Lab-2: RPC and Lock Server

Due: 10-25-2020 23:59 (UTC+8)

General Lab Info can be found in Lab Information.

Introduction

- This lab includes three parts. In **Part1**(60 points) you will use RPC to implement a single client file server. In **Part2**(60 points), you will implememnt a lock server and add locking to ensure that concurrent operations to the same file/directory from different yfs_clients occur one at a time. In **Part3**(80 points), you will use lock cache to improve system performance.
- If you have questions about this lab, either in programming environment or requirement, please ask TA: Yao Zihang (zihang010509@gmail.com).

Getting started

- Please backup your solution to lab1 before starting the steps below
  - At first, please remember to save your lab1 solution:

    ```
    % cd lab-cse/lab1
    % git commit -a -m “solution for lab1”
    ```

  - Then, pull from the repository:

    ```
    % git pull
    remote: Counting objects: 43, done.
    …
    [new branch]      lab2      -> origin/lab2
    Already up-to-date
    ```

  - Then, change to lab2 branch:

    ```
    % git checkout lab2
    ```

  - Merge with lab1, and solve the conflict by yourself (mainly in fuse.cc and yfs_client.cc):

    ```
    % git merge lab1
    Auto-merging fuse.cc
    CONFLICT (content): Merge conflict in yfs_client.cc
    Auto-merging yfs_client.cc
    CONFLICT (content): Merge conflict in ifs_client.cc
    Automatic merge failed; fix conflicts and then commit the result
    ......
    ```

  - After merge all of the conflicts, you should be able to compile successfully:

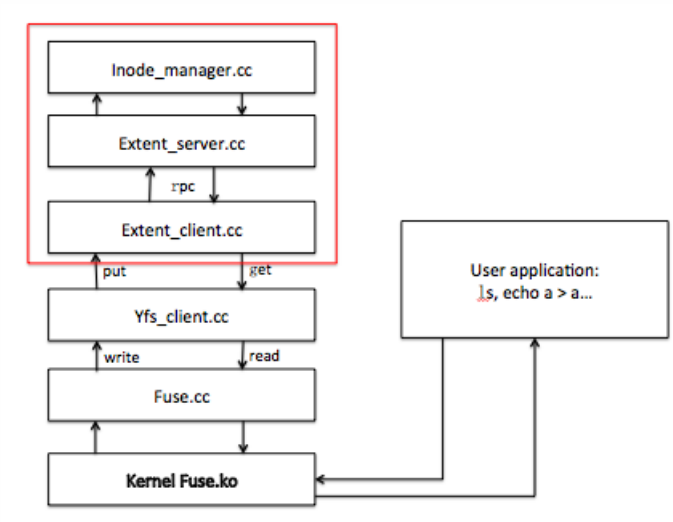
    ```
    % make
    ```

  - Make sure there's no error in make.
- Note: For Lab2 and beyond, you'll need to use a computer that has the FUSE module, library, and headers installed. You should be able to install these on your own machine by following the instructions at FUSE: Filesystem in Userspace (see Lab Information)
- Note: Both 32-bit and 64-bit librpc are provided, so the lab should be architecture independent.
- Note: 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.

Distributed FileSystem (Strawman's Approach)

- In lab1, we have implemented a file system on a single machine. In this lab, we just extend the single machine fils system to a distributed file system.
- Separating extent service from yfs logic brings us a lot of advantages, such as no fate sharing with yfs client, high availability.
- Luckily, most of your job has been done in the previous lab. You now can use extent service provided by extent_server through RPC in extent_client. Then a strawman distributed file system has been finished.
- *You had better test your code with the previous test suit before any progress.*

Part 1

- In lab 2, your aim is to extend it to a distributed file server. And in part 1, it now moves on to the RPC part.
  - 

- 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.
  - Using the RPC system:
    - The RPC library. In this lab, you don't need to care about the implementation of RPC mechanisms, rather you'll use the RPC system to make your local filesystem become a distributed filesystem.
    - A server uses the RPC library by creating an RPC server object (rpcs) listening on a port and registering various RPC handlers (see `main()` function in demo_server.cc).
    - A client creates a RPC client object (rpcc), asks for it to be connected to the demo_server's address and port, and invokes RPC calls (see demo_client.cc).
    - **You can learn how to use the RPC system by studying the stat call implementation.**please note it's for illustration purpose only, you won't need to follow the implementation
      - use `make rpcdemo` to build the RPC demo
    - 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 asstd::string, int, and char. You should be able to complete this lab with existing marshall/unmarshall methods.

