Project 3: A Paxos Application

1 Overview

This project is due on **Wednesday, December 9th, 2015 at 11:59pm**. There is one checkpoint that is due on **Monday, November 30, 2015 at 11:59pm**. Keep in mind that there will be no office hours over Thanksgiving, so you will want to start early. For more information regarding what portion of the project is expected to be completed for the checkpoint, please refer to section 3.

The starter code for this project is hosted as a read-only repository on GitHub. For instructions on how to build, run, test, and submit your server implementation, see the `README.md` in the project's root directory. To clone a copy, execute the following Git command:

```
git clone https://github.com/cmu440-F15/p3
```

You must work on this project with **a partner who is also in the class**. You do not have to work with the same partner as you did in Project 2. As per course policy, you may turn the project in up to **two** days late, and each day will incur a 10% deduction from your score.

2 Paxos (65%)

The main focus of this project will be for you to implement the Paxos algorithm that was described in class. The Powerpoint for lecture 16 (Distributed Replication) gives a detailed description of Paxos.

2.1 Basic Functionality

In terms of the given API, your `PaxosNode` is one node in a ring of nodes that functions as a storage system. Processes interact with a node by making `GetNextProposalNumber`, `Propose` and `GetValue` RPC calls.

- `GetNextProposalNumber`: returns the next proposal number for a node. `GetNextProposalNumber` should always be called before `Propose`, and the result should be passed to `Propose` as the proposal number. Recall that no two nodes should propose with the same

proposal number, and for a particular proposer, proposal numbers should be monotonically increasing.

- `Propose`: adds a (key, value) pair. This function responds with the committed value in the reply of the RPC, or returns an error. It should not return until a value is successfully committed or an error occurs. Note that the committed value may not be the same as the value passed to `Propose`.
- `GetValue`: gets the value for a given key. This function should not block. If a key is not found, it should return `KeyNotFound`.

Your node must be able to function as a proposer, acceptor, and learner. In this implementation of Paxos, all nodes will be acceptors and learners. Therefore, a proposer should send proposals and commits to all nodes in the ring. Nodes also communicate with each other using RPC calls. For example, if a node wants to send a prepare message, it should call the `RecvPrepare` method on all nodes.

A node is created by calling `NewPaxosNode`. This function should not return until it has established a connection with all nodes in the map of server ID to hostport which will be passed to the function. The map includes all nodes, including the one being initialized. If a node fails to connect with another node, it should sleep for one second, and try again. The number of retries will also be specified as an argument. As a small simplification, each `srvId` will be a number from 0 to `n-1` where `n` is the number of Paxos nodes.

2.2 Key-Value Store

You may have noticed that typical Paxos implementations simply store values instead of key-value pairs. We want to take advantage of this fact by being able to store distinct keys concurrently. This means that for some pair of distinct keys, the process of agreeing on the value associated with one key should be independent of the process of agreeing on the value for the other key. For example, if Node 0 proposes a pair `(key1, val)` and Node 1 proposes a pair `(key2, val)`, Node 0 and Node 1 should not contend with each other. Since the keys are distinct, there should be separate instances of Paxos for each key, and thus separate bookkeeping for the highest proposal number seen, etc.

2.3 Failure Cases

Below, we have defined some common failure cases and their expected behavior:

- If a proposer can not achieve a majority (at least $\frac{n}{2} + 1$ nodes, where n is the total number of nodes), it will abandon the proposal. This includes the proposer failing to

become the leader during the Prepare phase, or faling to get a majority of accept-ok's during the Accept phase.

- `Propose` should return an error if a value has not been committed after 15 seconds.

We will NOT be testing you on adding additional nodes, recognizing dead nodes, handling dropped messages, nor be requiring that you ensure that a proposed key-value pair is always committed. However, you should still follow the liveness principle by ensuring that if there has been a proposal, *some* value will be committed.

2.4 Replacement Nodes

If a node dies, the system may create a new `PaxosNode` to take the place of the dead one. This new node will have the same `srvId` as the old node, and will be created with the `newPaxosNode` function with `replace` set as true. It is then up to you to add this node to the ring and teach this node all of the key-value pairs the other nodes have committed, thereby bringing it up to speed. To do so, we are asking you to implement two RPCs that your replacement node can call after being created.

- `RecvReplaceServer`: Acknowledges the existence of the replacement node.
- `RecvReplaceCatchup`: Returns an array of bytes to the replacement node.

