Halo 😇

A Self-Stabilizing Distributed Data Store

Varun Iyer

The University of Chicago

ABSTRACT

In a world where quality customer experience is an ever-increasing priority, high availability has become a necessity for the backend of the world's largest organizations. This need led to the creation of Amazon's Dynamo, Facebook's Cassandra, and more recently, ScyllaDB, all of which promise high throughput and low-latency with eventual consistency as a tradeoff. What unites these modern distributed data stores is their ring topology and their replication scheme, both of which derive from Chord, one of the first distributed hash table protocols. One of the downfalls of the Chord paper is the lack of clear specification for how to tolerate network partitions and how to recover from failures. Halo is an implementation of Chord that extends the core protocol to effectively recover from failures and heal partitioned rings. We first introduce the Chord protocol, then our implementation Halo in Rust, a blazingly fast and memory efficient language for performance-critical services. We discuss the different tests we used to gauge Halo's tolerance to different types of failures, and finally conclude with a discussion of the challenges faced while implementing Halo.

INTRODUCTION

When it comes to building a business at a massive scale on the web, reliability is one of the greatest challenges. It can mean the difference between profit and loss in a given year for some of the largest household technology companies, including Apple, Google, Amazon, Netflix, and Facebook. All of these companies rely on some sort of highly available key-value storage system to power their core services. Amazon pioneered this "always-on" experience with Dynamo for its shopping cart feature. Facebook soon followed with Cassandra for its Inbox search. Netflix uses Cassandra for its video streaming services. To achieve this level of availability, these systems sacrifice consistency under certain failure conditions to drive the reliability and scalability of their software.

Chord

The choice of algorithm for Halo was inspired by the predecessor to all of the technologies named above: Chord. What made Chord influential as one of the first distributed hash table protocols was its low-latency lookup time. It is also self-stabilizing under high scalability in that it can handle concurrent joins and leaves. It is one of the first applied distributed algorithm that uses the hash ring topology to divide its key-space in a way that balances load among different nodes. Although the lookup strategy was modified for further speed, both Dynamo and Cassandra utilize the hash ring topology from Chord.

Rust

The choice of language for Halo was inspired by the need for a highly-performant language that still was developer-friendly. Rust was the perfect choice. Like C and C++, Rust has no runtime or garbage-collector, enabling it to operate at extremely low latency. However, unlike other systems languages, Rust uses a unique type system and ownership model that guarantees memory safety and thread safety at compile time. This prevents the countless memory bugs that are prone to languages like C and C++. Rust also has an integrated package manager, great documentation, and a friendly compiler with very useful error messages. Rust is being used by Mozilla to rewrite Firefox, Microsoft to rewrite part of

Windows, Amazon for its web services, and other distributed key-value stores such as TiKV. The language has a great future ahead of it as an alternative systems language.

ZeroMQ

ZeroMQ is an open-source messaging library. It provides sockets that carry message across various transports such as in-process, inter-process, TCP, and multicast. It is extremely fast and delivers messages atomicly. In the Halo implementation, ZeroMQ runs on the same machine as the nodes. Chistributed, the project used to simulate the network, creates a message broker using ZeroMQ that allows nodes to subscribe to messages destined for them and allows messages to be published for other nodes.

Chord

Overview

Chord is a protocol for a distributed hash table that allows efficient key-lookup and scalability.

The Chord protocol specifies:

1. How to find the location of keys across computers in a system
2. How new nodes join the system
3. How to recover from the failure/departure of existing nodes
4. Concurrent joins/failures

All nodes receive roughly the same number of keys. The protocol guarantees with high probability that when an Nth node joins/leaves the network, only an *O*(1/N) fraction of keys are moved to a different location.

