6.824 - Spring 2012

6.824 Lab 6: Paxos

Due: Friday, April 13th, 5:00pm.

Introduction

In labs 6 and 7, you will replicate the lock service using the replicated state machine approach. See Schneider's RSM paper for a good, but non-required, reference. In the replicated state machine approach, one machine is the master; the master receives requests from clients and executes them on all replicas in the same order.

When the master fails, any of the replicas can take over its job, because they all should have the same state as the failed master. One of the key challenges is ensuring that everyone agrees on which replica is the master and which of the slaves are alive, despite arbitrary sequences of crashes or network partitions. We use Paxos to reach such an agreement.

In this lab, you will implement Paxos and use it to agree to a sequence of membership changes (*i.e.*, view changes). We will implement the replicated lock server in Lab 7. We have modified `lock_smain.cc` in this lab to start the RSM *instead of* the lock server; however, we will not actually replicate locks until Lab 7. As a result, in this lab, the `lock_server` processes are actually serving as the `configuration servers`. We will use the term `configuration server` and `lock_server` interchangably in the following text.

When you complete this lab and the next you will have a replicated state machine that manages a group of lock servers. You should be able to start new lock servers, which will contact the master and ask to join the replica group. Nodes can also be removed from the replica group when they fail. The set of nodes in the group at a particular time is a *view*, and each time the view changes, you will run Paxos to agree on the new view.

The design we have given you consists of three layered modules. The RSM and config layers make downcalls to tell the layers below them what to do. The config and Paxos modules also make upcalls to the layers above them to inform them of significant events (*e.g.*, Paxos agreed to a value, or a node became unreachable).

RSM module

The RSM module is in charge of replication. When a node joins, the RSM module directs the config module to add the node. The RSM module also runs a *recovery* thread on every node to ensure that nodes in the same view have consistent states.

In this lab, the only state to recover is the sequence of Paxos operations that have been executed. In Lab 7, you will extend the RSM module to replicate the lock service.

config module

The config module is in charge of view management. When the RSM module asks it to add a node to the current view, the config module invokes Paxos to agree on a new view. The config module also sends periodic heartbeats to check if other nodes are alive, and removes a node from the current view if it can't contact some of the members of the current view. It removes a node by invoking Paxos to agree on a new view without the node.

Paxos module

The Paxos module is in charge of running Paxos to agree on a value. In principle the value could be anything. In our system, the value is the list of nodes constituting the next view.

The focus of this lab is on the Paxos module. You'll replicate the lock server in the next lab.

Each module has threads and internal mutexes. As described above, a thread may call down through the layers. For instance, the RSM could tell the config module to add a node, and the config module tells Paxos to agree to a new view. When Paxos finishes, a thread will invoke an upcall to inform higher layers of the completion. To avoid deadlock, we suggest that you use the rule that a module releases its internal mutexes before it upcalls, but can keep its mutexes when calling down.

Getting Started

Begin by initializing your Lab 6 branch with your implementation from Lab 5.

```
% cd ~/lab
% git commit -am 'my solution to lab5'

Created commit ...
% git pull
remote: Generating pack...
...
% git checkout -b lab6 origin/lab6
Branch lab6 set up to track remote branch refs/remotes/origin/lab6.
Switched to a new branch "lab6"
% git merge lab5
```

This will add new files, `paxos_protocol.h`, `paxos.{cc,h}`, `log.{cc,h}`, `rsm_tester.pl`, `config.{cc,h}`, `rsm.{cc,h}`, and `rsm_protocol.h` to your `lab/` directory and update the GNUmakefile from your previous lab. It will also incorporate minor changes into your `lock_smain.cc` to initialize the RSM module when the lock server starts. Note that since the RSM and the lock server both bind on the same port, this will

actually disable your lock server until Lab 7, unless you change the relevant line in `lock_smain.cc` back. The lock server will now take two command-line arguments: the port that the master and the port that the lock server you are starting should bind to.

In `rsm.{cc,h}`, we have provided you with code to set up the appropriate RPC handlers and manage recovery in this lab.

