6.824 - Spring 2012

6.824 Lab 7: Replicated State Machine

Due: Friday, April 27th, 5:00pm.

Introduction

In this lab you will replicate your lock server using the replicated state machine (RSM) approach (see Schneider's RSM paper for a good, but non-required, reference. We have also discussed an example replicated state machine in Lecture.) In the replicated state machine approach, one machine is the master and the others are slaves. The master is in charge of receiving requests from clients and executing them on all replicas. To ensure that all replicas have identical state, the replicas must execute all requests in the same order and all requests must produce the same result on all replicas (*i.e.*, the handlers must be deterministic). The RSM uses the Paxos protocol implemented in the previous lab to agree on the current master and node membership to cope with failed and re-joined replicas.

To ensure all requests are executed in a unique total order, the master assigns each request a `viewstamp` number which dictates the total order. The `viewstamp` consists of two fields, the view number (obtained from Paxos) and a monotonically increasing sequence number. The viewstamps assigned to all RSM requests dictate a total order among them. In particular, viewstamps with a lower view number are ordered before those with a higher view number and within the same view number, viewstamps with lower seqnos are ordered before those with higher seqnos. How do we guarantee all viewstamps form a unique total order? This is because Paxos guarantees all view numbers form a total order. Additionally, within each view, all nodes agree on the current view's membership and thus each RSM node can use the agreed upon membership to agree on a unique master who is the only one that can assign each request an increasingly seqno to properly order requests within a view.

The primary task in the lab is building a RSM library on top of our existing RPC library so that you can plug in any RPC program you have written so far and replicate it. To ensure the appropriate behavior, however, there are certain constraints on the RPC handlers. Most importantly, the RPC handlers must run to completion without blocking, and be deterministic and idempotent. These constraints will ensure that all replicas execute all requests in the same order and with the same result. Once you have built the RSM library we will ask you to replicate the lock server you built in previous labs using RSM.

Getting Started

Begin by updating your lab directory with the new infrastructure code for lab 7. Since you are building on the past labs, make sure your code passes all tests for previous labs before starting in on this lab.

```
% cd ~/lab
% git commit -am 'my solution to lab6'

Created commit ...
% git pull
remote: Generating pack...
...
% git checkout -b lab7 origin/lab7
Branch lab7 set up to track remote branch refs/remotes/origin/lab7.
Switched to a new branch "lab7"
% git merge lab6
```

We provide you with some skeleton code of the RSM library. The library has the client and server class for RSM in files `rsm_client.{cc,h}` and `rsm.{cc,h}`.

The RSM client class (`rsm_client`) is used by a client program to request service from the master replica in the RSM. The RSM client takes in its constructor the address of a known replica, and immediately contacts the node to learn the addresses of all replicas as well as the current master. The client program (e.g. the `lock_client` class) can use the `call` method on the RSM client object (just as if it were an RPC client). The `call` method on RSM client will marshall RSM request and send it via the `rsm_client_protocol::invoke` RPC to the master replica. (The RPC protocol between the RSM client and RSM server (replica) is defined in the `rsm_client_protocol` class in file `rsm_protocol.h`).

To turn any server program into a replica in the RSM service, your application (e.g. `lock_server_cache_rsm` class) creates an RSM server object (`rsm`) and uses it in place of the normal RPC server `rpcs` object. The RSM server constructor creates a `config` object with arguments consisting the id of the first replica ever created and the id of this server. The RSM server registers a number of RPC handlers and spawns off a recovery thread to synchronize state with the master replica when Paxos has agreed on a stable view.

Once the master is in a stable state, it can process `invoke` RPCs from RSM clients. For each request, the master assigns it the next `viewstamp` number with an increasing seqno. The master then issues an invoke RPC on all replicas in the current view. The replicas unmarshall the request, and execute the registered handler. Note that the replicas must execute requests in the same total order as dictated by the requests' viewstamps **without any gaps in seqno**. If the master has succeeded in executing a request on **all** replicas (including itself), it will reply to the client. If the master has encountered replica failures during this process, it should instruct its `config` object to initiate a view change. Occasionally, an RSM client might send its request to a non-master node, in which case the node should reject the client's request by replying with `rsm_client_protocol::NOTPRIMARY`. The client will then call the `members` RPC to get an updated list of replicas.

