NuRaft: A Python Implementation of Raft

Xuangui Huang David Stalfa

June 8, 2022

1 Introduction

NuRaft is an implementation of the central components of the Raft consensus protocol. This report documents the work done in constructing NuRaft, as well as the reasoning behind several design decisions made in the course of project.

Section 1 provides a brief overview of Raft, the functionality we chose to build on top of the Raft protocols, and the components of Raft that were implemented in NuRaft. Section 2 describes in more detail the functionality of our program, some technical details of the implementation, and any deviations of NuRaft from the proposed project. Section 3 reviews some design decisions that were made in the course of constructing the program and provides some rationale for those decisions. Finally, Section 4 describes some of the tests that were run on NuRaft to ensure it works correctly.

Raft Overview The Raft consensus protocol is Diego Ongaro's and John Ousterhout's [4, 3] attempt at defining a more understandable alternative to Paxos. Raft, like Paxos, is a protocol for coordinating a replicated log across many servers. A client may read or write to this log and the changes are made in parallel across the whole set of servers. A carefully designed protocol is needed in this model due to latency in communication, faulty servers, or a faulty network which drops messages. Therefore, these changes must be propogated carefully in order to ensure that the logs of all servers remain consistent.

Raft's main protocols can be understood through three basic concepts: strong leadership, randomized leader elections, and joint consensus for membership changes. Raft uses a notion of strong leadership in log replication and in client interaction. For log replication, the leader forces the log's of all other servers to match its own. For client interaction, Raft enforces that all client requests are directed to leader. Raft uses randomized leader elections to simplify the election process. Follower servers have randomly set election timers, which help ensure that no two servers timeout at the same time meaning that, typically, the first server to timeout will win an election before any other servers detect a failure. Finally, membership changes are done through a joint consensus protocol, which uses an intermediate phase in which the leader is managing separate majorities in old and new configurations. One key aspect of the Raft membership change is that the system remains online throughout the change. That is, the leader can continue to service client requests throughout a membership change.

Two further parts of Raft are persistent storage and log compaction. Servers frequently save parts of its current states to the persistent storage so that they can use this storage to restore their logs when they recover after a crash. Log compaction in Raft is done by servers independently. To compact its log, a servers stores in persistent memory its current state machine state along with the last log entry and the most current configuration. The server then deletes all committed log entries.

Underlying State Machine NuRaft implements a key-value store on top of the Raft protocols. This means that each server should hold a replicated key-value store identical to the store held on all other servers (the Raft protocols help ensure the stores are eventually the same). A client can interact with the system to put a key-value pair in the store or get the value from the store by a key.

Implemented Protocols NuRaft implements the log replication, leader election, persistent storage, client interaction, and membership change protocols. For membership changes, joint consensus is used only to add servers. For removing servers, NuRaft uses a single-server removal described in Section 2.1. Persistent storage allows for a crashed server to recover and restore its log to the last state it was in before crashing.

2 Implementation

In this section, we describe how the protocols described in Section 1 are implemented in NuRaft. In Section 2.1 we describe the functionality of NuRaft. In Section 2.2 we describe the technical details of NuRaft. Finally, in Section 2.3 we describe the deviations in NuRaft from the project proposal.

2.1 Functionality

In this section, we describe the functionality of NuRaft. Below we list the protocols that were implemented as well as some details of the implementations. The parenthesized name after each protocol indicates who was responsible for implementing it.

- Log Replication (Xuangui)

  The leader forces the logs's of all followers to match its own. This is achieved through the use of an AppendEntry message from the leader to each follower f, which allow the leader to do two things. First, the leader learns the last entry at which the its own log matches the log of f. Second, once the logs match up to a certain index, the leader overwrites all following indices in f's log with the corresponding entries in its own log.

