- Extended Due: Oct 3, 2023, 23:59 PDT
- Original Due: Sep 28, 2023, 23:59 PDT

Introduction

This is the first in a series of labs in which you'll build a fault-tolerant key/value storage system. In this lab you'll implement Raft, a replicated state machine protocol. In the next lab you'll build a key/value service on top of Raft.

A replicated service achieves fault tolerance by storing complete copies of its state (i.e., data) on multiple replica servers. Replication allows the service to continue operating even if some of its servers experience failures (crashes or a broken or flaky network). The challenge is that failures may cause the replicas to hold differing copies of the data.

Raft manages a service's state replicas, and in particular, it helps the service sort out what the correct state is after failures. Raft implements a replicated state machine. It organizes client requests into a sequence, called the log, and ensures that all the replicas agree on the contents of the log. Each replica executes the client requests in the log in the order they appear in the log, applying those requests to the replica's local copy of the service's state. Since all the live replicas see the same log contents, they all execute the same requests in the same order, and thus continue to have identical service state. If a server fails but later recovers, Raft takes care of bringing its log up to date. Raft will continue to operate as long as at least a majority of the servers are alive and can talk to each other. If there is no such majority, Raft will make no progress, but will pick up where it left off as soon as a majority can communicate again.

In this lab you'll implement Raft as a C++ class with associated methods, meant to be used as a module in a larger service. A set of Raft instances talk to each other with RPC to maintain replicated logs. Your Raft interface will support an indefinite sequence of numbered commands, also called log entries. The entries are numbered with *index numbers*. The log entry with a given index will eventually be committed. At that point, your Raft should send the log entry to the larger service for it to execute.

Your Raft instances are only allowed to interact using RPC. For example, different Raft instances are not allowed to share variables. Your code should not use files at all.

Objective

In this lab you'll implement most of the Raft design described in the extended paper. You will not implement: saving persistent state, cluster membership changes (Section 6), log compaction / s ⓘ shotting (Section 7).

- Read and understand the Raft paper and the Raft lecture notes before you start. Your implementation should follow the paper's description closely, particularly Figure 2, since that's what the tests expect.

Getting Started

Checking out code

Use the following link to get the assignment from GitHub Classroom.
https://classroom.github.com/a/TkjESYSH

Setup

First, make sure you are working on the right branch. Make sure you use `lab-raft-solution` as the branch name.

```
$ git checkout -b lab-raft-solution origin/raft-23fa
$ git submodule update --init
$ git push -u origin lab-raft-solution
```

We supply you with skeleton code and tests in `src/deptran/raft`.

Before compiling, you need to install all needed software dependencies.

```
sudo apt-get update
sudo apt-get install -y \
    git \
    pkg-config \
    build-essential \
    clang \
    libapr1-dev libaprutil1-dev \
    libboost-all-dev \
    libyaml-cpp-dev \
    libjemalloc-dev \
    python2 \
    python3-dev \
    python3-pip \
    python3-wheel \
    python3-setuptools \
    libgoogle-perftools-dev
sudo pip3 install -r requirements.txt
```

To get up and running, execute the following commands:

```
$ python3 waf configure build --enable-raft-test # compile
```

```
TESTS FAILED
```

ARM Mac users (unsupported by staff):

This lab is tested on x64 version of linux, but you may try ubuntu ARM as well. It looks the Multipass setup on this website installs aarch64 Ubuntu for ARM Mac computers.
For compilation, please use the following command instead.

```
$ python3 waf configure build --enable-raft-test --boost-libs=/usr/lib/aarch64-linux-gnu
```

The Code

Most of the Raft implementation will be located in `src/deptran/raft/`. In this directory, there are 3 `.cc` files (and respective headers) that you need to modify:

- `server.cc` and `server.h`
- `commo.cc` and `commo.h`
- `service.cc` and `service.h`

Besides these three files, there is also `src/deptran/raft/raft_rpc.rpc`, where you will find `abstract service Raft` containing sample prototypes for the `RequestVote` and `AppendEntries` RPCs. Here, you can modify the arguments and return values for the `RequestVote` and `AppendEntries` RPCs and add any other RPCs as you see fit.

