CPSC 536N: Randomized Algorithms 2014-15 Term 2

Lecture 6

Prof. Nick Harvey *University of British Columbia*

Today we discuss two other uses of randomized algorithms in computer networking and distributed systems.

1 Consistent Hashing

It is an approach for storing and retrieving data in a distributed system. There are several design goals, many of which are similar to the goals for peer-to-peer systems.

- There should be no centralized authority who decides where the data is stored. Indeed, no single node should know who all the other nodes in the system are, or even how many nodes there are.
- The system must efficiently support dynamic additional and removal of nodes from the system.
- Each node should store roughly the same amount of data.

We now describe (a simplification of) the **consistent hashing** method, which meets all of the design goals. It uses a clever twist on the traditional hash table data structure. Recall that with a traditional hash table, there is a universe U of "keys" and a collection B of "buckets". A function $f : U \to B$ is called a "hash function". The intention is that f nicely "scrambles" the set U. Perhaps f is pseudorandom in some informal sense, or perhaps f is actually chosen at random from some family of functions.

For our purposes, the key point is that traditional hash tables have a *fixed* collection of buckets. In our distributed system, the nodes are the buckets, and our goal is that the nodes should be dynamic. So we want a hashing scheme that can gracefully deal with a dynamically changing set of buckets.

The main idea can be explained in two sentences. The nodes are given random locations on the unit circle, and the data is hashed to the unit circle. Each data item is stored on the node whose location is closest. In more detail, let C be the unit circle. (In practice we can discretize it and let $C = \{\, i/2^K \,:\, i = 0, ..., 2^K - 1 \,\}$ for some some sufficiently large K.) Let B be our set of nodes. Every node $x \in B$ chooses its "location" to be some point $y \in C$, uniformly at random. We have a function $f : U \to C$ which maps data to the circle, in a pseudorandom way. But what we really want is to map the data to the nodes (the buckets), so we also need some method of mapping points in C to the nodes. To do this, we map each point $z \in C$ to the node whose location y is closest to z (i.e., $(y - z) \bmod 1$ is as small as possible).

The system's functionality is implemented as follows.

- **Initial setup.** Setting up the system is quite trivial. The nodes choose their locations randomly from C, then arrange themselves into a doubly-linked, circular linked list, sorted by their locations in C. (Network connections are formed to represent the links in the list.) Then the hash function $f : U \to C$ is chosen, and made known to all users and nodes.
- **Storing/retrieving data.** Suppose a user wishes to store or retrieve some data with a key k. She first applies the function $f(k)$, obtaining a point on the circle. Then she searches through the linked list of nodes to find the node b whose location is closest to $f(k)$. The data is stored or

retrieved from node b. (To search through the list of nodes, one could use naive exhaustive search, or perhaps smarter strategies. See Section 1.2.)

- **Adding a node.** Suppose a new node b is added to the system. He chooses his random location, then inserts himself into the sorted linked list of nodes at the appropriate location.

  And now something interesting happens. There might be some existing data k in the system for which the new node's location is now the closest to $f(k)$. That data is currently stored on some other node b', so it must now **migrate** to node b. Note that b' must necessarily be a neighbor of b in the linked list. So b can simply ask his two neighbors to send him all of their data which for which b's location is now the closest.

- **Removing a node.** To remove a node, we do the opposite of addition. Before b is removed from the system, it first contacts its two neighbors and sends them the data which they are now responsible for storing.

1.1 Analysis

By randomly mapping nodes and data to the unit circle, the consistent hashing scheme tries to ensure that no node stores a disproportionate fraction of the data.

Suppose there are n nodes. For any node b, the expected fraction of the circle for which it is responsible is clearly $1/n$. (In other words, the arc corresponding to points that would be stored on b has expected length $1/n$.)

Claim 1 *With probability at least $1-1/n$, every node is responsible for at most a $O(\log(n)/n)$ fraction of the circle. This is just a $O(\log n)$ factor larger than the expectation.*

Proof: Let $a = \lfloor \log_2 n \rfloor$. Define overlapping arcs $A_1, \ldots, A_{2^a}$ on the circle as follows:

$$A_i \;=\; [\; i \cdot 2^{-a},\; (i+3a)\cdot 2^{-a} \bmod 1 \;].$$

We will show that every such arc probably contains a node. That implies that the fraction of the circle for which any node is responsible is at most twice the length of an arc, i.e., $6a2^{-a} = \Theta(\log n/n)$.

Pick n points independently at random on the circle. Note that A_i occupies a $3a2^{-a}$ fraction of the unit circle. The probability that none of the n points lie in A_i is:

$$(1-3a2^{-a})^n \;\leq\; \exp(-3a2^{-a}n) \;\leq\; \exp(-3a) \;\leq\; n^{-2}.$$

By a union bound, the probability that there exists an A_i containing no node is at most $1/n$. $\square$

This claim doesn't tell the whole story. One would additionally like to say that, when storing multiple items of data in the system, each node is responsible for a fair fraction of that data. So one should argue that the hash function distributes the data sufficiently uniformly around the circle, and that the distribution of nodes and the distribution of data interact nicely.