In addition, here is no requirement that every node know about every other node. A node needs only a small amount of routing info. In an N-node network, each node maintains info about only *O*(log N) other nodes. A lookup requires only *O*(log N) messages. For a very large set of nodes, *O*(log N) in all practical purposes is approximately linear. Updating routing information when a node joins/leaves the network, however, requires $O(\log^2 N)$ messages.

Consistent Hashing

Every node and key are assigned an *m*-bit identifier using the SHA-1 hashing algorithm. A node's identifier is chosen by hashing the node's IP address, and a key identifier is produced by hashing the key. *m* must be large enough to make the probability of two nodes or keys hashing to the same identifier negligible.

Identifiers are ordered in an *identifier circle* of size 2^m. Key *k* is assigned to the first node whose id is equal to or follows *k*'s identifier in the identifier circle. This node is called the *successor node* of key *k*. Visually, this is the first node clockwise from *k*.

When a node *n* joins the network, certain keys assigned to *n*'s successor are assigned to *n*. When a node *n* leaves the network, all of its assigned keys are reassigned to *n*'s successor.

Scalable Key Location

Each node need only be aware of its successor node on the circle and some additional node information. Queries for a given identifier can be passed around the circle via successor pointers until they first encounter a node that succeeds the identifier. This is the node the query maps to. All that is needed to maintain correct lookups is that every node know its successor. Just this strategy, however, will render lookup time to *O*(N), which is very cumbersome when there are millions of nodes.

In order to improve lookup time, Chord uses the finger table. Each node *n* maintains a routing table with (at most) *m* entries, called the *finger table*. The *i*th entry in the table at node *n* contains the identifier of the first node *s* that succeeds *n* by at least 2^{i-1} on the circle. In other words:

$$s = successor\left(n + 2^{i-1}\right), 1 \leq i \leq m$$

Note that all arithmetic is modulo 2^m, such that the keys wrap around the circle in the clockwise direction. This entry is the *i*th finger in *n*'s finger table, and contains both the node's id and the IP address of the node. The first finger in a node's finger table is always the node's successor.

What happens when a node *n* does not know the successor of a key *k*? If *n* can find a node whose identifier is closer than its own to *k*, then that node will know more about the id circle in the region of *k* than *n* does. *n* searches its finger table for the node *j* whose identifier most immediately precedes *k*, and asks *j* for the node it knows whose identifier is closest to *k*. By repeating this process, *n* learns about nodes with identifiers closer and closer to *k*. This is the basic key search algorithm.

Concurrent Node Joins

Chord preserves the ability to locate every key in the network in the presence of node joins/leave under two conditions:

1. Every node's successor is correctly maintained
2. For every key *k*, node *successor*(*k*) is responsible for *k*.

Each node in Chord maintains a *predecessor pointer*. This pointer contains the Chord identifier and IP address of the immediate predecessor of that node. It can be used to walk counterclockwise around the identifier circle.

When a node *n* joins the network, it learns its successor by asking a node already on the network to look it up. It does not make the rest of the network aware of itself just yet.

Chord periodically runs a stabilization procedure within all its nodes to detect changes in ring topology as it arises. When a node *n* runs *stabilize*, it asks its successor for the successor's predecessor *p*. If node *p* recently joined the system, it decides that *p* should be *n*'s successor instead. *Stabilize* then notifies node *n*'s successor of *n*'s existence, giving the successor the chance to change its predecessor to *n*. The successor only does this if it knows no closer predecessor than *n*.

At this point, all predecessor and successor pointers are correct and lookup calls will work as expected. Chord also periodically refreshes random finger table entries, so that its lookup latency can be maintained along with its correctness.

When ring topology changes, keys are transferred to the new node that they belong to. Node *n* can become the successor only for keys that were previously the responsibility of the node immediately following it. Node *n* simply needs to contact its successor to get the keys/values its responsible for, and this will maintain Condition 2.

Failures and Replication

When a node *n* fails:

1. Nodes whose finger tables include *n* must find *n*'s successor
2. The failure of *n* must not be allowed to disrupt the queries that are in progress as the system is restabilizing.

