Section 14: Distributed KV Stores and 2PC

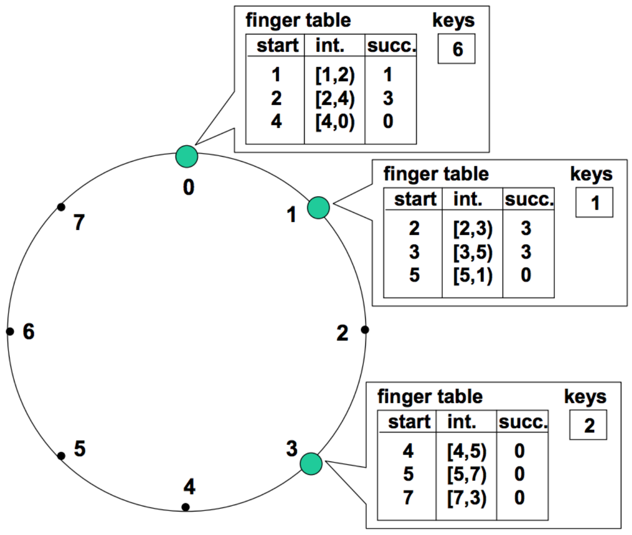
CS 162

August 10, 2020

Contents

1 Vocabulary

- **Distributed Hash Table (DHT) -** A distributed hash table, or a distributed key value store, is a system which follows the semantics of a regular key value store, but in which the data is distributed over multiple machines.
- **Recursive Query -** A DHT query strategy in which requests are made to a central directory, which acts as a proxy to reroute the request to the appropriate data server. Recursive queries tend to have lower latency, and provide for an easier consistency model, but don't scale as well.
- **Iterative Query -** A DHT query strategy in which a lookup occurs to resolve a node name. The client then directly connects to the node to continue the query. Iterative queries tend to have higher latencies, and are more difficult to design for consistency, but provide more scale. Many GFS based KV Stores follow this model.
- **Consistent Hashing -** A technique for assigning a K/V Pair to a node. With consistent hashing, a new node can be added to a DHT while only moving a fraction (K/N) of the total keys. With consistent hashing Nodes are placed in the key space. A node is responsible for all the keys less than it, but greater than its predecessor. When a new node joins, it copies its necessariy data from its successor.
- **Chord -** Chord is a distributed lookup protocol for efficiently resolving the node corresponding to a key in a DHT. Chord uses a finger table which contains pointers to exponentially further nodes provide $\log(N)$ lookup time. It periodically updates the fingertable to provide for eventual consistency.



- **Replication** A strategy for fault tolerance. With replication, one or more **replicas** are responsible for trying to maintain the same state.
- **Primary/Secondary** or **Coordinator/Worker**. A scheme for separation of responsibilities. In this scheme, there is typically a single active primary and one or more secondaries. The primary is typically responsible for being the source of truth and directing operations towards the secondaries.
- **Failover** A fault tolerance procedure invoked when a component fails. Typically this involves switching over to, or promoting a secondary.

- **2PC** - Two Phase Commit (2PC) is an algorithm that coordinates transactions between one coordinator and many workers. Transactions that change the state of the worker are considered 2PC transactions and must be logged and tracked according to the 2PC algorithm. 2PC ensures atomicity and durability by ensuring that a write happens across ALL replicas or NONE of them. The replication factor indicates how many different workers a particular entry is copied among. The sequence of message passing is as follows:

  ```
  for every worker replica and an ACTION from the coordinator,
  origin [MESSAGE] -> dest :
  ---
  COORDINATOR [VOTE-REQUEST(ACTION)] -> WORKER
  WORKER [VOTE-ABORT/COMMIT] -> COORDINATOR
  COORDINATOR [GLOBAL-COMMIT/ABORT] -> WORKER
  WORKER [ACK] -> COORDINATOR
  ```

  If at least one worker votes to abort, the coordinator sends a GLOBAL-ABORT. If all worker vote to commit, the coordinator sends GLOBAL-COMMIT. Whenever a coordinator receives a response from a worker, it may assume that the previous request has been recognized and committed to log and is therefore fault tolerant. (If the coordinator receives a VOTE, the coordinator can assume that the worker has logged the action it is voting on. If the coordinator receives an ACK for a GLOBAL-COMMIT, it can assume that action has been executed, saved, and logged such that it will remain consistent even if the worker dies and rebuilds.)