When a failed replica re-joins a running RSM, it has potentially missed many requests and must do a state transfer to bring its state in sync with the other replicas before it can process any requests. Additionally, when the master has encountered a failure during the process of invoking the client request at various replicas, some replicas might have executed the request while others not. Thus, the RSM servers must be able to synchronize its state properly from the agreed upon master node before processing any client requests. We provide some skeleton code to do this; the interface is defined in `rsm_state_transfer.h`.

Your Job

Your job is to turn the cache lock service into a RSM service. Our measure of success is surviving failed master and slaves and incorporating re-joined replicas back into a running RSM. For this lab, you'll need to pass tests 8-16 of `rsm_tester.pl` (as well as making sure all the file system tests from previous labs work).

The tester picks random ports for the lock server replicas, and starts them up. It redirects output to log files, named as `lock_server-[master_port]-[my_port].log`. The log for the tester is `lock_tester-[master_port].log`. Here is the output of a successful run of `rsm_tester.pl`:

```
% ./rsm_tester.pl 8 9 10 11 12 13 14 15 16
test8: start 3-process lock service
...
./lock_tester: passed all tests successfully
test9: start 3-process rsm, kill first slave while tester is running
...
./lock_tester: passed all tests successfully
test10: start 3-process rsm, kill second slave while tester is running
...
./lock_tester: passed all tests successfully
test11: start 3-process rsm, kill primary while tester is running
...
./lock_tester: passed all tests successfully
test12: start 3-process rsm, kill first slave at break1, continue with 2, add first slave
...
./lock_tester: passed all tests successfully
test13: start 3-process rsm, kill slave at break1 and restart it while lock_tester is running
```

```
...
./lock_tester: passed all tests successfully
test14: start 5-process rsm, kill slave break1, kill slave break2
...
./lock_tester: passed all tests successfully
test15: start 5-process rsm, kill slave break1, kill primary break2
...
./lock_tester: passed all tests successfully
test16: start 3-process rsm, partition primary, heal it
...
./lock_tester: passed all tests successfully
tests done
%
```

When debugging, you might want to run the tests individually by just specifying a single test number. You can also specify the same random seed values across run to make `rsm_tester.pl` choose the same set of random ports. (e.g. `./rsm_tester.pl -s 89362 8`) Once your lab works, make sure it is able to pass all (including test 0-7) tests of `./rsm_tester.pl` many times in a row. Note that we won't run the file system tests for grading to focus on RSM in this lab, although your lab 7 should be able to pass all the tests with slight modification to Makefile and if previous labs are correct.

Important: As in the previous lab, 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

Step One: Replicable caching lock server

At first step, you have to redesign the lock protocol so that the caching lock server is replicable. The implementation of such caching lock server and client may differ substantially from the ones in lab 4. Therefore, we suggest you implement them from scratch. We have provided you with the skeleton of the caching lock server in `lock_server_cache_rsm.{cc,h}`, and caching lock client in `lock_client_cache_rsm.{cc,h}`. We have also modified `lock_smain.cc` and `lock_tester.cc` to use these new classes too. Note that you should not include both `lock_server_cache.h` and `lock_server_cache_rsm.h` (or `lock_client_cache.h` and `lock_client_cache_rsm.h`).

The caching lock server should address the following three challenges in order to be replicated. For convenience, in the following context, we use `SVR_HDL` to refer "RPC handlers of the caching lock server" (e.g. `acquire`, `release`) , and `CLT_HDL` for "RPC handlers of the caching lock client" (e.g. `retry`, `revoke`).

