GEORGIA INSTITUTE OF TECHNOLOGY

CS 6210 Project 4 Report: GTStore

Manas George, Paras Jain

April 24, 2017

1 Partitioning

1.1 Design notes

We utilize a consistent hashing strategy[1] that is resilient to the addition and removal of nodes. A hash indicates a position on a virtual ring upon which nodes lie. Following the ring to the nearest node allows reaching the node or an adjacent node in the case of changing cluster membership. Each node gets a random value in this space representing position on the ring. Each data item is assigned to a node by hashing the client ID concatenated with the key to get its position on the ring, and assigning it to the first node with a position larger than the item's position.

When the client determines which partition and node their data is allocated to, it contacts said node. In case of failiure (e.g. coordinating node is down) the client proceeds to query subsequent nodes around the ring. In this way, arrival or departure of nodes only affects immediate neighbours. We assume homogenous nodes and accept the cost of non-uniform data and load distribution that results from the randomized position assignment. This issue is not significant as MD5 has a roughly even distribution across the hash space[2].

1.2 Discussion of pros and cons of design

This design works very well in practice. Large scale services like Akamai utilize consistent hashing at a global scale and it proves quite resilitent to node failiures. This is a very desirable property in our system. We have gone beyond the suggested design and used the structure of the ring to increase availability by cycling through adjacent nodes if the primary hash result is down.

However, there are some potential drawbacks of this strategy. One notable limitation is a cap on the number of servers that can be added to the ring. As of now, we cap the maximum size of the ring. In order to increase capacity further, we would have to allocate a new hash ring and reallocate servers onto the ring. This can be an expensive operation. Another drawback is performance - unlike hashing strategies (such as those that rely on modular arithmetic over primes), this strategy is not constant time. Realistically, this turns out to not be a major limitation for our service as the ring is not sized too large which limits the maximum amount of time a hash operation could take.

2 Replication

1. Replicate in triplicate, or the the number of data nodes alive, whichever is smaller (triplicate works well empirically, CITE THIS).

2. Each data item is replicated on the node it is assigned to, as well as the two nodes following it on the ring.

3. Replication is handled by data nodes, not the client; the data nodes contact the two successive data nodes in order to replicate data items. This is done in order to minimize traffic between clients and data nodes, restricting that purely to transfer of data. There is also the assumption that the data nodes have more processing power vailable to them than the clients and therefore should do more of the work.

[1]`http://michaelnielsen.org/blog/consistent-hashing/`

[2]`http://michiel.buddingh.eu/distribution-of-hash-values`

4. The Manager keeps track of membership, modifying the membership lists of each data node appropriately when leaving/additional nodes are detected.

2.1 VERSIONING

1. Eventual consistency, just like Dynamo. A single put call returns before the update is applied on all replicas.
2. There is an upper bound on propagation times given no failures.
3. Treat modifications to data item as a new, immutable version; multiple versions of a data item may be present in the system at any given point of time.
4. Use vector clocks; each data item has a list of (node, version) pairs that stores the versioning information across branches.
5. **Reconciliation** If there are two versions in the system that cannot be ordered, we return all the objects in the leaves and treat the subsequent put as the final version of the data item. We effectively push the responsibility of reconciliation in such cases to the user. If ordering is possible, we pick the latest version.
6. Have not implemented clock truncation yet, assume that there aren't multiple catastrophic server failures.

2.2 PUT/GET OPERATIONS

1. We store key value pairs, both of which are blobs of bytes. The key is MD5 hashed after prepending the client id to it, borrowing a trick from the Dynamo paper, in order to get an unique identifier that is used for all internal operations.
2. The client determines which node to send get/put requests to based on the hash as described in the section on consistent hashing before.
3. For put requests, the node generates the vector clock for the new version and writes the new version locally, It then sends the new version to the two successor nodes on the ring. The write is considered successful if both nodes respond.
4. For get requests, the node asks the two ring successor nodes for all existing versions of the data and waits for both to respond before reconciling and returning the result. If reconciliation takes place, the reconciled versions are written back into the replication nodes. If multiple leaves are returned, the context returns this, and no write-back takes place. Subsequent puts with the same key treat the value submitted by the put as the canonical version, and the put automatically overwrites old versions of all replicas with this reconciled version.

HINTED HANDOFF

(NOT IMPLEMENTED YET) Borrowing a technique from the Dynamo paper, if a put operation detects that one of the nodes it tries to write to is unavailable, it moves on to the next node in the ring, marking it as a "false" write. False writes are stored separately in the data nodes, and when the failed node comes back online, the Manager asks all nodes to write stored false writes back to their intended locations. We assume low membership churn and that node failures are transient, and do not implement measures to deal with permanent node failures.