Distributed Data Storage System

Ayan Patel
Computer Science
California Polytechnic State University
San Luis Obispo, CA, USA
apatel60@calpoly.edu

Nick Papadakis
Computer Science
California Polytechnic State University
San Luis Obispo, CA, USA
npapadak@calpoly.edu

***Abstract*—Distributed data stores are very important in large data centers that require a reliable, scalable, and efficient solution for getting and storing data. There are many algorithms and protocols that exist to enable this functionality.**

Our paper begins with a short survey of landmark papers in the field of distributed systems that created the different mechanisms that enable our simulation to function.

We have built a functional distributed data storage system simulation in GoLang, based on Amazon's Dynamo key-value store. We use a gossip-based heartbeat protocol for membership, both between nodes in a cluster and between different clusters on our hash ring. We have different clusters, each with a set of nodes that select a leader based on RAFT, so that each node can act as a valid replica. We use a hash ring to enable consistent hashing, with multiple tokens assigned to each cluster. This hash ring allows for easy horizontal scalability of the system. If more request need to be handled at a time, more clusters can be added to the ring. We implemented log replication to ensure that the majority of nodes in a cluster have the same log before committing to the data store.

Since we developed a simulation that simply runs on the cores of one machine, we did not think that performance testing was a valid form of evaluation. Instead, we focused on creating and presenting tests that show that the desired properties of our system are correctly implemented.

I. Introduction

A distributed data store is important because it allows for saving and replicating of data amongst multiple clusters of machines. In each cluster of replicas, an election determines the master node, so the system is persistent when a node goes down. Data is replicated on multiple nodes, which can all respond to a get request. Breaking up the request load evenly across multiple clusters enables horizontal scalability. This solution is scalable, reliable, and efficient. Our implementation draws guidance from multiple seminal works in distributed systems.

A. Epidemic Algorithms

The article, 'Epidemic Algorithms for Replicated Database Maintenance' [1], examined different algorithms that are used to propagate updates within a replicated database system in terms of the time it takes to propagate and the network traffic produced. The three algorithms that the authors analyze are direct mail, anti-entropy, and rumor mongering. In direct mail, each new update of a database is immediately sent to all other sites. The biggest issues with this algorithm is that each site does not necessarily know every other site that needs to be updated and mail is sometimes lost. Anti-entropy is where every site periodically chooses another site and resolves all differences between the two. Anti-entropy is very reliable, but expensive, and propagates updates slower than direct mail. In rumor mongering, each site starts ”ignorant”. When a site receives a new update, it starts sharing the new update with random sites. When another site receives the rumor for the first time, it also starts sharing it. When a site has shared the rumor too many times with other sites, it will stop sharing the rumor. Rumor cycles are much more lightweight than anti-entropy exchanges, but there is some chance that a new update will not reach every site.

A good epidemic was measured by looking at residue, traffic, and delay. Residue is the remaining susceptible sites when an epidemic is finished, also known as the number of sites that the update did not reach. Traffic is the number of updates between sites. Delay is quantified with an average and a worst case: the average time it takes the update to reach every site, and the amount of time it takes the update to reach the last site that it reaches. The main issue of a complex epidemic algorithm like rumor mongering is when to stop propagating while ensuring that the update has reached all the sites. Different methods to identify this are either blind or use feedback. Keeping a counter requires an additional piece of information that must be updated and stored. Using a counter and feedback decrease delay and improve residue. In terms of push vs pull methods, pull may require extra traffic overhead, for example, when there is no update. When using a connection limit, pull gets worse while push gets better.

Using some of these protocols together seem to be the best choice to ensure efficiency and accuracy. Anti-entropy can be used to back up data infrequency, while using an epidemic algorithm like rumor mongering for propagating updates. Death certificates can be used to delete items from the database. Dormant death certificates can ensure deletion with low storage cost. Choosing what sites to connect to for rumor mongering or anti-entropy should be decided based on a spatial distribution. It is difficult to maintain a hierarchical structure, but spacial distributions play a significant role in traffic rates and delay. More work needs to be done for spacial distribution for a nonuniform topology.

In the end, the researchers state that they implemented the randomized anti-entropy algorithm on the Xerox corporate system. They state that this solution uses some knowledge of the spatial distribution of sites as an optimization. They also say that using rumor mongering for updates and anti-entropy as a less periodic backup shows promise as another solution.

B. Paxos

The 'Paxos Made Simple' [2] article outlines the Paxos algorithm which is based on consensus, in order for a distributed system to agree on its next task and maintain consistency. The idea is that you want only a value that has been proposed to be chosen, only a single value is chosen, and processes never learn a value is chosen unless it actually has been. To make this work there are three types of agents: proposers, acceptors, and learners. A proposer proposes a value to a set of acceptors. For example, this could be the id for a specific operation. Consider a proposal as a proposal number and a proposal value. Acceptors may accept the proposed value. Acceptors can only accept one proposal and remember what they accepted. The goal is for a majority of the nodes to accept the same proposal. Multiple proposals can be run, each time with a higher proposal number.