- First, the lock protocol should avoid deadlocks that may be caused by locks held by RSM layer. The RSM layer holds the `invoke_mutex` while calling `SVR_HDL` to guarantee sequential execution of RPC requests (see "Step Two"). As a result, the caching lock server can not process more than one RPC request at the same time. This has two implications.
    - First, `SVR_HDL` should not call `CLT_HDL` which may call another `SVR_HDL` . Such RPC call chain may lead to a deadlock as follows. Client A sends an `acquire` to server, server sends `revoke` to client B in its `acquire` handler (with `invoke_mutex` held), client B sends `release` to server in `revoke_handler`, then the server blocks forever trying to acquire the `invoke_mutex` when dispatching the `release` request.

      To avoid such deadlock, one approach is to follow the rule of "do not call RPCs in an RPC handler", which means you send RPC in background threads rather than in `SVR_HDL` or `CLT_HDL` directly. We have provided 2 background threads in `lock_server_cache_rsm` (`retryer` and `revoker`) and 1 in `lock_client_cache_rsm` (`releaser`) for you. You may find `rpc/fifo.h` helpful to the communication between the background thread and RPC handlers.

Note that there are protocols that use less than 3 background threads. The reason is that sometimes it is benign to call RPC in RPC handlers, so that you do not need a background thread. For example, if the server sends a `revoke` in a background thread, then the client's `revoke_handler` is OK to send `release` RPC to the server. In this case, the `releaser` thread of the `lock_client_cache_rsm` is not needed. You are free to devise and use such protocols.

- Second, one `SVR_HDL` should not wait for the arriving of another RPC request. For example, in lab 4, your `lock_server_cache` may send a `revoke` to the lock owner, and wait for the owner to send a `release` RPC to the server. Such design leads to deadlocks in lab 7, and you should avoid it.

- Second, the caching lock client should handle failures of the primary lock server. The reason is that once a primary fails, the client has no idea whether its outstanding RPC requests are processed by the new primary or not.

  To handle the primary failure, the client should assign sequence numbers to all requests. That is each request should have a unique client ID (e.g., a random number or the id string) and a sequence number. For an `acquire`, the client picks the first unused sequence number and supplies that sequence number as an argument to the `acquire` RPC, along with the client ID. Same as lab 4, you probably want to send no additional `acquire`s for the same lock to the server until the outstanding one has been completed. The corresponding `release` (which may be much later because the lock is cached) should probably carry the same sequence number as the last `acquire`, in case the server needs to tell which `acquire` goes along with this `release`. This approach requires the server to remember at most one sequence number per client per lock. The server should discard out-of-date requests, but must reply consistently if the request is a duplication of the latest request from the same client.

  We have defined the type of sequence number as `lock_protocol::xid_t`, and added the sequence number argument into the RPC handler in both `lock_server_cache_rsm` and `lock_client_cache_rsm`.

- Third, in case of no failure, the state of the caching lock server should be consistent. Such requirement has two implications. First, input that is applied to all replicas should change the state in the same way. Second, input that is only applied to the primary should not change the state of the primary. Such input includes the response of the `revoke` or `retry` RPC, because only primary can send RPCs to client and receive the RPC responses from the client.

Upon completion of step one, you should be able to pass lab1 test using `lock_server` and `lock_tester`. You should provide the same port twice to start the `lock_server` (unless you temporarily change `lock_smain.cc` for this step to accept only one argument). A succesful run looks like this:

```
./lock_server 3772 3772 &
...
./lock_tester 3772
...
lock_tester: passed all tests sucessfully
```

Note that if you copy the code from `lock_server_cache` to `lock_server_cache_rsm`, you may see a segmentation fault because `lock_server_cache_rsm::lu` is NULL in this step. To fix this, just check for a non-NULL value before using lu.

Step Two: RSM without failures

Before starting Step Two, make sure you finished Step One. Then comment out the "#define STEP_ONE" line in `lock_main.cc` so that the `lock_server` uses rsm layer from now on.

In this step, just get the RSM working, assuming that none of the replicas will fail. The basic protocol is:

- The RSM client sends its request to the master by calling the `invoke` RPC.
- The master assigns the client request the next viewstamp in sequence, and sends the `invoke` RPC to each slave.
- If the request has the expected viewstamp, the slave executes the request locally and replies OK to the master.
- The master executes the request locally, and replies back to the client.