In files `paxos.{cc,h}`, you will find a sketch implementation of the `acceptor` and `proposer` classes that will use the Paxos protocol to agree on view changes. The file `paxos_protocol.h` defines the suggested RPC protocol between instances of Paxos running on different replicas, including structures for arguments and return types, and marshall code for those structures. You'll be finishing this Paxos code in this lab.

The files `log.{cc,h}` provide a *full* implementation of a `log` class, which should be used by your `acceptor` and `proposer` classes to log important Paxos events to disk. Then, if the node fails and later re-joins, it has some memory about past views of the system. Do not make any changes to this class, as we will use our own original versions of these files during testing.

`config.cc` maintains views using Paxos. You will need to understand how it interacts with the Paxos and RSM layers, but you should not need to make any changes to it for this lab.

In the next lab we will test if the replicated lock service maintains the state of replicated locks correctly, but in this lab we will just test if view changes happen correctly. The lab tester `rsm_tester.pl` will automatically start several lock servers, kill and restart some of them, and check that you have implemented the Paxos protocol and used it correctly.

Understanding how Paxos is used for view changes

There are two classes that together implement the Paxos protocol: `acceptor` and `proposer`. Each replica runs both classes. The `proposer` class leads the Paxos protocol by proposing new values and sending requests to all replicas. The `acceptor` class processes the requests from the `proposer` and sends responses back. The method `proposer::run(nodes, v)` is used to get the members of the current view (`nodes`) to agree to a value `v`. When an agreement instance completes, `acceptor` will call `config`'s `paxos_commit(instance, v)` method with the value that was chosen. As explained below, other nodes may also attempt to start Paxos and propose a value, so there is no guarantee that the value that a server proposed is the same as the one that is actually chosen. (In fact, Paxos might abort if it can't get a majority to accept its `prepare` or `accept` messages!)

The `config` module performs view changes among the set of participating nodes. The first view of the system is specified manually. Subsequent view changes rely on Paxos to agree on a unique next view to displace the current view.

When the system starts from scratch, the first node creates view 1 containing itself only, i.e. view_1={1}. When node 2 joins after the first node, node two's RSM module joins node 1 and transfers view 1 from the node one. Then, node 2 asks its `config` module to add itself to view 1. The config module will use Paxos to propose to nodes in view_1={1} a new view_2 containing node 1 and 2. When Paxos succeeds, view_2 is formed, *i.e.*, view_2={1,2}. When node 3 joins, its RSM module will download the last view from the first node (view 2) and it will then attempt to propose to nodes in view 2 a new view_3= {1,2,3}. And so on.

The config module will also initiate view changes when it discovers that some nodes in the current view are not responding. In particular, the node with the smallest id periodically sends heartbeat RPCs to all others (and all other servers periodically send heartbeats to the node with the smallest id). If a heartbeat RPC times out, the config module calls the `proposer`'s `run(nodes, v)` method to start a new round of the Paxos protocol. Because each node independently decides if it should run Paxos, there may be several instances of Paxos running simultaneously; Paxos sorts that out correctly.

The `proposer` keeps track of whether the current view is stable or not (using the `proposer::stable` variable). If the current view is stable, there are no on-going Paxos view change attempts by this node. When the current view is not stable, the node is initiating the Paxos protocol.

The `acceptor` logs important Paxos events as well as a complete history of all values agreed to on disk. At any time a node can reboot and when it re-joins, it may be many views behind. Unless the node brings itself up-to-date on the current view, it won't be able to participate in Paxos. By remembering all views, the other nodes can bring this re-joined node up to date.

The Paxos Protocol

The Paxos Made Simple paper describes a protocol that agrees on a value and then terminates. Since we want to run another instance of Paxos every time there is a view change, we need to ensure that messages from different instances are not confused. We do this by adding instance numbers (which are not the same as proposal numbers) to all messages. Since we are using Paxos to agree on view changes, the instance numbers in our use of Paxos are the same as the view numbers in the config module.

Paxos can't guarantee that every node learns the chosen value right away; some of them may be partitioned or crashed. Therefore, some nodes may be behind, stuck in an old instance of Paxos while the rest of the system has moved on to a new instance. If a node's `acceptor` gets an RPC request for an old instance, it should reply to the `proposer` with a special RPC response (set `oldinstance` to true). This response informs the calling `proposer` that it is behind and tells it what value was chosen for that instance.

