COS 521: Advanced Algorithm Design **Hashing / Content Distribution**
02/09/12 **Scribe: Shilpa Nadimpalli** **Professor: Sanjeev Arora**

1 Introduction to Internet Content Distribution

1.1 The "Hot-Spot" Problem

[1] Certain web sites (e.g. New York Times) are accessed thousands of times per day by unique users from multiple locations, which causes severe network congestion around such "hot-spots" on the web. A simple solution would be to simply copy the hot-spot site and cache it at multiple other locations around the web, and redirect users to these new URLs. However, web sites have dynamic content. We need a scheme to efficiently keep track of and update all site copies dynamically.

1.2 Characteristics of an Ideal Content Delivery Service:

- has no centralized control
- is robust to errors – caches can be added and removed from the network at any time
- reduces traffic on Internet backbone (repetitive long-distance data retrieval)
- prevents swamping of "hot spots" (overloaded servers)
- updates automatically (dynamic)
- is transparent to browsers (Different browsers have different views on which caches are accessible at a given time.)

2 Possible Solutions Using Hashing

- Let $\boldsymbol{URL}$ denote the set of all possible webpage URLs
- Let $\boldsymbol{P} \subseteq \boldsymbol{URL}$ denote the set of all webpages served around the world at any given point in time. Notice that P is a huge set, which changes dynamically.
- Let $\boldsymbol{C}$ denote a set of caches (run by a company such as Akamai) that will store and serve this content.

Idea 1: Use a single hashing function to map each URL to a specific cache.

$$h : URL \to C \tag{1}$$

Problem with Idea 1: This doesn't solve our congestion problem; New York Times site is always mapped to the same cache. This is useful for a task like web hosting.

Idea 2: Cache each URL k times using different hash functions. Redirect URL request to one of the k copies.

$$h_1, h_2, h_3, ..., h_k : URL \to C \tag{2}$$

[1] Modified from old scribe notes by Alina Ene in 2005.

Problem with Idea 2: Adding or removing caches changes the range of the hash functions, forcing us to remap *all* data if one cache goes down.

Idea 3: Use **Consistent Hashing**, an algorithm implemented by the Akamai Company. Let `circle` denote the unit circle in $\Re^2$. Caches and URLs are hashed to `circle` using two hash functions.

$$h_A : C \rightarrow \texttt{circle} \tag{3}$$

$$h_B : URL \rightarrow \texttt{circle} \tag{4}$$

Each URL is stored in the cache that is hashed closest to it in the clockwise direction on `circle`. This way, if one cache crashes, only $\frac{1}{\#\ caches}$ of the URLs need to be remapped (assuming caches are roughly uniformly distributed around circle).

The domain nameserver computes the two hash values and asks the cache for the page. If the requested page is not in the cache, the cache is responsible for looking up the page and serving it. This is a first cut of the consistent hashing idea; we extend it below.

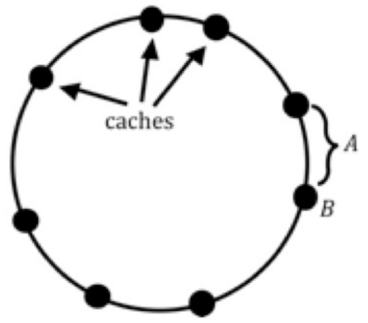

Figure 1: Caches and URLs are hashed to the unit circle, `circle`. For all webpages $w \in URL$, if $h_B(w)$ is in the region of `circle` marked by A, w is stored in cache B.

3 Analysis of Above Idea

Let us assume that h_A, h_B are random hash functions. A summary and analysis of random hash functions can be found in Section 6.

Claim 3.1 The number of URLs to fall in any interval on `circle` between two caches is proportional to the length of that interval.

Argument: We can assume this is true. Assuming the hash $h_B : URL \rightarrow \texttt{circle}$ is random, a complete proof is possible using the Strong Law of Large Numbers or Chernoff Bounds.

Claim 3.2 The load on each cache is approximately well-balanced. Specifically, the portion of `circle` assigned to each cache is $\leq O\left(\frac{lg(C)}{C}\right)$ with high probability, where C is the number of caches.

Proof:

- Define some bad event, B, where a single cache controls $> \frac{8 \cdot lg(C)}{C}$ of `circle`.
- Consider the arc of this length. However we split this arc into two pieces, one of those pieces must have length $> \frac{4 \cdot lg(C)}{C}$. Define a little interval L to be found on this larger piece and to have length $\frac{2 \cdot lg(C)}{C}$. If B occurs, then $\exists L$.

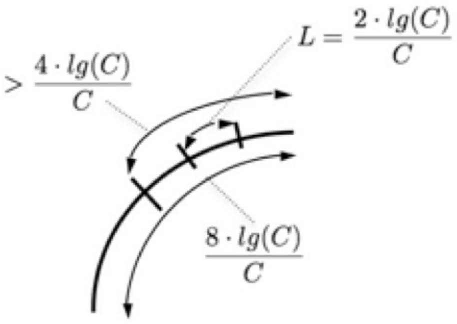
$> \frac{4 \cdot lg(C)}{C}$
$L = \frac{2 \cdot lg(C)}{C}$
$\frac{8 \cdot lg(C)}{C}$

Figure 2: The arc on `circle` in the bad event that a single cache controls a proportion of `circle` $> \frac{8 \cdot lg(C)}{C}$.

