Final Report

M120: Distributed Systems

Raft consensus protocol implementation

June 2019

Dimitris Sideris
[cs3180007]

Activities that were set out for this project

This project is about implementing the Raft consensus protocol as described in "In Search of an Understandable Consensus Algorithm" by Diego Ongaro and John Ousterhout. The Raft protocol was designed as a Paxos alternative and of its main objectives is the understanability of a distributed consensus algorithm.

The activities that were set out for this project are the main activities that are described in the paper:

- Leader election
- Log replication
- Safety

Our project was based in the MIT Distributed Systems class of Spring 2018 (https://pdos.csail.mit.edu/6.824/labs/lab-raft.html)

In this class a skeleton code was provided and one should implement the raft protocol based on the aforementioned code skeleton. We chose this approach because there were some very useful features that helped us deal with the core of the implementation and not the network and infrastructure setup. Specifically there was provided:

- An rpc package named labrpc that is an implementation of basic rpc commands in go language for ease of use
- A config.go file that makes the infrastructure setup to run many nodes in one machine.
- A perister.go file that implements the persistence logs of the nodes. This is needed to save logs that can be retrieved from nodes that have failed and are later resumed.
- A raft.go file that only has the names of the base methods and a skeleton ready for implementation.
- A test_test.go file that was the most useful one. This file has 18 test cases that checks the implementation done in raft.go file and its correctness. This test suite proved to be very helpful as it guided us though the assertions in every step so that we can have a correct implementation of the raft algorithm.

Activities that were achieved

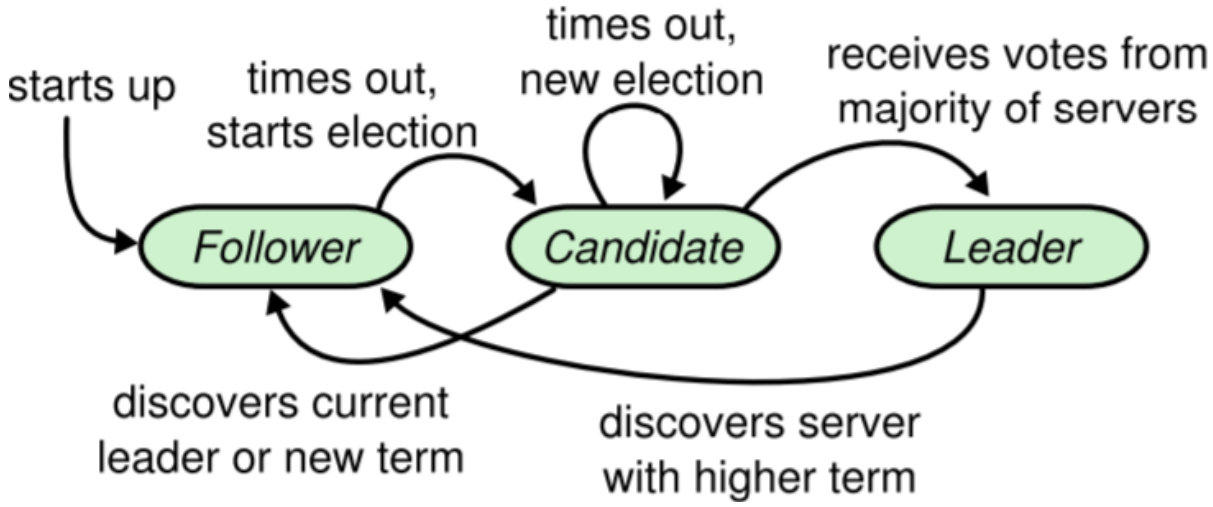
- **Leader election** is a process that one node of the distributed setup of server becomes a leader and is responsible for characterizing its leadership period as one term. Inside main loop of the code in the Make function a node becomes a candidate and sends to all other nodes RequestVote rpcs in order to obtain leadership. If current candidate has

a current term that is bigger than other nodes' terms and has a persisted log that outmatches other nodes' logs then it becomes the leader. It then sends HeartBeat rpcs in order to preserve system liveness and see if some other node has failed.

- **Log replication** is achieved with AppendEntries rpc that a leader sends to other nodes when an external client send a value request to the system to preserve. The leader sends commit commands to other nodes and matches each other nodes' index to the leader's index and replicates the correct log until it reaches a nextIndex that has no other values.