The goal is to maintain correct successor pointers to enable these conditions. Chord accomplishes this by having each node maintain a "successor list" of its *r* nearest successors on the ring. Maintaining this successor list is not clearly defined in the Chord protocol, but it is hinted that a modified version of the *stabilize* routine will suffice.

If node *n* notices that its successor has failed, it replaces it with the first live entry in its successor list. Node *n* can then direct ordinary lookups for which the failed node was the successor to the new successor. *stabilize* will correct finger table entries and successor list entries pointing to the failed node over time.

The Chord paper shows that if we use a successor list of length $r = O(\log N)$ in a network that is initially stable, and every node fails with a probability ½, then:

1. With high probability, *successor*(*k*) will return the closest living successor the query key
2. Th expected time to execute *successor*(k) in the failed network is $O(\log N)$.

Replicas of the data associated with a key can be stored at the nodes succeeding the key. The successor list aids in duplicating the keys and making dynamic replicas in the presence of successor failures as well.

Partition Healing

Chord has no specific mechanism to heal partitioned rings. Rings appear locally consistent to the stabilization procedure. The paper recommends an approach that checks global consistency. Each node *n* periodically asks other nodes to do a Chord lookup for *n*. If the lookup does not yield node *n*, there must be a partition. However, this will only detect partitions whose nodes know of each other. The paper recommends either every node should know of the same set of initial nodes or nodes maintain a long-term memory of a random set of nodes they have encountered in the past. Both prescriptions are only for detecting a partition; there is no procedure stated for a healing a partition once detected. One of the challenges of implementing Halo was strategizing an effective way to detect and heal partitions.

Halo

Overview

Halo was implemented as a multi-threaded Rust application. The implementation relied on the assumption that nodes on the network are aware of other nodes at startup and no other nodes enter into the network during execution. It also assumes a reliable network, with no dropped messages or malicious actors, and messages are always delivered except for failures and partitions. These conditions were guaranteed by the Chistributed testing software.

Setup

The *main* module parses command line arguments presented by Chistributed, including the name of the node, peer names, and the broker endpoints. Next, the message handler from the *handler* module is constructed.

The Handler object is a wrapper to an Automatic Reference Counted read-write lock (RWLock) that controls access to the handler's thread-safe data – HandlerInner. A read-write lock enables multiple readers to acquire the lock simultaneously, or just one writer. The Handler is initialized with a subscriber socket to receive messages from the message broker, a request socket to send messages through the broker, the names of other peers, and the node object itself.

Node Initialization

The Node is constructed from four values presented by the handler: M, the name of the node, a hashed identifier, and a tau value. M is set by Halo arbitrarily at 8, signifying that the number of unique keys in the key-space is 2^8 = 256. For the purposes of this implementation for testing a 4-node cluster, this key-space sufficed. The identifier was created by hashing the node's name using the hash function in the *hash* module. This function uses an external Rust package *sha-1* to hash the node name and then truncate the value to the last 8 bits. Tau represents the number of successors in the successor list, defined by the Chord protocol as *O*(log N).

The *node* module creates the new node, and handles all read and write operations to the node's local storage. It initializes the finger table, the successor list, and values such as the successor, predecessor, store, and replica store as if the node is the only node in the ring.

The First Join

Finally, the *main* module instructs the newly constructed handler to begins listening to messages from the ZeroMQ message broker. The message loop begins by reading a new message from the subscriber socket, acquiring read access to the lock. After reading the message, a fair unlock is performed to allows any other threads waiting to gain write access the lock (this is done using an external package called *parking-lot*). The message is parsed and then handled by acquiring a write lock again.

The node is first acknowledged by the message broker by receiving a *hello* message and making a corresponding *helloResponse*. It is at this point that the node begins the process of the joining the hash ring.

