Project 2: A Paxos Application

1 Overview

Important Dates:

Project release: **Thursday, October 26, 2017 at 11:59pm**
Checkpoint due: **Thursday, November 2, 2017 at 11:59pm**
Final Test due: **Saturday, November 11, 2017 at 11:59pm**
Application Layer due: **Sunday, November 19, 2017 at 11:59pm**
Submission limits: **15 Autolab submissions per checkpoint**

In this project you will implement the Paxos Algorithm to propose $< key, value >$ pair in a distributed storage system and then build an application upon it. There will be one checkpoint for the implementation of your Paxos. Keep in mind that this is an individual project, so start early! For more information regarding what portion of the project is expected to be completed for the checkpoint and the final test, please refer to section 2.5 and 2.6.

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` file in the project's root directory. To clone a copy, execute the following Git command:

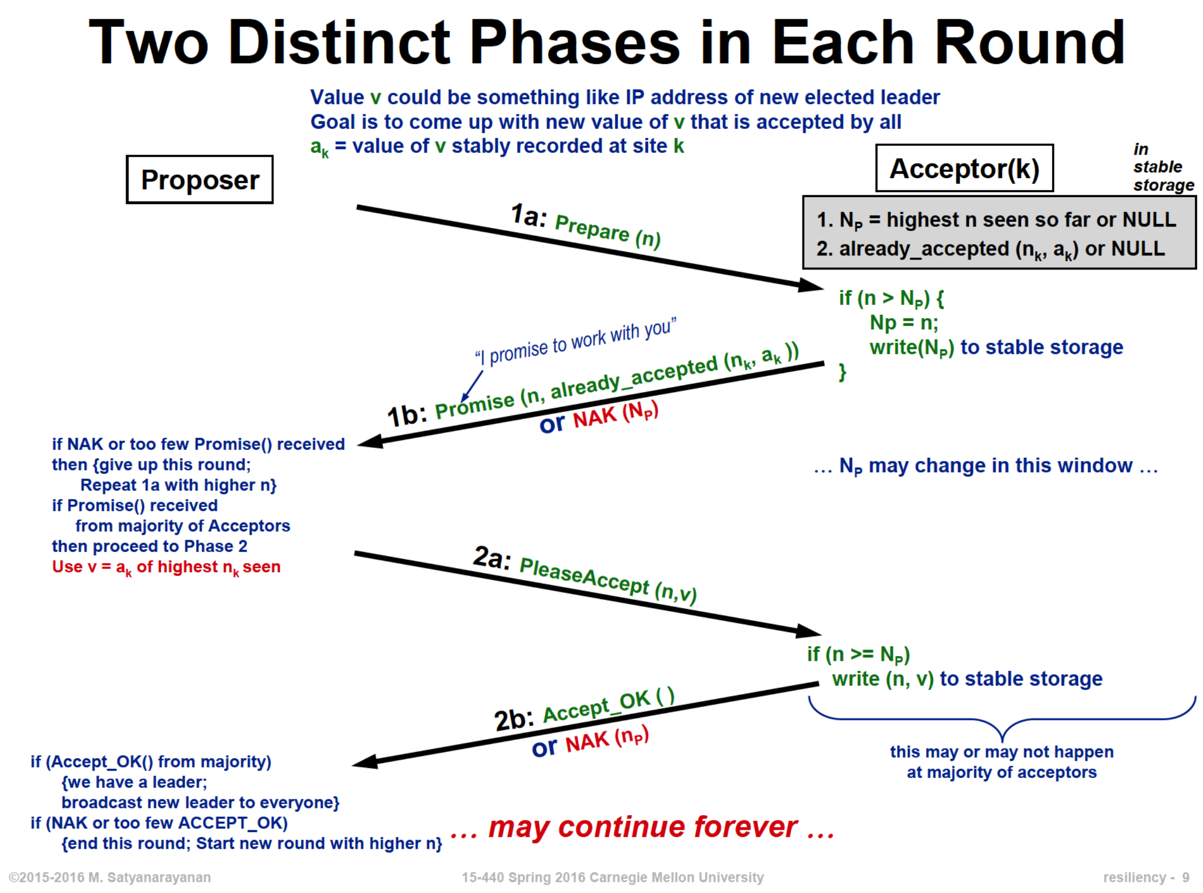
```
git clone https://github.com/CMU-440-F17/P2.git
```

This project is an individual project and you must work on this project on your own. 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

The main 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 on Oct. 5th (Section A Distributed Replication) also gives a detailed description of Paxos.

Figure **??** is a very concise illustration of the Paxos Algorithm that you want to implement.



