Dynamo notes

Main points:

Distributed hash tables
Consistent hashing
Lookup methods: O(1) vs. O(log N)
Consistency in DHTs

Suppose we have a server with a large workload (so we need to spread the work across multiple servers). How do we figure out which server handles which part of the workload?

By workload I mean anything: search queries, book orders, email, facebook pages, …

If the servers have no state, then it is simple: just pick one at random for each incoming request. If you operate at random, the difference between the highest and the average workload will be ~ O(sqrt(N)) – this is the same as asking for a hash table, how much larger will the largest bucket be than the average?

There are several reasons you might want more consistency in the assignment of work to processors:

1) cache effects – even if the server has no hard state, it might have items cached in its buffers, or in its processor cache, that will mean it will run faster if given similar queries. Example: search terms in google are directed to the same nodes, so that those nodes can keep the results of popular terms in memory.

2) state – if the server needs to maintain state, repeated requests to the same data need to go to the same server for correctness, and/or performance. For example, a particular user should be sent to the same email server, as long as it is still up.

A problem is how to provide both consistency in assignment, and load balance.

In BigTable, they solved this by using multiple level lookup, and caching the result on clients. If a particular tablet becomes overloaded, they can split that tablet in half, splitting the workload into more manageable chunks. This works because failures/changes are uncommon, so the client cache can have a high hit rate.

OK, but can we do this lookup without going through a lookup table?

A simple approach would be to use a hash table (for all the same reasons we can use a hash table instead of a tree in the data structures class): hash on the user ID, or the search term (the "primary key" in Megastore terminology), and let that determine the assignment.

Here's where failures make life harder: normally, you hash modulo the number of buckets – that is, the number of servers (N). But what if there's a failure? if you simply change N, the hash function will reassign everything, so each failure has a cascade effect on the rest of the system – everyone's cache becomes useless!

So what you need is "consistent hashing" – the assignment of work doesn't change as you vary the number of servers, yet you still have efficient lookup and load balancing.

How? You might come up with several on your own. Here's one: use a secondary hash if the first one is to a node that has failed. This requires everyone to keep trying the failed node, which you may not like, so here's a slightly more clever approach:

Hash both the server ID, and the lookup key, into a large address space (say, 128 bits).

You can think of that as a line, where the servers land at random points on the line, and the keys also land at a random points on the line.

Assign each server all the keys to its immediate left.

Now you might think this will cause the workload to be wildly unbalanced, but it turns out this is equally balanced as a normal hash table ~ O(sqrt N).

Further, failure recovery is entirely local – if a node enters, it takes keys from only one node; if it leaves, it gives up its keys to only one node.

If we want to replicate the servers for a key, as in Dynamo, we can use a group of nodes near the key's hash value.

If we want slightly better load-balancing, we can use virtual nodes (as in Dynamo), so that each real server is hashed to several places, e.g., 10-100 – this decreases the workload per virtual node, with the added benefit that failure recovery is now spread over more real nodes.

There's still a problem if the work per key is unbalanced. The O(sqrt N) applies is the work per key is uniform, but if some keys have much greater workload than others, then the result in practice may be worse. (This is why BigTable used managed sharding, rather than a hash table.)

Cool! A remaining issue is lookup: if we have a relatively small number of servers, each client can know the entire set, so that they can directly go to the right server for each key. This is how Dynamo works.

But suppose we have a really large number of nodes, e.g., in a p2p system? We don't want each client to have to keep track of millions of servers, as this set will be changing as nodes come and go.

Turns out there's a particularly clever way to do this, that requires O(logN) state at each node – that is, each node only needs to keep track of a small number of other nodes in the hash table, yet is able to find any node efficiently (in O(log N) time).

Suppose instead of thinking of the hash space as a line, we think of it as a hypercube, where all keys that share a particular bit are in the same plane of the hypercube. Given a key to find, we can do the lookup in O(log N) time if we have a way of reducing the distance by half each step – that is, first, figure out which half of the hypercube we need to go to, and keep iterating until we're done.

If we assume each node keeps a table of nodes which differ from the node's keys in each bit, we're done. It can populate this table by crawling other nodes, or just by listening to requests that other nodes make.

The BitTorrent DHT works in this manner.

What if some of the nodes in this table have failed? No worries! We can either keep a redundant array of neighbors that differ in each bit (we only need one of those to be available), or we can keep it up to date by pinging neighbors to find their neighbors, and replacing them as needed. And in the worst case, we can just send a request to any of our neighbors, and they likely have a path.

Some DHT's (e.g., Chord) maintain all nodes in the system on a circular list. It turns out maintaining this list is hard, in the presence of node failures, and so the best known techniques for it use Paxos voting on the forward and backward pointers.

So far we've talked about this as a matter of finding a server, but the same thing holds for a storage system -- conceptually, given a key, lookup the stored value, or given a key, store a value: get (key) or put (key, value) – this is the DHT interface.

At this point we're faced with all the same tradeoffs we've already seen for managing consistent storage in the presence of failures.

Dynamo uses eventual consistency, because Amazon believes that the most important thing is responsiveness – better to accept a write (e.g., customer tells you they want to buy X), than to delay it for consistency. If the storage system is down, or parts of it are down, we accept the write and deal with the reconciliation later.