Raft Made Live: A Simple Python Implementation

Xifan Yu, Yiyang Ou, Zixuan Alex Zhao
xifan@uchicago.edu, yiyangou@uchicago.edu, zhaozixuan@uchicago.edu

June 14, 2020

1 Introduction

Consensus algorithms are essential to building a consistent distributed system. Paxos [1] was an early consensus algorithm developed to achieve data consistency. However, Paxos is also considered as a hard-to-understand protocol, and since its publication, scientists have been searching for consensus algorithms that are more intuitive and are more suitable for building practical distributed system. Raft [2] was among one of them. It is a protocol equivalent to (multi-)Paxos, and it is as efficient as Paxos. Its different structure from Paxos makes Raft more understandable than Paxos, and also provides a better foundation for building practical systems. Raft separates the key elements of an consensus algorithm, such as leader election, log replication, and safety, and this greatly enhances its understandability.

Section 2 covers an overview of the Raft algorithm and consensus algorithms in general. Section 3 describes the implementation details about our projects and section 4 shows the types of faults that our program can tolerate and how they are tolerated. Section 5 provides a detailed description of our testing toolkits and testcases. Section 6 discusses various issues, challenges, and lessons in this project. Finally, section 7 gives a summary of this project.

2 Raft Overview

2.1 Properties of Consensus Algorithms

Consensus algorithms for practical systems typically have the following properties [2]:

- They ensure safety (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering.
- They are fully functional (available) as long as any majority of the servers are operational and can communicate with each other and with clients. Thus, a typical cluster of five servers can tolerate the failure of any two servers. Servers are assumed to fail by stopping; they may later recover from state on stable storage and rejoin the cluster.
- They do not depend on timing to ensure the consistency of the logs: faulty clocks and extreme message delays can, at worst, cause availability problems.
- In the common case, a command can complete as soon as a majority of the cluster has responded to a single round of remote procedure calls; a minority of slow servers need not impact overall system performance.

Raft is a protocol that implements a consensus algorithm, which satisfies the above properties.

2.2 The Raft Consensus Algorithm

A Raft cluster contains several servers: five is a typical number, which allows the distributed system to tolerate two failures. In Raft algorithm, each server is in one of three states: *leader*, *follower*, or *candidate* (Figure 1). In normal operation, there is exactly one leader, and all other servers act as followers. Followers only respond to requests from other servers. If a follower cannot identify the presence of a leader for a while, it will transit to candidate state, and initiate a new election through `RequestVote` RPC. If the candidate receives votes from a majority of the full cluster, then it becomes a new leader and periodically sends out heartbeats to establish its authority.

Raft implements consensus by first electing a distinguished leader, and then asking the leader to manage all client requests and ensure consistency across the servers. All other servers that receive requests will redirect the requests to the leader they are following. The leader is responsible for accepting log entries from clients, replicating these entries on other servers, and telling the servers when to safely apply the log entries.

Following the leader based approach, Raft decomposes the consensus problem into three relatively independent subproblems:

- **Leader election**: a new leader must be elected when there is no current leader.
- **Log replication**: the leader must accept log entries from clients and replicate them across the cluster, enforcing log consistency.
- **Safety**: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index.

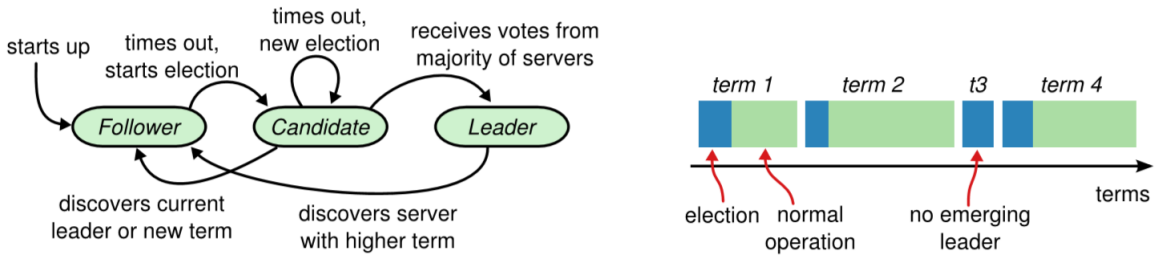

Figure 1: Server states and terms illustration [2]

2.3 Leader Election

When a leader is present in the system, it periodically sends heartbeats to all followers in order to maintain their authority. If a follower does not hear from a leader over a period of time called the election timeout, then it transits to candidate state and initiates a new election.