This will involve filling in the various functions in `rsm.cc` mentioned above. In particular:

- `rsm::client_invoke()`. This RPC handler is called by a client to send a new RSM request to the master. If this RSM replica is undergoing Paxos view changes (i.e. `inviewchange` is true), it should reply with `rsm_client_protocol::BUSY` to tell the client to try again later. If this RSM replica is not the master, it should reply with the `rsm_client_protocol::NOTPRIMARY` status. If it is the master, it first assigns the RPC the next viewstamp number in sequence, and send an `invoke` RPC to all slaves in the current view. As in the previous lab, you should supply a timeout to the `invoke` RPC in case any of the slaves have died. To execute a RSM request, you need to use the provided method `execute()` which unmarshalls the RSM representation of a client's RPC and executes it using the registered handler.

  The master must ensure that all client requests are executed in order at all slaves. One way to achieve this is for the master to process each request serially in lockstep. An easy way to ensure that requests are processed one at a time is to hold a mutex in `client_invoke()` while a request is being processed. However, it would be a bad idea to hold `rsm_mutex` while calling the `invoke` RPC, since the `rsm_mutex` protects the internal data structures of the RSM as well, and nothing else can happen in the RSM while that mutex is held. Instead, you can hold a separate mutex called `invoke_mutex`, which is used *only* to serialize calls to `client_invoke()`. You need to be careful of two things if you serialize requests this way. First, you shouldn't hold `rsm_mutex` across RPCs. Second, you shouldn't try to acquire `invoke_mutex` while you are holding `rsm_mutex`, since that would effectively cause you to hold `rsm_mutex` for the duration of an RPC.

  Once all slaves in the current membership list reply successfully, the master can execute the invoked RPC locally and reply success to the client. If the invoke RPC on a slave fails, the primary returns `rsm_client_protocol::BUSY` so that the `rsm_client` could resend the request to the new primary later. The slave failure indicated by the time out would be picked up by the config layer to form a new view, allowing the system to make progress. Apparently we are assuming that the failure is caused by a crash of the slave; otherwise the system may hang up since the replicas are probably inconsistent from now on. Thus, you have to set the RPC timeout sufficiently large to avoid any other conditions that could have caused the RPC time out. The staff implemnetation is able to pass the tests using timeout of 1 second.

- `rsm::invoke()`. This RPC handler is invoked on slaves by the master. A slave must ensure the request has the expected sequence number before executing it locally and reply back to the master. It should also ensure that the slave is in a stable view (`rsm::inviewchange` is false) and it is indeed a slave under current view; if not, it should reply with an error.

- The `rsm::inviewchange` variable keeps track of whether the current replica has successfully synchronized its state with the current master upon the latest view change. If a node has not finished state synchronization, it should not process any RSM requests. For this first step, we do not yet worry about replica failures nor state synchronization.

To change your lock server/client to use the RSM objects:

- Eliminate any randomness in the lock server if there is any, or at the very least make sure the random number generator in each replica is seeded with the same value.

- `lock_server_cache_rsm` has been modified to take in a `rsm` object in its constructor (e.g., as a pointer) and save it, so that each server can inquire about its master status using the `rsm::amiprimary()` method. Only the primary `lock_server_cache_rsm` should communicate directly with the client.

Therefore, you should modify `lock_server_cache_rsm` to check if it is the master before communicating with the lock client(s).

- Modify `lock_client_cache_rsm` to create a `rsm_client` object in its constructor. Then make `lock_client_cache_rsm` to use the `rsm_client` object to perform its RPCs, in place of the old `rpcc` object (no longer needed). The `lock_client_cache_rsm` sends RPCs as usual with the `call` method of the `rsm_client` object. The method will further call `rsm_client::invoke` with marshalled request.