We will not prove this. But, assume that it is true, each node stores nearly-equal fraction of the data. There is a nice consequence for data migration. When a node b is added (or removed) from the system, recall that the only data that migrates is the data that is newly (or no longer) stored on node b. So the system migrates a nearly-minimal amount of data each time a node is added or removed.

1.2 Is this system efficient?

To store/retrieve data with key k, we need to find the server closest to $f(k)$. This is done by a linear search through the list of nodes. That may be acceptable if the number of nodes is small. But, if one is happy to do a linear search of all nodes for each store/retrieve operation, which not simply store the data on the least-loaded node, and retrieve the data by exhaustive search over all nodes?

The original consistent hashing paper overcomes this problem by arguing that, roughly, if the nodes don't change too rapidly then all users' "opinions" about where the data is stored don't differ too much. So we can store each piece of data on just a few machines, and store/retrieve operations don't need to examine all nodes.

But there is another approach. We have just discussed the peer-to-peer system SkipNet, which forms an efficient routing structure between a system of distributed nodes. Each node can have an arbitrary identifier (e.g., its location on the unit circle), and $O(\log n)$ messages suffice to find the node whose identifier equals some value $f(k)$, or even the node whose identifier is closest to $f(k)$.

Thus, by combining the data distribution method of consistent hashing and the routing method of a peer-to-peer routing system, one obtains an highly efficient method for storing data in a distributed system. Such a storage system is called a **distributed hash table**. Actually our discussion is chronologically backwards: consistent hashing came first, then came the distributed hash tables such as Chord and Pastry. SkipNet is a variation on those ideas.

2 Leader Election

Suppose we have n nodes in a distributed system, of which k are good and the rest are malicious. We would like to somehow elect a leader that is good.

The model of communication is:

- Nodes communicate by broadcasting a message. All other nodes reliably receive that message, together with the identity of the sender.
- Each node has access to a random number generator.
- The malicious players secretly communicate with a computationally unbounded adversary that controls their actions.

Feige presented a very simple and elegant algorithm for the leader election problem.

Theorem 2 (Feige '99) *Suppose $k \geq (1+\alpha)n/2$. There is an algorithm that elects a good leader with success probability at least $\alpha^{O(\log(1/\alpha))}$.*

So, for example, if 10% of the nodes are malicious then we can take $\alpha = 0.8$, and elect a good leader with success probability $\alpha^{\Omega(\log(1/\alpha))}$, perhaps roughly 0.25.

To understand why this is impressive, it helps to see why a naive strategy doesn't work. If we know that only 10% of the node are malicious, then why not just pick $x \in \{1, \ldots, n\}$ uniformly at random and designate node x the leader? The success probability would be is 0.9, which is better than Feige's result!

The issue is: how can we randomly choose x? There is no centralized source of randomness. We can't designate a fixed node, say node 1, to randomly choose x for us, because node 1 might be malicious.

Perhaps we can designate a good node to choose x randomly for us, then use x as the leader? But this is circular logic: "designating a good node" is the same problem as "choosing a good leader". In fact, Feige proves that the problem of collectively flipping a random coin is *equivalent* to the problem of choosing a good leader.

Still, maybe there are some useful ideas here. One would think that initially there are many good nodes, so if we "combine" all their randomness, we should be able to make progress towards picking a random node. For example, maybe we could pick one bit of x, which amounts to saying that half of the nodes are still eligible to be x and the other half aren't. Hopefully half of the remaining nodes are still good. Then we can repeat this process until we have just one node left, which is hopefully good.

Feige's algorithm, shown in Algorithm 1, follows this strategy. But the way in which it selects half of the nodes is quite clever. It selects half of the nodes by **choosing the lightest bin**. This is a nice strategy because, if the malicious nodes collude and distribute themselves in an unbalanced way, then choosing the lightest bin should actually decrease the fraction of malicious nodes!

Algorithm 1: Feige's Lightest Bin algorithm. Bin i consists of the nodes who broadcast bit i.

```
Function LightestBin():
    Let $B \leftarrow \{1, \ldots, n\}$
    repeat
        Every player broadcasts a single bit. For the good players, this is a random bit.
        Let $B_i$ denote the players who broadcast $i$.
        if $|B_0| \leq |B|/2$ then
            $B \leftarrow B_0$
        else
            $B \leftarrow B_1$
    until $|B| \leq 1$;
    If $|B| = 1$, that node is the leader
```

Let's begin with a simple claim that illustrates the main ideas.