The election proceeds in the following way:

1. The follower increments its current term, and transits to candidate state.
2. It votes for itself, and issues `RequestVote` RPCs.
3. The candidate remains in current state until one of the following happens:
    - It wins the election by receiving votes from a majority of servers. In this case, the candidate transits to leader state.

- Another server establishes itself as leader, and has term at least at large as the candidate's current term. In this case, the candidate reverts to follower state.
- Neither does it receive votes from a majority of servers, nor has it identified a new leader in the cluster. In this case, the candidate restarts an election.

The third case above happens when votes are split evenly across the cluster. Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. Election timeouts for each server are chosen randomly from an interval. [2] shows that this approach elects a leader rapidly after split votes happened.

2.4 Log Replication

Once leaders are elected, leaders maintain the log by appending the incoming commands from the clients and issues `AppendEntries` RPC so that other servers could replicate the entry. The entries are applied to the state machine once it is safely replicated.

A log entry is committed once the leader has replicated it on a majority of the servers. If followers crash or packets are lost, leaders would retry `AppendEntries` RPC indefinitely.

The leaders keep track of the highest index it knows to be committed, and it includes the index in future `AppendEntries` RPCs so that the other servers eventually would find out. Once the followers learn that a log entry is committed, it applies the entry to its local state machine in log order as well. The Log Matching Property is constituted by the following two rules: if entries in different logs have the same index and term,

- They store the same command
- The logs are identical in all preceding entries.

These two requirements would ensure that when `AppendEntries` returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

In the case of leader failures, Raft handles inconsistencies by forcing the followers' logs to duplicate its own. This means that the conflicting entries in the follower logs will be overwritten with entries from the leader's log. Specifically, the leader keeps track of `nextIndex` for each follower, which is the index of the next log entry the leader will send to that follower. The leader then decrements the `nextIndex` value until it reaches a point where the follower and leader logs match. `AppendEntries` then succeed and removes conflicting entries in the follower's log. Note that this also follows the Leader Append-Only Property as logs only flow from the leaders to the followers.

2.5 Safety

In the section 5.4 [2], the authors talked about additional restrictions of the Raft algorithms, including which servers maybe selected as leaders and which logs could be committed from previous terms. They also gave a proof sketch for Leader Completeness Theorem.

Raft restricts the election process by preventing candidates winning the election unless its log contains all committed entries. During voting, candidate must contact a majority of the clusters to be elected and if the candidate's log is at least up-to-date as any other log in that majority, then it will hold all the committed entries.

If leader crashes before committing an entry, future leaders will attempt to replicate the entry. However, terms from previous terms are committed not by counting replicas, but instead replying on the Log Matching Property - all prior entries are committed indirectly. Raft does this additional requirement in commitment rules because log entries retain their original term numbers when leaders replicate entries from previous terms.

3 Implementation

In this section, we briefly discuss our implementation details. Our implementation is built on `chistributed` and `pyzmq`. The core part of our code is built on the `Node` class provided within `node.py`, which provides handlers for various message types, including but not limited to:

- `appendEntries / appendEntriesResponse`
- `requestVote / requestVoteResponse`
- `redirectSet / redirectGet / redirectSetResponse / redirectGetResponse`
- `set / get`
- `hello`

In this section, we will talk more specifically about their functionalities and how they work together in the context of Raft.

3.1 General

3.1.1 Leader Detection

Following the leader based approach, we need to design a method for each server to check for heartbeats sent from the leader. We achieve this by starting a new thread at the startup of each node that runs a function called `leader_detection_loop()`. This loop keeps running in the background till the node shuts down at the end of the program.

In this loop, after an election timeout the server checks for heartbeats received if it is in follower state. If a heartbeat was detected, the follower recognizes the sender of that heartbeat as the leader, and will redirect all get and set operations to the leader. If a heartbeat was not detected over the period, the follower assumes that there is no leader present in the cluster, and calls RequestVote RPCs as a candidate to initiate a new election.