Upon completion of step two, you should be able to pass '`./rsm_tester.pl 8`'. This test starts three `lock_server`s one after another, waits for Paxos to reach an agreement, then performs tests on the lock service using `lock_tester`.

Step Three: Cope with Backup Failures and Implement state transfer

In this step, you will handle node failures as well as joins in a running RSM. Upon detecting failure or a new node joining, the underlying Paxos protocol is kicked into action. When Paxos has reached an agreement on the next new view, it calls the rsm object's `commit_change()` to indicate that a new view is formed. When a new view is first formed, the `rsm::inviewchange` variable is set to true, indicating that this node needs to recover its RSM state before processing any RSM requests again. Recovery is done in a separate `recoverythread` in the `rsm::recovery()` method.

After a view change, each replica should recover by transferring state from the master. Its state must be identical to the master's before processing any RSM requests in the new view. Once recovery is finished, the replica should set its `rsm::inviewchange` variable to false to allow the processing of RSM requests. The master should not send any requests to the backups until all the backups have recovered.

To implement state transfer, first make `lock_server_cache_rsm` into a subclass of `rsm_state_transfer` interface. Second, implement the `marshal_state` and `unmarshal_state` methods for `lock_server_cache_rsm`. Use the YFS RPC marshalling code to turn various internal state into strings and vice versa. For example, if state of your lock server consists of a `std::map` called `locks` that mapped lock name (`std::string`) to a list of clients waiting to grab the lock (`std::vector`), the code might look roughly as follows:

```
std::string
lock_server_cache_rsm::marshal_state() {

  // lock any needed mutexes
  marshall rep;
  rep << locks.size();
  std::map< std::string, std::vector >::iterator iter_lock;
  for (iter_lock = locks.begin(); iter_lock != locks.end(); iter_lock++) {
    std::string name = iter_lock->first;
    std::vector vec = locks[name];
    rep << name;
    rep << vec;
  }
  // unlock any mutexes
  return rep.str();

}

void
lock_server_cache_rsm::unmarshal_state(std::string state) {

  // lock any needed mutexes
  unmarshall rep(state);
  unsigned int locks_size;
  rep >> locks_size;
  for (unsigned int i = 0; i < locks_size; i++) {
    std::string name;
```

```
    rep >> name;
    std::vector vec;
    rep >> vec;
    locks[name] = vec;
  }
  // unlock any mutexes
}
```

In the `lock_server_cache_rsm` constructor, call the `rsm`'s `set_state_transfer` method with `this` as the argument so that `rsm` can call `lock_server_cache_rsm`'s `marshal_state` and `unmarshal_state` function later.

Then you have to implement the following functions to syncrhonize the states among primary and backups.

- `rsm::sync_with_backups`. After a view change, the recovery thread of the new primary calls this function to wait until all backups the the new view. The primary has to wait because it cannot process any request until all backups are synchronized with the new primary.
- `rsm::sync_with_primary`. After a view change, the recovery thread on each of the backups calls this function to synchronize with the new primary.
- `rsm::statetransferdone`. Once the slave has synchronized with the primary, it calls this function to inform the primary that it has syncrhonized.
- `rsm::transferdonereq`. This is the corresponding RPC handler of the above RPC call. The primary keeps track of the number of synchronized backups in this function, and wakes up the recovery thread once all has syncrhonized so that the primary can start processing requests from client.

See the comment in these functions for more instructions.

Now you should be able to pass '`./rsm_tester.pl 9 10`'. These tests starts three lock servers and kills or restarts the second slave while running the `lock_tester` simultaneously.

Step Four: Cope with Primary Failures

The `rsm_client::invoke()` method handles two special cases. First, if the replica that the client sends the `invoke` RPC to is no longer the primary, that replica returns `rsm_client_protocol::NOTPRIMARY`. In this case, the client calls `init_members()`, which sends a `members` RPC to the old primary to update its list of replicas. Then the client retries its request.

The second case is where the replica isn't responding at all (so the `invoke` RPC fails). In this case, `init_members()` won't work because the `members` RPC will also fail. In this case, the client calls `rsm_client::primary_failure()`, which should forget about the failed primary and choose a different replica as the new primary to contact with. Then `invoke()` should retry as before.

