Consistent Hashing

CS273 - Spring 2003
4-9-2003
Scribe Notes by Will Plishker

What is normal hashing? Putting items into n buckets. This is often solved by a static solution (e.g. h(key) mod n).
But what happens to this problem if

1) n changes? In the naive solution, everything must move. We will target instead moving only #things/#buckets on average.
2) we don't know how many buckets there are?

Properties of a consistent hashing scheme
Let A & B be hashing. They send an object k to two different buckets, say y and z, respectively. Then either A cannot see bucket z or B cannot see bucket y.

Each bucket is a assigned a number in (0,1), and each item gets a value (0,1). You could then bin according to least distance or always round down (or up) and wrap at the edges. If a new bucket is added, then we need only move elements which exist in the bucket immediately before it.

If we let each bucket get m values in the (0,1) range and let m = log n, then each bucket gets it's fair share of things, so the target of moving only #things/#buckets is satisfied. In other words, if we add a new bucket, then, on average, only the average contents of the bucket are moved.

Distributed Hash Tables
Consider the buckets to be computers on the internet and that given a key, they should find the value associated with it (i.e. lookup).
The challenges with such a system are: Load balancing, scalability, dynamic nature, no critical point, deterministic (if you don't find it, it's not there).

Chord

Chord solves this problem by the following setup. Give each node an ID selected randomly from the range $[0, 2^{m-1}]$ and each object gets one as well $[0, 2^{m-1}]$. If there are to be n objects, pick m such that $2^m >> n$. Collisions are possible, but we will consider these cases to be negligible. Each node stores a pointer to its successor node. A naïve scheme for getting to the node responsible for a given key k is simply following successor pointers until n<k<n.successor (i.e. you arrive at the node who owns k).

To improve upon this linear time algorithm, each node stores series of fingers. n.finger[i] is defined as the first node to fall after $n+2^{i-1}$ on the circle. When searching for halving the distance to the destination on each round. We can therefore find a key location in log time.

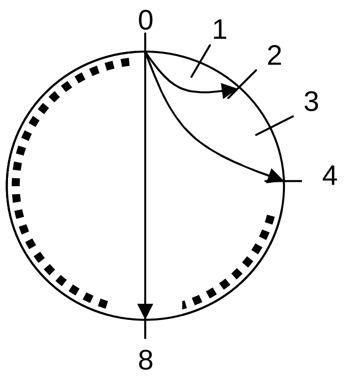

Figure 1. An example configuration of a Chord configuration with fingers for node 0 shown.

A little more precisely, we can find the closest_preceding_node(id) with the following pseudo-code:

```
//search the local table for the highest predecessor of id
n.closest_preceding_node(id) {
    for i from m to 1
        if(finger[i]∈(n,id))
            return finger[i]
}
```

We can then use this procedure to find the node which owns a certain id:

```
//ask node n to find the successor of id
n.find_successor(id) {
    if(id∈(n,n.successor])
        return n.successor
    else
        n’:= closest_preceding_node(id)
        return n’.find_successor(id)
}
```

If a node leaves the network, then it shifts all of its items to its successor and its predecessor updates its successor pointer. For a join, the new node assigns it successor to be the next node in the ring, creates a finger list, and notifies its predecessor to update its successor and move those objects which will now be controlled by the new node. It also inserts itself in other nodes finger lists.

```
//join the system using information from node n’
n.join(n’) {
    predecessor:=nil
    successor := n’.find_successor(n)
    predecessor := successor.predecessor
    build_fingers(n’)
}
```

```
//update finger table via searches by node n'
n.build_fingers(n') {
    i0 := [log(successor-n)]+1
    for each i≥i0 index into finder[]
        finger[i] := n'.find_successor(n+$2^{i-1}$)
}
```

But since this is a distributed system, many joins may occur simultaneously and independently. This can lead to situations in which the previously described invariants (how the fingers and successors point across the network) do not hold in the system. To solve this, each node can periodically fix its fingers and successors.

```
n.maintence() {
    x:=successor.predecessor
    if(x∈(n,successor))  //if x is a better successor
        successor:=x
    successor.notify(n)

n.notify(p)
    if(predecessor=nil or p∈(predecessor, n)) //if p is a better predecessor
        predecessor:=p

n.fix_fingers
    pick i at random
    finger[i] := find_successor($2^i$+n mod $2^m$)
```

As a further optimization, if a node finds that it has a finger update, it can notify its successor to the fact since there is a good chance that it will make a similar adjustment (especially if the update interval was large).

Let us now consider how this system compares to an optimal with respect to the number of hops to a destination. Suppose you have a network of size n with average degree d, what is the minimum number of hops?

$d^{\#hops} \geq n$
$\#hops \geq \log_d(n)$

So for this network d=log n

$\#hops \geq \log_{\log n}(n)$
$\#hops \geq \log(n)/\log(\log n)$
$\#hops \geq O(\log(n)/(\log\log n)))$

Since we are doing O(log(n)) hops, we're not optimal, but it's pretty good.

References:
David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-topeer systems. In Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing, pages 233--242. ACM Press, 2002.

Karger, D., Lehman, E., Leighton, F., Levine, M., Lewin, D., And Panigrahy, R. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In Proc. STOC (1997).