White Water

Don't Forget Your Liftvest

A Distributed Key-Value Store

Max Demers, Joseph Marques, Eric Volpert

The University of Chicago

Abstract

In the age of web-scale applications that serve millions of users and run on copious numbers of machines across the world, storage systems that maintain a level of consistency, availability, and partition-tolerance are desirable to persist vital information. One such proposed system for persisting data in these conditions is Raft, a simplified alternative to the Paxos consensus algorithm that offers availability, partition tolerance, and eventual consistency. We explore Raft as an algorithm, describe what it is good at and where it struggles, then implement it in Golang in the form of White Water. We discuss the different methods of testing used to gauge tolerance to different types of failures in different situations. Finally, we talk about the challenges we faced in building a Golang implementation of the Raft algorithm, working with ZeroMQ, and even the merits of the Chistributed library.

Introduction

The past few decades have seen an incredible increase in the expected audience of applications being developed [1]. With this increase in audience, we've also seen an increase in the need for storing application information relating to both users and products which can be observed in an increase in datacenter sizes and energy usage [2]. To solve these issues, companies have employed strategies that involve distributing storage across clusters of nodes in order to keep lookups quick and applications fault-tolerant.

Our Choice: Raft

Our choice of algorithm for White Water was the fairly popular Raft [3], an alternative of the Paxos consensus algorithm [4]. We chose Raft because of its claimed simplicity, offers of similar performance to Paxos, and its numerous implementation details specified in the paper. Raft is widely used even in modern systems like the Redis key-value store system, where it is used to replicate data between instances in a Redis cluster [5]. Redis is used for mission critical systems serving hundreds of millions of users showing Raft's viability on a large scale [6].

Why We Chose Golang

For this project, we decided to use Golang for our implementation. We picked Golang for a number of reasons, one being a personal interest of the authors. Golang has proven to be a very powerful systems programming language and is in frequent use in large distributed systems at internet companies like Twitch [7]. Valkov et al. researched the potential of Golang as a highly concurrent programming language in deployments across the web from the streaming services provided by Twitch to internal services at a large high-traffic internet company, Netflix. Go is built for high concurrency and is perhaps best

exemplified in its usage in the Docker ecosystem. The Docker distribution repository, where the Docker binaries are built from, is written in almost 100% Golang according to GitHub's language statistics for that repo. Given that Docker is the most widely used tool for deployment management and hardware agnostic software deployment [8], we knew Golang would be up to the standards modern software development requires of a production-ready programming language.

ZeroMQ
ZeroMQ is a messaging system designed for routing distributed system RPCs (remote procedure calls). ZeroMQ delivers messages atomicly, making it an ideal choice for a distributed system that demands a messaging framework mimicking a real-world network. In our implementation, ZeroMQ is running on the same machine as the nodes. Chistributed, the service we use to simulate the network, creates a message broker that allows nodes to subscribe to any messages destined for them, and allows them to publish messages to specific nodes [9].

Raft

The Raft algorithm at its heart involves an election of a single leader, where every other node is a follower, allowing the leader to have authority over total order of events in the system. Some other consensus protocols require a consensus on order as well as a consensus on value, however Raft overrides the need for that by having the leader as a total authority such that the values that arrive at the leader are stored in order of arrival and no consensus is required other than the leader making sure every node got the right value. Individual nodes don't talk to each other aside from during elections. This cuts down on network activity between nodes.

Setting a Value
When the leader node wants to set a value, it first appends it to its own log. Then, through periodic successful heartbeat messages, the update to its log is propagated to the other nodes in the system.

Once the leader sees that a majority of the nodes have added the value to their logs, the leader updates its commit index to the index of that log entry. Then it applies the transaction to its local state machine. As the follower nodes get the periodic heartbeat messages, they will see the new commit index and commit the value themselves.

This system gives us the Leader Append Only property and the Leader Completeness property as stated in the Raft paper and reiterated below. We also get the Log Matching property, due to the log checking performed on the heartbeat messages, which we will describe more of in our implementation.