Do not create new files, and do not modify any existing files not mentioned.

RPCs

Class `RaftServiceImpl` (`service.cc`) implements receiver-end handlers for the RPCs declared in `src/deptran/raft_rpc.rpc`. To register a handler for an RPC in `RaftServiceImpl`, use the `RpcHandler` macro.

`RpcHandler` usage: `RpcHandler(RPC_NAME, N_PARAMS, PARAMS...) { DEFAULTLOGIC }`

- `RPC_NAME`: should match the name of the RPC declared in `raft_rpc.rpc`
- `N_PARAMS`: number of RPC arguments + number of RPC return values
- `PARAMS`: the RPC arguments and return values in the same order as in `raft_rpc.rpc`, with comma parations between the type and name.
- `DEFAULTLOGIC`: write code to assign default values to the RPC's return value in these brackets

when they happen.

Class `RaftCommo` (`commo.cc`) is meant to handle sending RPC requests. See the example sender functions to understand how to send RPCs in `RaftCommo`.

Server logic

Class `RaftServer` (`server.cc`) is your starting point for writing most of the server logic.

Some useful member variables (inherited from a parent class of `RaftServer`):

- `loc_id_`: id of the server.
- `commo()`: function that returns the server's `RaftCommo` instance.
- `mtx_`: a `std::recursive_mutex` you can use for protecting data from concurrent access.

Required Functionality

There are a few specific functionalities in `RaftServer` you need to implement in order to pass all tests.

```
bool Start(shared_ptr<Marshallable> &cmd, uint64_t *index, uint64_t *term)
```

- Implement this member function
- If server is not the leader, return false
- Else, start agreement on `cmd` in a new log entry, set `index` and `term` with the server's current index and term, and return true

```
void GetState(bool *is_leader, uint64_t *term)
```

- Implement this member function
- Populate is_leader with true if the server thinks it is the leader
- Populate term with the server's current term number

```
function<void(Marshallable&)> app_next_
```

- A member variable of a parent class of `RaftServer`.
- Each server must pass each committed command to `app_next_` exactly once, in the correct order, as soon as each command is committed on each server.

Your first implementation may not be clean enough that you can easily reason about its correctness.

implementation.

You are recommended to do this lab following these two steps:

Part 1A

Implement leader election and heartbeats (AppendEntries RPCs with no log entries). The goal for Part 1A is for a single leader to be elected, for the leader to remain the leader if there are no failures, and for a new leader to take over if the old leader fails or if packets to/from the old leader are lost.

- Add any state you need to the RaftServer class (but do not use static member variable to share states between different `RaftServer` instances) in server.cc. You'll also need to define a struct to hold information about each log entry. Your code should follow Figure 2 in the paper as closely as possible.
- In `RaftServer`, create the function to handle `RequestVote`. Fill in the `RequestVote` RPC handler in `service.cc`, connect the RPC handler and the `RaftServer`.
- When initializing the `RaftServer`, create a backgrond coroutine that will kick off leader election periodically by sending out RequestVote RPCs when it hasn't heard from another peer for a while. This way a peer will learn who is the leader, if there is already a leader, or become the leader itself.
- To implement heartbeats, implement a separate `EmptyAppendEntries` RPC (`AppendEntry` without log entries), and have the leader send them out periodically. Write an `EmptyAppendEntries` RPC handler method that resets the election timeout so that other servers don't step forward as leaders when one has already been elected.
- Make sure the election timeouts in different peers don't always fire at the same time, or else all peers will vote only for themselves and no one will become the leader.
- The tester requires that the leader send heartbeat RPCs no more than ten times per second.
- The tester requires your Raft to elect a new leader within five seconds of the failure of the old leader (if a majority of peers can still communicate). Remember, however, that leader election may require multiple rounds in case of a split vote (which can happen if packets are lost or if candidates unluckily choose the same random backoff times). You must pick election timeouts (and thus heartbeat intervals) that are short enough that it's very likely that an election will complete in less than five seconds even if it requires multiple rounds.
- The paper's Section 5.2 mentions election timeouts in the range of 150 to 300 milliseconds. Such a range only makes sense if the leader sends heartbeats considerably more often than once per 150 milliseconds. Because the tester limits the RPC count (you are recommended to set the heartbeat interval to 100ms), you will have to use an election timeout larger than the paper's 150 to 300 milliseconds, but not too large, because then you may fail to elect a leader within five seconds.

election is spread over multiple parts of the figure.

