CS 5412 - Cloud Computing

Recitation 03/07: DHTs

Sagar Jha

1 Key-Value Stores (DHTs)

DHTs (also called key-value stores) are a type of a distributed database. Members (nodes) manage a portion of the data. One very generic way to organize data is through key-value pairs: (K, V). The key K, like an index, identifies the value it stores.

What are the basic operations you would want for a key-value store? You want to insert and retrieve data. For insert, you have $Put(K, V)$ which inserts a new key K with value V. To retreive data, you have $Get(K)$. It returns the value associated with K. You can also search for data using Get depending on its semantics. For example, Get can return $NULL$ if the key is not present in the database.

What are the main questions for designing a DHT? With anything distributed, you want to decide the distribution itself. Here you want to decide how to distribute the key-value pairs among the nodes. One obvious idea is to partition on the basis of key. The node assigned for a given partition can be the primary replica with other replicas as backups. Recall that this is very much similar to our discussion about Sharding + Replication.

Given this, we now have the familiar questions of membership management:

- How do decide the distribution of keys?
- How to decide the replicas?
- How to handle membership changes?

We will now see how the Chord system answers these questions.

2 Chord

Chord maps the key-space to $[0, 2^m)$ for a large m. Given a key, its position in the key-space can be found out by hashing it to an integer and then taking modulo 2^m. Similarly, nodes are mapped to the same key space by hashing the node identifier and taking modulo 2^m. Think of the key-space as a ring with equally spaced integers $0, 1, \ldots, 2^m - 1$ labeled clockwise. The position in this ring for keys and nodes is referred to by the position of that key, node etc.

Given a key at position k, its primary node is obtained by finding the closest node with position $\geq k$ in the ring. That node stores the key and value associated with it. Visualizing it in the ring, the node positions $n_1, n_2, \ldots$ partition the key-space. The portion of the space between two consecutive nodes n_i and n_{i+1} oriented clockwise is owned by n_{i+1}. Now, it is easy to answer what happens when a new node joins the system. Suppose, a node joins the system. We first find its position in the ring, say n. Let us assume that it lies between two nodes n_1 and n_2. By our partitioning scheme, n should store the keys lying between n_1 and n. They previously belonged to n_2, thus addition of n in the system involves a state transfer of a portion of n_2's keys from n_2 to n. Similar is the case for node failures or leaves. If n fails in the future, the keys it stored will move back to n_2 (to be more precise, n_2 will ask another replica of n for the keys. n_2 may itself be a replica).

It can be shown that with high probability that nodes will roughly handle $1/n$ of the key-space, if n is the total number of nodes. Similarly, with high probability, when a new node joins ($(n+1)^{st}$ node), about $O(1/n)$ fraction of the key-space will need to be moved. Thus, Chord implements consistent hashing. Consistent hashing is a hashing scheme which remaps approximately K/n keys upon resize, where K is the number of keys, n is the number of slots (ref. Wikipedia).

Now, we look at some of Chord's protocols. The single most important functionality is being able to locate keys. This is required for both Put and Get!. To be able to locate a key k, the client at least needs to contact a member of Chord. Each member stores its successor information (a successor is the next node, clockwise, in the ring) which is consistent with respect to membership changes. Thus, a naïve scheme is to hop through successors one by one until you find a node with position $N \geq k$ in the ring (Recall that the node closest to k with position $n \geq k$ stores k). This scheme is inefficient because it is linear in the number of nodes in the system. Chord uses a scheme that only requires logarithmic number of hops (logarithmic in the size of the key-space i.e. $O(log2^m) = O(m)$). To enable this, each node stores a finger table consisting of m entries (as opposed to store $O(1)$ information, that of the successor, in the previous scheme). The $m-$entries at a given node n, are for nodes,

$$succ(n + 2^i), i = 0, 1, \ldots, m - 1$$

That is, you find positions $n+1, n+2, \ldots, n+2^{m-1}$ (modulo m) and find the node which stores these keys. You store routing information for these nodes in the finger table. Now, how do you lookup a key? Given key k and the search context at n, n looks through its finger table to find the closest position $< k$, and transfers the query to the node that corresponds to this position. The search stops if n itself is that node. Then, the successor of n must store k. This can be shown to work in $O(m)$ time by arguing that the distance between n and k halves after each extra hop.

The finger table is updated everytime the membership changes. The basic idea is to correct finger tables by notifying nodes using a reverse lookup $n - 2^i$.

A joining node can initialize its finger tables by asking another existing node n' or copying a finger table from its neighbor and correcting the entries in the future. See the chord paper for more details.

Replication is done by storing the keys in the next $k-1$ nodes, if the replication factor is k. This helps in cases of failures. The state transfer step for joins/failures we discussed earlier can be suitably modified with replication in mind. If a node fails, for every key that it was a replica of (a total of k partitions of the key-space), the node next to the end of the replica chain will become a replica, for each different partition. Assuming that k nodes can never fail simultaneously, Chord then guarantees that keys will never be lost. Replication helps with failures, but also with load balancing. Write requests must go through the primary and be replicated to the backups, but reads can be serviced from any of the replicas. This allows for dynamic load-balancing opportunities where requests are routed to a fairly unloaded node.