Getting a Value
To get a value, the client must send a request to the leader of the cluster. If the client sends a get request to a follower node, the request fails and the client is given the ID of the current leader. When the leader receives a get request, they provide the value from their local store, because the leader is the ultimate authority in the Raft cluster and is guaranteed to have the most updated value by the Leader Completeness Property of the Raft algorithm.

Leader Elections
Leader elections are the only source of non-determinism in the Raft algorithm. While a leader is elected, the leader broadcasts a heartbeat to all follower nodes containing any new transactions to be prepared or committed. When a follower node receives a heartbeat, it does two things: restart the timeout timer, and interpret any commands in the heartbeat. All follower nodes have timeout timers that run for a random duration within implementation specific boundaries. When the timer times out, the follower node starts an election cycle in which it converts itself into a candidate for election.

When a candidate has risen, it sends vote messages to all of its peers with a new term number which is the previous cycle number incremented by one. When a follower or leader receive this vote request message with a great

term than its current term number, it will send a vote confirmation to the candidate. Because these timers are random, typically a candidate will be converted and receive enough votes (a majority) before another node times out and thus becomes the leader. This may not occur in one round of elections: if all nodes are candidates, all nodes will refuse to vote for others, but with random timer intervals, Raft guarantees that a leader will be elected eventually.

Below are a handful of properties that using Raft guarantees us.

Election Safety: at most one leader can be elected in a given term.
Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries.
Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. [3]

White Water

Implementation

We chose to write our Raft implementation as a single-threaded Go application. We can divide it into two portions of execution, setup and running. The setup phase is described below, with the rest of the sections describing the running phase of the implementation.

Our implementation relies on a few assumptions based on our testing software Chistributed. We assume that a node on the network is aware of all other nodes at startup, and that no knew nodes will enter into the network during execution. We also assume a reliable network, where no messages can be dropped, and messages will always be delivered barring node failures or network partitions.

Setup

The setup of the algorithm consists of a number of steps. All of these setup steps are performed from the main control flow of the main function in the node.go file.

First, the network client is created. This is done using the ZMQ module. See the Messaging section of White Water or the documentation of the ZMQ module for more details.

Second, we initialize the Sailor, which is our own custom struct that contains all of the information needed to run an instance of the Raft algorithm. We will describe more about the Sailor struct in the *Structures* section of the paper.

Third, we initialize the storage state machine. Our state machine is simply a key-value store implemented through the use of Golang's built in Map data type, which is simply a hashmap. However, we designed the interface to be abstracted out completely from the state machine, so that the Raft code simply applies arbitrary transactions to the state machine. This abstraction can be useful for future changes, in case we choose to implement a more complicated state machine.

Fourth, the hello message is responded to. This is a strange quirk of Chistributed, as the hello message must be responded to before any other messages can be considered or sent. So we respond to the message, ignoring all other messages until we receive such a hello.

Fifth, the we spin up the message handler function, which makes up the running phase of our program. From that moment on, we can begin the real work of the Raft algorithm.

Structures

White water's fundamental consensus structure is the Sailor. Each node in the network has such a struct, where pointers to the node's outer client struct, as well as

internally maintained fields described in Raft can be found. Vital items like the log, timer, current term, and current leader are all housed in Sailor, and are necessary for both elections and appending new entries to the log. In addition, Sailor houses a “leader” structure, only used if the node is the current leader, where nextIndex and matchIndex, as described in Figure 2 of the Raft paper, are stored [3].

A Golang timer is also stored in Sailor. In the main message handling function, described below, the timer channel is always the first item to be checked. If the timer has signaled, a handle_timeout helper is called to either send out a heartbeat to the network if the Sailor’s current state is leader. Otherwise, this timeout triggers the Sailor to become a candidate in a new election, sending requestVote RPCs to all other nodes on the network.