2 Distributed Key-Value Stores

1. Consider a distributed key-value store using a directory-based architecture.

   Keys are 256 bytes, values are 128 MiB, each machine in the cluster has a 8 GiB/s network connection, and the client has a unlimited amount of bandwidth. The RTT between the directory and data machines is 2ms and the RTT between the client and directory/data nodes is 64ms.

   (a) How long would it take to execute a single GET request using a recursive query?

   There would be 66 ms of latency + $\frac{2^{27}}{2^{33}} = 0.0156$ seconds of transfer time. The total time would be 82ms.

   (b) How long would it take to execute 2048 GET requests using recursive queries?

   Assuming we are able to carefully implement pipeline parallelism, we would still have 66 ms of latency. The transfer time would now be $\frac{2^{11} \times 2^{27}}{2^{33}} = 2^5 = 32$ seconds.

   (c) How long would it take to execute a single GET request using an iterative query?

   There would be 128 ms of latency and $\frac{2^{27}}{2^{33}} = 0.0156$ seconds of transfer time so the total time would be 143ms.

   (d) How long would it take to execute 2048 GET requests using an iterative query?

   Assuming we can take advantage of pipeline parallelism for resolving nodes, it would take 64 ms + $\frac{2^{11} \times 2^{8}}{2^{33}}$ seconds to resolve all the keys.

   We assume each data request can be executed in parallel, so it would take 64 ms + $\frac{2^{27}}{2^{33}} = 0.0156$ seconds to transfer all the data.

   This is a total of 143 ms. Notice that in the recursive query case, the directory server was a much larger bottleneck.

   Note it's reasonable to assume that we can execute our requests in parallel because we can use a hash function to effectively map a key to a uniform random address in our hash table (assuming a non adversarial client). We are also asssuming that there are a large number of nodes, so no single node is limited by bandwidth.

   Here's a proof that our objects should be conveniently distributed across our nodes: https://inst.eecs.berkeley.edu/ cs70/sp17/static/notes/n15.pdf

   Pay special attention to the assumptions made, and whether or not they make sense in practice!

   Also note that the peak bandwidth in this scenario exceeds 1TiB/s, which would require careful client design.

   (e) Now imagine our client is located in the same datacenter, and the RTT between all components is the same (this is a common assumption when modeling datacenter topology).
   Briefly describe how your results would change.

   In broad terms, the RTT latency would now become far smaller than the actual transfer time, removing the previous bottleneck on latency for small transfers.

   From a performance perspective, it is almost strictly better to use iterative querying under these assumptions about latency.

   Note that there could still be other reasons to use a recursive query strategy even under these conditions. For example, one could try to simplify their design, ensure ordering/-

timestamp their outputs, aggregate the data, etc, but perhaps this provides insight into why iterative querying is a popular design for DHT's within datacenters (such as GFS and the many systems based on it).

(f) What are some advantages and disadvantages to using a recursive query system?

Advantages: Faster, easier to maintain consistency.
Disadvantages: Scalability bottleneck at the directory/coordinator server.

(g) What are some advantages and disadvantages to using an iterative query system?

Advantages: More scalable.
Disadvantages: Slower, harder to maintain consistency.

2. **Quorum consensus:** Consider a fault-tolerant distributed key-value store where each piece of data is replicated N times. If we optimistically return from a put() call as soon as we have received acknowledgements from W replicas, how many replicas must we wait for a response from in a get() query in order to guarantee consistency?

We must wait for at least $R > N - W$ responses. If we have any fewer than this number, there is a possibility that none of our responses contain the latest value for the key we are requesting.

3. In a distributed key-value store, we need some way of hashing our keys in order to roughly evenly distribute them across our servers. A simple way to do this is to assign key K to server i such that $i = \text{hash}(K) \mod N$, where N is the number of servers we have. However, this scheme runs into an issue when N changes — for example, when expanding our cluster or when machines go down. We would have to re-shuffle all the objects in our system to new servers, flooding all of our servers with a massive amount of requests and causing disastrous slowdown. Propose a hashing scheme (just an idea is fine) that minimizes this problem.

We can treat the possible hash space as a circle, where every possible hash maps to some point on the circle. We then roughly evenly distribute our servers across this circle, and have each hash be stored on the next closest server on the circle. Then, when we add or remove servers, we need only move a portion of the objects on one server adjacent to the server we just added or removed. This technique is commonly known as **consistent hashing**.