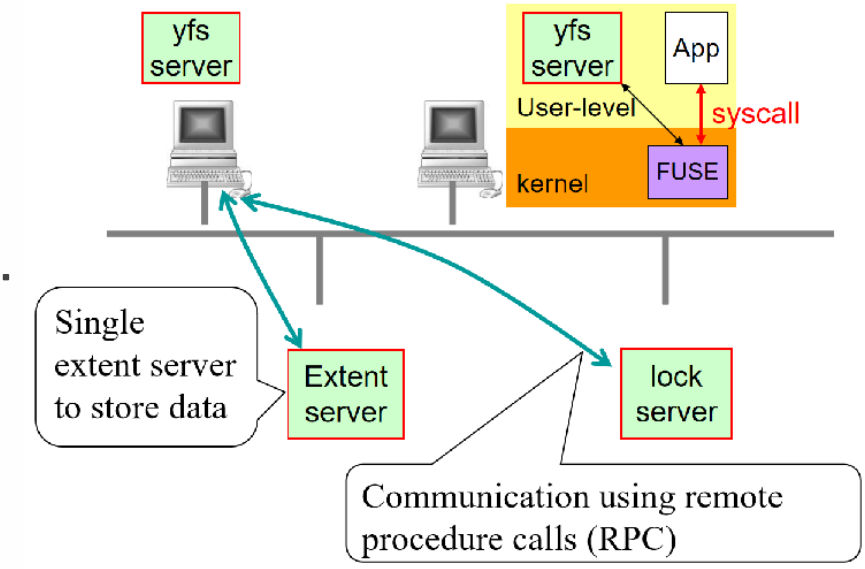
Test

- To grade this part of lab, a test script `grade.sh` is provided. Here's a successful grading.

```
% ./grade.sh
Passed A
Passed B
Passed C
Passed D
Passed E
Passed G (consistency)
Lab2 part 1 passed
......
```

Part 2

- In part 2, you will implement a locking service to coordinate updates to the file system structures. Part 2 is further devided into two parts. In **Part2A** you should implement a simple lock server. Then in**Part2B** you will use the lock service to coordinate yfs clients.



- Reference: Distributed Systems (G22.3033-001, Fall 2009, NYU)

Part 2A: Lock Server

- 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 1386312245
  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 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.

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.

  - Using the RPC system:

- 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 asstd::string, int, and char. 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 tolocked, then returns to the client, which indicates that the client now has the lock. The valuer returned by acquiredoesn'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 Lab Information 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.

Part 2B: Locking

- Next, you are going to ensure the atomicity of file system operations when there are multiple yfs_client processes sharing a file system. Your current implementation does not handle concurrent operations correctly. For example, your yfs_client's create method probably reads the directory's contents from the extent server, makes some changes, and stores the new contents back to the extent server. Suppose two clients issue simultaneous CREATEs for different file names in the same directory via different yfs_client processes. Both yfs_client processes might fetch the old directory contents at the same time and each might insert the newly created file for its client and write back the new directory contents. Only one of the files would be present in the directory in the end. The correct answer, however, is for both files to exist. This is one of many potential races. Others exist: concurrent CREATE and UNLINK, concurrent MKDIR and UNLINK, concurrent WRITEs, etc.
- You should eliminate YFS races by having yfs_client use your lock server's locks. For example, a yfs_client should acquire a lock on the directory before starting a CREATE, and only release the lock after finishing the write of the new information back to the extent server. If there are concurrent operations, the locks force one of the two operations to delay until the other one has completed. All yfs_clients must acquire locks from the same lock server.

Your Job

- Your job is to add locking to yfs_client to ensure the correctness of concurrent operations. The testers for this part of the lab are test-lab2-part2-a and test-lab2-part2-b, source in test-lab2-part2-a.c and test-lab2-part2-b.c. The testers take two directories as arguments, issue concurrent operations in the two directories, and check that the results are consistent with the operations executing in some sequential order. Here's a successful execution of the testers:

```
% ./start.sh
% ./test-lab2-part2-a ./yfs1 ./yfs2
Create then read: OK
Unlink: OK
Append: OK
Readdir: OK
Many sequential creates: OK
Write 20000 bytes: OK
Concurrent creates: OK
Concurrent creates of the same file: OK
Concurrent create/delete: OK
Concurrent creates, same file, same server: OK
test-lab2-part2-b: Passed all tests.
% ./stop.sh
```

```
% ./start.sh
% ./test-lab2-part2-b ./yfs1 ./yfs2
Create/delete in separate directories: tests completed OK
% ./stop.sh
```

- If you try this before you add locking, it should fail at "Concurrent creates" test in test-lab2-part1-a. If it fails before "Concurrent creates", your code may have bugs despite having passed previous testers; you should fix them before adding locks.