Other necessary Raft messaging structures are also used. These include the requestVote RPC, appendEntries RPC, and their replies. These structured are marshalled into strings to be stored in ZMQ payloads when being sent over the network. See structures.go for more details on the specific structures used by White Water.

The Message Handler Function

The message handler function in our raft implementation processes every event that runs through our system. This includes incoming messages, election timers, and heartbeat timers.

The heartbeat and election timers are implemented through the Golang timer described above. They take priority over the incoming messages to the Raft node.

If a message is received, then it is sorted to the appropriate function first by the Raft state, as stored in the state field of the Sailor struct, then by the message type, as given in the ZMQ module’s message type field. The type field of the message is populated by predefined strings. Then we dispatch them to the appropriate function, returning to the message handler function once it has been processed.

In the event that there is no timer going off and no message incoming on the network, we instead let the thread sleep for a few milliseconds before checking again. We chose to let the threads sleep due to the observation that if they had no messages, that would continue to be the case for quite some time. This way, we reduce our observed CPU usage from 100% per Raft node down to around 13%. This proved very useful for testing on our local systems, which are severely limited in the number of cores available. We discuss this decision in detail in the *Challenges* section below.

Elections

The election process is handled by code in the requestVote.go file. Here, all election behavior functions for any Sailor state are stored.

After a Sailor has become a candidate and sent out a requestVote broadcast, it waits for replies. When a follower receives a vote request, it compares the last log index and last log term of the candidate, send in the message, against its own log. If the follower confirms that the candidate’s log is at least up to date with its own, it responds to the candidate indicating that the follower granted the vote, and will reject vote requests from other candidates with the same (or smaller) term number. If a Sailor is in a leader or follower state, and this request vote message has a term number greater than their own term number, they revert to follower states, grant a vote to this candidate, and await further messages. This process effectively maintains the Leader Completeness Property as required by Raft, where if a node has an entry that does not match the voting node, a vote will never be granted.

Once a candidate has been granted votes from a majority of nodes on the network, it becomes the leader, sends out its first heartbeat appendEntries message, and resets it's timer to be on heartbeat durations (specified in timer.go). By requiring a majority of nodes to

vote for a leader, the Election Safety Property is maintained by only one node being able to get a majority of votes.

The first heartbeat contains a special no-op log entry, which helps to confirm what log entries have been committed when a node first becomes a leader. This not only helps to inform the leader of which entries have been committed, but will also finally commit entries in the log that were uncommitted by previous leaders: in the event that a leader has sent appendEntries for some item but failed prior to informing all nodes, the next leader, who has received this value, will re-confirm this commit to all nodes who may not have it yet through this no-op heartbeat. This no-op is supported by the process of followers choosing to grant votes or not, where a follower would reject the vote if it has a different item in its log for a recent, but potentially uncommitted, entry.

Append Entries

The Append Entries RPC is handled by the code in the append.go file. There are three main bits of logic: how to send an Append Entries RPC, how to reply to an Append Entries RPC, and how to process a reply to an Append Entries RPC.

For how to send an Append Entries RPC, there are a few small tricks to the logic. First, since our implementation uses Golang's zero-indexed slice to implement Raft's one-indexed log structure, we have to convert every log access to be the entry we want to access, minus one. Second, since we have to deal with very short or empty logs, we had to add a number of if-statements to consider edge cases which the authors of Raft never explicitly set out for us. As for the overall structure, when we send the Append Entries RPC, we add log entries at and beyond the NextIndex that we have stored for that peer node. We also send what we expect are the previous log entry's term and index, so that the follower can verify that we are editing their log at the correct location.