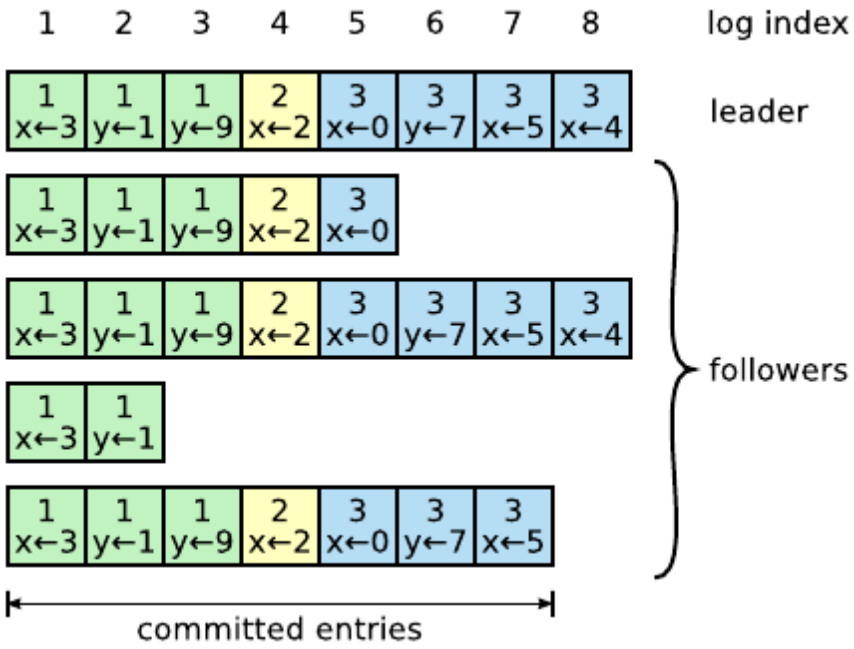
- **Safety** is achieved by choosing the correct leader each time an election occurs meaning that a candidate leader must have the majority of previous entries in its log or it cannot become leader and the election process is reinitialized. Also in AppendEntries process when a value is commited and failure of leader occurs then after the new leader election, the new leader if it has the majority of the quorum of previously commited values it matches the index of each other node forcing them to commit to the current leader status. In case some previously commited values never reached majority in nodes persistence it can be overwritten by new values that leaders commit to all nodes.

Pictures with text explaining the architecture of your system and how it works

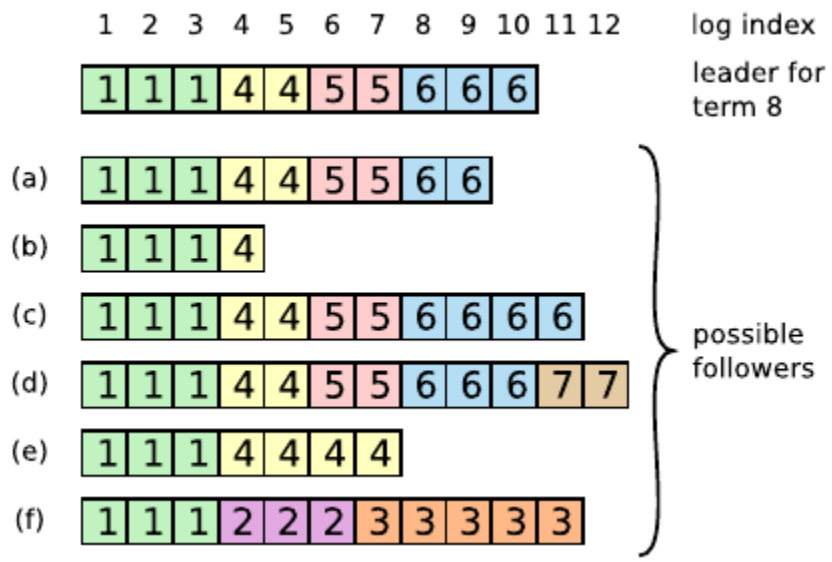


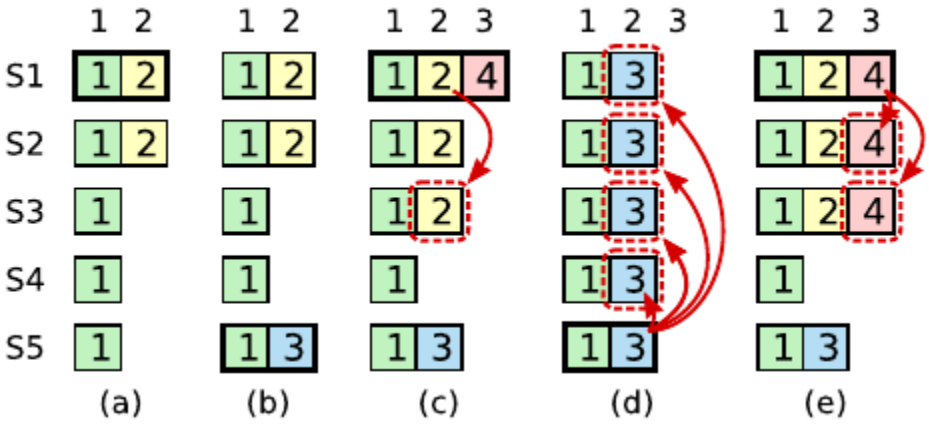
The above picture is the process of the leader election. Any node at the beginning of the protocol has a role of follower. Each time a Timeout occurs any node can become a candidate and sends requestVote rpc to other nodes. If a node send an rpc to another node and in the meantime the sending node receives a requestVote rpc with a greater term, then the first node steps back to the follower role. Else a node that send a requestVote rpc and does not receive

anything from another node it steps forward from candidate role to leader if it receives a vote from the majority of the nodes including its own vote for itself.



