Lab 1: Lock Server

COSI 147A – Distributed Systems: Spring 2023
Due: January 31, 2023, 11:55pm

1 Introduction

The labs in this course will give you experience designing, building, and debugging distributed systems. The labs will focus on fault tolerance, since that is perhaps the most difficult and important aspect of distributed systems.

In this lab you'll build a lock service that can keep operating despite a single "fail-stop" failure of a server. Clients will talk to your lock service via RPC. Clients can use two RPC requests: `Lock(lockname)` and `Unlock(lockname)`. The lock service maintains state about an open-ended set of named locks. If a `Lock(lockname)` request arrives at the service and the named lock is not held, or has never been used before, the lock service should remember that the lock is held and return a successful reply to the client. If the lock is held, the service should return an unsuccessful reply to the client. If a client sends `Unlock(lockname)`, the service should mark the named lock as not held. If the lock had been held, the service should return a successful reply to the client; otherwise an unsuccessful reply.

Your lock service will be replicated on two servers, a primary and a backup. If there has been no failure, clients will send lock and unlock requests to the primary, and the primary will forward them to the backup. The point is for the backup to maintain state identical to that of the primary (i.e., the primary and backup states should agree about whether each lock is held or free).

The only failure your system is required to tolerate is a single fail-stop server failure. A fail-stop failure means that the server halts. There are no other kinds of failure in this lab (for example, clients don't fail, the network doesn't fail, all network messages to non-crashed servers are delivered, and a server fails only by halting, not by computing incorrectly). Servers are never repaired in this lab – once a primary or backup has failed, it will stay failed.

If a client cannot get a response from the primary, it should contact the backup instead. If the primary cannot get a response from the backup while forwarding client requests, the primary should stop forwarding client requests to the backup.

The failure model in this lab (no network failures, no server repair) is so restricted that your lock service won't be much use in practice; subsequent labs will be able tolerate a much wider range of failures.

2 Collaboration Policy

You must write all the code you hand in, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution. You may discuss the assignments with other students, but you may not look at or copy each others' code. You are not allowed to publish your code (e.g., on Github) or make it available to other students. If evidence of plagarism, cheating, or violation of these rules is found, you will not receive credit for the assignment.

3 Software

You'll implement this lab (and all the labs) in Go. The Go web site contains lots of tutorial information which you may want to look at. We supply you with a partial lock server implementation (just the boring bits) and some tests. You'll find the initial lab software on Latte. You are required to use Go version 1.19.5, which you can download and install here. You also must use the Visual Studio Code editor along with the Go extension.

4 Getting Started

Compile the initial software we provide you and run our tests:

```
$ cd ~/lab1/lockservice
$
$ go test
Test: Basic lock/unlock ...
--- FAIL: TestBasic (0.00 seconds)
test_test.go:14: Lock(a) returned false; expected true
Test: Primary failure ...
--- FAIL: TestPrimaryFail1 (0.00 seconds)
test_test.go:14: Lock(d) returned false; expected true
Test: Primary failure just before reply #1 ...
--- FAIL: TestPrimaryFail2 (2.00 seconds)
...
```

You'll see there are five files in the `lockservice` directory. `common.go` contains RPC request and response definitions for the `Lock` and `Unlock` RPCs. `client.go` contains `Lock` and `Unlock` stubs that are to be linked into client programs; these stubs talk to the lock service. The client code we've supplied you only contacts the primary; you'll have to modify `client.go` to switch to the backup if the primary seems to have failed. `server.go` contains the server code; it has a basic implementation of a Lock handler, but it has neither an Unlock handler nor code to forward operations from the primary to the backup. Finally, `test_test.go` contains our test code; you should not modify it, but you'll need to look at it to see the details of what we're testing.

We've given you code that sends RPCs via "UNIX-domain sockets". This means that RPCs only work between processes on the same machine. It would be easy to convert the code to use TCP/IP-based RPC instead, so that it would communicate between machines; you'd have to change the first argument to calls to `Listen()` and `Dial()` to "tcp" instead of "unix", and the second argument to a port number like ":5100".

5 Your Job

Your job is to modify `client.go`, `server.go`, and `common.go` so that they pass the tests in `test_test.go` (which `go test` runs). When you're done you should see:

```
$ go test
Test: Basic lock/unlock ...
  ... Passed
Test: Primary failure ...
  ... Passed
Test: Primary failure just before reply #1 ...
  ... Passed
Test: Primary failure just before reply #2 ...
  ... Passed
Test: Primary failure just before reply #3 ...
  ... Passed
Test: Primary failure just before reply #4 ...
  ... Passed
Test: Primary failure just before reply #5 ...
  ... Passed
Test: Backup failure ...
  ... Passed
Test: Multiple clients with primary failure ...
  ... Passed
Test: Multiple clients, single lock, primary failure ...
  ... Passed
PASS
```

The above output may omit some benign Go rpc errors.

Only RPC may be used for interaction between clients and servers, between different servers, and between different clients. For example, different instances of your server are not allowed to share Go variables or files.

Your lock code is a library intended to be used in client and server programs. You can see source for simple lock server and client programs along with instructions in the `cmd` directory. Practice starting a primary and a backup server process, locking a few locks, then crashing the primary (with control-C). Observe that the backup server still thinks the locks are held. You must make sure your lock code works by building and running both `lockd` and `lockc`. Provide

`.png` screenshots of successful expected behavior in the `img` directory.

6 Hints

Start by modifying `client.go` so that `Unlock()` is a copy of `Lock()`, but modified to send `Unlock` RPCs. You can test your code by inserting `log.Printf()`s into `server.go`. Next, modify `server.go` so that the `Unlock()` function unlocks the lock. This should require only a few lines of code, and it will let you pass the first test.

Modify `client.go` so that it first sends an RPC to the primary, and if the primary does not respond, it sends an RPC to the backup.

Modify `server.go` so that the primary tells the backup about `Lock` and `Unlock` operations. Use RPC for this communication.

Now do something about the case in which the client sends to the primary, the primary forwards to the backup and then crashes, and the client re-sends the RPC to the backup – which has now already seen the RPC.

It is OK to assume that each client application will only make one call to `Clerk.Lock()` or `Clerk.Unlock()` at a time. Of course there may be more than one client application, each with its own Clerk.

Remember that the Go RPC server framework starts a new thread for each received RPC request. Thus if multiple RPCs arrive at the same time (from multiple clients), there may be multiple threads running concurrently in the server. You will need to use `sync.Mutex` correctly to complete the assignment.

The easiest way to track down bugs is to insert `log.Printf()` statements, collect the output in a file with `go test > out`, and then think about whether the output matches your understanding of how your code should behave. The last step is the most important. If you fail a test, you may have to look at the test code in `test_test.go` to identify the failure scenario.

7 Handin Procedure

Submit your code via LATTE Upload your code as a gzipped tar file by the deadline at the top of the page. To do this, execute these commands:

```
$ cd ~/lab1
$ tar czvf lab1-handin.tar.gz .
```

That should produce a file called `lab1-handin.tar.gz`. You will receive full credit if your software passes the `test_test.go` tests.