Paxos

Deadline 1: Wednesday, 13th November 2019 at 11:50 PM
Deadline 2: Sunday, 24th November 2019 at 11:50 PM

You will be implementing Paxos in this assignment for solving consensus in a network.

Note: process and node will be used interchangeably in this handout.

Note: the Evaluation section explains the deadlines.

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

- Students must not share 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.

Note: You can work on this assignment with a partner who is also in the class and submit the assignment in pairs or work individually. In case you submit the assignment as a pair, please make sure when you submit code files, names and roll numbers of both the students are written. Please note both the students should be able to explain any program code or documentation they submit.

1. Introduction

In this project you will implement the Paxos Algorithm to propose $< key, value >$ pair in a distributed storage system. For instructions on how to build, run, test, and submit your server implementation, see the Logistics section.

2. Paxos

The focus of this project will be for you to implement the Paxos algorithm that was described in class. An illustrated example of Paxos can be found at this blog to get you started. The slides for lecture 12 (on LMS) also give 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.

- `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`.
- `GetNextProposalNumber`: returns the next proposal number for a node. It 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 for a given key, and for a particular key in a particular proposer, proposal numbers should be monotonically increasing. However, because the proposals of different keys are independent, it's acceptable to have the same proposal number for 2 proposal of different keys.
- `GetValue`: gets the value for a given key. This function should not block. If a key is not found, the Status in reply should be `KeyNotFound`.

You also need to implement `RecvPrepare`, `RecvAccept` and `RecvCommit` RPC APIs in your Paxos implementation.

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 commit 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.

We provide much more detailed specification of all the API you need to implement in the starter code **paxos_impl.go**, please make sure you go through it carefully.

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 cannot achieve a majority (at least $\left\lfloor \frac{n}{2} \right\rfloor + 1$ nodes, where n is the total number of nodes), it should not return a non-nil error immediately. It should ideally return committed value from the contending proposal. This includes the proposer failing to become the leader during the Prepare phase, or failing to get a majority of accept-ok's during the Accept phase etc.
- `Propose` should return an error if a value has not been committed after 15 seconds.

As a simplification, 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.

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. You can also assume that when you try to restore, the data in other nodes are in consistent status – it will not be in the middle of the commit phase of some proposal.

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. Requirements

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

1. You must work on this project individually or **ONLY** with 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/or your partner's own work.
2. You may use any of the synchronization primitives in Go's `sync` package for this assignment.
3. 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.

4. Evaluation

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

Deadline 1 (total 10 points): the tests will assume that there will be no dueling proposers – proposers that proposes values for same key concurrently – 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 correct proposal numbers.

Deadline 2 (total 20 points): You need to implement your code not only correctly but also efficiently to pass these tests. We will test dueling proposers, dropped messages, proposal of multiple keys and continuous proposal of a single proposer etc. We also have tests for the replacement of nodes.

Please make sure you read Notes and Hints before you get stuck.

Deadline 1 & 2 (total 5 points)

We will manually grade the style of your code in this project. This means you should have **well-structured**, **well-documented** code that can be easily read and understood. For example, you should not have dead code and you should not write a lengthy function. The Effective Go guide is a great resource to look at should you have any questions.

Also note the following:

- Race conditions will be checked.
- You have a total pool of 5 days for late submissions, without deductions. Late days from this pool can be used for any of the four assignments. Once you have used up all the 5 days, no late submissions for the assignments will be accepted.

5. Logistics

You will need to implement the functions inside **paxos_impl.go** file.

We have provided the basic tests (for Deadline 1). However, tests for the final assignment evaluation (Deadline 2) will be given as executables.

You are encouraged to write some of your own tests as well to check the functionality of your implementation. It will be up to you to thoroughly test your code before submitting on LMS.

Starter code

The starter code for this assignment is contained in the folder **`paxosapp/`**. The code is roughly organized as follows:

```
paxos/            Implement Paxos

tests/            Source code for the official tests
    paxostest/    Tests your Paxos implementation

runners/          Used to launch an instance of your Paxos node
    prunner/

rpc/
    paxosrpc/     Paxos RPC helpers / constants

paxostest.sh  Shell script to run the tests (Deadline 1)
paxostest_race.sh  paxostest.sh but with race test (Deadline 1)
paxostest_pack1.sh  Shell script to run Test Pack 1 (Deadline 2)
paxostest_pack2.sh  Shell script to run Test Pack 2 (Deadline 2)
paxostest_pack3.sh  Shell script to run Test Pack 3 (Deadline 2)
```

Folder placement

Place **`paxosapp/`** in your **`$GOPATH/src`** directory.