The reply to an Append Entries RPC is slightly more complicated. The first thing we do is check whether or not we need to change our state. If the RPC came from a leader in this term or a leader in a future term and we are not already following them, then we need to change ourselves to do so. Next, we have to consider situations in which we don't accept the Append Entries RPC. We refuse the Append Entries RPC when the term is too old or when the last log entry cited in the Append Entries RPC does not match up with the last log entry in our own log. When we reply with a rejection, we tell the sender of the Append Entries RPC of our current term, that way they can update themselves. This rejection of Append Entries RPCs when the logs don't match is what ensures the Log Matching property described in the Raft paper. On the other hand, if we accept the Append Entries RPC, we drop the old values from our log (but not before considering the implications for Set, as described below). Then, once the log entries are added to our own log, we can consider the commit index that the leader sent us. We commit entries from the log up to the commit index or our own log's end, whichever is smaller. As we commit entries, we apply them to the state machine.

A note about the Append Reply RPC that we send: it's a deviation from the official Raft algorithm. Rather than simply sending back a confirmation or a failure, as specified in the Raft paper, we also send back the range of values that we added to our logs on that RPC call and the range of values that we committed on that RPC call. This is functionally the same as the procedure from the Raft paper, but easier in that it allows us to not store the state of outgoing requests in from the leader for each follower node, dramatically simplifying our own codebase.

Finally, the logic to process a reply to an Append Entries RPC is some of the most tricky. Upon receiving a reply, we first check if the call was successful or not. If the call was successful, then we update our local vote totals for each log entry. Those vote totals keep track of how many nodes have replicated or committed each log entry. Once the log entries have a majority replicated, we begin to

send out commit messages. This level of replication is known in the raft paper as the Match Index of each node. Once we reach a majority of nodes committed on a given index, we reply to the client node the success of our algorithm, as described in the Set section below. On the other hand, in the event of the reply containing a signal that our Append Entries RPC has failed, we don't consider new vote totals. Instead, we consider the term that the follower sent back. If it is beyond our own term, we fallback to being a follower and wait for a RPC from the leader. In the event that the term is correct, but we are still rejected, we roll back one index of what we plan to send next from our log, which is done by decrementing the NextIndex of the node in the leader's Sailor data. Then, we try to resend the appendEntries RPC with the new NextIndex.

Set

Set requests from the client are only processed by the current leader. If a non-leader receives a set request, it replies to the client indicating the proper leader to direct such a request to. We chose to reject set requests to non-leader nodes in order to maintain ordering of the client's set values. Should a client send multiple set requests to a follower and a leader to edit the value of the same key, the order in which the leader processed such requests could be inconsistent. We find it more simple to only allow the leader to process such requests, as described in the original Raft paper.

When a leader gets a set request, it creates a new log entry with the leader's current term, the transaction to be applied to the storage structure (using the key and value indicated by the set request), and the ID number of the set request. To simplify the leader's overhead of looping over all of matchIndex on every heartbeat round, we add a votes count to the log entries. The count is initialized to 1, since the leader is appending it to its log immediately.

In subsequent append replies, the leader uses the upper and lower prepare indices from the follower to increment the number of votes for each log entry within that range if it is less than the commit index. When a majority of prepare votes have been collected, this log entry can be committed. The leader resets the number of votes to 1, indicated one node who has committed the transaction (itself), and moves the commit index up to that log entry.

The leader then counts the number of commits received from followers by using the commit upper and lower indices in a similar fashion. Finally, when a leader has once again received a majority of votes for a log entry in the current commit range, it applies the transaction to its own storage, and informs the client of a successful set.

In the event that a leader has uncommitted log entries and fails, other nodes on the network may commit entries at that same index. To prevent violation of the State Machine Safety Property, the old leader's uncommitted log entries will be overwritten to match the most up to date log when it recovers. When this occurs, the now-follower must detect that this older log entry is a set request it must respond to. The vote count of a log entry is never sent to follower nodes, so if a follower encounters a log entry with a non-zero vote count, it is indicative that this log entry was appended while it was in a leader state. With this, a follower can check the vote count of log entries that it is overwriting, and if the count is non-zero, the follower knows that this transaction was uncommitted and will inform the client that the set request failed. This ensures that our set requests always eventually succeed or indicate an error to the client in the event of fail-recovery.

