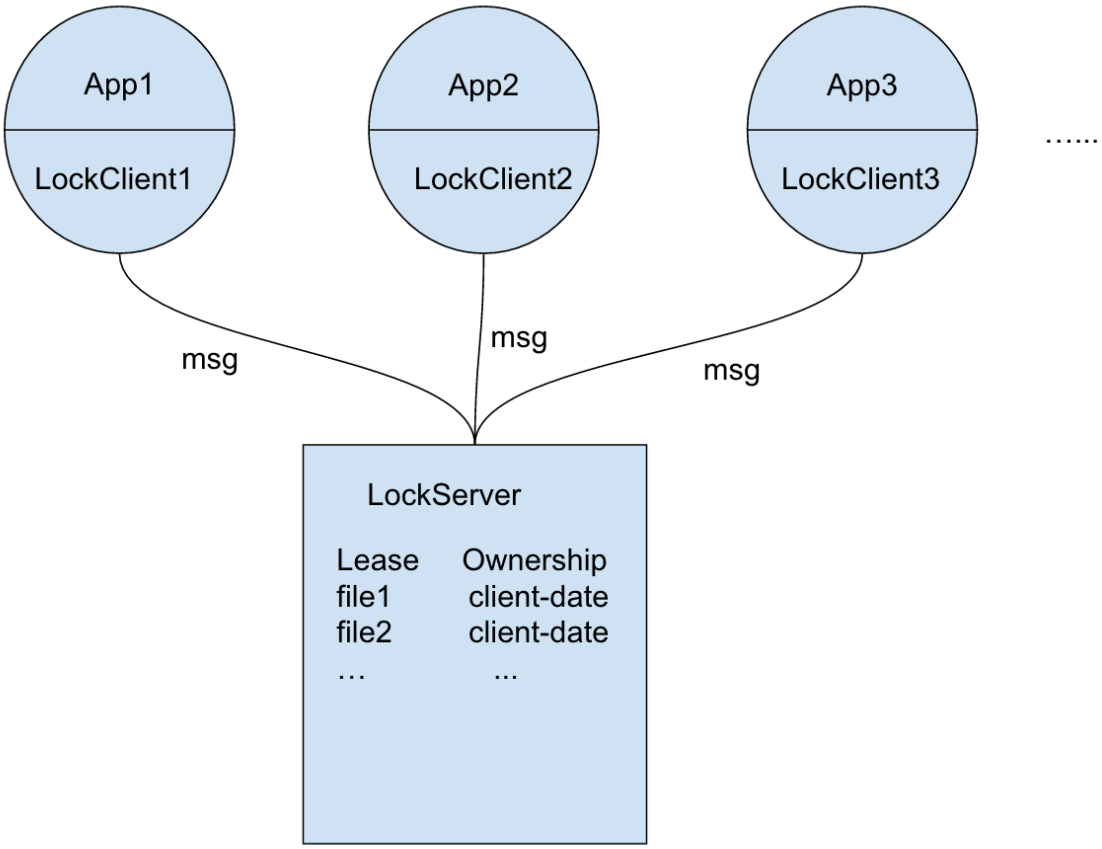
CPS512 HW2 Lock Service

Guotao Dong gd50@duke.edu **Meilong Pan** mp293@duke.edu

Introduction:

In this assignment, we implmented a simple distributed lock service using Scala and Akka. The general idea is that in this service, there are one lock server and multiple lock clients(as many as needed), the lock server is responsible for managing leases and each client is responsible for acquire/release/renew the lease which the application on top of it needs. Our main goal is to guarantee the safety(mutual exclusion) and liveness(eventually released) feature of leases.



The figure above shows the architecture of our lock service, like Chubby, there is one Lock server maintaining all leases and multiple lock clients on applications. Because Scala/Akka is the language and framework used to implement this service, the communication between lock server and lock clients is via messages.

In following paragraphs, we will discuss how we implement lock server, lock client and test results.

