ECE 360P: Concurrent and Distributed Systems **Vijay K. Garg**

Spring 2023

Assignment 5: Paxos

Due: Midnight April 18 (Tuesday)

The goal of this assignment is to fully understand and implement the Paxos protocol. You will be implementing a library to achieve agreement on a certain value between multiple nodes. You should consult the paper, Paxos Made Simple by Leslie Lamport. **Please use the starter code template located at https://github.com/vijaygarg1/ECE-360P.**

1 Starter Code

The starter code contains a number of files that should guide your implementation. Here is a high level overview of each file:

- **Paxos.java:** contains the code for the library you will implement. Each instance of this class corresponds to a Paxos peer that will perform agreement with other peers. You will need to implement the RMI request handlers and other helper functions as described in the code comments and briefly below. The majority of your work for this assignment will be here.
- **PaxosRMI.java:** contains the interface that your Paxos library should implement.
- **Request.java:** contains the data that will be sent to each RMI request handler. You will need to add appropriate fields as necessary for your implementation.
- **Response.java:** contains the data that will be returned by each RMI request handler. You will need to add appropriate fields as necessary for your implementation.
- **State.java:** Provides an enum data type to represent the state of a Paxos peer on a particular instance of agreement.
- **PaxosTest.java:** JUnit test cases that your Paxos library should pass. The results of this test case file (with slightly different numbers) is what you will be graded on.

The starter code also contains a Makefile to compile and run your code with JUnit. To compile your code:

```
$ make
```

To run the testcases with your compiled code:

```
$ make test
```

There are many tutorials online on how to install make if you do not already have it on your machine, or you can manually compile and run the program with JUnit.

2 Requirements

The comments in Paxos.java contain a description of each function that you will implement. Here is an overview of the Paxos class:

```
public Paxos(int me, String[] peers, int[] ports);
```

This is the constructor of a Paxos peer. The *peers* argument contains the hostnames of all the peers (including this one). The *ports* argument contains the ports of all the peers (including this one), and the *me* argument is the index of this peer in the *peers* array. Each paxos peer serves as both a proposer and an acceptor.

```
public void Start(int seq, Object value);
```

An application will call Start on a created Paxos peer when it wants to start an agreement on instance *seq*, with a proposed *value*. This method should not wait for completion of the protocol, and your implementation should be able to make progress on agreement for multiple instances at the same time. To accomplish this, this method should start a new thread to run the agreement protocol (using the Runnable interface) and return without waiting for that thread to complete.

A proposer needs a way to choose a higher proposal number than any seen so far. This is a reasonable exception to the rule that proposer and acceptor should be separate. It may also be useful for the RMI handlers to return the highest known proposal number if it rejects a request, to help the caller pick a higher one next time. The *me* value will be different in each Paxos peer, so you can use it to help ensure that proposal numbers are unique.

```
public Response Prepare(Request req);
public Response Accept(Request req);
public Response Decide(Request req);
```

These are RMI request handlers that will be used to facilitate communication between Paxos peers. They serve as the functions for a Paxos acceptor. You should implement these according to the Paxos pseudo-code below.

```
public retStatus Status(int seq);
```

An application calls Status to find out whether the Paxos peer thinks the corresponding instance *seq* has reached agreement, and if so what the agreed value is. Status() should consult the local Paxos peer's state and return immediately; it should not communicate with other peers. The application may call Status for old instances (however, see the discussion of Done() below).

```
public void Done(int seq);
```

An application will call Done on a Paxos peer when it no longer needs to query the result of an agreement instance *seq*. The reason we need this functionality, is that we want our Paxos server to free memory on instances that are no longer needed. Note that even after calling Done, the Paxos peer can't discard the instances yet, since some other Paxos peer might not have agreed to the instance yet. So each Paxos peer should tell every other peer the highest Done argument supplied by its local application. Then, each Paxos peer will have a Done value from every other peer. It should find the minimum (see Min below), and discard all instances with sequence numbers $\leq$ that minimum.

It's OK for your Paxos to piggyback the Done value in the agreement protocol packets; that is, it's OK for peer P1 to only learn P2's latest Done value the next time that P2 sends an agreement message to P1.

```
public int Min()
```

The Min method returns the minimum of the Done values supplied by every other Paxos peer. This should be used to implement forgetting (see Done above). Additionally, if Start() is called with a sequence number less than Min(), the Start() call should be ignored. If Status() is called with a sequence number less than Min(), Status() should return Forgotten.

```
public int Max()
```

The Max method returns this peer's largest known sequence number.

3 Paxos Pseudocode

Algorithm 1 Paxos

```
 1: proposer($v$):
 2: while not decided: do
 3:    choose $n$, unique and higher than any $n$ seen so far
 4:    send prepare($n$) to all servers including self
 5:    if prepare_ok($n, n_a, v_a$) from majority then
 6:       $v' = v_a$ with highest $n_a$; choose own $v$ otherwise
 7:       send accept($n, v'$) to all
 8:       if accept_ok($n$) from majority then
 9:          send decided($v'$) to all
10:       end if
11:    end if
12: end while
13:
14: acceptor's state:
15: $n_p$ (highest prepare seen)
16: $n_a, v_a$ (highest accept seen)
17:
18: acceptor's prepare($n$) handler:
19: if $n > n_p$ then
20:    $n_p = n$
21:    reply prepare_ok($n, n_a, v_a$)
22: else
23:    reply prepare_reject
24: end if
25:
26: acceptor's accept($n, v$) handler:
27: if $n >= n_p$ then
28:    $n_p = n$
29:    $n_a = n$
30:    $v_a = v$
31:    reply accept_ok($n$)
32: else
33:    reply accept_reject
34: end if
```

4 Suggested Plan

1. Add elements to the Paxos Class in Paxos.java to hold the state you'll need, according to the above pseudo-code. You'll need to define a class to hold information about each agreement instance.
2. Modify the RMI Request and Response classes for Paxos protocol messages, based on the lecture pseudo-code. The RMIs should include the sequence number for the agreement instance to which they refer.
3. Write the proposer function that drives the Paxos protocol for an instance, and RMI handlers that implement acceptors. Start a proposer function in its own thread for each instance, as needed (e.g. in Start()).
4. At this point you should be able to pass the first few tests. Now implement forgetting.

Clarifications / Hints:

- In order to pass tests assuming unreliable network, your Paxos should call the local acceptor through a function call rather than RMI.
- You do not need to write code to handle the situation where a Paxos peer needs to re-start after a crash. If one of your Paxos peers crashes, it will never be re-started.
- The tester calls Kill() when it wants your Paxos to shut down; Kill() sets px.dead. You should call px.isDead() in any loops you have that might run for a while, and break out of the loop if px.isDead() is true. It is particularly important to do this any in any long-running threads you create.

5 Submission Instructions

Please follow this handin procedure exactly. The *paxos/* directory containing your source files should be in a directory named $EID1\_EID2$. Please do not remove the jar files. Then zip your $EID1\_EID2$ dir into $EID1\_EID2$.zip and submit through Canvas.

```
zy6363_yh28827.zip
└─ zy6363_yh28827
   ├─ paxos
   │  ├─ Paxos.java
   │  └─ ...
   ├─ junit-4.12.jar
   ├─ hamcrest-core-1.3.jar
   └─ Makefile
```

6 Assignment Reference

MIT Paxos Lab: http://nil.csail.mit.edu/6.824/2015/labs/lab-3.html