3 Checkpoint (10%)

To get full points for this section, your code must pass the Autolab tests for the checkpoint by the due date, **Monday, November 30, 2015 at 11:59pm**. The tests will assume that there will be no dueling proposers and no dropped messages. The tests will not test node replacement. Essentially, all you need to have done is to go through the motions of sending out a proposal, having the rest of the nodes respond, sending out an accept, having the rest of the nodes accept it, sending out a commit, and having all nodes commit the key-value pair to storage. You must also be able to start up a node ring correctly, and implement generating unique proposal numbers.

4 Application Layer (20%)

After finishing your Paxos implementation, you must build a small application that uses your Paxos Key-Value store in some way. We are leaving this part of the project very

open-ended, so you have a lot of freedom in choosing what to build. You may use external libraries if you wish (excluding distributed systems libraries which implement Paxos for you). You may also use external packages and SDKs to support the application or user interface (i.e., GUI toolkits/frameworks, image/video/audio processing packages, online mapping APIs, etc.). This wiki page on writing web apps in Go might be useful: https://golang.org/doc/articles/wiki/. Some ideas from the Spring 2014 iteration of this class are:

- Projects with a real-time multi-user component are a good match for Paxos you could either have the individual users be part of the Paxos quorum, or use Paxos for the replicated state that stores their updates.
- Shared document editing, in the style of Google docs. The system should support real-time editing and viewing by multiple participants. Multiple replicas would be maintained for fault tolerance. Caching and/or copy migration would be useful to minimize application response time.
- A reservation system (airline, train, restaurant, etc)
- A multi-player real-time game.
- A low-latency notification system. E.g., watch a whole bunch of RSS feeds and send all subscribers an email when one is updated.
- Projects involving multiple agents or users coordinating updates that can be shown on a map (e.g., Google Maps).

To receive full credit, your application must use your Paxos implementation (can run multiple nodes) and we must be able to run it successfully on a 64 bit Linux or OSX machine. We will be evaluating your application based on how well thought out, interesting, and unique it is, as well as the code complexity required to implement it. Your application must heavily rely on Paxos or use it in an interesting way.

Your project should be reasonably documented. You will need to hand in a PDF describing what your application is, its functionality, and how we can run the application. You should describe and make a note of any scripts that should be used for running the application.

You are free to add additional directories and packages for your application, but you must not change the provided Paxos API, or remove any of the fields defined in the structs of `paxosrpc/proto.go`. This is to ensure compatibility with our test system. All of your application code must be included in the `paxosapp` directory, which you will be submitting to autolab.

5 Style (5%)

We will grade on the style of your code in this project. This means you should have well-structured, well-documented code that a TA can easily read and understand. For example, you should extract out utility methods into a separate file and You will also have received feedback from the course staff from Project 2 on what is considered good style. The Effective Go guide (https://golang.org/doc/effective_go.html) is a great resource to look at should you have any questions.

6 Starter Code

The starter code for this project can be found in the https://github.com/cmu440-F15/p3/src/github.com/cmu440-F15/paxosapp/paxos. You will need to implement the functions inside `paxos_impl.go` file as well as your application layer.

7 Testing

We have provided the basic checkpoint tests. However, we will not be releasing the tests for the final project evaluation. We will also not open up the submission for the final project until the checkpoint due date. You should write some of your own tests to check the functionality of your implementation. It will be up to you to thoroughly test your code before submitting on Autolab, though the number of submissions will be unlimited.

8 Hand In

You must register your group on Autolab before submitting anything!!!

8.1 Checkpoint

You will need to hand in a `paxosapp.tar` file of the `paxosapp` folder. You can generate this file by runnning the following command in the `p3/src/github.com/cmu440-F15` folder:

```
tar -cvf paxosapp.tar paxosapp
```

8.2 Full Project

You will need to hand in a `paxosapp.tar` file on Autolab of the `paxosapp` folder. Your folder should contain:

- Your complete Paxos implementation
- Your application code
- A PDF called `application.pdf` describing what your application is, what functionality it has, and **how we can run your code/use your application**.

9 Project Requirements

As you write code for this project, also keep in mind the following requirements:

- You must work on this project with only your partner. You are free to discuss high-level design issues with other people in the class, but every aspect of your implementation must be entirely you and your partner's own work.
- You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.
- You may use any of the synchronization primitives in Go's sync package for this project.