3.1.2 Heartbeat Propagation

The leader needs to periodically send out heartbeats to maintain its authority over its term. We achieve this by starting a new thread when the server has become leader, running a function called `heart_beat_loop()`.

In this loop, as long as the server is still in leader state, it sends out heartbeats (AppendEntry RPCs without log entries) to all other servers in the cluster. When it exists leader state, the thread running this loop is joined.

3.1.3 Redirection

According to Raft paper, the leader handles all the client requests. When client contacts a follower, the follower would redirect it to the leader. This is implemented not only by including a leader id field in the `AppendEntries` responses but also a custom message type for `set` and `get` operations. When follower nodes received `set` and `get` operations, it sends a redirect message to the current term leader. It sets up a timer concurrently and will timeout if leader does not respond timely. This could happen when the leader fails, the follower fails, or they are in different partitions. In these cases, error message will be returned saying leader timeout. It might also be possible the follower doesn't have a current term leader if a election is ongoing. In this case, error message is also returned saying there is no leader.

3.1.4 Round Trip

According to Raft paper in section 8, to make sure linearizable reads don't return stale data, the leader must check if it's still a leader before processing `get` request. In our implementation, the leader achieves this by running a "round trip" before processing `get` request. In a round trip, the leader sends a message to all other nodes and wait for response. A follower node receiving the message will respond only if the leader is still its leader. Once leader receives more than majority of response, it will process the `get` request. On the other hand, if leader doesn't receive enough response in given time, it will timeout and respond "leader stale" error message to `get` request and also revert to follower state. This mechanism is important to ensure we don't read stale data when partitioning happens, which will be described in section 4.2.

3.2 Leader Election

In Raft algorithm, the leader election process starts when a follower has not received a heartbeat in the function `check_heart_beat()` during the execution of `leader_detection_loop()` over a period time. Then, the follower transits to candidate state and calls the function `request_vote()`.

In the function `request_vote()`, the candidate will enter a loop that keeps restarting a new election until the function `check_vote()` indicates the election result has been resolved, i.e., either the candidate has been elected as leader, or the candidate has reverted back to follower state because another leader appeared.

When a new election starts, the candidate first increments its term by 1, and sets the vote count `self.votes` to 1 since it votes for itself. Then, the candidate sends out messages of the type `requestVote` to all other servers in the cluster, and calls the function `check_vote()` after an election timeout to determine if it needs to exit the loop (when the election has been resolved), or stay in the loop to restart a new election (when a split votes happened).

On receiver side, when a server receives `requestVote` message from a candidate, it checks the following conditions before responding to the request. If the log entries of the candidate (sender of the `requestVote` message) is at least as up-to-date as the receiver's log, and either the receiver has not voted in its current term or the receiver has voted for this same candidate before, then the receiver will grant its vote to the candidate by sending back a message of the type `requestVoteResponse` with the `vote` field in the message set to `True`. Otherwise, the receiver sends back a message of the type `requestVoteResponse` with the `vote` field in the message set to `False`. The `requestVoteResponse` message also sends back the term of candidate running the election, in order for the candidate (receiver of `requestVoteResponse` message) to distinguish between responses to previous elections and current election it is running.

On candidate side, when a candidate receives `requestVoteResponse` message, it first checks if the message is responding to its current election by comparing its current term with the `term` field in the response message. If the message is responding to its current election, it will check for the `vote` field in the response message, and increment `self.votes` by 1 if the `vote` field is set to `True`. If the candidate gathers votes from a majority of the cluster after receiving this message, it will transit to leader state and start a new thread running the function `heart_beat_loop()` that sends out heartbeats to other servers in the cluster to establish its authority.

A candidate might also receive an `appendEntries` message during its election. In this case, if the `term` of the sender of this message (the leader) is at least as large as the candidate's, the candidate quits its election and reverts back to follower state. Otherwise, the candidate stays in its election.

3.3 Log Replication

In section 3.1, we talked about the implementation of redirect mechanism. When followers received get and set messages, they are handled by the `getFollowerHander` and `setFollowerHanlder`, where timers

are initiated and redirect messages are sent to the broker. When leaders received set and get messages, they are handled by `setOperation` and `getOperation`.