Detailed Guidance

- What to lock?
  - At one extreme you could have a single lock for the whole file system, so that operations never proceed in parallel. At the other extreme you could lock each entry in a directory, or each field in the attributes structure. Neither of these is a good idea! A single global lock prevents concurrency that would have been okay, for example CREATEs in different directories. Fine-grained locks have high overhead and make deadlock likely, since you often need to hold more than one fine-grained lock.
  - You should associate a lock with each inumber. Use the file or directory's inum as the name of the lock (i.e. pass the inum to acquire and release). The convention should be that any yfs_client operation should acquire the lock on the file or directory it uses, perform the operation, finish updating the extent server (if the operation has side-effects), and then release the lock on the inum. Be careful to release locks even for error returns from yfs_client operations.
  - You'll use your lock server from part 1. yfs_client should create and use a lock_client in the same way that it creates and uses its extent_client.
  - **(Be warned! Do not use a block/offset based locking protocol! Many adopters of a block-id-as-lock ended up refactoring their code in labs later on.)**
  - **(Notice: If you don't implement a reentrant lock, be careful not to recursively acquire the same lock in a thread.)**
- Things to watch out for:
  - This is the first lab that creates files using two different YFS-mounted directories. If you were not careful in earlier labs, you may find that the components that assign inum for newly-created files and directories choose the same identifiers.
  - If your inode manager relies on pseudo-randomness to generate unique inode number, one possible way to fix this may be to seed the random number generator differently depending on the process's pid. The provided code has already done such seeding for you in the main function of fuse.cc.

Part 3: Lock Cache

- 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 2 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.

In this part, the following files will be used:

- **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 2, 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.

Notice: Before testing, you also need to modify some codes in lock_tester.cc, lock_smain.cc, yfs_client.cc to use lock_client_cache and lock_server_cache.

Your Job

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

1. 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().
2. 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.
3. 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.
4. 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.
5. 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.
6. 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 sent 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.
7. 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.
8. **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?
  How do you handle a revoke 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.

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

- If you finished your design, or decided to refer to the design presented in the comments of the handout code, please move on.

- 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 part 2. 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 tests in part1&2. We will run your code against all tests from part1&2.
- 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 previous 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 part2 and part3 code during the test-lab2-part3-b 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 part 2 tester passes. If it doesn't, then the issues are most likely with YFS under RPC_LOSSY, rather than your caching lock client.

Evaluation Criteria

- Our measure of performance is the number of acquire RPCs sent to your lock server while running yfs_client and test-lab2-part3-b.
- 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-lab2-part3-b 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 part 2 and for your part 3 (which means you should run test-lab2-part3-b in your part 2 code). If your part 3 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 part 3 tests (a, b) with RPC_LOSSY=0 and RPC_LOSSY=5.

- Your part 3 code generates about a tenth as many acquire RPCs as your part 2 code on test-lab2-part3-b.

Grading

After you have implement part1&part2, run the grading script:

```
% ./grade.sh
Passed part1 A
Passed part1 B
Passed part1 C
Passed part1 D
Passed part1 E
Passed part1 G (consistency)
Lab2 part 1 passed
Concurrent creates: OK
Concurrent creates of the same file: OK
Concurrent create/delete: OK
Concurrent creates, same file, same server: OK
Concurrent writes to different parts of same file: OK
Passed part2 A
Create/delete in separate directories: tests completed OK
Passed part2 B

Score: 120/120
```

We will test your lock server with cache following the evaluation criteria above. You part3 score is proportional to the RPC reduction factor(a factor of 10 will be the full score).

Tips

- This is also the first lab that writes null ('\0') characters to files. The std::string(char*)constructor treats '\0' as the end of the string, so if you use that constructor to hold file content or the written data, you will have trouble with this lab. Use the std::string(buf, size) constructor instead. Also, if you use C-style char[] carelessly you may run into trouble!
- Do notice that a non RPC version may pass the tests, but RPC is checked against in actual grading. So please refrain yourself from doing so ;)

Handin procedure

- After all above done:

  ```
  % make handin
  ```

- That should produce a file called lab2.tgz in the directory. Change the file name to your student id:

  ```
  % mv lab2.tgz lab2_[your student id].tgz
  ```

- Then upload lab2_[your student id].tgz file to ftp://esdeath:public@public.sjtu.edu.cn/upload/cse/lab2/ before the deadline. You are only given the permission to list and create new file, but no overwrite and read. So make sure your implementation has passed all the tests before final submit.
- You will receive full credits if your software passes the same tests we gave you when we run your software on our machines.