Claim 3 *Suppose k is a power of two and $k > n/2$, i.e., there is a strict majority of good nodes. Then* `LightestBin()` *elects a good leader with probability at least $k^{-O(\log k)}$.*

Suppose we flip $2c$ random coins. What is the probability that exactly c of them are heads? By Stirling's formula, it is

$$\binom{2c}{c} \cdot 2^{-2c} \;=\; \frac{(2c)!}{c!c!} \cdot 2^{-2c} \;\overset{c\to\infty}{=}\; \frac{\sqrt{2\pi 2c}(2c/e)^{2c}}{\left(\sqrt{2\pi c}(c/e)^c\right)^2} \cdot 2^{-2c} \;=\; \frac{1}{\sqrt{\pi c}}.$$

In fact $\binom{2c}{c}2^{-2c} \geq 1/\sqrt{4c}$ for all $c \geq 1$.

Proof: The probability that B_0 and B_1 each have exactly $k/2$ good nodes is at least $1/\sqrt{2k}$. In this case, regardless of how the malicious nodes divide themselves amongst the bins, the lighter bin will have at most $(n-k)/2$ malicious nodes. But $k > n/2$ implies $k/2 > (n-k)/2$. So the lightest bin still has a strict majority of good nodes and the number of good nodes is still a strict power of two. The argument can continue inductively for $\ell = \log_2 k$ rounds.

In the i^{th} round there are $k/2^i$ good nodes so the probability of an even split at least $\sqrt{2^i/4k}$. The rounds are independent, so the probability of a good split in *every* round is at least

$$\prod_{i=0}^{\ell-1} \sqrt{2^i/4k} \;=\; (1/4k)^{\ell/2} \prod_{i=0}^{\ell-1} 2^{i/2} \;=\; (1/4k)^{\ell/2} 2^{(1+\cdots+\ell-1)/2} \;=\; (1/4k)^{\ell/2} 2^{\ell(\ell-1)/4} \;\approx\; k^{-\ell/4}.$$

□

This might not seem too impressive. After all, $k^{-O(\log k)}$ is a pretty small success probability. But, if we make stronger guarantees about the number of good nodes, we can do better.

Proof (of Theorem 2). (Sketch)

Instead of insisting that the good players split with exactly half in each bin, let's just ask that they are fairly well balanced. Specifically, we want that each bin has at least $k/2 - k^{2/3}$ good players.

We can analyze the probability of this happening by a Chernoff bound. Let X_i be the indicator of the event that the i^{th} good node chooses bin 0. Let $X = \sum_{i=1}^{k} X_i$. Then $\mathrm{E}[X_i] = 1/2$ and $\mathrm{E}[X] = k/2$. Set $\delta = 2k^{-1/3}$. The probability that some bin has fewer than $k/2 - k^{2/3} = (1+\delta)k/2$ good players is

$$\Pr\left[|X - k/2| > k^{2/3}\right] \;\leq\; 2\exp(-\delta^2\,\mathrm{E}[X]/3) \;<\; 2\exp(-k^{1/3}/2) \;<\; 1/k,$$

as long as k is sufficiently large.

Let k_j be the number of good nodes at the end of the j^{th} round. Let $\mathcal{E}_j$ be the event that, in the j^{th} round, either B_0 or B_1 has less than $k_j/2 - k_j^{2/3}$ good players. If *none* of the events $\mathcal{E}_1, \ldots, \mathcal{E}_j$ occur then $k_j \geq k/2^j - O(k^{2/3})$.

By our previous Chernoff bound, $\Pr[\mathcal{E}_j \mid k_j] \leq \frac{1}{k_j}$, so

$$\Pr\left[\mathcal{E}_j \mid \overline{\mathcal{E}_1} \wedge \cdot \wedge \overline{\mathcal{E}_{j-1}}\right] \;\leq\; \frac{1}{k/2^j - O(k^{2/3})} \;\approx\; \frac{2^j}{k}$$

If the algorithm runs for ℓ rounds, a union bound gives that the probability of an unbalanced split in any round is

$$\Pr[\mathcal{E}_1 \vee \cdots \vee \mathcal{E}_\ell] \;\lessapprox\; \sum_{j\leq\ell} 2^j/k \;\approx\; 2^\ell/k.$$

This bound is small enough so long as ℓ is slightly less than $\log_2 k$.

To conclude, let n_j be the number of nodes after the j^{th} round. Since we choose the lightest bin, we necessarily have

$$n_j \;\leq\; n/2^j.$$

On the other hand, if all splits are good then

$$k_j \geq k/2^j - O(k^{2/3}).$$

So we can inductively maintain that $k_j \geq (1+\Omega(\alpha)) \cdot n_j/2$.

The argument breaks down when k_j gets too close to $n_j/2$, which will happens when $k_j = (1/\alpha)^{O(1)}$. When that happens, we can use the analysis of Claim 3 to finish the proof. ■