6.824 - Spring 2012

6.824 Lab 1: Lock Server

Due: Friday, February 17th, 5:00pm

Introduction

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

The core logic of the lock service consists of two modules, the lock client and the lock server, which 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 (if any) waiting to acquire the lock.

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.

Duplicate RPC delivery, when not handled properly, often violates application semantics. Here's a example of duplicate RPCs causing incorrect lock server behavior. A client sends an `acquire` request for lock x, the server grants the lock, the client releases the lock with a `release` request, a duplicate RPC for the original `acquire` request then arrives at the server, the 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.

Software

The files you will need for this and subsequent lab assignments in this course are distributed using the Git version control system. To learn more about Git, take a look at the Git user's manual, or, if you are already familiar with other version control systems, you may find this CS-oriented overview of Git useful.

The URL for the course Git repository is http://am.lcs.mit.edu/6.824-2012/yfs-class.git. To install the files in your Athena account, you need to *clone* the course repository, by running the commands below. You must use an x86 or x86_64 Athena machine; that is, `uname -a` should mention `i386 GNU/Linux` or `i686 GNU/Linux` or `x86_64 GNU/Linux`. You can log into a public i686 Athena host with `ssh -X linerva.mit.edu`.

```
athena% mkdir ~/6.824
```

```
athena% cd ~/6.824
athena% add git
athena% git clone http://am.lcs.mit.edu/6.824-2012/yfs-class.git lab
Initialized empty Git repository in ../6.824/lab/.git/
got c9c80a1686710307b99b2c8642311e001920641f
walk c9c80a1686710307b99b2c8642311e001920641f
got 468233b68dff78915966975f5e9aad54beeea84e
...
got 60e68c3bfa43d5d6d1f2b8a2c7a1d5db5fa0e860
Checking out files: 100% (44/44), done.
athena% cd lab
athena%
```

Git allows you to keep track of the changes you make to the code. For example, if you are finished with one of the exercises, and want to checkpoint your progress, you can commit your changes by running:

```
athena% git commit -am 'my solution for lab1 exercise9'
Created commit 60d2135: my solution for lab1 exercise9
 1 files changed, 1 insertions(+), 0 deletions(-)
athena%
```

You can keep track of your changes by using the `git diff` command. Running `git diff` will display the changes to your code since your last commit, and `git diff origin/lab1` will display the changes relative to the initial code supplied for this lab. Here, `origin/lab1` is the name of the git branch with the initial code you downloaded from our server for this assignment.

Getting started

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:

```
% cd lab
% 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:

```
% cd lab
% ./lock_demo 3772
stat returned 0
%
```

`lock_demo` asks the server for the number of times a particular 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 lock granting or releasing. Your job is to implement this functionality in the server, and to arrange for the client to send RPCs to the server.

Your Job

Your first job is to implement a correct lock server assuming a perfect underlying network. Correctness means obeying this invariant: at any point in time, there is at most one client holding a lock with a given identifier.

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 release a
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 XXX twice", the problem is probably that `lock_tester` sent two simultaneous requests for the same lock, and the server granted both requests. A correct server would have granted the lock to just one client, waited for a release, and only then sent granted the lock to the second client.

Your second job is to augment the RPC library to guarantee **at-most-once** execution. You can tell the RPC library to simulate a lossy network by setting the environment variable `RPC_LOSSY`. 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
% ./rpc/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
```

Your code must pass both ./rpc/rpctest and lock_tester; you should ensure it passes several times in a row to guarantee there are no rare bugs. 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 malicious or buggy applications.

Detailed Guidance

In principle, you can implement whatever design you like as long as it satisfies the requirements in the "Your Job" section and passes the testers. In practice, you should follow the detailed guidance below. You might want to look at the general programming tips in the lab overview page.

Step One: implement the lock_server assuming a perfect network

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

- Using the RPC system:

  The RPC library's source code is in the subdirectory rpc/. A server uses the RPC library by creating an RPC server object (rpcs) listening on a port and registering various RPC handlers (see lock_smain.cc). A client creates a RPC client object (rpcc), asks for it to be connected to the lock_server's address and port, and invokes RPC calls (see 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. You must register handlers for these RPCs with the RPC server object (see lock_smain.cc).

  You can learn how to use the RPC system by studying the stat call implementation in lock_client and lock_server. RPC handlers have a standard interface with one to six request arguments and a reply value implemented as a last reference argument. The handler also returns an integer status code; the convention is to return zero for success and to return positive numbers for various errors. If the RPC fails in the RPC library (e.g.timeouts), the RPC client gets a negative return value instead. The various reasons for RPC failures in the RPC library are defined in rpc.h under rpc_const.

  The RPC system marshalls objects into a stream of bytes to transmit over the network and unmarshalls them at the other end. Beware: the RPC library does not check that the data in an arriving message have the expected type(s). If a client sends one type and the server is expecting a different type, something bad will happen. You should check that the client's

RPC call function sends types that are the same as those expected by the corresponding server handler function.

The RPC library provides marshall/unmarshall methods for standard C++ objects such as `std::string`, `int`, and `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.