Below is the pseudocode for Paxos. The `acceptor` and `proposer` skeleton classes

contain member variables, RPCs, and RPC handlers corresponding to this code. Except for the additions to handle instances as described above, it mirrors the protocol described in the paper.

```
proposer run(instance, v):
 choose n, unique and higher than any n seen so far
 send prepare(instance, n) to all servers including self
 if oldinstance(instance, instance_value) from any node:
   commit to the instance_value locally
 else if prepare_ok(n_a, v_a) from majority:
   v' = v_a with highest n_a; choose own v otherwise
   send accept(instance, n, v') to all
   if accept_ok(n) from majority:
     send decided(instance, v') to all

acceptor state:
 must persist across reboots
 n_h (highest prepare seen)
 instance_h, (highest instance accepted)
 n_a, v_a (highest accept seen)

acceptor prepare(instance, n) handler:
 if instance <= instance_h
   reply oldinstance(instance, instance_value)
 else if n > n_h
   n_h = n
   reply prepare_ok(n_a, v_a)
 else
   reply prepare_reject

acceptor accept(instance, n, v) handler:
 if n >= n_h
   n_a = n
   v_a = v
   reply accept_ok(n)
 else
   reply accept_reject

acceptor decide(instance, v) handler:
 paxos_commit(instance, v)
```

For a given instance of Paxos, potentially many nodes can make proposals, and each of these proposals has a unique proposal number. When comparing different proposals, the highest proposal number wins. To ensure that each proposal number is unique, each proposal consists of a number and the node's identifier. We provide you with a struct `prop_t` in `paxos_protocol.h` that you should use for proposal numbers; we also provide the `>` and `>=` operators for the class.

Each replica must log certain change to its Paxos state (in particular the `n_a`, `v_a`, and

n_h fields), as well as log every agreed value. The provided log class does this for you; please use it without modification, as the test program depends on its output being in a particular format.

Add the extra parameter rpcc::to(1000) to your RPC calls to prevent the RPC library from spending a long time attempting to contact crashed nodes.

Your Job

The measure of success for this lab is to pass tests **0-7** of rsm_tester.pl. (The remaining tests are reserved for the next lab.) The tester starts 3 or 4 configuration servers, kills and restarts some of them, and checks that all servers indeed go through a unique sequence of view changes by examining their on-disk logs.

```
% ./rsm_tester.pl 0 1 2 3 4 5 6 7
test0...
...
test1...
...
test2...
...
test3...
...
test4...
...
test5...
...
test6...
...
test7...
...
tests done OK
```

Important: If rsm_tester.pl fails during the middle of a test, **the remaining lock_server processes are not killed** and the log files are not cleaned up (so you can debug the causes.). Make sure you do 'killall lock_server; rm -f *.log' to clean up the lingering processes before running rsm_tester.pl again.

Detailed Guidance

We guide you through a series of steps to get this lab working incrementally.

Step One: Implement Paxos

Fill in the Paxos implementation in paxos.cc, following the pseudo-code above. Do not worry about failures yet.

Use the RPC protocol we provide in paxos_protocol.h. In order to pass the tests, when the proposer sends a RPC, you should set an RPC timeout of 1000 milliseconds. Note that though the pseudocode shows different types of responses to each kind of RPC, our protocol combines these responses into one type of return structure. For example, the prepareres struct can act as a prepare_ok, an oldinstance, or a reject message, depending on the situation.

You may find it helpful for debugging to look in the paxos-[port].log files, which are written to by log.cc. rsm_tester.pl does not remove these logs when a test fails so that you can use the logs for debugging. rsm_tester.pl also re-directs the stdout and stderr of your configuration server to lock_server-[arg1]-[arg2].log.

Upon completing this step, you should be able to pass 'rsm_tester.pl 0'. This test starts three configuration servers one after another and checks that all servers go through the same three views.

Step Two: Simple failures

Test whether your Paxos handles simple failures by running 'rsm_tester.pl 0 1 2'. You will not have to write any new code for this step if your code is already correct.