As mentioned in section 3.1.4, the leader first checks if it's still the leader by calling `roundTrip` before executing the `getOperation`. `getOperation`, which implements the actual reading of data store, is straightforward, just by checking if its datastore contains the requested key. In `setOperation`, we wait until hearing back from a majority of the peers so that we can append the entries to the log. This propagation of message is implemented by checking the `AppendEntriesReponse` message and `append_entries_has_log` function. We also used the optimization mentioned in section 5.3 [2] where a match index array is used to apply chunk of logs at the same time.

From the follower's side, after checking the leader's current term and `preLogIndex` is matched, we apply the entries sent from the leaders and update commit index. The commit index is updated to the minimum of length of log and the `leaderCommit` field in the message. Then we respond to the leader an `appendEntriesResponse` message and set the flag as `True`. If either of the two criterion failed, we returned with flag as `False`. From the leaders's side, if the append flag is set as `True`, we used the `update_commit` function to update commit index for the leaders. If a majority of the match indexes are larger than the current log index, we apply the current log block.

When applying the log entries, the node stores the key value pair in the `store`, which is implemented dictionary simulating a stores.

4 Fault Tolerance

In this section, we list the errors our program can tolerate, and briefly discuss how each error is tolerated. There are two errors that our program can handle:

- Fail-stop: this is an abstraction of the crash error. It happens when a server no longer responds to the messages from other servers in cluster.
- Network-partition: this happens when the cluster is split into several components. In this partition, communication between different components is blocked, and servers can only communicate to servers in the same component.

4.1 Fail-Stop

When the leader failed, other servers in the cluster no longer receives heartbeats from the leader, and assumes that no leader is present. Therefore, some of them will transit to candidate state and start an election. As long as the number of functional servers in the cluster is more than half of the total number of servers in the cluster, the remaining servers will be able to elect a new leader and proceed in normal operation.

If a follower fails, since in Raft, log replication only flow from the leader to follower, this would not cause a problem if we are only dealing with fail-stop problems. If the log exists on a majority of the nodes, it would be applied in a later term according to the election restriction in section 5.3 [2]. If it does not exist on a majority of the nodes, it could be overwritten in a later section. The new leader will automatically simulates as it has received a dummy set request, which triggers a round of log replication to its quorum, which will commit entries that are the uncommitted but already existing on majority of nodes. Therefore, our code should be resilient to fail-stop errors during log replications.

4.2 Network Partition

When the network is partitioned into components, exactly one of the following happens:

- In the partition, there is one component that contains a majority of servers in the original cluster, and the remaining components do not contain a majority of servers.
- No component in this partition contains a majority of servers in the original cluster.

In either case, the components in the partition that do not contain the leader in the original cluster will not be able to reach the leader. The servers in those components will not detect heartbeats, and transit to candidate state to initiate new elections. If an election takes place in a component that does not contain a majority of servers, the election will never resolve. If an election takes place in a component that contains a majority of servers, the election will eventually resolve. If the old leader is in a partition without majority of nodes, it is not able to commit any logs because there are not enough servers in its component and if client calls `get` on this leader, it will not get enough response from round trip which will cause it to timeout and return to a follower. This step of round trip is necessary because if a new leader is already elected in the larger partition, but a client chooses to read from the old leader, then it will read stale data.

Therefore, in the former case, the component that contains a majority of servers will be fully functional, and other components either do not have a leader, or have a leader will respond error to `get` and `set`. In the latter case, no component will be able to process any requests, and all requests will eventually timeout and errors will be returned.

5 Testcase Analysis

We include 5 test scripts to illustrate functionalities discussed before. **test1 test2 test3** have configuration of 5 nodes, **test4** has 4 nodes and **test5** has 7 nodes. We explain the purpose of each test and how our system behaves in each test. We call the nodes as `node-1`,`node-2`... And at the beginning stage of each test, we will start each node and use `wait -t` to wait for the first leader to be elected.