An acceptor must accept the first proposal it receives. When an acceptor accepts a proposal, it promises to not accept a proposal with a lower proposal number. If an acceptor has already accepted a proposal in the past, then it will respond to the current proposal with the highest numbered previous proposal it has accepted. When the proposer has received a majority of responses from the nodes, then it sends an accept request message, containing the proposal number and either whatever proposal value it wants or, if a node had responded to say that it had already accepted another proposal, the operation value for that proposal. Acceptors only accept the accept request if they have not responded to another higher numbered accept request. If the acceptor accepts, it will broadcast to all learners that it has accepted. Since the majority of nodes agreed to this value, consistency between the nodes can be maintained.

C. RAFT

RAFT [3] is a consensus algorithm inspired by Paxos that is designed to be easier to understand and implement. To determine a leader, an election is run. Each node first starts as a follower. After a randomized timeout, followers become candidates and increment their term counter. A candidate then votes for itself and asks each node for a vote. If a node has not yet voted for the current term, it will reply with a vote. After a timeout, a candidate will tally up its votes. If they have a majority, they have become the leader and send a leader heartbeat to all other nodes.

D. Dynamo

Amazon's Dynamo [4] is a key-value store focusing on reliability and scalability. It is a NoSQL style database in which values are only stored key-value, where values are binary blobs, which are usually less than 1MB in size. It does not provide ACID guarantees, rather trading some consistency for better availability. Dynamo can be described as "eventually consistent". Many data stores reconcile conflicts during writes, in order to keep reads available. Dynamo does the opposite, keeping it "always writeable" to optimize for user experience. The authors like to describe some of the major driving design goals being incremental scalability, symmetry, decentralization, and heterogeneity.

There are two methods including get(key) and put(key, context, object). It uses a partitioning algorithm to scale incrementally. It uses consistent hashing to do this and distributes loads to virtual nodes in a ring. Nodes are placed on a logical ring based off of a hash of their id. If a value is to be written, its key indexes into the ring, and the next node counterclockwise on the ring is the node that will receive it. Each virtual node is assigned to multiple physical nodes to provide load handling when nodes become unavailable. To achieve high availability and durability, data is replicated on multiple hosts. A coordinator node is in charge of replication of data in a given range. Data consistency is provided by using a Lamport-clock based object versioning system. Each put request contains a context including the version of the data. A pull-based system was chosen for scalability reasons.

Temporary failure of nodes does not affect read and writes due to the hinted hand-off. This is when a node receives a hint in the meta data of a replica to identify which node the data was meant for. Once the failed node recovers, the node with the replica will attempt to send the replica to the original node. In the instance that a failed node goes down for an extended period of time, identifying what replica data it needs is time consuming. Instead a Merkle tree that keeps hashes of keys and its data is used to identify what part of the tree needs updating. Dynamo uses a gossip based distributed failure detection and membership protocol. Data will be replicated at N hosts. R is the minimum number of nodes needed to provide a successful read operation and W is the minimum number of nodes required for successful write operation. Client applications can tune values of N, R, and W to achieve desired levels of performance, availability, and durability.

E. Cassandra

Facebook's Cassandra [5] is a distributed storage system that is reliable, scalable, and available, handling high write rates and efficient reads. Cassandra was designed originally to fill the needs of Facebook's messaging inbox search feature. The authors noted that Dynamo would be limiting for their use case because of the read overhead required for managing vector timestamps is harmful for high write throughput.

Tables have two column families: Simple and Super (a column family within a column family). Rows can be sorted by time or name for different purposes. Cassandra has three methods in its API: insert(table, key, rowMutation), get(table, key, columnName), delete(table, key, columnName). For partitioning, Cassandra uses consistent hashing with an order preserving hash function. Nodes with low loads move on the ring to alleviate pressure on high load nodes. For replication, different policies include rack or data center awareness.

ZooKeeper is used to manage awareness and node locations on the ring. Data is usually replicated on N-1 successor nodes on the ring. Each row is replicated among different data centers for durability.

Cassandra uses Scuttlebutt, a gossip based membership protocol. For failure detection, it uses Accural Failure Detection where there is no boolean up/down for a node, but a sliding scale and a suspicious level variable for each node. A manual command line interface is used for adding and removing nodes. Tokens are assigned to new nodes randomly such that a heavy loaded node is relieved. Each node has a rolling commit log that gets written to before actually writing a value. Before going to disk, an in-memory data structure is updated. For on disk files, a bloom filter is used that summarizes the keys on disk and makes it easier to jump right to the key location.