- 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 request the same lock, the lock server must grant the lock to one client 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;
  - free: no clients own the client
  - locked: some client owns the lock

  The RPC handler for `acquire` should first check if the lock is locked, and if so, the handler should block 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` should change the lock state to free, and notify any threads that are waiting for the lock.

  Consider using the C++ STL (Standard Template Library) `std::map` class to hold the table of lock states.

- 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 should send and receive RPCs. Multiple threads in the client program can use the same `lock_client` object and request the same lock. See `lock_demo.cc` for an example of how an application uses the interface. `lock_client::acquire` **must not return until it has acquired the requested lock.**

- Handling multi-thread concurrency:

  Both `lock_client` and `lock_server`'s functions will be invoked by multiple threads concurrently. On the lock server side, the RPC library keeps a thread pool and invokes the RPC handler using one of the idle threads in the pool. On the lock client side, many different threads might also call `lock_client`'s `acquire()` and `release()` functions concurrently.

You should use pthread mutexes to guard uses of data that is shared among threads. You should use pthread condition variables so that the lock server acquire handler can wait for a lock. The general tips contain a link to information about pthreads, mutexes, and condition variables.

Threads should wait on a condition variable inside a loop that checks the boolean condition on which the thread is waiting. This protects the thread from spurious wake-ups from the `pthread_cond_wait()` and `pthread_cond_timedwait()` functions.

Use a simple mutex scheme: a single pthreads mutex for all of `lock_server`. You don't really need (for example) a mutex per lock, though such a setup can be made to work. Using "coarse-granularity" mutexes will simplify your code.

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

The RPC code we provide you has a complete client implementation of at-most-once delivery: the client code times out while waiting for a response, re-sends the request, and accompanies each request with information the server will need for its part of at-most-once delivery. However, the code is missing some of the server at-most-once code, in particular the implementation of the functions `rpcs::checkduplicate_and_update` and `rpcs::add_reply`. It is your job to implement those two functions.

After your lock server has passed `lock_tester`, test it with a simulated lossy network: type "`export RPC_LOSSY=5`", restart your lock_server, and try lock_tester again. Very likely 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.

Skim 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 before proceeding. `rpc.cc` already contains some of the code required to cope with duplicate requests; your job will be to complete that code.

The `rpcc` class manages RPC calls for clients. At its core lies the `rpcc::call1` function, which accepts a marshalled RPC request for transmission to the RPC server. `call1` attaches additional RPC fields to each marshalled request:

```
// add RPC fields before the RPC request data
req_header h(ca.xid, proc, clt_nonce_, srv_nonce_, xid_rep_window_.front());
req.pack_req_header(h);
```

What's the purpose for each field in req_header? (Hint: many 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. Also, if the underlying (TCP) connection to the server fails, rpcc automatically re-connects to the server again (in function `get_refconn`) in order to retransmit.

The `rpcs` class manages RPC calls for the server. When a connection receives an RPC request, it calls `rpcs::got_pdu` to dispatch the request to a thread from the pool. The thread pool (`class`

ThrPool) consists of a fixed number of threads that call rpcs::dispatch to dispatch an RPC request to the relevant registered RPC handler. rpcs::dispatch extracts RPC fields from the request, including the RPC procedure number which is used to find the corresponding handler. The header fields also provide sufficient information for you to ensure that the server eliminates all duplicate requests.

Question: The partial lock server we provide you uses "blocking" RPC handlers that sometimes wait (for lock releases). How many concurrent "blocking" lock acquire requests can the server handle? (Hint: our implementation of rpcs currently uses a thread pool of 10 threads).

How to 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 original return value for each RPC so that the server can re-send it in response to a duplicate request. This strawman guarantees at-most-once, but is not ideal since the memory holding the RPC ids and replies grows 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. slide the window forward? What if a retransmitted request arrives while the server is still processing the original request?

Once you figure out the basic design for at-most-once delivery, go ahead and implement your design in rpc.cc and (if needed) rpc.h; you should not need to modify any other files. You need to add code in two places. First, rpcs::checkduplicate_and_update should 1) check if a request is a duplicate and return information about the remembered reply if it is, 2) remember that a new request has arrived if it is not a duplicate, and 3) trim the window of remembered requests and reply values. Second, rpcs::add_reply should remember the RPC reply values for an RPC call that the server has completed. You should store the remembered RPC reply values in rpcs::reply_window_, declared in rpc.h.

After you are done with step two, test your RPC implementation with ./rpc/rpctest and RPC_LOSSY set to 0 ("export RPC_LOSSY=0"). Make sure ./rpc/rpctest passes all tests. Once your RPC implementation passes all these tests, test your lock server again in a lossy environment by restarting your lock_server and lock_tester after setting RPC_LOSSY to 5 ("export RPC_LOSSY=5"). The RPC_LOSSY environment variable must be set for both lock_server and lock_tester.

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 ~/6.824
% tar czvf `whoami`-lab1.tgz lab/
```

or

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

That should produce a file called `[your_user_name]-lab1.tgz` in your `lab/` directory. Attach that file to an email and send it to the address above.

You will receive full credit if your software passes the same tests we gave you when we run your software on our machines.

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