- A good way to debug your code is to insert print statements when a peer sends or receives a message, and redirect the output in a file `compile_cmd &> out`. Then, by studying the trace of messages in the out file, you can identify where your implementation deviates from the desired protocol. You might find `Log_info` and `Log_debug` in the framework useful to turn printing on and off as you debug different problems.

Part 1B

We want Raft to keep a consistent, replicated log of operations. A call to Start() at the leader starts the process of adding a new operation to the log; the leader sends the new operation to the other servers in AppendEntries RPCs.

Implement the leader and follower code to append new log entries. This will involve implementing Start(), completing the AppendEntries RPC structs, sending them, fleshing out the AppendEntry RPC handler, and advancing the commitIndex at the leader.

- You will need to implement the election restriction (section 5.4.1 in the paper).
- One way to fail the tests is to hold un-needed elections, that is, an election even though the current leader is alive and can talk to all peers. This can prevent agreement in situations where the tester believes agreement is possible. Bugs in election timer management, or not sending out heartbeats immediately after winning an election, can cause un-needed elections.
- You may need to write code that waits for certain events to occur. Do not write loops that execute continuously without pausing, since that will slow your implementation enough that it fails tests.

The tests for upcoming labs may fail your code if it runs too slowly. Try to write efficient code.

Be Caution & Hints

1. Try to avoid creating coroutines inside Start() function due to ill-prepared coroutine runtime.
2. Sleep the coroutine/thread when it seems necessary to avoid spinning and burning all CPU resources.
3. If you constantly see multiple elections happen almost at the same time throughout several consecutive tests or elections trigger frequently, even when servers are all live and stable, you may want to adjust the random number for the timer and heartbeat frequency. (Think about why?)
4. ⓘ p_next_ should be called on both leader and followers. (Why?)
5. Be aware of outdated RPC replies. Due to (simulated) network delay, the packet might be

information regarding all peers.

7. When calling "sending" functions from the `RaftCommo` instance, you can use some heuristic timer to wait for the reply. Reply won't be immediately available.

8. In the file "`src/rrr/reactor/event.h`", there are multiple useful event types provided. You can use them as you see fit.

Compile and Test

Compile:

```
$ python3 waf configure build --enable-raft-test
```

Test:

```
$ build/deptran_server -f config/raft_lab_test.yml
TEST 1: Initial election
TEST 1 Passed
TEST 2: Re-election after network failure
TEST 2 Passed
TEST 3: Basic agreement
TEST 3 Passed
TEST 4: Agreement despite follower disconnection
TEST 4 Passed
TEST 5: No agreement if too many followers disconnect
TEST 5 Passed
TEST 6: Rejoin of disconnected leader
TEST 6 Passed
TEST 7: Concurrently started agreements
TEST 7 Passed
TEST 8: Leader backs up quickly over incorrect follower logs
TEST 8 Passed
TEST 9: RPC counts aren't too high
TEST 9 Passed
TEST 10: Unreliable agreement
TEST 10 Passed
TEST 11: Figure 8
TEST 11 Passed
ALL TESTS PASSED
Total RPC count: 6609
```

The source for the tests is in `src/deptran/raft/test.cc` and `src/deptran/raft/testconf.cc`.

Submit

do not push any commits or changes after the official deadline. We will check out the latest commit that comes before the deadline for the grading.

DO NOT

Make sure you do not do any of the following; otherwise you will fail the labs, even if the code may pass the testing scripts.

- Change/create files other than the files allowed.
- Use shared variables between Raft instances.
- Use file/network/IPC interface other than the RPC interface given to you.

Acknowledgment

Thank Shuai Mu, Julie Lee for sharing this lab. It was originally inspired by MIT 6.824, and the code is based on the academic prototypes of previous research works, including Rococo/Janus/Snow/DRP/Rolis/DepFast.