Get

When a client sends a get request, Raft guarantees through the Leader Completeness Property that the leader has the most up-to-date information. The leader will access the storage module to get the value associated with the indicated key, and respond with that value, or and error indicating no such key, back to the client.

We chose to further improve the availability of White Water by allowing non-leaders to also

process get requests. When a non-leader receives such a request, it also attempts to get the associated value for a key from the storage module. The success or failure of this request is stored in the ZMQ message's error field in the reply. This helps to indicate to the client that the information of that get request may be stale, since there is no guarantee that a non-leader has the most up-to-date log and state machine. We also include the proper leader Id to the client, so future get requests may be routed to the most accurate node in the network. This allows for the client to choose whether they want more consistency or higher availability out of White Water, since they may choose to use or ignore stale get responses from the non-leaders.

Fault Tolerance

White Water is tolerant to both network partitions and fail-recovery. Failure-recovery is guaranteed through both the leader election process, and by having followers check log entries before overwriting them as described in White Water's set behavior. We describe this tolerance in detail in the failure testing below.

Partition tolerance is similar, where White Water guarantees eventual correctness by requiring a majority of nodes to respond before committing entries. This prevents a partition with a minority of nodes from ever committing an entry, while allowing partitions with a majority of nodes to continue operating as required. Partition tolerance behavior is also described in detail below.

Messaging

The messaging library we built uses ZMQ4, a Golang library for connecting to and interacting with ZeroMQ daemons. We originally designed the messaging library to be running in another Goroutine and interact with other threads of the process through channels. I'll describe the initial implementation here, but note we made some changes to keep non deterministic behavior from popping up with chistributed and concurrency. I will refer to this version in the past tense as it is not current.

The model we used for a messaging service started with creating a single subscriber and a single publisher for the given node. The subscriber would subscribe with the broker to the stream of messages with the topic "node_name" so that node-3 would subscribe to the topic "node-3". Then, the subscriber would start a listening goroutine that handled all incoming messages on that topic.

The subscriber listening routine was built to act as a micro pub/sub service provided to the rest of our application. When a new message came in that we had subscribed to, the listening routine would iterate over all subscribers who had registered listeners for the given node. This list contained channel and type pairs such that we would compare the type of the incoming message with the type specified by the subscriber and, if they matched, we would convert the message from JSON to a Go object and pass it through the given channel. This allowed us to provide a thread safe method for piping messages into other threads as they came in.

The other threads would run goroutines for each listener that would block on new channel entries. When a new entry came down the channel, the thread would unblock and process the new message. This concurrency model gave us the ability to register as many listeners as necessary to perform basic functions like interpreting raft messages while continuing to run timer threads.

After we ran into concurrency issues, we moved logic for handling messages into the Raft library where Raft would call to receive a message from the message buffer and send it to the right handler inside the Raft library. While this method works, it isn't ideal and our original method would provide a cleaner interface between the ZMQ and Raft modules.

The publisher for our messaging process is pretty straightforward and hasn't changed since the concurrency issues we ran into. The publisher is registered at start time with the subscriber client. The publishing client then is passed back to the primary node thread and

can be used in any other library we built to call specific functions. The functions we built for the publisher are as follows: Send to Broker, Send to Peer, Send to Peers. These methods do what the title states, sending either to the broker, or to one or more peers.

We ran into some complications when working on the publishing functions in that the documentation provided for chistributed was incomplete and incorrect. While we first tried to send messages to multiple peers via a list in the destination inside our JSON message, we learned that this functionality wasn't actually implemented in Chistributed and one must send to each peer individually. Because we had built some of our infrastructure assuming Send to Peers worked as intended, we had to proxy this behavior by building subclasses for single destination messages.

Send to Peers was modified to take a standard message object we had defined, but changed such that it could split a list of destinations and create submessages for each destination listed, and send each submessage individually. This was a pretty big setback in our design and ended up complicating our code more than we wished it to. Given some more time and motivation, we would love to refactor this particular bit of code and make the messaging library cleaner.