- Pr[no cache lands in L] $= \left(1 - \frac{2 \cdot lg(C)}{C}\right)^C$. This value has C in the exponent because we are using *random mapping.*
- Recall that $\left(1 + \frac{x}{n}\right)^n \approx e^x$ if x does not increase as quickly as n. Using this, we know Pr[no cache lands in L] $\approx e^{-2 \cdot lg(C)}$, which is $< \frac{1}{c^2}$.
- Therefore, $\Pr[B] \leq \frac{\{L\}}{c^2} \ll \frac{1}{c^2} \ll 1$ as $C \to \infty$.

□

Thus the hashing scheme has the following property:

DEFINITION 1 ***Monotonicity:*** *If we add or remove a cache, the only web page URLs that need to migrate are those that are assigned to the new cache, or were assigned to the deleted cache. No other web pages are reassigned, and so changes in the number of caches causes minimal disruption.*

Problem with Consistent Hashing: However, this method *still* does not solve the congestion problem; The New York Times website is still mapped to only one cache.

4 Random Abstract Trees

Each URL needs to be stored in multiple caches to prevent swamping of hot spots. Additionally, the lookup for a cache given a URL must be robust, because caches may go down at times. An extension of the original Consistent Hashing Algorithm, described below, addresses these issues.

- Represent each URL by $\dfrac{d^{k+1}-1}{d-1} \approx d^k$ pseudo-URLs for some small d, k. These pseudo-URLs refer to the nodes of a tree with height k and branching degree d.
- Each tree node is assigned to a cache using a consistent hash function. The cache at the root node of the tree keeps a copy of the page pointed to by the original URL.
- Upon looking up a URL, the user is directed to one of the leaves of the tree randomly. If the cache referred to at that leaf does not contain a copy of the page, the request is redirected up the path until a node is reached that refers to a cache with a copy of the page.
- There is a threshold q such that if some tree node is accessed at least q times in the above process, then the corresponding cache acquires the URL content and caches it.

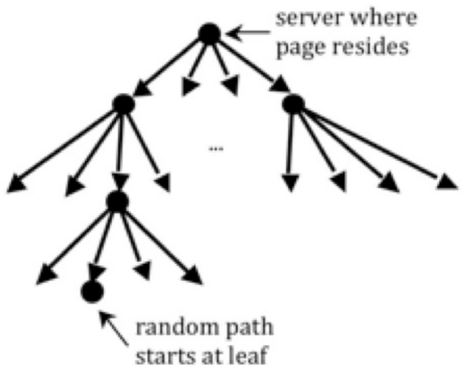

Figure 3: Assignment of a page URL to a tree, before any copies of the page have been stored at caches referred to by internal and leaf nodes. Each node of the tree is assigned to a cache using a consistent hash function.

Notice that if a page is popular, copies of the page will quickly migrate from the root cache to the caches at all the internal nodes and leaves. We can see that the latency of this algorithm is $O(\log_d(C))$, and hot spots are avoided. Furthermore, if a new hot spot develops, then copies of the root quickly percolate to the caches corresponding to the tree nodes. All this can be mathematically analyzed, but we won't do it.

Claim 4.1 The consistent hashing algorithm with URL storage in trees is robust.

Proof: It is important to assume that the adversary controlling caches does not see the random coins used by the algorithm. An adversary can make ρ fraction of the caches go down (different for different users). Let P_d be the event that a path dies. Let L_d be the event that a leaf goes down.

- $\Pr[P_d] = \Pr[L_d]$
- $\Pr[\neg L_d] = (1-\rho)^{\log_d(C)}$

□

5 Details of Akamai's Real-World Implementation

Akamai caches mainly pictures and embedded files, because (1) these files are usually big files and they are responsible for the high latency in retrieving a web page, (2), by caching only pictures, the content provider is still able to monitor the traffic at their site. The company uses a nameserver hack that allows file retrieval from the nearest cache.

Akamai uses distributed caches that employ consistent hashing. The hashing is done in two steps. First, the content provider hashes the URLs to a serial number; when the document is requested, the local nameserver first asks the Akamai high-level nameserver for the IP address of the low-level nameserver. Second, the low-level nameserver evaluates the consistent hash function for the current view at the given serial number and 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 maps URLs to serial numbers.

6 Analysis of Random Hash Functions

Claim 6.1 If n elements are mapped to n bins with a random hash function, the maximum bin size $= O\left(\frac{lg(n)}{lg(lg(n))}\right)$.

Proof:

- Fix bin. Let B_k be the event that the bin has k balls.
- $\Pr[B_k] = \binom{n}{k} \cdot \frac{1}{n^k}$
- Recall by **Sterling's Approximation** that $\binom{n}{k} \approx \left(\frac{ne}{k}\right)^k$, and so $\Pr[B_k] \approx \left(\frac{e}{k}\right)^k$.
- Let $k = O\left(\frac{lg(n)}{lg(lg(n))}\right)$. Recall that $\left(\frac{lg(n)}{lg(lg(n))}\right)^{c \cdot \frac{lg(n)}{lg(lg(n))}} = n^c$. Then $\Pr[B_k] < \frac{1}{n^2}$.

□

Ideas for more random hash functions:

- Any "naive" hash functions using randomness (key strokes, system time, composition of random functions). Unfortunately these don't always work.
- Low-degree polynomial instead of linear function
- Apply 1-way (cryptographic) function, followed by linear hash function. This probably works if the cryptographic function is indeed 1-way.