Lab #2: Cached Lock Service

Schedule: **Week #3 – Week #5**

Deadline: **Oct 28, 2025**

Points: **10/40**

Prerequisites

- Lab #1

Introduction

You will build a server (the Lock Service) and client (a lock client within the DFS Service) that cache locks at the client, reducing the load on the server and improving client performance. For example, when client C_1 asks for lock A repeatedly and no other client wants the lock, then all acquire, and release calls can be performed on C_1 without having to contact the server.

The challenge is the protocol between the clients and the server. For example, when client C_2 acquires a lock that client C_1 has cached, the server must revoke that lock from client C_1 by sending a revoke remote call to client C_1. The server can give client C_2 the lock only after client C_1 has released the lock, which may be a long time after sending the revoke (e.g., if a thread on client C_1 holds the lock for a long period).

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 are not looking at but maintains correctness when the same files and directories are concurrently read and updated on multiple hosts.

Protocol and implementation hints

Implement a protocol in the style suggested below. You may think there is a simpler protocol, and you are probably right; you will have to trust us that our protocol makes things easier when we replicate the lock server for fault tolerance.

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). This allows the server to be replicated using the replicated state machine approach in Lab #4. If you change the protocol, make sure that handlers on the server run to completion.

On the client a lock can be in several states:

- *None* – The client knows nothing about this lock.
- *Free* – The client owns the lock, and no thread has it.

- *Locked* – The client owns the lock, and a thread has it.
- *Acquiring* – The client is acquiring ownership.
- *Releasing* – The client is releasing ownership.

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).

When a client asks for a lock with an `acquire()` remote call, the server grants the lock and responds with `true` (which means *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 `false` (which means *RETRY*). At some point later (after another client has released the lock using a `release()` remote call), the server sends the client a `retry()` remote call. The `retry()` remote call informs the client that the lock may be free, and therefore it ought to try another `acquire()` remote call.

Note that *RETRY* response and `retry()` remote call are two different things. On the one hand, *RETRY* is the value the server returns for `acquire()` remote call to indicate that the requested lock is not currently available. On the other hand, server sends `retry()` remote call to 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()` remote call 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()` remote call 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 service interface `LockCacheService` in protobuf to use when sending remote calls from the server to the client. This protocol contains definitions for the `retry()` and `revoke()` remote calls.

A good way to implement releasing locks on the client is using a separate concurrent *Releaser* task. When receiving a revoke request, the client adds the revoke request to a list and wakes up the *Releaser* task. The *Releaser* task will release the lock (i.e., send a `release()` remote call to the server) when the lock becomes free on the client. Using a separate task is good because it avoids potential distributed deadlocks and ensures that `revoke()` remote calls from the server to the client run to completion on the client.

On the server, handlers should not block either. A good way to implement this on the server is to have concurrent *Revoker* and *Retrier* tasks that are in charge of sending `retry()` and `revoke()` remote calls, 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 (if there are clients who want the lock) and wakes up the *Retrier* task. This design ensures that the handlers run to completion. Blocking operations are performed by the *Retrier* and *Revoker* tasks, and those blocking operations are just remote calls to the client, whose handlers also should run to completion without blocking.

A challenge in the implementation is that `retry()` and `revoke()` remote calls 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., any unique name as a string – you can use UUID/GUID –, but you need also hostname or IP address, and port of the client) 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()` remote call, along with the client ID. You probably want to send no additional acquires for the same lock to the server until the outstanding 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 does not require sequence numbers. We suggest that you use sequence numbers anyway, for two reasons. One is that sequence numbers are easy to reason about, whereas thinking of all possible ways in which reordering might cause problems is harder. The second reason is that you will need sequence numbers anyway when we replicate the lock service to handle lock server failures.

Task #2.1: 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()` remote call to client *B*. Then, before client *B* can send another `acquire()` to the server, client *C* sends `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 should not block or invoke remote calls.

- If the server receives a `release()` remote call, and there is an outstanding `acquire()` call from another client, what is the handler going to do? Remember again that the handler should not block or invoke remote calls.
- If a thread on the client is holding a lock and a second thread calls `acquire()`, what happens? You should not need to send a remote call to the server.
- How do you handle a `revoke()` call 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()` call for this lock, what happens? Other clients trying to call `acquire()` on 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()` call showing up on the client before the response on the corresponding `acquire()` call?
- How do you handle a `revoke()` call showing up on the client before the response on the corresponding `acquire()` call?
- When do you increase a sequence number on the client? Do you have one sequence number per lock and client 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()` call 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()` call also results in the lock being returned to the server? (You could alternatively send a `revoke()` remote client immediately after granting the lock in this case.)

Task #2.2: Implement the lock cache functionality

- In your DFS Service server application create a Lock Cache module which implements `LockCacheService` gRPC interface procedures.
- Your DFS Service server must host both `DfsService` and `LockCacheService` gRPC interfaces on the same port.
- Implement *Releaser* task which runs together with your Lock Cache module and activates releasing of unused cached locks.
- Client ID (`ownerID`) should contain 3 kinds of information: hostname or IP address of the client, port of the client (same as DFS Service), and a client name (any unique string) separated by colon (`:`) character (the client ID looks like e.g., `110.120.130.140:5001:abcd`). This way the Lock Service server can connect back to the client and call its `retry()` and `revoke()` remote methods. When looking for the IP address of your computer, do not select localhost loopback address (127.0.0.1), but look for real IP address instead. Real address can be either your local IP address or external/public IP address. When you use localhost

loopback address, the client can be connected only from your local computer where the client is running. Real IP addresses are assigned either manually or by the router which your computer is connected to, and the address is only usable within the address space of the local area network defined by the router. When your computer has an external IP, you can use this one to allow your service to be used within larger address space, i.e. whole internet. When you are not able to dynamically find computer's IP address, you can also configure your DFS Service server application to use a specific IP address defined in some application configuration.

- Do not forget update `stop()` method of the DFS Service server to stop concurrent tasks *Releaser*.
- Rewrite your Lock Service server implementation to perform new `acquire()` and `release()` functionality. It differs from Lab #1 basic functionality. Both `ownerID` and `sequence` parameters must be considered during both acquire and release operations.
- Lock Service server remotely calls back DFS Service server (its Lock Cache) by `retry()` (to block acquire method call in Lock Cache) and `revoke()` (to add a release request for a cached lock).
- *Revoker* task and *Retrier* task must be implemented and running in parallel within the Lock Service server. They must be stopped when Lock Service server is stopped by `stop()` method too.

Bonus exercises

Here are a few things you can do if you finish the lab early and feel like improving your code. These are not required, and there are no associated bonus points, but some of you may find them interesting. Note that you should only consider extensions to the lab if your code passes all the tests with no errors. If you decide to add extensions and you are concerned that you might have introduced new bugs, you can submit a basic version for grading purposes and keep the extended version separate.

Implement read/write locks instead of the current exclusive locks and extend lock client to take advantage of them. That is, if you know that client is not going to modify an extent, your code could acquire a read lock, instead of an exclusive lock. Your lock server can allow several clients to have a read lock at the same time. If a client asks for a write lock, you will have to revoke all read locks. This modification should give better performance because read operations can proceed in parallel, instead of being serialized by the lock service.

Notes

- When running services using shell scripts, start each service and client in a separate terminal.
- When running tests, all your services and clients must be stopped.