- Leader Election (Xuangui)

  Leaders maintain their position through the use of empty AppendEntry messages, which serve as heartbeats. Each follower f holds a randomized timeout parameter t_f (different for each follower) where $t_f \in [T, 2T]$ and T is the minimum election timeout. If f does not receive a heartbeat within time t_f of the last heartbeat, then it assumes the leader has failed and starts a new election. f starts an election by voting for itself and sending out RequestVote messages to all other servers in the configuration. Each server can vote for at most one server in an election. If a server votes for a candidate c, then the server sends c a RequestVoteReply message indicating that the vote was granted. If a candidate receives votes from a majority of servers in the configuration then it becomes the leader. If a server does not receive enough votes and does not receive heartbeats from an elected leader, then eventually it will restart the election after the election timer times out.

  NuRaft assumes that server 0 is the leader in the first term, term 0.

- Persistent Storage (Xuangui)

  Each server makes a persistent copy of its log, current term, and current vote each time before it responses and commits. Upon recover, the server can restore its log to its state prior to crashing by using the `-r` option in its startup command.

- Client Interaction (David)

  - Find Leader

    In NuRaft, the Client initially assumes the leader is server 0. On sending a request to a a server s it believes is the leader, the request either (a) times out or (b) is told that s is not the leader. In case (a), the client randomly choose server from the configuration to believe is the leader and retries the message to that server. In case (b), the reply includes the identity of a server s believes is the leader and the client retries the message at this server.

- Issue Get and Put Requests

  The client uses a queue to store user commands, and then issues these as requests to the network. This allows the client to receive commands faster than it can process them and submit the requests in the order they were given to the client.

- Issue Add and Remove Requests

  In NuRaft the client issues add and remove requests to the system. In this way, NuRaft combines the role of client and system administrator. These roles could easily be separated since the code to handle membership change requests is mostly separate from that which handles read/write requests.

  The client stores a separate queue which stores the user commands to add or remove servers and issues these requests to the system in the order in which they were given. These requests are made in parallel with the read/write requests, so several read/write requests could be made between two add/remove requests.

  The separate queues also means that we cannot reason about the order of read/write commands and add/remove requests. For example, if the user begins by issueing several put commands and then an add command, the add command may get executed before some of the put requests because it immediately is put at the front of its queue. This behavior seems desirable in that it mimics what would happen if there were a separate system administrator issuing membership change independently of the client.

- Membership Changes

  NuRaft supports the addition and removal of servers from the configuration. All servers added or removed are from a pool of servers with known addresses. Specifically, NuRaft supports adding servers with addresses *vdi-linux-0*$\langle n \rangle$*.ccs.neu.edu* for $n = 30, \ldots, 39$ from the server pool, which is set in `config.py`. Any servers that are added can also be removed.

  NuRaft uses different protocols for adding and removing servers: one protocol supports adding multiple servers and the other supports removing a single server. The decision to use different protocols was done mostly for pedagogical reasons. The protocol to add/remove multiple servers has better safety properties, but the algorithm is slightly more complicated to impelement. The protocol to add/remove a single server uses a simpler algorithm, but there are known bugs in the published specification which have to be accounted for. We thought it would be helpful for us to understand both approaches to the problem.

  - Add Multiple Servers (Xuangui)

    NuRaft supports adding multiple servers to the configuration without interruption in availability of the replicated store. This is initiated with a call `add` $\langle n_1 \rangle$ $\langle n_2 \rangle \ldots$ to the client, where n_i is the integer index of a server you would like to remove.

    To safely add the servers without interruption in availability, NuRaft uses Raft's joint consensus algorithm. This algorithm runs in two phases. Suppose the configuration $\{a, b, c\}$ wants to add two servers, d and e, and so transition into the configuration $\{a, b, c, d, e\}$. In the first phase of joint consensus, the intermediate configuration $\{\{a, b, c\}, \{a, b, c, d, e\}\}$ is committed. During this period, any requests must be approved by majorities in *both* the old and new configurations. Once the intermediate phase is committed, the second phase begins in which the new configuration $\{a, b, c, d, e\}$ is committed. Committing this configuration completes the membership change.

