Project 2: Part 1: RPC and Locks

Due: 11:59PM March 1, 2011

1 Introduction

In this series of labs, you will implement a fully functional distributed file server. To work correctly, the yfs servers need a locking service to coordinate updates to the file system structures. In this lab, you'll implement the lock service.

The core logic of the lock service is quite simple and consists of two modules, the lock client and lock server that communicate via RPCs. A client requests a specific lock from the lock server by sending an acquire request. The lock server grants the requested lock to one client at a time. When a client is done with the granted lock, it sends a release request to the server so the server can grant the lock to another client who also tried to acquire it in the past.

In addition to implementing the lock service, you'll also augment the provided RPC library to ensure at-most-once execution by eliminating duplicate RPC requests. Duplicate requests exist because the RPC system must re-transmit lost RPCs in the face of lossy network connections and such re-transmissions often lead to duplicate RPC delivery when the original request turns out not to be lost, or when the server reboots.

Duplicate RPC delivery, when not handled properly, often violates application semantics. Here's an example of duplicate RPCs causing incorrect lock server behavior: A client sends an acquire request for lock x, server grants the lock, client releases the lock with a release request, a duplicate RPC for the original acquire request then arrives at the server, server grants the lock again, but the client will never release the lock again since the second acquire is just a duplicate. Such behavior is clearly incorrect.

2 Development Environment

For Project 2, will provide you with a pre-configured development environment in the form of a VM (Virtual Machine) image.

The image we are providing contains an installation of Ubuntu Linux, pre-configured with FUSE to allow you to test out your filesystem.

The image can be found at:

`http://moo.cmcl.cs.cmu.edu/~okatkova/440.vdi.bz2`

You are welcome to set up and use any environment, but we will only be supporting this configuration.

Please see the additional handout on installing and using VirtualBox, and let us know as soon as possible if you have issues with getting set up.

3 Important information

Please read this section carefully.

Starter code for this lab is provided in the handout directory. The starter code can be found in:

```
/afs/andrew/course/15/440-sp11/handout/lab2/
```

There is a subdirectory for each part of the lab. When starting subsequent parts of the lab, please make sure to copy in the additional files corresponding to that part.

Please hand in your submission to:

```
/afs/andrew/course/15/440-sp11/handin/proj2/your_andrew_id/part1/
```

Copy all the files that you used, including the Makefile. We should be able to compile and test your code directly from your /part1 folder. Do not hand in object files, binaries, or core dumps.

As with project 1, you can make multiple submissions by naming the directory for subsequent turnins `your_andrew_id.1`, `your_andrew_id.2`, etc.

However, please do make sure that your last submission contains the most up-to-date version of each of the 6 parts of the lab.

4 Getting started

In this lab, we provide you with a skeleton RPC-based lock server, a lock client interface, a sample application that uses the lock client interface, and a tester. Now compile and start up the lock server, giving it a port number on which to listen to RPC requests. You'll need to choose a port number that other programs aren't using. For example:

```
% make
% ./lock_server 3772
```

Now open a second terminal on the same machine and run lock_demo, giving it the port number on which the server is listening:

```
% ./lock_demo 3772
stat request from clt 1450783179
stat returned 0
%
```

lock_demo asks the server for the number of times a given lock has been acquired, using the stat RPC that we have provided. In the skeleton code, this will always return 0. You can use it as an example of how to add RPCs. You don't need to fix stat to report the actual number of acquisitions of the given lock in this lab, but you may if you wish.

The lock client skeleton does not do anything yet for the acquire and release operations; similarly, the lock server does not implement any form of lock granting or releasing. Your job in this lab is to fill in the client and server function and the RPC protocol between the two processes.

5 Your Job

Your first job is to implement a correct lock server assuming a perfect underlying network. In the context of a lock service, correctness means obeying this invariant: at any instance of time, there is at most one client holding a lock of a given name.

We will use the program lock_tester to check the correctness invariant, i.e. whether the server grants each lock just once at any given time, under a variety of conditions. You run lock_tester with the same arguments as lock_demo. A successful run of lock_tester (with a correct lock server) will look like this:

```
% ./lock_tester 3772
simple lock client
acquire a release a acquire a release a
acquire a acquire b release b releasea
test2: client 0 acquire a release a
test2: client 2 acquire a release a
. . .
./lock_tester: passed all tests successfully
```

If your lock server isn't correct, lock_tester will print an error message. For example, if lock_tester complains "error: server granted a twice!", the problem is probably that lock_tester sent two simultaneous requests for the same lock, and the server granted the lock twice (once for each request). A correct server would have sent one grant, waited for a release, and only then sent a second grant.

