Project 2: Part 5: Caching Locks

Due: 11:59PM Thursday, March 24, 2011

1 Introduction

In this part you will build a server and client that cache locks at the client, reducing the load on the server and improving client performance. For example, when client 1 asks for lock 42 repeatedly and no other client wants the lock, then all acquire and releases can be performed on client 1 without having to contact the server.

The challenge in the lab is the protocol between the clients and the server. For example, when client 2 acquires a lock that client 1 has cached, the server must revoke that lock from client 1 by sending a revoke RPC to client 1. The server can give client 2 the lock only after client 1 has released the lock, which may be a long time after sending the revoke (e.g., if a thread on client 1 holds the lock for a long period). The protocol is is further complicated by the fact that packets may be lost, duplicated, and delivered out of order. The at-most-once RPC server you implemented in Part 1 will take care of most of these problems, but RPCs may still be delivered out of order. We will test your lock server code with RPC_LOSSY set to 5, like in Part 1.

To make the lock service fault tolerant, we will want to put a constraint on the lock server operations. This constraint will have implications for the protocol between the lock clients and lock server. The constraint is that handlers on the server should run to completion without blocking. That is, a server thread should not block on condition variables or remote RPCs. Of course, the thread can wait to take out locks as long as it can be sure that the lock will never be held by another thread across an RPC, and once it has acquired the lock it should run to completion. Your implementation of the lock server for this part should adhere to this non-blocking constraint.

Your server will be a success if it manages to operate out of its local lock cache when reading/writing files and directories that other hosts aren't looking at, but maintains correctness when the same files and directories are concurrently read and updated on multiple hosts. We will test with both RPC_LOSSY set to 0 and RPC_LOSSY set to 5.

2 Getting started

As in part 4, copy the additional files found in the part5 subdirectory of the handout into your working directory. **Make sure you choose to keep the new Makefile.**

The following is some additional information about the new files given to you and other changes made.

- `lock_client_cache.cc,h`: This will be the new lock client class that the lock_tester and your

yfs_client should instantiate. Because the client will now be receiving asynchronous RPCs from the server, it will now need to listen to RPCs, and also have some way of notifying the server of its network address for these future RPCs. We have provided you with some code in the lock_client_cache constructor that picks a random port to listen on, creates an RPC server for that port, and constructs an id string that the client can send to the server when requesting a lock. The server can then use the make_sockaddr method (as in lock_client.cc, for example) to make a sockaddr_in object suitable for use with rpcc.call.

We also provide code that launches a thread to wait for locks to be revoked by the server, and when the higher class releases the lock locally, this thread should send a release RPC to the server.

Note that although lock_client_cache extends the lock_client class from Part 1, you probably won't be able to reuse any code from the parent class; we use a subclass here so that higher classes can use the two implementations interchangeably. However, you might find certain member variables useful (such as lock_client's RPC client cl).

`lock_server_cache.cc,h`: This class does not extend lock_server. Again, we provide with code that launches threads: one that will notify clients that a previously-requested lock may now be available, and one that sends revoke RPCs to the holders of locks requested by another client. This class should be instantiated by lock_smain.cc, which should also register the RPC handlers for the new class.

- `lock_tester.cc`: The lock tester will now instantiate lock_client_cache objects.
- `lock_protocol.h`: In part 1, we defined a protocol in which the client sends RPCs to the server. In this part, we have defined a second protocol for the RPCs that the server sends to the client.
- `lock_smain.c`: #include lock_server_cache.h instead of lock_server.h

In order to evaluate the caching performance of your caching lock server and clients, the RPC library has a feature that counts unique RPCs arriving at the server. You can set the environment variable RPC_COUNT to N before you launch a server process, and it will print out RPC statistics every N RPCs. For example, in the bash shell you could do:

```
% export RPC_COUNT=25
% ./lock_server 3772
RPC STATS: 7001:23 7002:2
...
```

This means that the RPC with the procedure number 0x7001 (acquire in the original lock_protocol.h file) has been called 23 times, while RPC 0x7002 (release) has been called twice.

2.1 Testing Performance

Our measure of performance is the number of acquires that your lock clients send to the lock server. You can use RPC_COUNT, as mentioned above, to print out these stats every N calls.

The workload on which we'll evaluate your lock protocol's performance is generated by test-lab-4-c. It creates two subdirectories and creates/deletes 100 files in each directory, using each directory through only one of the two YFS clients.

Using the lock server of part 4, you will see that the number of acquires is at least a few thousand. With the caching lock client, you will see that the number of acquires is only a few hundred (i.e., less than a thousand). We are a bit vague in this performance goal because the exact numbers depend a bit on how you use locks in yfs_client. Suffice it to say, the drop in acquires should be significant.

Of course, your system must also remain correct. We will require the code you hand in to pass all of the part 1 and part 4 testers as well as getting good performance on the test-lab-4-c tester.