Storage

The storage library is a very simple Golang map of keys to values. This map contains pairings of keys to Value Stamps. A Value Stamp is an object reflecting a tuple of a value and timestamp. We decided to store timestamps inside our model because in the event of a partition or a if data has gone stale, the user might be interested to know when a particular value was last updated. This idea was drawn from standard "last updated" columns seen in standard database models.

For methods, the storage model offers local get and set methods as well as get and set methods that talk to the Raft library to communicate with the cluster, telling us who the current leader is (if not us) and if we are the leader, starting the process of committing a new value to the global state.

The storage module also provides a generic "transaction" model, which performs the exact same operations as the regular API, but is helpful for abstracting away knowledge of the state machine from the Raft code.

Challenges

Single-Threaded vs. Multi-Threaded

In our original implementation, we had a multi-threaded version of the program, with RPC messages being passed on a series of channels. The multi-threaded version of the ZMQ module had a method to take a Golang channel and subscribe it to a message type, so that it would send the message on the channel whenever it arrived. The original message handler function selected the first channel that contained a message and the processed that message.

However, this more interesting design ultimately led to more trouble than it was worth as it made it much harder to debug things like the timer thread and make it more difficult to reason about how our code was functioning. In the end, we decided to restructure our project for the sake of simplicity and made it all run from a single thread. This led to much simpler debugging.

CPU Cores Available

Throughout both our single-threaded and multi-threaded implementations of Raft, we ran into issues with the number of CPU cores available causing strange scheduling bugs. This problem was exacerbated by the fact that we were running the program on drastically different machines, from a high power laptop down to a CS-department virtual machine, which had very limited processing power available.

Ultimately, we mitigated these issues by debugging on a single machine and limiting the number of nodes in use at any given time, as this allowed for simpler scheduling leading to more predictable behavior.

Another fix we used to help mitigate this issue is our change in the timer lengths chosen for timeouts. We make the random window of the timeout timer twice as wide as the recommended 150-300 ms timer by the Raft authors, since that allows us to get greater spacing between node timeouts, since our system resources are so heavily taxed by all the messages already going through the system slow down the election process enough to make elections somewhat difficult to resolve.

CPU Power Available

Another issue we ran into was that once we moved to the single threaded implementation, each Raft node began to consume 100% of the available CPU power as it ran. Ultimately, we realized that this was due to the architecture change. Before, we were waiting for messages to appear in channels, an operation managed by the Golang runtime, which is cleverly coded to keep the CPU usage low. But now that we were working single threaded, we were explicitly checking every possible path in a constantly running infinite loop. This meant that it was consuming of the 100% CPU available to each node.

Ultimately, we fixed this by making the node thread sleep for a few milliseconds whenever it went through the loop and found absolutely nothing to do. That way, we waited a bit for new input, rather than spinning endlessly while doing nothing of value.

Testing

General Testing Information

There are a few points of note with our tests.

First, when our system starts, node-1 always becomes the first leader. This happens because Chistributed starts all the nodes in order, and since each node takes a small while to start up, the timer for node-1 always goes off first, meaning that it always starts as the first leader. This is very convenient for testing, as it gives our test scripts a place to start.

Second, testing this is very difficult to do automatically as the leader elections after the first one are completely random. And since the Chistributed syntax doesn't really provide any syntax to do the parsing of our output that we want, we are simply forced to try performing sets and gets on every node until we find the one who is the leader.

Case One: Fail-Recover Testing

We tested White Water's resistance to to failures with two main tests. For both, we used a four node network, varying which nodes fail during the test and ensuring that they are their storage states are made consistent when recovering.

For the first test, simple-fail.chi, we initialized each of the four nodes, node-1 becomes the leader, and node-1 sets four keys (A - D). We then fail node-1, and give White Water time to re-elect a new leader from the remaining nodes. We then send a get request to node-2 for keys A - D, confirming that they were committed to storage by the followers and indicating if node-2 has become the leader. Since White Water uses random timeout intervals for each node and all non-failed nodes have equal logs, each have a chance to become the leader. To account for this in our next set, we attempt to set a new key E to all three of the non-failed nodes, of which one set will always be successful, and the other two will return rejections.