  - Remove One Server (David)

    NuRaft also supports removing a single server. This can be done with a call `rem` $\langle n \rangle$ where n is the index of the server you would like to remove.

    To safely remove the server without loss of availability, NuRaft uses a two phase process. The first phase commits a no-op entry in the log. This ensures that the logs' of any servers outside the configuration are not up to date with the majority of servers in the configuration [2]. The second phase commits the new configuration with the requested server removed.

– Ignoring Disruptive Servers (David)

NuRaft also allows for servers outside the configuration to run in and send messages without disrupting the availability of the system. This is a desirable feature if servers are being adding to the configuration, and is essential if servers are being removed. When adding servers, we would like to be able to start them up and possibly set their configurations *before* adding them to the configuration, but we would not want them to start broadcasting messages which interrupt availability (e.g. by starting unnecessary elections). Similarly, when removing servers we need to know that the removed server will not continue broadcasting to the configuration and interrupting availability.

NuRaft is able to ignore disruptive, out-of-configuration servers by carefully timing when each server will accept messages which request votes. Specifically, each server's randomized election timer has a minimum value, say $t_{\min}$. If any server receives a request vote message with time $t_{\min}$ of the last received heartbeat, then the server rejects the request. Otherwise, the server grants the request (assuming it meets the criteria for granting). This ensures that the only times an out-of-configuration server is able to disrupt a healthy leader is if that leader's heartbeats have already been delayed by time $t_{\min}$ (and so some server might already have timed out anyway). Thus the availability of the system is unaffected.

2.2 Technical Details

NuRaft is implemented in Python 3.6 using the Asynchronous I/O (asyncio) library. We know of nine previous implementations of Raft, only one of which uses the asyncio library [1]. We implement the communications using connectless UDP, making it compatible with asyncio, using the newly-introduced features of futures and coroutines. Our implementation also allows for simulating dropped or delayed messages. The client and servers run on Northeastern's VDI's. All addresses are of the form `vdi-linux-0`$\langle n \rangle$`.ccs.neu.edu` with servers using addresses with $n = 30, \ldots, 39$ and the client using server $n = 40$. The client accepts commands through standard input.

2.3 Changes from Proposal

In constructing NuRaft, there were only minor changes from the proposed project. First, since Xangui is, it turns out, a much more experienced and faster programmer in Python than David, more of the initial server side functionality fell to him, while David implemented the client program and the final steps (single-server removal and ignoring disruptive servers), prepared the slides, and wrote the report. Second, because of compatibility issues with David's computer, it was impossible to use Docker for the project as proposed, and so it was decided to use Northeastern's VDI's. Third, in the proposal, after finishing basic functionalities we planned to implement log compaction first and then membership changes if time permits, but it turns out that membership changes is a much more interesting functionality in Raft so we decided to invest all of our remaining time to implement it properly.

3 Design Decisions

In this section, we describe and rationalize some of the decisions that were made in the design and implementation of NuRaft.

- Joint Consensus for Add-Only Membership Changes

  In implementing the Raft joint consensus algorithm, we decided to only allow it to support adding multiple servers to a configuration. We verified that, even in this weaker scenario, the configuration cannot be committed directly and something like joint consensus is needed.