3 Protocol and Implementation Hints

There are many ways to correctly implement a protocol, so we've provided a sample protocol for you. You are welcome to add additional states or remove states as needed.

This protocol has most of the complexity on the client. All the handlers on the server run to completion and threads wait on condition variables on the client when a lock is taken out by another thread (on this client or another client).

On the client a lock can be in several states:

- none: client knows nothing about this lock

  free: client owns the lock and no thread has it

- locked: client owns the lock and a thread has it

- revoked: client has received a revoke from the server

In many of the states there may be several threads waiting for the lock, but only one thread per client ever needs to be interacting with the server; once that thread has acquired and released the lock it can wake up other threads, one of which can acquire the lock (unless the lock has been revoked and released back to the server). If you need a way to identify a thread, you can use its thread id (tid), which you can get using pthread_self().

When a client asks for a lock with an acquire RPC, the server grants the lock and responds with OK if the lock is not owned by another client (i.e., the lock is free). If the lock is owned by another client, the server responds with RETRY. At some point later (after another client has released the lock using a release RPC), the server sends the client a retry RPC. The retry RPC informs the client that the lock may be free, and therefore it ought to try another acquire RPC.

Note that RETRY and retry are two different things.On the one hand, RETRY is the value the server returns for a acquire RPC to indicate that the requested lock is not currently available. On the other hand, retry is the RPC that the server sends the client when a previously requested lock becomes available.

Once a client has acquired ownership of a lock, the client caches the lock (i.e., it keeps the lock instead of sending a release RPC to the server when a thread releases the lock on the client). The

client can grant the lock to other threads on the same client without interacting with the server. The server will inform the client when it wants a lock back.

The server sends the client a revoke RPC to get the lock back. This request tells the client that it should send the lock back to the server when it releases the lock or right now if no thread on the client is holding the lock.

For your convenience, we have defined a new RPC protocol called rlock_protocol in lock_protocol.h) to use when sending RPCs from the server to the client. This protocol contains definitions for the retry and revoke RPCs.

A good way to implement releasing locks on the client is using a separate releaser thread (as mentioned above, the skeleton code already launches one for you). When receiving a revoke request, the client adds the revoke request to a list and wakes up the releaser thread. The releaser thread will release the lock (i.e., send a release RPC to the server) when the lock becomes free on the client. Using a separate thread is good because it avoids potential distributed deadlocks and ensures that revoke RPCs from the server to the client run to completion on the client.

On the server, handlers shouldn't block either. A good way to implement this on the server is to have revoker and retrier threads that are in charge of sending retry and revoke RPCs, respectively. When a client asks for a lock that is taken by another client, the acquire handler adds a revoke request to a queue and wakes up the revoker thread. When a client releases a lock, the release handler adds a retry request to a list for the retrier (if there are clients who want the lock) and wakes up the retrier thread. This design ensures that the handlers run to completion. Blocking operations are performed by the retrier and revoker threads, and those blocking operations are just RPCs to the client, whose handlers also should run to completion without blocking.

A challenge in the implementation is that retry and revoke RPCs can be out of order with the acquire and release requests. That is, a client may receive a retry request before it has received the response to its acquire request. Similarly, a client may receive a revoke before it has received a response on its acquire request.

A good way to handle these cases is to 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. You probably want to send no additional acquires for the same lock to the server until the oustanding 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. You may be able to think of a strategy that doesn't require sequence numbers. We suggest that you use sequence numbers anyway, because sequence numbers are easy to reason about, whereas thinking of all possible ways in which reordering might cause problems is harder.

Unless your server in Part 1 has non-blocking handlers already and uses retry and revoke RPCs, you are probably best off starting from scratch for this part. Part 1 didn't require you to write much code, and morphing it into something that is suitable for part 5 is likely to be more work than doing it right from scratch.

As in the previous labs, remember not to hold pthreads mutexes across remote procedure calls.

Holding mutexes across RPCs might seem to work, and it might even pass our tests, but it's a really bad idea! Suppose the lock server does an RPC while holding a mutex. Then the server receives an RPC from a client, and the RPC handler tries to acquire the same mutex. The result is that the client's RPC can't even be processed until the first RPC returns. Not only is this bad for performance, but it can lead to distributed deadlock, which is one of the most tortuous problems you may ever have to debug.

Pthreads mutexes are intended for synchronizing multiple threads' access to shared memory. If you use them correctly, you will only need to hold the mutex for a few microseconds, and this in turn means that you could have a single pthreads mutex protecting the entire state of the lock server without limiting its scalability. This will make your job much simpler.

4 Step One: Design the Protocol

You should design the protocol and basic system structure on paper (after playing perhaps a little bit around with the code). In particular, carefully think through the different scenarios due to reordered messages. Changing the basic system structure and tracking down errors in your implemented protocol is painful. If you have thought through all scenarios before you start implementing and have the right system structure, you can save yourself much time.