After that set, we recover node-1, which now has an out-of-date log and storage module, and give White Water time to catch it up. We confirm that node-1 has received key E transaction with a get request, where node-1 has the value for E but will also respond with the correct leader.

Next, we fail node-2. This test's behavior may split here because any node other than node-1 may be the current leader. If node-2 was the leader prior to failing, a new election will once again take place, where any non-failed node is a viable candidate. After allowing for a new election, we test White Water's resilience to overwriting key values in the presence of failures by setting a new value for key E. We similarly test resilience to new keys by setting the transaction G.

We also attempt to set to node-2 (which only responds to the chistributed client, but will not receive messages from any other nodes), since node-2 was the leader prior to failing in this example, in its failed state it still thinks that it is a leader, so we add two new transactions to overwrite A and B's values. When node-2 recovers, the two uncommitted transactions (A and B) will be overwritten by the transactions E and G in its log, prompting a failure to set message back to the client. Finally, we confirm the correctness of node-2's storage module by sending get requests for A, E, and G: node-2 reports that it has the original value for A, the overwritten value for E, the brand new transaction G, and is now fully up-to-date with the rest of the network.

In the event that node-2 was *not* the leader, failing it will cause it to transition from a follower to a candidate. When this occurs, the sets to A and B, described above, will immediately fail since it is not in a leader state. Still, upon recovering, node-2 demonstrates the same final values for keys A, E, and G.

In the next test, multi-fail.chi, we initialize all nodes in the same fashion, but node-3 and node-4 immediately fail. Node-1 is still the leader, so when it receives set requests for keys A - D, it will attempt to have them committed. In this instance though, it adds these transactions to its log, but never commits them to the storage module because it never receives a majority of prepare and commit indications. We confirm this by attempting to get the value of each transaction from node-1, which all results in failure.

During this time, node-3 and node-4 have been stuck in a candidate state, frequently incrementing their term numbers since they are unable to talk to any other nodes. When node-3 recovers, it will initiate a new leader election cycle because of its greater term number, but will be unable to become leader since its log is behind node-1 and node-2. After a new leader is elected, it will update node-3 with transactions A - D, and all non-failed nodes will finally be able to commit these transactions to their storage modules because a majority of nodes are now on the network. This is confirmed by having each non-failed node successfully fulfilling get requests for keys A - D.

Finally, node-4 recovers with no transactions stored but a larger term number. This initiates one more election, after which node-4 will be updated by the new leader. This is confirmed with one final round of get requests to node-4 for all transactions, which successfully indicate the values for each key and inform of the newly elected leader.

Case Two: Partition Testing

We tested White Water's resistance to partitioning with three different tests, two of which are described below. For both, we used a network of five nodes, and had varying set and gets to the network. In the first test, big-1-partition.chi, we first initialize all five nodes and allow time for proper leader election and initial stabilization of White Water. Node-1 becomes the current leader of the network, a single key is set to all nodes, to represent the base log and to help differentiate the outputs, then the partition of the network occurs.

This partition consists of a three-two split of the network, where the current leader remains in control of a majority of the nodes. During the partition, node-1 is successfully able to set three transactions to the store: (b, 43), (c, 44), and (d, 45). After those successful sets, Chistributed attempts a get to all nodes for each of those keys in succession. As expected, node-1 always has the proper values for each key. Similarly, node-2 and node-3 also have the values, and also report that node-1 is still the leader of the network.

Node-4 and node-5, who are partitioned from the network, report never having that key in their storage, since the previous set requests were never delivered to those partitioned nodes. Additionally, they report being in an election cycle, since they have not received heartbeats from the leader, they each time out, become candidates, but can never

successfully become new leaders because they can never receive a majority of votes from the network (each can only ever receive a maximum of ⅘ votes).