The node first sends a *join* message to each of its peers. The message constructs are

all defined in the *msg* module. Upon receiving a *joinAck* from another live node, the node asks the acknowledging node to find a successor for it in the ring.

Finding a Successor

When a node receives a *findSucc* message, it asks its internal storage to find a predecessor for the query in the *node* module. It uses a function *in_range* defined in the *hash* module to determine whether the key is in the current node's range. If so, the current node is the successor and it sends back a *findSuccResponse* to the source. If not, the node looks inside its finger table for the closest preceding finger. It then routes the *findSucc* message to that finger, keeping the original source node who wanted the query. Thus, the *findSucc* message is routed recursively around the ring.

Since there are many reasons why a node may want to find a successor, the reasons are distinguished using a *QueryType* enum. Outgoing *findSucc* lookups are pushed in a *current_queries* queue inside the node's local storage, and are popped when a *findSuccResponse* is received. For example, the *joinAck findSucc* lookup query type is *JoinAck*. When a successor is found for a new node, it has almost finished its process of joining the ring. Receiving a predecessor, getting its keys, and stabilizing its finger table/successor list are left up to stabilization.

Ring Stabilization

When a node receives a *hello* message from the broker, along with starting a *join* operation the node also begins to periodically stabilize. Using the external *chan* package, a tick is sent every 1 second to a separate stabilizing thread. On receiving the tick, the thread requests read access to the data of the handler. It then calls a set of methods to help stabilizes the ring.

To stabilize the topology itself, the ring periodically verifies the current node's immediate successor by requesting its predecessor with the *getPred* message. Upon receiving the successor's response, it uses the successor's predecessor to determine who its current successor is. If the successor's predecessor is not the current node, and is closer to the current node than the successor, the current node sets the successor's predecessor as its new successor. It updates this information in its finger table and successor list as well. It then sends its successor a *notify* message, to inform it that it may want to update its predecessor.

Upon receiving the *notify* message, a node will set itself a new predecessor if:

1. its predecessor is nil (a newly joining node),
2. if the predecessor candidate is closer to it than its old predecessor, or
3. its old predecessor has failed.

We will discuss possibility 3) in a subsequent section. In case 1), the data for the newly joined node must then be transferred from its successor from the interval between its old predecessor's key and its key. In case 2), the current node's key-space has become smaller, so it must send some of its extra keys to its old predecessor. The *TransferType* enum defines these ways that a node can transfer/be transferred keys from other nodes after a predecessor change. The handler uses the *TransferType* of a predecessor change to either send its excess keys/values out with a *TransferKeys* call or request keys/values within a given range with a *TransferRequest* call.

The result of running this stabilization procedure every 1 second is a consistent ring topology, where each node has the correct successor, predecessor, and key/values, in the presence of node joins/leaves.

Finger Table Stabilization

To guarantee a faster *findSucc* lookup speed, local finger tables need to be stabilized as well. The stabilizing thread makes this call by receiving write access to the data of the handler. The node than chooses a random integer *i* between 1 and M-1 to index into the finger table. It then pushes a query of type *FixFinger* to its *current_queries*, and finds a successor for that finger's start interval. Upon receiving the *findSuccResponse* for that query,

it replaces the *i*th entry of the finger table with the new successor.

Get

When a *get* message is received from the client to the node, it hashes the key given and pushes a query of type *Get*(k) to *current_queries*, where *k* is the unhashed key. It then proceeds to find the successor of the hashed key with a *findSucc*.

Upon receiving a *findSuccResponse*, the node sends the successor of the key a *retrieve* message with the *k* key from the stashed query type.

Upon receiving a *retrieve* message, a node returns either a *getSuccessResponse* with the retrieved value or a *getFailureResponse* if the value for the key is not in the local store.

Set

When a *set* message is received from the client to the node, it immediately sends a *setResponse* to the client. It then hashes the key given and pushes a query of type *Set*(k, v) to *current_queries*, where *k* is the unhashed key and *v* is the value. It then proceeds to find the successor of the hashed key with a *findSucc*.