Step Three: Logging Paxos state

Modify your Paxos implementation to use the log class to log changes to n_h, and n_a and v_a when they are updated. Convince yourself why these three values must be logged to disk if we want to re-start a previously crashed node correctly. We have provided the code to write and read logs in log.cc (see log::logprop(), and log::logaccept()), so you just have to make sure to call the appropriate methods at the right times.

Now you can run tests that involve restarting a node after it fails. In particular, you should be able to pass 'rsm_tester.pl 3 4 '. In test 4, rsm_tester.pl starts three servers, kills the third server (the remaining two nodes should be able to agree on a new view), kills the second server (the remaining one node tries to run Paxos, but cannot succeed since no majority of nodes are present in the current view), restarts the third server (it will not help with the agreement since the third server is not in the current view), kills the third server, restarts second server (now agreement can be reached), and finally restarts third server.

Step Four: Complicated failures

Finally, you need to verify that your code handles some of the tricky corner cases that Paxos is supposed to deal with. Our test scripts do not test all possible corner cases, so you could still have a buggy Paxos implementation after this step, but you will have a good feel for the protocol.

In paxos.cc, we use two methods: breakpoint1() and breakpoint2() to induce

complex failures. The `proposer::run` function calls `breakpoint1()` just after completing Phase 1, but before starting Phase 2. Similarly it calls `breakpoint2()` in between Phases 2 and 3. The RSM layer runs a small RPC server that accepts the `rsm_test_protocol` RPCs defined in `rsm_protocol.h`. The tester uses `rsm_tester` to sends RPCs to cause the server to exit at the respective breakpoint.

- **Test 5**: This test starts three nodes and kills the third node. The first node will become the leader to initiate Paxos, but the test will cause it to crash at breakpoint 1 (at the end of Phase 1). Then the test will restart the killed third node, which together with the remaining node should be able to finish Paxos (ignoring the failed first node) and complete the view change successfully. The script will verify that the Paxos logs show the correct view changes.

- **Test 6**: This test starts four nodes one by one and kills the fourth node. The first node initiates Paxos as a leader, but the test causes it to fail at breakpoint 2 (after phase 2.) When the fourth node re-joins the system, the rest of the nodes should finish agreeing on the view originally proposed by the first node, before making a new view of their own.

- **Test 7**: This test is identical to test 6, except that it kills *all* the remaining nodes after the first node exits. Then it restarts all slaves and checks that they first agree on the first node's proposed view before making a new view of their own.

By now, your code should now reliably pass all required tests, i.e. '`rsm_tester.pl 0 1 2 3 4 5 6 7`'.

Debugging Hints

- Make sure you remove all the log files and then kill any remaining lock servers (`killall lock_server; rm -f *.log`) before you start a new test run.
- If a test fails, first check the Paxos logs (`paxos-*.log`) and make sure the sequence of proposals and views make sense. Do all the nodes (that didn't crash) go through the same sequence of views? Does the sequence of views make sense given what the test does?
- If a server gets stuck, check whether one of the lock servers (particularly the leader) deadlocked or crashed when it shouldn't have. You can get a list of the process ids of running lock servers with `` `pgrep -s0 lock_server` ''. You can debug a running lock server with `` `gdb -p*pid* ``.
- Use `printf`s and check the relevant `lock_server-*.log` files to see what is going on. Some particularly important events are view changes, RSM requesting to add a node, and heartbeat events to remove nodes. In Paxos, print out important state variables (e.g., `my_n`, `n_h`, `n_a`, `instance_h`) and verify that they make sense.

Handin procedure

E-mail your code as a gzipped tar file to 6.824-submit@pdos.csail.mit.edu by the deadline at the top of the page. To do this, execute these commands:

```
% cd ~/lab
% ./stop.sh
% make clean
% rm core.*
% rm *.log
% cd ..
% tar czvf `whoami`-lab6.tgz lab/
```

or

```
% cd ~/6.824/lab
% make handin
```

That should produce a file called `[your_user_name]-lab6.tgz` in your `lab` directory. Attach that file to an email and send it to the 6.824 submit address.

Please post questions or comments on Piazza.
Back to 6.824 home.