Your second job is to augment the RPC library to guarantee at-most-once execution. We simulate lossy networks on a local machine by setting the environmental variable RPC_LOSSY. **A positive RPC_LOSSY value will result in message either being dropped, delayed or sent twice**. If you can pass both the RPC system tester and the lock_tester, you are done. Here's a successful run of both testers:

```
% export RPC_LOSSY=0
```

```
% ./rpctest
simple test
. . .
rpctest OK

% killall lock_server
% export RPC_LOSSY=5
% ./lock_server 3722 &
% ./lock_tester 3772
simple lock client
acquire a release a acquire a release a
. . .
./lock_tester: passed all tests successfully
```

For this lab, your lock server and RPC augmentation must pass the both rpctest and lock_tester; you should ensure it passes several times in a row to guarantee there are no rare bugs. lock_tester will succeed with RPC_LOSSY=0 if you implement the lock server functionality properly.

You should only make modifications on files rpc.cc,h, lock_client.cc,h, lock_server.cc,h and lock_smain.cc. We will test your code with with our own copy of the rest of the source files and testers. You are free to add new files to the directory as long as the Makefile compiles them appropriately, but you should not need to.

For this lab, you will not have to worry about server failures or client failures. You also need not be concerned about security such as malicious clients releasing locks that they don't hold.

6 Detailed Guidance

In principle, you can implement whatever design you like as long as your implementation satisfies all requirements in the "Your Job" section and passes the tester. To be nice, we provide detailed guidance and tips on a recommended implementation plan. You do not have to follow our recommendations, although doing so makes your life easier and allows maximal design/code re-use in later labs.

Step One: implement the lock_server assuming a perfect network

First, you should get the lock_server running correctly without worrying about duplicate RPCs under lossy networks.

- **Using the RPC list:** The RPC library's source code is in the files rpc.cc, chan.cc,, and host.cc. To use it, the lock_server creates a RPC server object (rpcs) listening on a port and registers various RPC handlers (see an example in lock_smain.cc). The lock_client creates a RPC client object (rpcc), binds it to the lock_server's address (127.0.0.1) and port, and invokes RPC calls (see an example in lock_client.cc).

  Each RPC procedure is identified by a unique procedure number. We have defined the acquire and release RPC numbers you will need in lock_protocol.h. Other RPC numbers defined there are for use in later labs. Note that you must still register handlers for these RPCs with the RPC server object.

You can learn how to use the RPC system by studying the given stat call implementation across lock_client and lock_server. All RPC procedures have a standard interface with x+1 (x must be less than 6) arguments and an integer return value (see the example in lock_server::stat function). The last argument, a reference to an arbitary type, is always there so that a RPC handler can use it to return results (e.g. lock_server::stat returns the number of acquires for a lock). **Remember that the reference must always be the last argument : it will be important in Parts 2-4**. The RPC handler also returns an integer status code, and the convention is to return zero for success and to return positive numbers otherwise for various errors. If the RPC fails at the RPC library (e.g.timeouts), the RPC client gets a negative return value instead. The various reasons for RPC failures at the RPC library are defined in rpc.h under rpc_const.

The RPC system must know how to marshall arbitary objects into a stream of bytes to transmit over the network and unmarshall them at the other end. The RPC library has already provided marshall/unmarshall methods for standard C++ objects such as std::string, int, char (see file rpc.cc). If your RPC call includes different types of objects as arguments, you must provide your own marshalling method. You should be able to complete this lab with existing marshall/unmarshall methods.

Suggested implementation step 1: Add the acquire and release calls to the client and the server. For starters, don't have them do anything — just have the server print out that it got an acquire request (release request) for lock ID x. This will require changes to `lock_server.h`, `lock_server.cc`, `lock_smain.cc` and `lock_client.cc`. When you run `lock_tester`, you should see the server print out some acquire and release messages.

- **Implementing the lock server:** The lock server can manage many distinct locks. Each lock is identified by an integer of type lock_protocol::lockid_t. The set of locks is open-ended: if a client asks for a lock that the server has never seen before, the server should create the lock and grant it to the client. When multiple clients simultaneously request a given lock, the lock server must grant the lock to each client one at a time.

  You will need to modify the lock server skeleton implementation in files lock_server.cc,h to accept acquire/release RPCs from the lock client, and to keep track of the state of the locks. Here is our suggested implementation plan.

  On the server, a lock can be in one of two states: 1) free: no clients own the client; or 2) locked: some client owns the lock. The RPC handler for acquire first checks if the lock is locked, and if so, the handler blocks until the lock is free. When the lock is free, acquire changes its state to locked, then returns to the client, which indicates that the client now has the lock. The value r returned by acquire doesn't matter.

  The handler for release changes the lock state to free, and notifies any threads that are waiting for the lock.

- **Implementing the lock client:** The class lock_client is a client-side interface to the lock server (found in files lock_client.cc,h). The interface provides acquire() and release() functions that are supposed to take care of sending and receiving RPCs. Multiple threads in the client program can use the same lock_client object and request the same lock name. See lock_demo.cc for an example of how an application uses the interface. **Note that a basic requirement of the client interface is that lock_client::acquire must not return until that lock is granted.**

- **Handling multi-thread concurrency:** Both lock_client and lock_server's functions will be invoked by multiple threads concurrently. In particular, the RPC library always launches a new thread to invoke the RPC handler at the RPC server. Many different threads might also call lock_client's acquire() and release() functions simultaneously.

