Lecture 24

1 Overview

We will discuss consistent hashing and the Chord implementation of a distributed hash table.

2 Consistent Hashing.

The idea of consistent hashing is to be able to hash with varying number of buckets. In particular, we are concerned with the case that moving items among buckets is very expensive. (For example, we are adding or removing a computer.)

Traditionally, changing the size of a hash table involved moving all the items in the hash table, since the hash function changed and it was by definition "random".

We wish instead to develop a hashing scheme where buckets can be added/or removed very few items are moved to new buckets and the number of items in each bucket is approximately the same.

Consistent hashing proceeds as follows. It assigns items and buckets to the $[0, 1]$ interval using a hash function. Each item is assigned to the closest bucket of greater size.

Now, the load balance and the movement is just the size of a bucket. A removal simply assigns the items in that bucket to the "next" bucket. An addition splits the items in a single other bucket. Cool!!

The following lemma bounds the load and thus, the overheads in the scheme.

Lemma 1
For b buckets and n items, with high probability (under a random choice of hash functions) that the number of items assigned to any bucket is at most $\Omega(n \log^2 n/b)$, assuming $n > b$.

Proof:
With high probability the length of an interval is at most $\Omega(\log n/b)$ since the probability that an interval of this length is not hit by any of the b buckets is at most

$$(\frac{\log n}{b})^b.$$

This can be made

Furthermore, the probability that any interval of length ℓ is hit by more than $\ell \log n$ fraction of items is very small as well.

Question 1: Finish this proof.
□

One can further improve the load balance of the scheme, by assigning a superbucket to contain several of the buckets. This will increase the cost of superbucket insertion/deletion.

For example, if one is willing to have $\log n$ superbuckets, then one can achieve constant load balance, I believe.

An implementation note; The $[0, 1]$ interval can be considered the interval $[0, 2^m]$ for an m that is much larger than $\log n$. The properties will hold nicely since in this case, no two items will likely be assigned the same key.

3 Chord.

One can build a distributed hash table based on consistent hashing. (Indeed, that seems almost to be its intent.) What is a distributed hash table. Computers can join/leave the system. When they join, they will become a bucket. When they leave, they transfer their items to other computers.

Thus, a distributed hash table can be built on the circle topology that underlies consistent hashing. When joining the system, each node hashes itself to a place on the circle, finds a successor and predecessor, and then participates in the hash table in the natural manner. For example, looking up an item can be accomplished by following successor pointers to the appropriate bucket.

The state that each node keeps, is simply two pointers. This is quite efficient, but does lead to long lookup times. An optimization is to keep finger pointers.

In more detail, the distributed hash table assigns a number in $[0, 2^m]$ to each processor. Each processor keeps a pointer to the closest successor processor, the procssor with the next higher hash value. Moreover, it keeps a finger pointer for each i from 1 to m. For a processor hashed to value v, finger pointer i points to the processor responsible for the value $v + 2^i$, that is has a higher value than $v + 2^i$ and is closest among those that meet this condition.

Now, we can search such a network efficiently. For example to find the object whose hashed identifier is *id*, we find its owner as follows. We search downward in our finger list to find a bucket which is before *id*. This has a number that is likely to be at most $2^m/2$ away from *id*. We jump to that node, and search its finger lists as well. After each jump, we should have halved the distance in identifier space to the id. When the distance is around $2^m/b$, we will be very close to the computer that is responsible for the *id*. Thus, after $O(\log b)$ steps, we should get to the bucket that is responsible for the id.

Question 2: Why is this $O(\log b)$ instead of m steps?

To join the network, the joiner generates its identifier v and searchs for the node responsible for it. It updates the successor and predecessor pointers in the list, as appropriate. It then builds its finger table by searching for the owners of $v + 2, v + 4, \ldots$.

To leave, it informs only its predecessors and successors. The fingers are updated lazily to avoid too much traffic. That is, every once in a while each node recomputes its finger tables.