Upon receiving a *findSuccResponse*, the node sends the successor of the key a *store* message with the *k* key and *v* value from the stashed query type.

Upon receiving a *store* message, a node sets the key-value pair to its local store, and then sends a replica to the successors in its successor list. Replication and keeping the successor lists up-to-date will be discussed in the following sections.

Successor List Stabilization

Each node makes a call to fix successors in its successor list on every tick of the stabilizing thread. It then chooses a random successor of index *i* from the indexes of its live successors. The node pushes a query of type *FixSuccessor* to its *current_queries* and finds the successor to that live successor. Upon receiving a *findSuccResponse*, the received successor is added to index *i + 1* in the node's successor list. Over time, the entire successor list will be up to date in a stable ring topology.

Replicas

Complete replicas of a node's local store are sent to its successors three possible scenarios:

1. On storing a client's *set* request,
2. On receiving a key transfer from another node,
3. On receiving a new successor in its successor list (in this case, a replica is only sent to that successor).

The node sends a *duplicate* message to the nodes receiving replicas, containing all of its keys and values. The receiver of the *duplicate* message stores the replica in the node's local storage, in a hashmap of node identifiers to key/value stores called *replica_store*.

Failure/Partition Detection

Up until this point, the Halo algorithm has been a slight modification of the Chord protocol. However, the Chord protocol does not clearly explain how to detect and handle failures/partitions, so we came up with our own algorithm.

At every tick of the stabilize thread, a node will send a *ping* message to its successor. The successor will then respond with a *pong* message. The node locally keeps track of how many outstanding *pings* were sent before the last *pong* was received. If that number of *pings* crossed a threshold (set at 2 pings in our implementation), the node declares its successor as failed. It then takes the next successor in the successor list as its own and notifies that node that the current node is its new predecessor.

It is in this case that Condition 3 of the *Ring Stabilization* section comes into play. The new successor's old predecessor had failed. Thus, its key space is now larger than it was before. However, it cannot request any live nodes for the missing data, as that data had died with its old predecessor. The data, however, is contained in the replica of the old predecessor contained in its local replica store. Thus, the new successor transfers the keys from its

replica store so that its key-space is consistent.

Failure/Partition Recovery
When a successor fails, the node also stores the successor as its *last_failed_successor*. On every tick of the stabilize thread, the node will try to rejoin with its *last_failed_successor* with a *rejoin* message. When the successor is failed or in a separate partition, this message is never received. However, when the partition is removed or the successor is revived, the successor will receive the *rejoin* message. It will then perform a similar procedure to a node join, by finding a successor for the node in its partition (or in the case of a node failure, itself). Thus, the ring is reunited and data from separated key-spaces will become redistributed.

Testing

All tests are performed using a 4-node cluster configuration in Chistributed. We believe that this is sufficient enough to show how the protocol handles both node failures and network partitions.

In analyzing the output of these tests, it will be helpful to know the hash keys of the following nodes and values:

- node-1: 21
- A: 27
- X: 75
- node-4: 156
- node-2: 170
- Y: 172
- node-3: 251
- Z: 253

Fail-Recover Testing
The first test case tests how Halo handles nodes failing one at a time. First, four nodes are initialized. The client sets key-values {"X": 1, "Y": 2, "Z": 3} on node-1. These values are stored locally on nodes 4, 3, and 1, respectively. Now, node-1 fails. Thus, node-1's key "Z" must be transferred to its new successor, node-4. We can see from the response of node-2's *get* calls that all 3 values are still available in the ring. We then set different values for the same key "A" on the 3 remaining nodes in the ring. We expect that the last written value (6) be the only one propagated. We then recover node-1. When we get the value "A" from the recently recovered node-1, it returns us 6, a sign that the node is successfully connected to the ring again.