2.1 Issues with Chord

Network Partitions : Central to the correctness of a Chord instance is that there's a single ring. If somehow, failures can trigger a partition of the nodes such that each node forms a different chord ring, then everything will collapse. Clients talking to one of the two partitions will see state determined by the partition they talk to, thus introducing inconsistency. This is the split-brain problem we've discussed earlier.

Scaling to WANs : The $O(logn)$ (m is about $\log(n)$) hops that Chord needs to make to locate the key adds $O(logn)$ rounds of latency. If that includes hops across datacenters, the latency just shoots up. For example, you search for a key at a node in a datacenter in San Francisco, and hop to a node in Paris based on the finger table of the node. That leads you to hop to Mumbai, then to Beijing etc. Each one of these hops adds miliseconds of latency (typical of communication across datacenters).

We really need to ask if an $O(logn)$ search time is really needed? Chord is motivated by the clean design choice of storing only $O(logn)$ routing entries. This is good in theory, but in practice, it is much more feasible to store information about all the nodes of the system. Even if you were storing a million entries, the amount of information will be few MBs at best. If you can maintain all key to node mappings at each node, the search is really efficient. You need to make at most one hop in the common case. But then, failure handling becomes a bit complicated. You might be trying to contact a node that does not exist anymore as part of the system (this can happen in Chord too).

This motivates the design of Dynamo, used by Amazon. It runs inside a single datacenter, stores all information about membership at all the members and deals with failures by contacting the replicas for reads, and writing to other nodes any updates if the intended node has failed. This introduces some inconsistency which Dynamo seeks to resolve by moving the key-values to their proper places in the background. This is more in line with the BASE semantics (Basically Available, Soft state, Eventual consistency). Contrast BASE with

ACID to appreciate the differences.

Chord provides a good example of how a DHT looks like in theory. In the next section, we look at some practical issues concerning DHTs.

3 Practical issues with DHT

3.1 Transactions on Sharded DHTs

Recall that a transaction (on a database) provides the guarantees of ACID:

- Atomicity : A transaction either commits in full or aborts. If it aborts, none of its parts has any effect on the database.
- Consistency: The database is in a consistent state before and after the transaction is run.
- Isolation: Transactions do not interfere with each other. This is equivalent to saying that the state of the database changes sequentially where each state change corresponds to a state change from running aa single transaction on the existing state.
- Durability: The transactions persist their changes, so that the effects are not lost even in the case of failures.

Transactions are useful because they give the illusion of sequential execution, but in practice, they are often run concurrently. Thus, one can obtain nice guarantees (of ACID) and performance much better than that for a sequential execution.

How will a transaction on a DHT look like? It will be composed of arbitrary put and get statements. While transactions on a centralized database are easy to support, transactions on DHT is complicated because of sharding. Recall that the key-space is partitioned among the nodes, so a transaction can potentially span multiple key partitions (or shards). Many DHTs take the easy way out and support single-shard transactions. So why does sharding complicate implementing transactions? A general multi-shard transactions will visit multiple shards to perform its operations which will increase latency and thus reduce performance. Nonetheless, one way of realizing transactional properties is through locking. First make sure that the individual operations, that of get and put, are atomic on a given shard. Then, a transaction can lock the keys it needs to access at every shard while it is running. If a lock for a key cannot be obtained (because some other transaction owns the key), the transaction can abort (thus releasing other locks it held) and retry again. If all transactions are visiting shards in a fixed order (called canonical ordering), then we may get away without using locks. We can process every transaction in the same order at all the shards guaranteeing isolation. This will not work for more complicated transactions that need to visit shards multiple times or whose pattern of vising shards cannot be guarantted to be in order.

Passing transactions through multiple shards sequentially still has high latency. Flattening is a technique which runs a sequence of actions as a parallel operation which can be executed concurrently. This helps achieve high performance. In our case, flattening will break supported transactions into transactions per shard and run each part concurrently at the shards.

3.2 Miscellaneous issues

Range queries is one operation that might want to support in your DHT. Such a query, if run as a transaction, can be costly since it might need to visit multiple shards. Some systems like HyperDex can support running range queries in a flattened way.

If multiple applications are making changes to the DHT, one might want a feature that undoes all changes made by one application. There's no information offhand in the key-value pairs that can tell the DHT to delete them in the future. To support such operations, one idea is to store leases with key-value pairs and periodically monitor to delete pairs for which the lease has expired.

Even though with consistent hashing, you get a uniform division of the key-space on expecation, but there is non-uniformity in the following two ways:

1. The keys may not map uniformly to the key-space, thus some nodes may store more data than others.

2. The access pattern for keys may be non-uniform, i.e. some keys may be accessed much more frequently than others.

This leads to two kind of hot spots: Storage and Requests. A node may be overloaded because it has to store many key-value pairs or it may be overloaded because it is getting too many requests for get and put. One way to deal with storage hot spots is map a single node to multiple logical nodes and store all the data assigned to the logical node at that node. This tends to balance load because on average some logical nodes will be storing more tuples and some less, but with a smaller variance, and the sum will be closer to the mean load on the system. This will also help reduce load for get and put operations. Another technique to spread load evenly is replication. We can dynamically increase the replication factor for shards that contain more popular keys increasing the number of nodes that can handle request for such keys.