  Consider the example given in Figure 1, where the current configuration has three servers and four servers are added. Suppose we tried to make this configuration change directly. In this case, if the

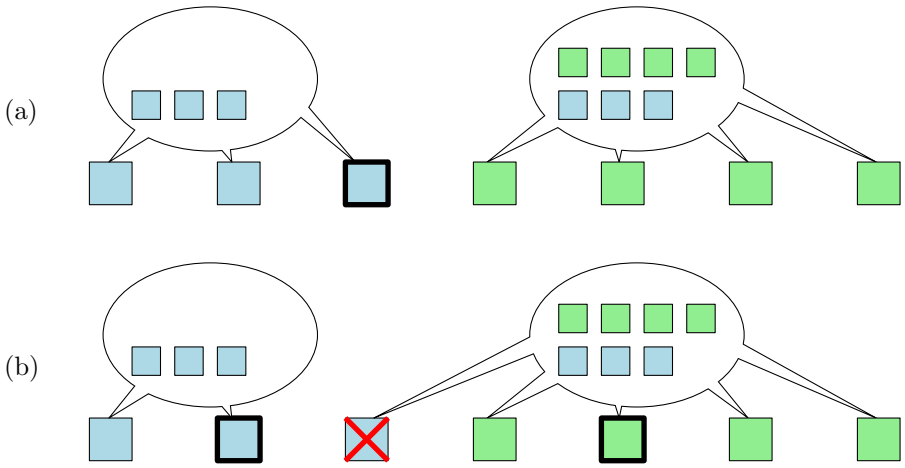

Figure 1: Old servers are shown in blue, new servers in green. In both figures, the elected leader are shown with a heavy border. (a) shows the old (blue) configuration before the change, with the new (green) servers waiting to be added. (b) shows the election of separate leaders after the old leader received the request to add the new servers, and then immediately crashed.

old leader crashes, then there is no way for the other old servers to learn about the new configuration. Since there are more added servers than old servers, both sets believe that they can attain a majority and so each set may elect its own leader.

- Storing Client Message ID's

  The Raft specification calls for servers to store client message ID's in the logs along with the logged client requests. NuRaft does not have the servers store this information. Here we explain why this is justified.

  In general, it is necessary to store client message ID's to avoid a situation in which client requests are executed twice. Consider a state machine that stores a counter. In this case, the client may request to increment the counter, and this request may be successfully committed. However, before the leader can respond with success, the leader fails. Then, according to the Raft protocol, the leader will resend the request to the new leader. If the servers are not storing the client message ID's, then this increment will occur twice, and the system will not behave correctly.

  Our system, however, holds a key-value store. In this case, a client message can be processed many times and the result is the same as if it were processed exactly once. Figure 2 depicts this situation. For this reason, we do not store client message ID's in the log, and allow client messages to be processed many times.

- Other Design Decisions

  - Put and Get Requests are Handled the Same by the Servers

    The Raft specification calls for optimizing read-only (get) requests by just exchanging a single heartbeat and then responding to the client directly rather than committing the request. Because of our architecture, this optimization would have been nearly as costly as committing each get request, so we opted for the conceptually simpler model where all request are handled the same way.

  - One Log Entry is Appended at a Time

    The Raft specification calls for optimizing appends by appending entries in batches. We chose not to implement this optimization to keep out code simpler.

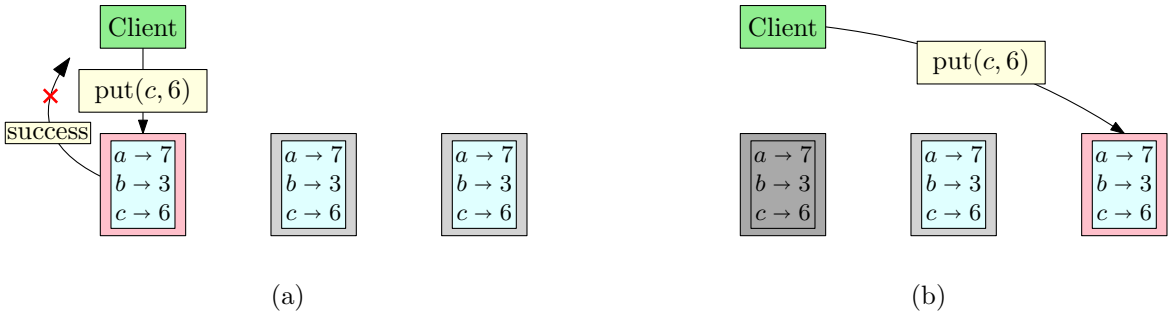

Figure 2: Servers are shown along the bottom and the client at the top. A red bordered server denotes the leader, and a dark gray server indicates that it has crashed. In (a), the client requests that $(c, 6)$ be put into the store. The request is granted, but the leader crashes before it can report success. In (b), the leader re-sends its request to the new leader, and the resulting store is unchanged.