Finally, the partition is healed and node-4 and node-5 resume conversation the more up-to-date nodes. Since nodes 1, 2, and 3 will all have the most up-to-date logs, only they may become leaders, and can then update the previously partitioned nodes with the prior committed transactions. This is exhibited by this test's final round of get requests, where after giving White Water some time to stabilize, gets for all keys will return the correct values for all nodes, and node-1, node-2, or node-3 will be the new leader.

The second partition test, big-2-partition.chi, involves the opposite type of partition than the first: a partition occurs, but the current leader is isolated to the partition with only minority of nodes in the network. Here, node-1 (the leader) and node-2 are partitioned, and the rest of the nodes timeout and hold a new election.

After allowing some time for the majority partition to stabilize with a new leader (node-3, node-4, or node-5), the test will attempt to set the same keys as the prior test to the new leader. The new leader, who can contact a majority of nodes, successfully has these transactions committed to the nodes in the majority partition. Node-1, who thinks it is still a leader, can also attempt to process set requests as well. We test this by having node-1 set three keys, all a variations of (shouldFail, Fail). This essentially makes node-1 and node-2 have different (uncommitted) logs of the same length as node-3, node-4, and node-5.

When the partition is healed, the same leader in the majority partition will remain in control of the network since it has a higher term number than node-1, forcing node-1 to turnover to a follower state. When this occurs, node-1 and node-2 begin receiving updates on the logs of the leader which overwrite all of the (shouldFail, Fail) uncommitted log entries. This is confirmed in two ways. First, when node-1 goes to overwrite these log indices, it detects that it had tried to set some key there in a prior term, and therefore will report back to the client that each of those (shouldFail, Fail) set requests have failed. Finally, the test sends a get request to all nodes for keys b, c, and d, and each node reports having the key, even if they are not the current leader. This confirms that the logs of node-1 and node-2 were properly corrected upon the partition being healed, and that they have committed those transactions to their key store.

Conclusion

White Water is a fault and partition-tolerant keystore implemented using the Raft algorithm. Our implementation allows for a client to choose for themselves whether to have consistency or availability when performing reads by allowing follower nodes to fulfill such requests.

We test our implementation using The University of Chicago's Chistributed library, which is a program to test distributed systems algorithms. Using it, we demonstrated White Water's tolerance to partitions and fail-recovery in multiple different scenarios.

In future work, White Water can be extended to run on other distributed systems software, such as mininet, to further test resilience using different environments. This includes, but is not limited to, performance testing, testing on larger networks, testing varying performance enhancing attributes by passing new arguments to White Water nodes at startup, and further testing of a our multi-threaded White Water.

References

1. M. Shaw and P. Clements, "The golden age of software architecture," in *IEEE Software*, vol. 23, no. 2, pp. 31-39, March-April 2006.
2. Jonathan Koomey. 2011. Growth in data center electricity use 2005 to

2010. Oakland, CA: Analytics Press. July
3. Diego Ongaro and John Ousterhout, “In Search of an Understandable Consensus Algorithm (Extended Version)”, Stanford University, 2014
4. Leslie Lamport, “The Part-Time Parliament”, in in ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.
5. https://redis.io/topics/cluster-spec
6. Jing Han, Haihong E, Guan Le and Jian Du, "Survey on NoSQL database," *2011 6th International Conference on Pervasive Computing and Applications*, Port Elizabeth, 2011, pp. 363-366.
7. Valkov et al. “Comparing Languages for Engineering Server Software: Erlang, Go, and Scala with Akka”, SAC 2018: Symposium on Applied Computing, April 9–13, 2018. Pau, France.
8. Preeth E N, F. J. P. Mulerickal, B. Paul and Y. Sastri, "Evaluation of Docker containers based on hardware utilization," *2015 International Conference on Control Communication & Computing India (ICCC)*, Trivandrum, 2015, pp. 697-700.
9. https://github.com/uchicago-cs/chistributed