  To protect access to shared data in the lock_client and lock_server, you need to use pthread mutexes. Please refer to the general tips for programming using threads. As seen from the suggested implementation plan, you also need to use pthread condition variables to synchronize the actions among multiple threads. Condition variables go hand-in-hand with the mutexes.

  For robustness, when using condition variables, it is recommended that when a thread that waited on a condition variable wakes up, it checks a boolean predicate(s) associated with the wake-up condition. This protects from spurious wake-ups from the pthread_cond_wait() and pthread_cond_timedwait() functions. For example, the suggested logic described above lends itself to such an implementation (see how on the lock_client, a thread that wakes up checks the state of the lock.)

  In this and later labs, we try to adhere to a simple (coarse-grained) locking convention: we acquire the subsystem/protocol lock at the beginning of a function and release it before returning. This convention works because we don't require atomicity across functions, and we don't share data structures between different subsystems/protocols. You will have an easier life by sticking to this convention.

Step two: Implement at-most-once delivery in RPC

After your lock server has passed lock_tester under a perfect network, enable RPC_LOSSY by typing "export RPC_LOSSY=5", restart your lock_server and try lock_tester again. If you implemented lock_server in the simple way as described previously, you will see the lock_tester fail (or hang indefinitely). Try to understand exactly why your lock_tester fails when re-transmissions cause duplicate RPC delivery. Read the RPC source code in rpc/rpc.cc,h and try to grasp the overall structure of the RPC library as much as possible first by yourself without reading the hints below.

The `rpcc` class handles the RPC client's function. At its core lies the `rpcc::call1` function, which accepts a marshalled RPC request for transmission to the RPC server. We can see that `call1` attaches additional RPC fields to each marshalled request:

```
// add RPC fields before req
m1 << clt_nonce << svr_nonce << proc << myxid << xid_rep_window.front() << req.str();
```

What's the purpose for each of these fields? (Hint: most of them are going to help you implement at-most-once delivery) After call1 has finished preparing the final RPC request, it sits in a "while(1)" loop to (repeatedly) update the timeout value for the next retransmission and waits for the corresponding RPC reply or timeout to happen.

The rpcs class handles the RPC server's function. It creates a separate thread (executing rpcs::loop) that continously tries to read RPC requests from the underlying channel (e.g. a TCP connection). Once a request is read succesfully, it spawns a new thread to dispatch this request to the registered

RPC handler. The function rpcs::dispatch implements the dispatch logic. It extracts various RPC fields from the request. These fields include the RPC procedure number which is used to find the corresponding handler. Additionally, they also provide sufficient information for you to ensure the server can eliminate all duplicate requests.

How do you ensure at-most-once delivery? A strawman approach is to make the server remember all unique RPCs ever received. Each unique RPC is identified by both its xid (unique across a client instance) and clt_nonce (unique across all client instances). In addition to the RPC ids, the server must also remember the actual values of their corresponding replies so that it can re-send the (potentially lost) reply upon receiving a duplicate request without actually executing the RPC handler. This strawman guarantees at-most-once, but is not ideal since the memory holding the RPC ids and replies can grow indefinitely. A better alternative is to use a sliding window of remembered RPCs at the server. Such an approach requires the client to generate xid in a strict sequence, i.e. 0, 1, 2, 3... When can the server safely forget about a received RPC and its response, i.e. sliding the window forward?

Once you figure out the basic design for at-most-once delivery, go ahead and realize your implementation in rpc.cc (rpc.cc is the only file you should be modifying). Hints: you need to add code in three places, rpcc:rpcc constructor to create a thread to enable retransmissions, rpcs:add_reply to remember the RPC reply values and rpcs::checkduplicate_and_update to eliminate duplicate xid and update the appropriate information to help the server safely forget about certain received RPCs.

After you are done with step two, test your RPC implementation with ./rpctest and RPC_LOSSY set to 0 ("export RPC_LOSSY=0"). Make sure ./rpctest passes all tests. Once your RPC implementation passes all these tests, test your lock server and rpctest again in a lossy environment by restarting your lock_server and lock_tester after setting RPC_LOSSY to 5 ("export RPC_LOSSY=5" if using bash). Note that rpctest may take several minutes to complete with RPC_LOSSY=5.

Don't modify the file host.cc because we will replace it with the one you received to test your code.

7 C++ Tutorials and Resources

- C++ Tutorial
  `http://www.cplusplus.com/doc/tutorial/`
- C++ Reference
  `http://www.cppreference.com/wiki/start`

8 Common problems

- Remember to initialize the pthread_mutex_t and pthread_cond_t variables before using them. See the man pages for pthread_mutex_init and pthread_cond_init.