We provide the testing script that runs all following test cases: `run_test.sh`. Scripts would setup the corresponding configuration files and input the `chistributed` commands under `scripts/chi/`. The output file is stored under `scripts/output`. And our submission already contains the output we ran for **test1**, **test2 test3, test4, test5**. We also provided script that runs only 1 test. See more about testing at the `README.md` file under `script/`.

```
for test_case in $(seq 1 $tot_test); do
    cp ./scripts/conf/test$test_case.conf ./chistributed.conf
    echo "Running test case $test_case"
    test_file=./scripts/chi/test$test_case.chi
    output_file=./scripts/output/test_output$test_case.txt
    echo "stored at: "$output_file
    timeout 40 chistributed --pub-port 23310 --router-port 23311 < $test_file >
    $output_file
done
```

5.1 Set and Get

test1 shows `get` and `set` work properly when there are no node failures. Specifically, we `set` different key-value pairs on `node-1`,`node-2`,...,`node-5`. Suppose `node-1` is the leader. A follower node, such as `node-2`, will redirect its received `set` request (say key-value pair (B,2)) to leader, and the leader replicates the key-value pair and applies it to its store. After this, because of redirection and replication, we can successfully call `get` on `node-3` asking for the value of key `B` even though we didn't explicitly `set` (B,2) on `node-3`.

5.2 Fail Gracefully

test2 shows `get` and `set` return correct error message when called on a failed node: `node-1`. One caveat is since Raft uses randomized election timeout, this failed node can either be a leader or a follower. Hence, the error message might vary in different runs: if `node-1` is a leader, `set` will timeout after leader realizing it can't replicate the log on majority of nodes, and `get` will fail saying leader is stale because round trip can't get responses; on the other hand, if `node-1` is a follower, `set` and `get` will both say leader timeout because its redirect message can't be received by leader; last case is if `node-1` fails and starts a new election, but it can't resolve election because it can't receive votes from others. Then, `node-1` will say "no leader" error to both `set` and `get`, since Raft system can serve client only when a leader is present.

5.3 Fail-Stop

test3 shows the system is resilient to single node failure: `node-1`, whether it's a leader or a follower, if majority of nodes are still alive. After `node-1` fails, if it's follower fails, the system is not affected. If it's leader fails, one of `node-2`,`node-3`,`node-4`,`node-5` will cause election timeout because they can't receive heart beats from `node-1`. Then a new leader will be elected, say `node-2`, and it starts servicing requests with a new leader. It's possible there are log entries replicated on majority of machine and committed on the `node-1`, but not yet committed on the `node-2`. `node-2` will automatically simulates as it has received a dummy set request, which triggers a round of log replication to its quorum, which will commit the uncommitted entries. After this, because majority of nodes are still alive, the system behaves the same way as in **test1** in the eye of client, i.e. `get` and `set` should succeed.

5.4 Network Partition

test4 and **test5** both test network partition. In **test4**, we divide 4 nodes into two partitions each with 2 nodes, so neither partition contains a majority of node. As a result `set` and `get` on the partition with a leader will timeout after leader realizing it can't replicate the log on majority of nodes, and will say the stale leader since round trip will fail. On the partition without a leader, new leader election can't succeed because the partition doesn't have majority of nodes, so `set` and `get` will either say leader timeout (if they haven't started reelection and are in follower state) or no leader exists (if they enter reelection and become candidates). In **test5**, we divide 7 nodes into two partitions one with 2 nodes and one with 5 nodes (`node-3`,..,`node-7`), so 5-node partition has majority of nodes. Then 5-node partition will reelect a new leader if old leader is in (`node-1`,`node-2`) or is not affected if old leader is already in (`node-3`,..,`node-7`). Then, due to similar reasoning in **5.3**, the new leader will simulate a dummy `set` to commit any previously uncommitted but replicated logs. Then, 5-node partition will service requests with the new leader and `set` and `get` will both succeed.

6 More Discussion

6.1 Limitations

There are some limitations of script testing due to the randomness nature of Raft election and reelection. One is we do not know who becomes a leader. If one wants to test leader failure specifically, one can run `chistributed --show-node-output` in terminal mode, which allows one to see who becomes the leader. Then one could fail that node specifically to test leader failure.

Another limitation is we have to add `wait -t` after failing the leader or at start up to allow enough time for a new leader to be elected, before calling `get` and `set` command. Otherwise, `get` and `set` will generate "no leader" error, since election hasn't been completed.