The following questions might help you with your design (they are in no particular order):

- Suppose the following happens: Client A releases a lock that client B wants. The server sends a retry RPC to client B. Then, before client B can send another acquire to the server, client C sends an acquire to the server. What happens in this case?

  If the server receives an acquire for a lock that is cached at another client, what is that the handler going to do? Remember the handler shouldn't block or invoke RPCs.

- If the server receives a release RPC, and there is an outstanding acquire from another client, what is the handler going to do? Remember again that the handler shouldn't block or invoke RPCs.

- If a thread on the client is holding a lock and a second thread calls acquire(), what happens? You shouldn't need to send an RPC to the server.

- How do you handle a revoke on a client when a thread on the client is holding the lock?

- If a thread on the client is holding a lock, and a second thread calls acquire(), but there has also been a revoke for this lock, what happens? Other clients trying to acquire the lock should have a fair chance of obtaining it; if two threads on the current node keep competing for the lock, the node should not hold the lock forever.

- How do you handle a retry showing up on the client before the response on the corresponding acquire?

- How do you handle a revoke showing up on the client before the response on the corresponding acquire?

- When do you increase a sequence number on the client? Do you have one sequence number per lock_client object or one per client machine?

- If the server grants the lock to a client and it has a number of other clients also interested in that lock, you may want to indicate in the reply to the acquire that the client should return the lock to the server as soon as the thread on the client calls release(). How would you keep track on the client that the thread's release() also results in the lock being returned to the server? (You could alternatively send a revoke RPC immediately after granting the lock in this case.)

5 Step Two: Lock Client and Server, and Testing with RPC_LOSSY=0

A reasonable first step would be to implement the basic design of your acquire protocol on both the client and the server, including having the server send revoke messages to the holder of a lock if another client requests it. This will involve registering RPC handlers on the client, and devising a way for the server to receive and remember each client's location address (i.e., the id variable in lock_client_cache) and using it to send the client RPCs.

Next you'll probably want to implement the release code path on both the client and the server. Of course, the client should only inform the server of the release if the lock has been revoked. This will also involve having the server send a retry RPC to the next client in line waiting for the lock.

Also make sure you instantiate a lock_server_cache object in lock_smain.cc, and correctly register the RPC handlers.

Once you have your full protocol implemented, you can run it using the lock tester, just as in Part 1. For now, don't bother testing with loss:

```
% export RPC_LOSSY=0
% ./lock_server 3772
```

Then, in another terminal:

```
% ./lock_tester 3772
```

Run lock_tester. You should pass all tests and see no timeouts. You can hit Ctrl-C in the server's window to stop it.

6 Step Three: Testing the Lock Client and Server with RPC_LOSSY=5

Now that it works without loss, you should try testing with RPC_LOSSY=5. Here you may discover problems with reordered RPCs and responses.

```
% export RPC_LOSSY=5
% ./lock_server 3772
```

Then, in another terminal:

```
% export RPC_LOSSY=5
% ./lock_tester 3772
```

7 Step Four: Run File System Tests

In the constructor for your yfs_client, you should now instantiate a lock_client_cache object, rather than a lock_client object. You will also have to include lock_client_cache.h. Once you do that, your YFS should just work under all the Part 4 tests. We will run your code against all 3 tests (a, b, and c) from Part 4.

You should also compare running your YFS code with the two different lock clients and servers, with RPC count enabled at the lock server. For this reason, it would be helpful to keep your Part 4 code around and intact, the way it was when you submitted it. As mentioned before, you can turn on RPC statistics using the RPC_COUNT environment variable. Look for a dramatic drop in the number of acquire (0x7001) RPCs between your Part 4 and Part 5 code during the test-lab-4-c test.

The file system tests only need to pass with RPC_LOSSY set to 0.

8 Evaluation Criteria

If your caching lock protocol performs well and your system passes all the tests, you're done. In particular, we will be checking the following:

- Your caching lock server passes lock_tester with RPC_LOSSY=0 and RPC_LOSSY=5.

  Your file system using the caching lock client passes all the Part 4 tests (a, b, and c) with RPC_LOSSY=0.

- When you run test-lab-4-c with RPC_LOSSY=0 and your old lock server and client, there are several thousand acquire RPCs. When you run it with your caching lock server and client, the number of acquires is substantially less (e.g., several hundred). If your numbers are on that order of magnitude, you're fine.

9 Handin

Please submit all the files necessary for running Part 5, including the Makefile to:

`/afs/andrew/course/15/440-sp11/handin/proj2/your_andrew_id/part5/`

Please follow the same guidelines outlined in Part 1 for multiple submissions.

10 C++ Tutorials and Resources

- C++ Tutorial
  `http://www.cplusplus.com/doc/tutorial/`

- C++ Reference
  `http://www.cppreference.com/wiki/start`