Lab 4 : Caching Locks

Introduction

In this lab you will build a lock server and client that cache locks at the client, reducing the load on the server and improving client performance. For example, suppose that an application using YFS creates 100 files in a directory. Your Lab 3 yfs_client will probably send 100 acquire and release RPCs for the directory's lock. This lab will modify the lock client and server so that the lock client sends (in the common case) just one acquire RPC for the directory's lock, and caches the lock thereafter, only releasing it if another yfs_client needs it.

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 further complicated by the fact that concurrent RPC requests and replies may not be delivered in the same order in which they were sent.

We'll test your caching lock server and client by seeing whether it reduces the amount of lock RPC traffic that your yfs_client generates. We will test with both RPC_LOSSY set to 0 and RPC_LOSSY set to 5.

Getting Started

Since you are building on the past labs, ensure the code in your Lab 3 directory passes all tests for Labs 1, 2 and 3 before starting this lab.

Begin by merging your solution to lab 3 with the new code for lab 4:

```
% cd ~/lab
% git commit -am 'my solution to lab3'
% git checkout lab4
% git merge lab3
```

This will add these new files to your lab directory:

- lock_client_cache.{cc,h}: This will be the new lock client class that the lock_tester and your yfs_client should instantiate. lock_client_cache must receive revoke RPCs from the server (as well as retry RPCs, explained below), so we have provided you with code in the lock_client_cache constructor that picks a random port to listen on, creates an rpcs for that port, and constructs an id string with the client's IP address and port that the client can send to the server when requesting a lock.

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

- lock_server_cache.{cc,h}: Similarly, you will not necessarily be able to use any code from lock_server. lock_server_cache should be instantiated by lock_smain.cc, which should also register the RPC handlers for the new class.
- handle.{cc,h}: this class maintains a cache of RPC connections to other servers. You will find it useful in your lock_server_cache when sending revoke and retry RPCs to lock clients. Look at the comments at the start of handle.h. You can pass the lock client's id string to handle to tell it which lock client to talk to.
- tprintf.h: this file defines a macro that prints out the time when a printf is invoked. You may find this helpful in debugging distributed deadlocks.

We have also made changes to the following files:
- GNUmakefile: links lock_client_cache into lock_tester and yfs_client, and link lock_server_cache into lock_server.
- lock_tester.cc: creates lock_client_cache objects.
- lock_protocol.h: Contains a second protocol for the RPCs that the lock server sends to the client.
- lock_smain.cc: #include lock_server_cache.h instead of lock_server.h

Step One : Design the Protocol

Your lock client and lock server will each keep some state about each lock, and will have a protocol by which they change that state. Start by making a design (on paper) of the states, protocol, and how the protocol messages generate state transitions. Do this before you implement anything (though be prepared to change your mind in light of experience).
Here is the set of states we recommend for the client:
- 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
- acquiring: the client is acquiring ownership
- releasing: the client is releasing ownership

A single client may have multiple threads waiting for the same 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 not free, and there are other clients waiting for the lock, the server responds with a RETRY. Otherwise, the server sends a revoke RPC to the owner of the lock, and waits for the lock to be released by the owner. Finally, the server sends a retry to the next waiting client (if any), grants the lock and responds with OK.

Note that RETRY and retry are two different things. RETRY is the value the server returns for a acquire RPC to indicate that the requested lock is not currently available. retry is the RPC that the server sends the client which is scheduled to hold a previously requested lock next.

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 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.

The server's per-lock state should include whether it is held by some client, the ID (host name and port number) of that client, and the set of other clients waiting for that lock. The server needs to know the holding client's ID in order to send it a revoke message when another client wants the lock. The server needs to know the set of waiting clients in order to send one of them a retry RPC when the holder releases 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.

Hint: don't hold any mutexes while sending an RPC. An RPC can take a long time, and you don't want to force other threads to wait. Worse, holding mutexes during RPCs is an easy way to generate distributed deadlock.

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

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

Hint: a client may receive a revoke RPC for a lock before it has received an OK response from its acquire RPC. Your client code will need to remember the fact that the revoke has arrived, and release the lock as soon as you are done with it. The same situation can arise with retry RPCs, which can arrive at the client before the corresponding acquire returns the RETRY failure code.

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

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, and retry messages to the next waiting client.

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.

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 Lab 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.

A lock client might be holding cached locks when it exits. This may cause another run of lock_tester using the same lock_server to fail when the lock server tries to send revokes to the previous client. To avoid this problem without worrying about cleaning up, you must restart the lock_server for each run of lock_tester.

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
```

Again, you must restart the lock_server for each run of lock_tester.

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 Lab 3 tests. We will run your code against all 3 tests (a, b, and c) from Lab 3.

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 Lab 3 code around and intact, the way it was when you submitted it. As described below, you can turn on RPC statistics with the RPC_COUNT environment variable. Look for a dramatic drop in the number of acquire (0x7001) RPCs between your Lab 3 and Lab 4 code during the test-lab-3-c test.

The file system tests should pass with RPC_LOSSY set as well. You can pass a loss parameter to start.sh and it will enable RPC_LOSSY automatically:

```
% ./start.sh 5                # sets RPC_LOSSY to 5
```

If you're having trouble, make sure that the Lab 2 tester passes. If it doesn't, then the issues are most likely with YFS under RPC_LOSSY, rather than your caching lock client.

Please restart all the servers using stop.sh and start.sh for each run of each test.

Evaluation Criteria

Our measure of performance is the number of acquire RPCs sent to your lock server while running yfs_client and test-lab-3-c.

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.

test-lab-3-c creates two subdirectories and creates/deletes 100 files in each directory, using each directory through only one of the two YFS clients. You should count the acquire RPCs for your lab 3 and for your lab 4. If your lab 4 produces a factor of 10 fewer acquire RPCs, then you are doing a good job. This performance goal is vague because the exact numbers depend a bit on how you use locks in yfs_client.

We will check 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 Lab 3 tests (a, b, and c) with RPC_LOSSY=0 and RPC_LOSSY=5.
- Your lab 4 code generates about a tenth as many acquire RPCs as your lab 3 code on test-lab-3-c.

Handin procedure

E-mail your code as a gzipped tar file to TA and the teacher by the deadline mentioned on the course web. To do this, execute these commands

```
% cd ~/lab
% ./stop.sh
% make clean
% rm core*
% rm *log
% cd ..
% tar czvf your-student-d-your-name-lab4.tar.gz lab/
```