The second test case tests how Halo handles nodes failing at the same time. First, four nodes are initialized, and nodes 3 and 4 are failed simultaneously. "X", "Y", and "Z" are set to node-1 with values "X: INIT", "Y: INIT", and "Z: INIT", respectively. These values are also confirmed to be *get*table from node-2 as well. Next, node-3 is recovered. After checking receiving values from all 3 nodes, we verify that the values are consistent. Lastly, node-4 is recovered. We check the values for the keys at node-4 are verify that node-4 is also re-connected to the ring.

Partition-Heal Testing
The first test case tests how Halo handles a partition where data is set on one side of the partition. First, four nodes are initialized, and a partition is created. The first partition consists of nodes 1 and 2, and the second partition consists of nodes 3 and 4. Next, the client sets key-values {"X": 1, "Y": 2, "Z": 3} on node-1. We verify that "X", "Y", and "Z" can only be retrieved by nodes in the first partition and cannot be retrieved by nodes in the second partition. We then remove the partition. After the ring heals itself, we ask each node to *get* X", "Y", and "Z", and see that all 4 nodes can access the values set on the first partition. Thus, the two 2-node rings have merged back into one 4-node ring.

The second test case tests how Halo handles a partition where data is set on both sides of the partition. First, four nodes are initialized. The key "X" is set to node-1 with value "X: PRE-PARTITION", to show that the data will still be accessible after the partition from replica recovery. The partition is created. The first partition consists of nodes 1 and 2,

and the second partition consists of nodes 3 and 4. The values of "X" are received post-partition showing that the data set pre-partition is indeed available on both sides of the partition. Next, values for keys "Y" and "Z" are set on both sides of the partition, with values of "Y: PARTITION", "Z: PARTITION" in the first partition and "Y: PARTITION2", "Z: PARTITION2" in the second partition. We then remove the partition. Observing the values of each node for "Y" and "Z" after the partition, we see that each node returns a value of "Y: PARTITION" for "Y" and "Z: PARTITION2" for "Z". Hence, the data is consistent within the ring after the partition.

The testing shows that Halo is tolerant to both single node failures/recoveries and multi-node partitions/recoveries.

Conclusion

There were several challenges that had to be overcome in implementing Halo, particularly challenges involving the stabilization functionality.

First, we did not anticipate that a recurring timer in Rust would be blocked by the message handling loop. We realized we needed to share node data between both threads in a way where a timer event did not continuously block receiving a message, and waiting to receive a message did not block the timer. This was extremely difficult, given that locks by default are not fair. Some research presented the idea of using a read-write lock within both threads and allowing each thread to fairly unlock itself when it is finished either reading/writing the node data. This also required what is known in Rust as "unsafe" code, where the compiler does not memory check certain pieces of code that it trusts the developer with.

Dealing with failures and partitions required a substantial amount of work as well, given that the partition healing algorithm we proposed was not part of the Chord protocol. It required some thought and experimentation, and we were pleasantly surprised that it worked the way we expected. It required a clever trick of using a *pingSelf/pongSelf* message to keep a failed node's message loop from blocking it from sending out *rejoin* messages on the stabilizing thread.

While understanding the Chord protocol was relatively easy, implementing the protocol required a much deeper level of understanding. There are several nuances like replication, successor list maintenance, and partition tolerance that are left up to the reader to discover. We believe Halo is a step forward from the original Chord protocol in clarifying its ambiguity and building on the original self-stabilizing distributed data store's weaknesses.

References

1. Avinash Lakshman & Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44.2, pg. 35-40. ACM, 2010.
2. Giuseppe de Candia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulpati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on operating systems principles*, pg. 205-220. ACM, 2007.
3. Ion Stoica, Robert Morris, David Liben-nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11:17-32, 2003.
4. Borja Sotomayor. Chistributed: a distributed systems simulator. https://github.com/uchicago-cs/chistributed.