II. Implementation

We implemented a fully functional distributed key-value data store simulation in GoLang. We have simulated different clusters of nodes. Each cluster is in charge of running elections to elect a leader of the cluster. In addition, all nodes in a cluster have the same replicated data, in case a node goes down. We use a gossip style heartbeat protocol for membership.

A. Membership

We use a gossip based heartbeat protocol for membership. Each node has a heartbeat table of all nodes in the cluster. Periodically, each node will advertise its table to a subset of random neighbors in the cluster. If a node receives the table from another node, it updates entries in its own table. If a heartbeat is not received for a given period of time, that node is declared dead, and eventually removed from the table after an additional waiting time. Therefore, each node has a table of active nodes in the cluster. This same membership algorithm is utilized between the leaders of each cluster, as knowing the current state of each cluster as a whole is necessary for our consistent hashing scheme, which will be discussed in detail later.

B. Elections

We use RAFT to elect a leader of a cluster. This method has three types of members: leader, candidate, and follower. All nodes start out as followers. After a random timeout range, followers become candidates, increment the current election term, and send out vote request messages. If a node receives a vote request, it can send a vote for that node, as long as it has not voted already during the current election term. If a candidate receives enough votes within a specified election timeout, it becomes the leader node and sends a leader heartbeat to every node. Upon receiving a leader heartbeat, a node will become a follower. If no node becomes the leader because of too many candidates trying to become the leader at once, each candidate returns to the follower state and waits a randomized timeout range before becoming a candidate and asking for votes again. Due to the randomized nature of the timeouts, RAFT tends to quickly achieve consensus. The leader node is in charge of handling client requests.

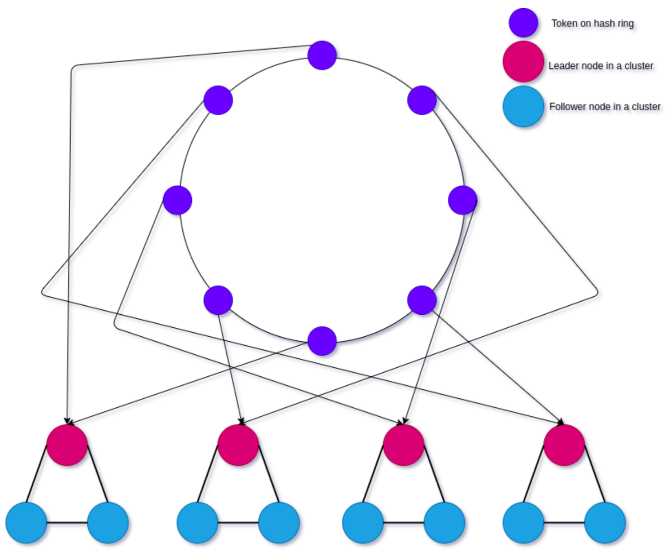

Fig. 1. Dynamo-style hash ring implementation

C. Consistent Hashing

Consistent hashing is achieved in a very similar way to Amazon's Dynamo system. We use tokens evenly distributed around a hash ring. A token on the hash ring maps to a cluster. When a leader node of a cluster receives a request, it hashes the key provided by the client. The hash is then looked up on the ring, where the next token clockwise on the ring corresponds to the cluster that is responsible for that data. That cluster will store the key value pair if the request is a put request or return the value if it is a read request. In our implementation, all the tokens are known to clusters, making this forwarding possible. Each cluster has a channel between them to forward requests that pertain to that cluster. The leader node will check this channel.

Each cluster is assigned multiple tokens on the ring. The total number of clusters and tokens per cluster are known ahead of time as an option provided to the programmer. Tokens are placed on to the ring so that when the ring is full, each token is responsible for an equal portion of the hash space. If an entire cluster goes down, the next token in the ring will be responsible for the hash space for the downed cluster. This is implemented so that if a node receives a request that is not for itself, it looks up the cluster that is responsible for that key using the hash ring. It then checks to see if that cluster is alive using the cluster-level heartbeat table. If that cluster is not alive, it will find the next cluster in the hash ring that is alive and forward the request to it.

D. Log Replication

All nodes in a cluster have the same data. Logs are updated first and sent to each follower node by the leader. Once the majority of the nodes respond with a successful ACK, the log entry can be committed to the data store. The log table consists of entries with an index, term, and command. In our case, for the command, we store the key and value pair as well as the type such as Put or Remove.

When a new log entry is sent to follower nodes, the message also includes the last log index and term in the leader's table. If the follower's last entry matches the leader's, then the new log is added and a successful ACK is sent back. However, if it does not match, the new log is not added and a failed ACK is sent notifying the leader of missing log entries. The leader will then respond with the missing entries to the table. The message also includes the commit index for the leader, which refers to the last committed log entry. This is sent to all the followers who can then commit all entries in their log table up to that point and update their own commit index.