6.2 Observations

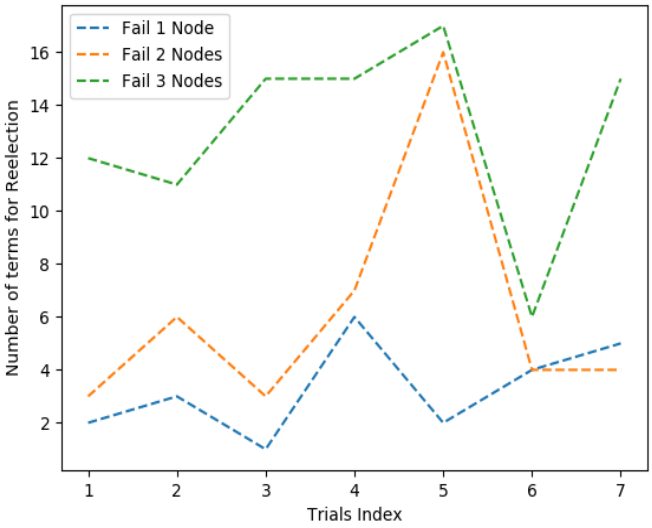

Figure 2: Number of terms needed for reelection

One more interesting observation about Raft algorithm is that the number of terms (time) needed for leader reelection varies according to the number of nodes alive. For example, we performed a experiment with 8 nodes. Each time, we start up all the nodes. After the leader has been elected, we fail $k = 1, 2, 3$ nodes including the leader, and we keep track of the number of terms that passed before a new leader is elected. Figure 2 shows that

- When 1 node is failed, there are 7 nodes alive. In this case, a candidate needs to gather 5 votes from the 7 nodes alive to become the new leader. The blue line in the figure shows that the remaining nodes reelect a new leader pretty rapidly, using only 3 or 4 terms.
- When 2 nodes are failed, there are 6 nodes alive. In this case, a candidate needs to gather 5 votes from the 6 nodes alive to become the new leader. The orange line in the figure shows that they need a little bit more time for reelection of the new leader. The number of terms needed is around 6 or 7.
- When 3 nodes are failed, there are only 5 nodes alive, which is exactly the number of nodes that constitutes a majority of a cluster of 8 nodes. In this case, a candidate needs to gather all votes from the 5 nodes alive to become the new leader. The green line in the figure shows that they took significantly longer time in order to reelect a new leader: the average number of terms needed is around 13.

6.3 Challenges and Lessons

During implementation, we encountered many times KeyError from the pyzmq library and we realized that it was due to concurrency issues at the startup of nodes. For example, pyzmq might send `get` messages to the nodes before it is initialized. Therefore, when pyzmq queried the name of the node, it is set to None, which caused the KeyError mentioned above. We solved this by only allowing `Hello` messages to be delivered to nodes when nodes are not connected.

Another thing was we were not very familiar Python threading library. For example, we forgot to wait for threads to finish when shutting down the node, which caused many strange error messages and missing of expected output. We got more experienced with how to handle threads throughout the project and become able to efficiently use Timer to implement request timeout, etc.

Lastly, since there are not enough documentations on the `chistributed` commands, we had some trouble using the infrastructure. It seems that `quit` command is not working from the `chistributed` framework and we have communicated with other groups and TA: we decided to use a GNU command `timeout` to terminate the bash script if it fails to stop. We also realized the importance of automated testing scripts: it saved a lot trouble when we are trying to conduct experiments for multiple times (provided under Impl). Another issue is that we cannot add custom fields to the `set` and `get` messages because they are instantiated from the `cmd` and `interpreter` module. To circumvent this, we used a custom message field `redirectGet` and `redirectSet` to redirect the messages to the leaders.

7 Conclusion

In this project, we implemented a simplified version of the Raft algorithm that is resilient to fail-stop errors and network partitions. The Raft algorithm is easily understandable in that it separates the key elements of consensus algorithms, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Our implementation of the `appendEntries` and `update_commit` logic follows directly from the Raft paper [2] and ensures that reelections would occur and system would go back to normal when leader fails. We also look into the relation between the numbers of failed nodes and the length of reelection process in section 5.

References

[1] Lamport, L. *Paxos made simple.* ACM Sigact News 32, no. 4 (2001): 18-25

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