Lock Server:

Lock server is the component, which maintains and manages leases centrally. It maintains a hashmap for all lease their ownership. The lock server provides two main functions to the lock client, which are lease acquire and lease renewal. Also, it provides the emulation of disconnect and reconnect condition.

acquire (acqMsg : AcqMsg)
When there is a lease acquire request, the server decide whether it grants the lease to the requester or not. Below is how we handle this request, there are three conditions.

1. If the requested lease is never been used, the server grants the lease directly (by updating hashtable and send back a message to the user.)
2. If the requested lease is granted and not expired, the server has to consult the current client to check if the application is actually using the lease. We check this because it's possible that the application has released the lease but the lease is still held in the clients' cache and not expired yet. If the lease is not being used, the server reclaims and grants the lease to the new client, otherwise, the server denies the lease request.
3. If the requested lease is granted but already expired, the server tells the previous holder to clear the lease and grants the lease to the new client.

One thing worth attention is that when server consults the current holder of lease, we used ask pattern instead of tell pattern, we do this to guarantee the safety of the lease, e.g. avoid the situation where lease is granted to a new holder while old holder still holding it.

renew(renMsg: RenMsg)
When a client come to renew a lease, the server grants this request no matter what, because in our distributed lock service the server honors renew. The server add T time on the original timestamp and send ack message back to requester.

disconnect(clientId: Int, timeLength: Long)
reconnect(clientId: Int)
The method we adapt to handle disconnection is by ignoring all requests from one specified client. We maintain a hashtable for disconnected clients, when one request comes to the server, the server checks that if the request is from a disconnected client. if so, server does nothing with the request and this is the exact situation where real disconnection happens in a distributed system(network is partitioned and no messages flow between two ends). Finally, when the disconnected client comes back, server removse it from the disconnect table.

check()
The method we used to check the correctness for the whole system. When server calls check method, server sends a *reportLease* message to all file lease holders (clients) and waits for replies. A reply message contains *modifiedTimes,* which is used to count the total requested times for a certain file.

Lock Client:

Lock clients associated with applications(like chubby library on user side), it provides API calls for applications so that each app can acquire and release leases they want. Therefore, the main function of lock client is to handle acquire request and release request from applications, another important thing the client has to deal with is the renewal of leases.

In order to reduce the pressure on a lock server, each client has its own lease cache. If the required lease is in this cache and is still valid, the client can complete an acquire request without bothering the lock server.

appAskLease(fileName: String)
When application requires a lease from its lock client, here is what might happen.

1. Lock client checks if the lease is valid in the cache, if it is, lock client returns cached lease directly.
2. If the client the lease is in the cache but it is expired, the client sends a renew request to the server. => **renewLease(fileName)**
3. If the client cannot find the lease in its cache, it has to consult the lock server for this lease. => **acqLease(fileName)**
4. After the client sends associated request and gets the response from server, the client checks whether I get the lease or not. If it gets, use it (make the lease as used), otherwise, schedule a **AskLease** (appAskLease) request again via akka scheduler.

appReleaseLease(fileName: String)
When application releases a lease, the client actually keeps the lease in lease table but mark the lease as unused. The advantage of this design is that the cached lease is convenient for both server's reclaim and current application's acquire. If the server trys to reclaim the lease and find that the lease is cached but not in use, the server can reclaim the lease and grant the lease to the requester. If current application trys to acquire the lease again before the lease is reclaimed by the server, the client will return the cached lease without consulting the server.

renewCheck()
renewLease(fileName: String)
Another significant function of lock client is auto renew. Lock clients could determine which lease requires renew. For those leases which are still in use but nearly expired time, lock clients will renew it automatically via **renewLease** function before its expiration. Applications don't have to care about this because this function is actually transparent to them.

acqLease(fileName: String)
Once a client receives that the application asks for a lease which is not in its cache, the client will execute this function to request lease from server. Because this function uses an ask pattern, it could undergo a timeout exception, which is used to judge whether the client is disconnected or not.

reclaim(recMsg: RecMsg)
Once a client receives a reclaim request from server, it checks the availability of that requested lease. If the lease isn't expired and still, the client deny this reclaim request; Otherwise, the client cleans the requested lease and reports essential info.