We have given you most of the code you need. Your job is simply to write `rsm_client::primary_failure()` to handle the second case.

The challenge of dealing with primary failure is handling duplicated requests. Consider what happens if the primary crashes while some replicas have executed the request and others have not. A view change will occur, and the client will re-send the request in the new view. If the new primary has already executed the request, the RPC handler in `lock_server_cache_rsm` will be invoked twice. A good way to handle these cases is to assign sequence numbers to all requests, as described in "Step One". Since you have already implemented sequence numbers in Step One, now it is a good time to test whether you are using sequence numbers correctly to tolerate primary failures.

Note that if the primary crashes while (or shortly after) executing an `acquire` or `release` RPC, after recovery it will be ambiguous as to whether the appropriate `retry` or `revoke` RPCs were sent in the previous view. A simple

way to address this is to have clients that are waiting to acquire locks retry automatically every 3 seconds, even in the absence of a `retry` RPC. The servers can use sequence numbers to identify duplicate `acquire` requests; however, when a server gets a duplicate `acquire` and another client holds the lock, it should send another `revoke` anyway, in case the first `revoke` got lost due to a crash.

Now you should be able to pass '`./rsm_tester.pl 11`'. This test starts three lock servers and kills the primary while running the `lock_tester` simultaneously.

Step Five: Complicated failures

In `rsm::client_invoke`, place the function `breakpoint1()` after the master has finished invoking RSM request on one slave and before it moves on to issue RSM request to other slaves. In the three server test scenario (test 12), this causes the master to fail after one slave has finished the latest request and the other slave has not seen the latest request yet. If you have implemented recovery correctly, the set of RSM servers in the new view resolve this case correctly and all master/slaves will start executing requests from identical state. Note that since the `rsm_client` has not heard back from the master in the previous view, it will retry its request in the new view (in `rsm_client::invoke()`). This might cause your lock server to execute duplicate requests, but that is OK as long as these requests are idempotent, meaning they can be executed multiple times in a row without affecting correctness.

Next, place the function `breakpoint1()` in `rsm::invoke` just after the slave has finished executing a request. In the three server test scenario (test 13), this causes the second slave to fail after it has finished the latest request. Again, if you have implemented recovery correctly, the set of RSM servers in the new view resolve this case correctly and all master/slaves will start executing requests from identical state.

Then, call the function `breakpoint2()` in `rsm::commit_change` just before exiting the function and only when the node is part of the new view. Test 14 starts five server, kills one slave at `breakpoint1()` and promptly kills another at `breakpoint2()`.

Test 15 is exactly like test 14 except that the primary is killed at `breakpoint2()` instead of a slave.

Finally, add the function `partition1()` right after `breakpoint1()` in `rsm::client_invoke`. This function induces a partition that splits the node away from the rest of the nodes. In the three server test scenario (test 16), this causes the primary to loose communication with the two slaves and allows the slaves to form a new view. After the slaves from a new view, the partition is healed and the old primary is allowed to join the system. Your code should be able to deal with the primary being partitioned from the slaves and with an old primary joining the system which has elected a new primary.

If your RSM works correctly, you should be able to pass '`./rsm_tester.pl 12 13 14 15 16`'.

Challenges

Here are a few things you can do if you finish the lab early and feel like improving your code. These are **not** required, and there are no associated bonus points, but some of you may find them interesting.

- There's no reason that the slaves replicas must receive the RSM requests and execute in series (in lockstep). You may be able to improve performance by executing them in parallel. However, this could complicate recovery.
- Modify the RSM library to log its received requests on disk. This way, even if all lock servers fail simultaneously, we can recover them later.
- Use the RSM library to replicate the extent server. As a further extension, you could improve performance by using a recovery scheme that is more efficient than transferring all of the file system state.

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`-lab7.tgz lab/
```

or

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

That should produce a file called [your_user_name]-lab7.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.