Implementing Raft's log consensus algorithm

Deadline 1: Wednesday, 18th November, 2020 at 11:55 PM

Deadline 2: Thursday, 26th November, 2020 at 11:55 PM

This is the second assignment, in a series of assignments, in which you will build a fault-tolerant key-value storage system. In the last assignment (assignment 2), you have implemented Raft's leader election algorithm. In this assignment, you will implement most of Raft's core features, such as the **log consensus algorithm**. Your implementation will be a key-value service that uses Raft's implementation as a building block.

We use Raft to keep a consistent replicated log of the operations. Raft servers accept client operations and insert them into the log. Only the leader is allowed to append to the log and should disseminate new entries to other servers by including them in its outgoing *AppendEntries* RPCs.

It is very important that your understanding of the Raft is crystal clear. You are highly encouraged to go back to reread the extended Raft paper, the Raft lecture slides, and the illustrated Raft guide. You should, of course, also review your work from Assignment 2, as this assignment directly builds off that.

This assignment is divided in two parts:

1. Raft's consensus algorithm
2. Fault tolerance with persistent storage (server crashes and then restarts)

Note: you are required to submit **ONLY** your implementation of the Raft.

Note: This assignment can be done in **pairs.**

Note: You should post any assignment related query ONLY on piazza. Do not directly email the course staff.

Note: Course policy about **plagiarism** is as follows:

- Students must not share the actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection.
- Students cannot copy code from the Internet.
- Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Part 1 – Consensus algorithm

In the first part of this assignment, you will implement the Raft consensus algorithm as described in Raft's extended paper. You won't have to implement the cluster membership changes, as described in section 6, and log compaction/snapshotting as described in section 7 of the Raft extended paper.

You can start by implementing the leader and follower code to append new log entries. You will have to implement the *start*() API, completing the *AppenEntries* API with relevant data structures. In the beginning, your goal should be to pass the basic tests, for example, *TestBasicAgree*() test (in test_test.go). Once the basic test is passed, try to pass all the tests until the basic persistence test.

You can only use RPCs (provided to you in the **labrpc** folder) for communication between different instances of the Raft. You are not allowed to share Go variables between the different instances of the Raft. Your implementation shouldn't use files.

Part 2 – Fault tolerance

The second part of this assignment is related to fault tolerance in Raft protocol. At this stage, your implementation should be robust to handle failures, such as server crashes, and then restart. Raft servers must be able to pick up from where it left off before restart. This means your Raft's implementation should store state on the persistent storage every time a state update happens. Please see Figure 2 of the paper which mentions which state should be stored on persistent storage.

In real situations, fault tolerance is achieved by saving state to the non-volatile memory on each state update and reading state back from the non-volatile memory on reboot. However, for this assignment, you are not required to write state to non-volatile memory. You will use the persister object (provided in persister.go file) to save and restore the state. When the tester code calls *Make*() (in raft.go), it provides a persister object which holds the most recent state of the Raft (if any). The Raft initializes its state from the persister and uses it to save its state every time a state update happens. You can use *readRaftState*() and *SaveRaftState*() for reading and writing state to persister object respectively.

To save state through the persister object, you will have to serialize the state before calling *persist*() API. Similarly, you will have to de-serialize the state after reading the state through the *readPersist*() API. You will also need to determine the places where storage to non-volatile memory is required and where to read state from the non-volatile memory. You can find some example code on how to use *persist*() and *readPersist*() in the raft.go file.