4. Consider a distributed key value store, in which for each KV pair, that pair is stored on a single node machine, and we use iterative querying (this is essentially what we looked at in the previous DHT problem).

(a) Describe some limitations of this system. In particular, focus on bandwidth and durability.

Bandwidth: This system could still be bottlenecked by bandwidth. If there is a popular key in the store, all reads/writes to that key will be directed to the same machine, and thus be limited by the bandwidth of a single machine.

Durability: Writes to this system are only durable, so long as the machine backing the node does not fail. If this machine is backed by RAID then it may have fault tolerance with respect to a disk failing, but if a flood damages the entire machine, data written to that node will be lost.

(b) Propose some strategies for overcoming these limitations.

One strategy for overcoming these limitations is replication.

If multiple replicas contain the same data, the directory server, or a load balancer, could direct GET requests to any machine which contains the data, thus the bandwidth of the node is now scaled by the number of replicas.

Note that this replication now introduces new challenges, such as consistency and consensus.

3 Two-Phase Commit

Quorum consensus is able to provide weak consistency. For this section, assume the DHT backs each node with multiple replicas. Further, assume that these machines use a two phase commit protocol to commit information in a strongly consistent manner.

1. 2PC requires a single coordinator and at least one worker. By default, all replicas can be workers. How should the coordinator be picked?

   The general problem here is leader election - the bootstrap process of choosing and agreeing a leader in distributed systems.

   Naively, we could pick a single machine to be the coordinator for all transactions and hard code that machine as the leader. If the leader crashes, then we must wait for that leader to restart before the system can make any forward progress.

   Modern leader election algorithms do not rely on a hard coded leader but rather allow the workers to choose who they want as their leader. In the event that the leader fails (as detected by some form of health checking), the subset of the machines that are still connected would rerun the algorithm and find a new leader.

   This provides better availability because a new leader can be choose on the fly, but it also introduces further complexities. For example what if there was a network partition and a single cluster of works split into two where each one elects their own leader. Then the partition heals how do you deal with the fact that there are now two leaders.

   Additionally in any situation where we choose a single leader, we would be bottle necked by its upload bandwidth, since it would have to send the request to all replicas. To mitigate this problem, we could pick a leader at random for each request. This would spread resource usage for the dominant resource (upload bandwidth).

   Note we typically don't consider consider GFS style chaining. It doesn't fit in the byzantine generals model, and may not be fast if we can't do a shortest hop tour (in the event of a failed machine).

2. Briefly describe the messages that the **coordinator** will send and receive in response to a PUT. Also describe when and what would be logged.

   - RECEIVE (from client): A PUT query
   - LOG (to journal): PREPARE query
   - SEND (to workers): N PREPARE PUT messages
   - RECEIVE (from workers): COMMIT or ABORT per worker
   - LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
   - SEND (to workers): GLOBAL-COMMIT if it receives COMMIT from all workers, otherwise GLOBAL-ABORT
   - RECEIVE (from workers): ACK
   - SEND (to client): SUCCESS if it sent GLOBAL-COMMIT, otherwise FAIL.

3. Briefly describe the messages that a **worker** will send and receive.

   - RECEIVE (from coordinator): A PREPARE PUT
   - LOG (to journal): PREPARE query

- SEND (to coordinator): COMMIT or ABORT
- RECEIVE (from coordinator): GLOBAL-COMMIT or GLOBAL-ABORT
- LOG (to journal): GLOBAL-COMMIT or GLOBAL-ABORT
- SEND (to coordinator): ACK

4. Under the current model, multiple PUT queries could be sent to separate coordinator machines. Propose set of worker routines which can handle this. In particular, consider the case in which 2 coordinators receive PUT queries for **the same key but different values**. The state of the system should remain consistent.

Upon receiving a PREPARE PUT request, `try_acquire` a lock on the key.

If acquire fails, ABORT, otherwise COMMIT.

Upon receiving a GLOBAL-COMMIT, set the value then release the lock. Release the lock without setting the value on a GLOBAL-ABORT.

The state of the system will always be consistent now. Note that a potentially unintuitive outcome of 2PC here is that both queries could fail here, leaving the key empty despite a PUT request.

One potential solution to this problem (which is out of scope) while maintaining strong consistency could include using PAXOS to determine an order of these queries, and accept only the first.

Note: Many popular DHTs in the real world take different approaches to handling this edge case. Most of these DHTs are eventually consistent.