The above picture shows how logs of different nodes can be. When a new command reaches the leader, the leader appends it to *log* and reply an index where command will exists if successfully replicated. Followers won't accept requests from clients, they simply redirect clients to the leader. The leader sends new log entries to each server, attaching index of previous log entry and the term of that entry. The log entries are determined by *nextIndex*. Followers check whether they have previous log entries and then append new log entries to certain location. If a majority of followers accept a command, the leader increases his *commitIndex* and replies to the client. The *commitIndex* will be sent in next *appendEntry* request, followers commit the commands known to have been replicated in a majority servers.





The leader sends *appendEntries* request to each follower periodically to keep the role state. In a request, if a follower has log entries not replicated, the next log entries will be attached in the request. Each request is executed in a new thread in case the network is slow or unreachable, which will block the loop. Every *appendEntries* contains more than one log entries. When a follower receives this request, it first confirms the role of the server claimed to be the server. Then it checks if it has already recorded the log entries before newer ones. If all pass, the server overwrite the log and replace log from certain index with given log entries in request. The leader receives reply from followers and increase *matchIndex*, which means logs entries known to replicated in a follower. If a majority followers have replicated a log entry, the leader increase the *commitIndex* by one and replies to the client. To prevent potential problems of unreliable network, the log entries are committed one by one in incremental order.

Evaluation

As stated in introduction the project skeleton was provided by the MIT distributed systems class. In the given skeleton there was a test suite provided to check all crucial raft functionality. There are 18 tests provided in this test suite that exhaustively checks leader election procedure in normal cases and when during election candidate leaders fail. Test suite also tests various permutations of log replication and corner cases when leaders try to apply entries and before getting replies they fail. The tests are asserting expected behavior from the theoretical paper and the test cases pass only when everything works as expected. Test cases can be found in test_test.go file in the attached code. As for scalability we tested the code with the above test suite for as much as 30 instances as our resources were limited. The code scaled easily from 3 to 30 nodes with no effort. Unfortunately our testing environment was limited and wider setups have to be made in future work to check the scalability in hundreds of server nodes.

Demo

In order for someone to run the the test suite, she should have labrpc package installed in src folder of go source and then navigate to the project's path and run:

go test -v

An example verbose execution of the whole process of a leader election with 5 nodes follows:

```
=== RUN   TestInitialElection
0 says: hello world!
1 says: hello world!
2 says: hello world!
3 says: hello world!
4 says: hello world!
Test: initial election...
3 tells 2 : vote me,  {1 3 0 0}
3 tells 1 : vote me,  {1 3 0 0}
3 tells 4 : vote me,  {1 3 0 0}
3 tells 0 : vote me,  {1 3 0 0}
2 says: higher term detected, term= 1
2 tells 3 : vote granted
4 says: higher term detected, term= 1
4 tells 3 : vote granted
3 says: I am the leader in term 1
0 says: higher term detected, term= 1
0 tells 3 : vote granted
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
1 says: higher term detected, term= 1
1 tells 3 : vote granted
```

```
0 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
0 says: my current state, { 1 , 3 , [] }
1 says: my current state, { 1 , 3 , [] }
2 says: my current state, { 1 , 3 , [] }
3 says: my current state, { 1 , 3 , [] }
4 says: my current state, { 1 , 3 , [] }
0 says: my current state, { 1 , 3 , [] }
1 says: my current state, { 1 , 3 , [] }
2 says: my current state, { 1 , 3 , [] }
3 says: my current state, { 1 , 3 , [] }
4 says: my current state, { 1 , 3 , [] }
3 tells 4 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
```

```
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
```

```
3 tells 2 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
```

```
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
2 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
4 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
0 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
1 tells 3 : pong, &{1 true 0}
```

```
4 tells 3 : pong, &{1 true 0}
3 tells 1 : ping, {1 3 0 0 [] 0}
3 tells 0 : ping, {1 3 0 0 [] 0}
3 tells 4 : ping, {1 3 0 0 [] 0}
3 tells 2 : ping, {1 3 0 0 [] 0}
4 tells 3 : pong, &{1 true 0}
0 tells 3 : pong, &{1 true 0}
1 tells 3 : pong, &{1 true 0}
2 tells 3 : pong, &{1 true 0}
0 says: my current state, { 1 , 3 , [] }
1 says: my current state, { 1 , 3 , [] }
2 says: my current state, { 1 , 3 , [] }
3 says: my current state, { 1 , 3 , [] }
4 says: my current state, { 1 , 3 , [] }
... Passed
```