Requirements

Your implementation must meet the following requirements:

1. This project can be done in pairs. You are not allowed to use any code that you have not written yourself.
2. You **MUST** format your code using gofmt and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

Starter code

You are provided with a skeleton code in the **raft** and **labrpc** folders. You won't need to modify anything in the **labrpc** folder. You will use labrpc (the RPC like system) for

communication between the Raft peers. Your implementation will be in the raft.go file in the **raft** folder. Understanding the skeleton code may take some time, therefore it is highly recommended to go over the provided code and digest it before starting the implementation.

```
└── src
    ├── labrpc
    │   ├── labrpc.go (RPC like system using go channels, no need to modify)
    │   └── test_test.go (For testing RPCs, no need to modify)
    └── raft
        ├── config.go (Used by test cases, no need to modify)
        ├── persister.go (APIs for persistent storage, no need to modify)
        ├── raft.go (Implement the APIs provided in this file)
        ├── test_test.go (Test cases for your implementation, no need to modify)
        └── util.go (Some helpful utility functions, no need to modify)
```

For instructions on how to build, run, test, and submit your Raft implementation, see the **Logistics** of this document.

Evaluation

You can earn up to 35 points from this assignment. There is no extra credit or bonus.

Deadline 1:

1. **Test basic agreement** - 2 points
2. **Test agreement despite a follower failure** - 2 points
3. **Test no agreement, too many followers fail** - 3 points
4. **Test concurrent start agreement requests** - 3 points
5. **Test rejoin of a partitioned leader** - 3 points
6. **Test leader's backup on incorrect follower log** - 3 points

Deadline 2:

1. **Test persistent storage 1** - 3 points
2. **Test persistent storage 2** - 3 points
3. **Test persistent storage 3** - 3 points
4. **Test figure 8** - 5 points

Code Style (Deadline 1 & 2):

1. **Go Formatting** - 1 point
2. **Manual Grading** - 4 points

There will be no hidden test cases in this assignment.

Late Submissions

You should submit your assignment on LMS before its due time.

If you submit your work late, we will award you a fraction of the score you would have earned on the assignments had it been turned in on time, according to this sliding scale:

- 90% for work submitted up to 24 hours late
- 80% for work submitted up to 2 days late
- 70% for work submitted up to 3 days late
- 60% for work submitted up to 5 days late
- 50% for work submitted after 5 days late

For example, if you should have earned 8/10 points but submitted 36 hours late, you will instead earn 6.4 points.

Besides the above policy, **you are allowed five "free" late days during the semester** (that can be applied to one of assignments 1 through 3; the final assignment will be due tentatively on the final day of classes [12 Dec] and cannot be turned late. **The last day to do any late submission is also the final day of classes, even if you have free late days remaining**). You do not need to tell us that you are applying your "late day" -- we'll remove the late penalty at the end of the semester from the assignment that benefits you the most.

Advice

- All the field names used within the data structures, which you will be sending over the RPCs (e.g., information about each log entry), should be in capital letters.
- Similar to how the RPC system only sends structure field names that begin with upper-case letters, and silently ignores fields whose names start with lower-case letters, the GOB encoder you'll use to save persistent state only saves fields whose names start with upper case letters. This is a common source of mysterious bugs since Go doesn't warn you.
- While the Raft leader is the only server that causes entries to be appended to the log, all the servers need to independently give newly committed entries to their local service replica (via their applyCh provided through *Make*() by the tester). Because of this, you should try to keep these two activities as separate as possible.
- To avoid running out of memory, Raft must periodically discard old log entries, but you do not have to worry about garbage collecting the log in this assignment.

Logistics

Running the official tests

To test your submission, we will execute the following command from inside the `$GOPATH/src/raft` directory:

```
$ go test
```

We will also check your code for race conditions using Go's race detector by executing the following command:

```
$ go test -race
```

To execute a single unit test, you can use the **`-run`** flag and specify a regular expression identifying the name of the test to run. For example,

```
$ go test -run TestBasicAgree
```

To run a particular test with a check on race condition, use –race flag with –run flag, for example,

```
$ go test -race -run TestBasicAgree
```

Submission

Your implementation should be in the ***raft.go*** file only. Rename ***raft.go*** to ***raft_<R>.go*** (where ***<R>*** should be replaced by your roll number, e.g. ***raft_17030010.go or raft_18030010_19030010.go*** in case assignment done in pair) before uploading it to the LMS. Please submit your code with all print statements removed.