- Membership Change Requests are Made from Client

  In general, membership change requests would be made from a system administrator. We chose to merge these two roles into the client role in order to simplify testing. Section 2.1 discusses the implications of this decision in more detail.

- All Servers are Taken from a Pool of Servers with Known Addresses

  NuRaft does not support using servers with arbitrary addresses. All servers used in the system must be taken from the pool of servers described in Section 2.2. This was done to simplify the code and free us up to focus on what we saw as being the more fundamental aspects of Raft.

- Commit No-Op Log Entry Before Removing Server

  Raft calls for committing a no-op log entry immediately after removing a single server. This ensures that a majority of servers in the new configuration have more up-to-date logs than the server that was just removed. Specifically, the problematics case occurs when several servers are removed without any other log entries being committed in between [2]. We address this problem by committing a no-op entry immediate *before* each server removal. That way, if several servers are removed in sequence, there will still be log entries separating their logs from the majority of remaining servers. Since only one server is being removed at a time, it can never achieve a majority vote to be elected leader against the remaining servers.

4 Testing

Many tests were run on NuRaft to test log replication, leader election, persistent storage, putting, getting, and adding servers. Some recorded tests are linked below with descriptions. We have many parameters in `config.py` that can be changed for testing.

1. `https://drive.google.com/file/d/1vwxE09CsJoMIZ-1j9xg5hNvnCJNwOIAS/view?usp=sharing`

   Here we test the general functionalities of the system in an integration test. We test subsequently the cases where key-values pairs are put into the store, values are retrieved by key, accessing non-existent keys, leaders crash, a minority of servers crash, a majority of servers crash, servers recovers, servers are added, a much larger minority of servers crash after configuration changes. In these cases the system remains online, available, and consistent, and the client can find the leader and interact with it. The system blocks if a majority of servers crash, but if these servers recover then the system becomes available again.

2. `https://drive.google.com/file/d/1we4QX9g0P_KZiqyHGwliSJmmu6tWnHpo/view?usp=sharing`

In this test, we slow down joint consensus such that there is lag between commitment of the intermediate and the new configurations. During this delay, we make several get/put requests to show that the system remains available and consistent during the change.

3. `https://drive.google.com/file/d/1PWhHiyuOtOEqNMZirqFsyxgGTR2QwJb5/view?usp=sharing`

   Here we test that joint consensus requires majorities from both the old and new sets of servers. We, again, slow down joint consensus and crash a majority of the old servers to show that, in this case, the system blocks.

4. `https://drive.google.com/file/d/1_slALcMnZH8yNUO2sg00BHBhTjNwocaX/view?usp=sharing`

   Here, we test that candidate servers whose logs are not up-to-date cannot be elected leader. We slow down the leader's heartbeats to that the election timers' constantly run out. Further, we crash server's strategically to ensure that each server makes one attempt to become leader. When the server with the too short log attempts to become leader, its request vote message is rejected.

A similar test to Test 1 was run to test removing a single server and ignoring disruptive servers, but since we completed this part of the code fairly late we did not have time run extensive tests.

References

[1] The raft consensus algorithm. `https://raft.github.io/`.

[2] Diego Ongaro. bug in single-server membership changes. `https://groups.google.com/forum/#!msg/raft-dev/t4xj6dJTP6E/d2D9LrWRza8J`.

[3] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014. (work in progress).

[4] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm (extended version), 2014. `https://raft.github.io/raft.pdf`.