Two Distinct Phases in Each Round

Value v could be something like IP address of new elected leader
Goal is to come up with new value of v that is accepted by all
a_k = value of v stably recorded at site k

Proposer

Acceptor(k)

in stable storage

1. N_P = highest n seen so far or NULL
2. already_accepted (n_k, a_k) or NULL

1a: Prepare (n)

```
if (n > N_P) {
    Np = n;
    write(N_P) to stable storage
}
```

"I promise to work with you"

1b: Promise (n, already_accepted (n_k, a_k)) or NAK (N_P)

... N_P may change in this window ...

```
if NAK or too few Promise() received
then {give up this round;
    Repeat 1a with higher n}
if Promise() received
    from majority of Acceptors
then proceed to Phase 2
Use v = a_k of highest n_k seen
```

2a: PleaseAccept (n,v)

```
if (n >= N_P)
    write (n, v) to stable storage
```

this may or may not happen at majority of acceptors

2b: Accept_OK ( ) or NAK (n_P)

```
if (Accept_OK() from majority)
    {we have a leader;
    broadcast new leader to everyone}
if (NAK or too few ACCEPT_OK)
    {end this round; Start new round with higher n}
```

... may continue forever ...

©2015-2016 M. Satyanarayanan 15-440 Spring 2016 Carnegie Mellon University resiliency - 9

Figure 1: An illustration of Paxos Algorithm from Professor Satya's slides of the 440 course in Spring 2016.

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. `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 for a given key, and for a particular key in a particular proposer, proposal numbers should be monotonically increasing. However, because the proposal 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`, the 3 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 out prepare messages, it should call the `RecvPrepare` method on all nodes.

Note: Learners determine whether any value has been accepted. For a value to be accepted by Paxos, a majority of acceptors must choose the same value. In most practical settings, every node is a proposer, acceptor and a learner.

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 can not achieve a majority (at least $\lfloor \frac{n}{2} \rfloor + 1$ nodes, where n is the total number of nodes), it will abandon the proposal and return a non-nil error immediately. 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.

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.

2.5 Checkpoint (13%)

To get full points for this section, your code must pass the Autolab tests for the checkpoint by the due date, **Thursday, November 2, 2017 at 11:59pm**. 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.

2.6 Final Test (62%)

To get full points for the final test, you must pass the final Autolab tests by the due date, **Saturday, November 11, 2017 at 11:59pm** . 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 section 8 for all the notes and hints before you get stuck.

2.7 Code Style (5%)

As before, we will manually grade your submissions based on your coding style 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 not have dead code and you should not a lengthy function. You have received feedback from us for previous projects 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.

3 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 previous 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).

Prior to deadline, you must sign up for a 10-minute presentation session with TAs to showcase your application. We will release link of the sign-up form on Piazza, so keep an eye on the latest instructor note. 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. During the presentation, TAs will evaluate your project from following criteria:

- **Correctness:** In your application, if there is a majority, all the nodes should agree on the same value proposed by some leader. Your application should also be able to handle the scenario when there is no majority, i.e., half/more than half of the nodes shut down.
- **Fault-tolerance:** We are expecting your Paxos application to still reach agreement when less than half of the nodes fail. **We will be explicitly asking you to manually kill some of the nodes in your application during the presentation**.
- **Robustness:** Your Paxos application should be able to handle considerable amount of nodes. We are expecting to see at **least 10 nodes** running at the same time in your application.
- **Visual:** Though UI is not the focus for this application, we are expecting to actually see the Paxos session result represented in some reasonable format.

Besides the aforementioned criteria, TAs love to be surprised by your creativity. We would like to see anything interesting within your application during the presentation. It could be some original application feature, additional situation handler in your Paxos implementation, or even some tricks you used when making the application. Think of the project as a platform to showoff what you have learnt from this course.

Your project should be reasonably documented. You will need to hand in a PDF named `application.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. **TAs should be able to run your application after reading your documents.**

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 define 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.

4 Testing

Please bear in mind that you will have a limit of 15 submissions for both checkpoint and final test. We have provided the basic checkpoint tests. We will provide you with the final tests after the checkpoint due date. 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.

Please refer to the `README.md` about how to run our test.

5 Hand In

5.1 Checkpoint & Paxos Final Test

You will need to hand in a `paxos.tar` file of the `paxos` folder. You can generate this file by runnning the following command in the `P2/src/github.com/cmu440-F17/paxosapp` folder:

```
tar -cvf paxos.tar paxos
```

5.2 Application

You will need to hand in a `application.tar` file on Autolab. 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.**

6 Project Requirements

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

- You are free to discuss high-level design issues with other people in the class, but every aspect of your implementation must be entirely your 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.

7 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` these 3 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 $n >= Np$. 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 the application, even though we are looking for visual display on the front-end of your application, a visually-appealing UI will just be the last few points to get you full-score, so our advice would be don't overdo it if you are short on time.