ (m=2 in this example). The closest bucket point clockwise of i's point is one associated with the bucket D. (ii) For any view $V_1 \in V_2$ containing the bucket D (here $V_1 = C, D$), the point closest to i's point will still be D. .*

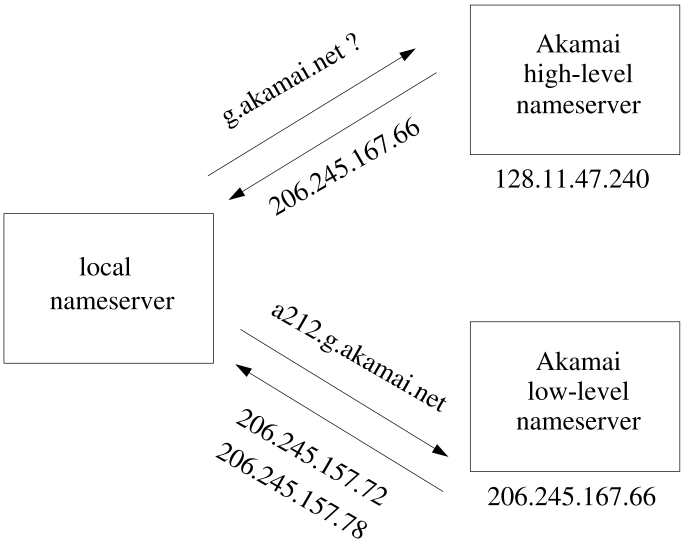

Figure 11: *The figure shows how a request for a212.g.akamai.net is resolved.*

```
; <<>> DiG 8.2 <<>> a1.g.akamai.net
;; res options: init recurs defnam dnsrch
;; got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 4
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 13, ADDITIONAL: 13
;; QUERY SECTION:
;;      a1.g.akamai.net, type = A, class = IN

;; ANSWER SECTION:
a1.g.akamai.net.        7S IN A         206.245.157.79
a1.g.akamai.net.        7S IN A         206.245.157.71

;; Total query time: 1 msec
;; FROM: gs116.sp.cs.cmu.edu to SERVER: default -- 127.0.0.1
;; WHEN: Thu Nov 30 20:46:01 2000
;; MSG SIZE  sent: 33  rcvd: 484
```

Figure 12: Using dig

```
 1  GIGROUTER.NET.CS.CMU.EDU (128.2.254.36)  0.462 ms  0.353 ms  0.350 ms
 2  RTRBONE-FA4-0-0.GW.CMU.NET (128.2.0.2)  0.713 ms  0.664 ms  0.562 ms
 3  killifish.psc.net (198.32.224.11)  1.388 ms  0.992 ms  1.582 ms
 4  Serial0-1-0.GW2.PIT1.ALTER.NET (157.130.19.241)  1.432 ms  2.164 ms  1.730 ms
 5  554.at-2-1-0.XR1.DCA1.ALTER.NET (152.63.40.94)  6.427 ms  6.419 ms  7.960 ms
 6  195.ATM6-0.GW4.PHL1.ALTER.NET (152.63.37.17)  10.451 ms  9.835 ms  10.468 ms
 7  fastnetoc-gw.customer.alter.net (157.130.251.174)  10.484 ms  10.804 ms  9.631 ms
 8  pos4-0-0-abepa.fast.net (206.245.159.113)  14.573 ms  15.731 ms  13.715 ms
 9  cust01-abe.fast.net (209.92.0.9)  15.190 ms  15.269 ms  16.962 ms
10  a206-245-157-79.deploy.akamaitechnologies.com (206.245.157.79)  16.912 ms  15.424 ms  16.164 ms
```

Figure 13: Using Tracroute