In order to ensure that the leader always has the most up to date log table, when elections occur, followers do not vote for candidates with a table smaller than their own. Each vote request includes the last index of the candidate's table. If the follower has a higher index, the vote is denied. This ensures that the elected leader has an up to date log table, so a follower would never have to update the log table for a leader.

III. Evaluation

We built this simulation as a demonstration rather that runs on multiple cores of a single machine rather than a usable system, so we did not believe that performance metrics would be meaningful to collect. Instead, we focused on making sure that the properties described here were functional. For an in depth look at the evaluation we performed, we have included our unit test file that follows standard Go testing conventions. The following is a list of our tests and their explanations:

- **TestHeartbeatAlive:** This test creates a single cluster with 4 nodes. We sleep for a given period of time to let the nodes in the cluster update their heartbeat tables. Then we check each node's heartbeat table to make sure all other nodes in the cluster exist, are labeled alive, and have an updated count and time.
- **TestHeartbeatDeath:** This test creates a single cluster with 4 nodes. We sleep for an interval so that the heartbeat tables are updated for each node. Then we down a node and sleep again. Then we check the remaining nodes' heartbeat tables to ensure that the downed node is marked as down and not alive.
- **TestHeartbeatCleanup:** This test creates a single cluster with 4 nodes. We sleep for an interval so that the heartbeat tables are updated for each node. Then we down a node and sleep again. This time for a longer period of time. The downed node should have been cleaned up, or removed, from the heartbeat table of the rest of the nodes in this time. So then we check the heartbeat tables of the rest of the nodes to ensure the downed node is removed.
- **TestElection:** This test creates a singe cluster with 4 nodes. We test that an election successfully chose a leader with a put request. If a leader has not been chosen then the request will not be processed. We send two put requests with a sleep interval to make sure the leader has enough time to receive ACKs and update its commit index. Then we check the data store of a follower node to make sure the key exists.
- **TestMultiplePuts:** This test creates a single cluster with 4 nodes. Three put requests are sent to different nodes, all forwarded to the leader. A sleep interval in between each request gives the leader enough time to receive ACKs from followers so that the next log update message includes an up to date commit index. Since the commit index is sent out on the next request, all three key value pairs should be on the leader node, but the follower nodes should only have the first two key value pairs. Therefore, we check the data stores of the followers for the first two and check the leader for the last one.
- **TestClusterHeartbeatAlive:** This test creates 3 clusters with 4 nodes each. Then it sleeps for an interval to let each cluster's leader update its cluster heartbeat. Then we check each cluster's heartbeat table to ensure all the other clusters are included and marked as alive with the correct count and time.
- **TestClusterPuts:** This test creates 3 clusters with 4 nodes each. The purpose of this test is to make sure that the hash-ring and cluster-level heartbeat protocol is working properly. First, 3 clusters are created. Then, one of them is brought down. Next, two puts are executed. The key for each of these puts maps to the cluster that is downed. The test then makes sure that the next cluster in the ring has saved the data for those two puts. Then, the downed cluster is brought back up. The test waits to make sure that there has been enough time for the changes to propagate, and then checks to make sure that the cluster that was brought up has the new puts. It also checks to make sure that the first cluster that saved the data has cleared it from itself after forwarding it to the cluster that was brought back up.
- **TestGet:** This test creates a single cluster with 4 nodes. Two put request messages are sent to node 0. We have a sleep in between requests to make sure the leader has enough time to receive ACKs for the first request, so that the log update from the second request contains an updated commit index. Then we send a get request for the first key. Since all requests are forwarded to the leader, we check the leader's data store to ensure the key value pair exist. Then we check the client response channel on the leader node and make sure the correct value is returned for the requested key.

IV. Limitations

One limitation of our design is that when a cluster forwards a message to another cluster, it does not wait for an ACK back from the other cluster to ensure that it had successfully received the message. It is true our system has a cluster-level membership protocol, so a cluster will never forward to a node that it believes to be down. However, it is possible that the other cluster goes down while the message is in flight, in which case a client request could get lost. Two possible solutions to this are: the cluster waiting for an ACK back before considering a forwarded message handled or push this work to the client, where if they have not received a response to resend their request.

Another limitation of the design is that we did not implement a clock system. This means that if a put for the same key is introduced at certain times, there is no guarantee that the commits will happen in the order in which they were received. Such a system could be implemented with Lamport clocks.

V. Conclusion

We created a simulation of a distributed data storage system that stores key value pairs. Our system is designed to enable horizontal scalability across many nodes to improve availability and resiliency. We utilized a heartbeat protocol for membership, RAFT for consensus, and a hash ring for consistent hashing. We have shown that our system exhibits its desired properties through our provided tests.

References

[1] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, 1987, pp. 1–12.

[2] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.

[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, 2014, pp. 305–319.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[5] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.