The `prunner` program

The `prunner` program creates and runs an instance of your `PaxosNode` implementation. Some example usage is provided below:

```
# Start a ring of three paxos nodes, where node 0 has port 9009,
# node 1 has port 9010, and so on.
./prunner -myport=9009 -ports=9009,9010,9011 -N=3 -id=0 -retries=5
./prunner -myport=9010 -ports=9009,9010,9011 -N=3 -id=1 -retries=5
./prunner -myport=9011 -ports=9009,9010,9011 -N=3 -id=2 -retries=5
```

You can read further descriptions of these flags by running

```
./prunner -h
```

Executing the official tests

The tests for Deadline 1 are provided as bash shell scripts in the `paxosapp/` directory. The scripts may be run from anywhere on your system (assuming your `$GOPATH` has been set and they are being executed on a 64-bit Mac OS X or Linux machine). Simply execute the following:

```
$GOPATH/src/paxosapp/paxostest.sh
```

To test for race conditions, execute the following:

```
$GOPATH/src/paxosapp/paxostest_race.sh
```

If you have other questions about the testing policy, please don't hesitate to ask us a question on Piazza!

Submission

You are only required to submit all contents of the `paxosapp/` folder. Please make a zip file, name it using the format **<R>.zip**, where **<R>** should be replaced by your roll number (e.g. **19030010.zip**) and submit it on LMS.

If you worked on this assignment with a partner, only a single LMS submission is required. Name the zip file **<R1>_<R2>.zip** where **<R1>** and **<R2>** are replaced by the roll numbers of each student (e.g. **19030010_19030011.zip**).

6. Notes and Hints

Although the Paxos algorithm is clear, there are still some details and hints we want to emphasize here to help you pass our tests:

- If there is no already accepted $< proposalnumber, value >$ pair for a given key when you response to a prepare message, you should response with $< -1, nil >$.
- To help us test your code, you **must** include the node Id of the caller as `RequesterId` in the arguments when you call `RecvPrepare`, `RecvAccept` and `RecvCommit` RPC APIs.
- When you initialize a Paxos node in `NewPaxosNode`, you CAN NOT assume `myHostPort == hostMap[srvId]` – because we set up a proxy server upon each node when we do the test. You should listen to `myHostPort` rather than `hostMap[srvId]` to wait for incoming RPC request; and you should dial to `hostMap[srvId]` rather than `myHostPort` before this node tries to make RPC call to itself.

- Please don't update the Highest Seen Proposal Number (Np) to the new Proposal number (n) when you received the accept message in the second phase, even if $n \geq Np$. You only want to update your Np in the first phase.
- Think about what information you need to update/delete when you commit a $< key, value >$ pair? What happens if some delayed prepare message or delayed accept message delivered to a client after this proposal has been committed?
- We do care about the efficiency of your implementation. You have to move on to next phase as soon as you can – for example, when you receive the majority of accepted, you don't have to wait for the response from the remaining nodes.
- For each paxos node individually, the proposal numbers should be strictly increasing. And across all paxos nodes, the proposal numbers should be unique.
- If a key-value pair (k, v1) was committed and a new proposal is seen for (k, v2), all nodes should now reach consensus on the newer value, i.e. v2 for key k.
- *func NewPaxosNode(...)* initializes a paxos node. It is here you need to establish a connection with all the nodes (including itself) so that RPCs can be made on them. All other functions you have to implement in **paxos_impl.go** will be invoked as RPCs. Hence, nodes should be able to call these functions on other nodes as RPCs as well.
- Errors around nodes not being able to connect with other nodes or listen on a specified port can be more frequent on slower machines or VM's. Within **paxostest*.sh** [* = _pack1, _pack2, _pack3, _race or nothing], you may include another flag "-retries 10" or "-retries 15". This is the number that is fed into *numRetries* in *func NewPaxosNode(...)*, i.e. the number of times a paxos node tries to connect with each of the other paxos nodes.
- This assignment uses the same RPC code structure that was used in Assignment 1 Model 2. You are advised to review your solution to Assignment 1 or the reference solution available on LMS.