Test & Results:

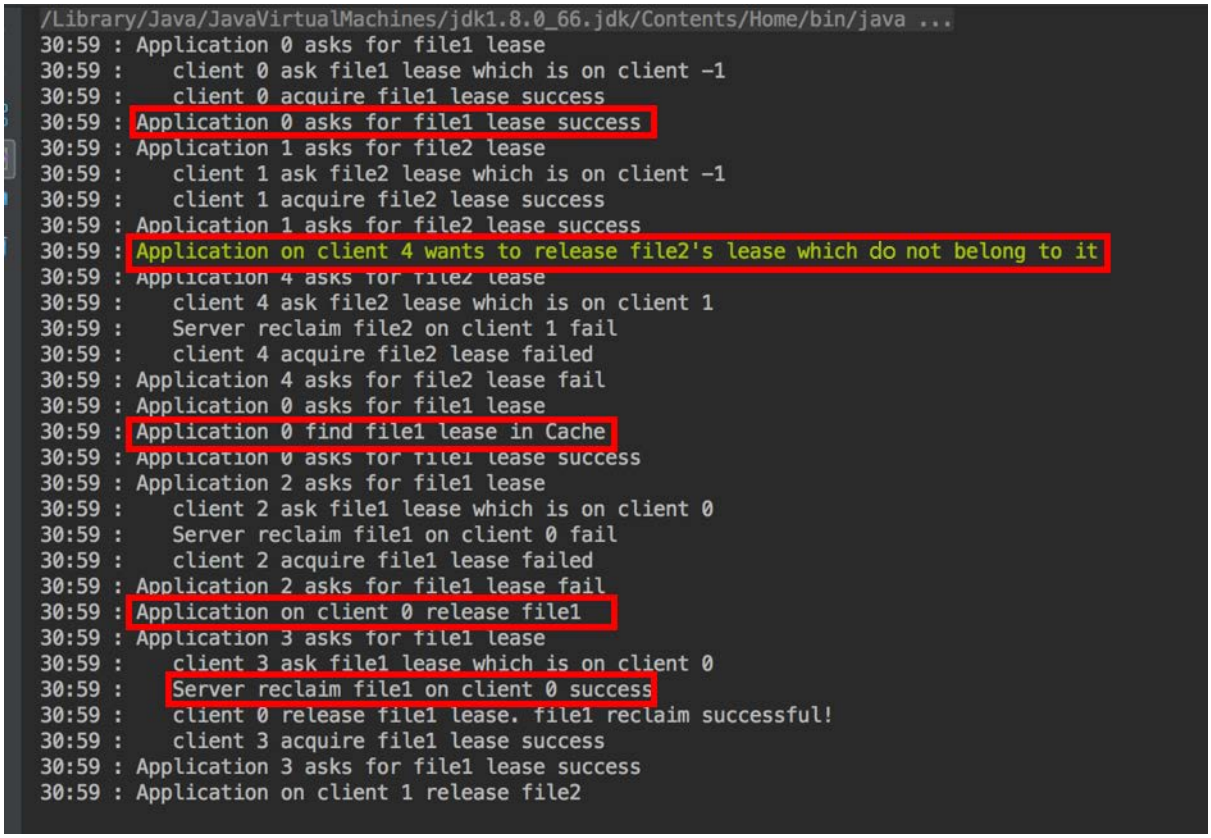
To simulate a real distributed lock service, we only operate on applications. Thus, in test cases, we could only send, **AskLease** and **ReleaseLease,** which represent one application calls for a lease and one application releases the lease respectively.

Basic Function Test:

In first test, We focus on testing normal operations of this lock service. In our test case there are 5 clients for 5 applications and 2 leases for 2 files. This is a logical use case where 5 applications acquire/release these leases in a logical order. The figure below shows the test operations.

```
lockClients(0) ! AskLease("file1") // client0 try to acquire lease for file1
Thread.sleep(20)
lockClients(1) ! AskLease("file2") // client1 try to acquire lease for file2
Thread.sleep(20)
lockClients(4) ! ReleaseLease("file2") // client4 try to release the lease for file2, but it actually dont have the lease
Thread.sleep(20)
lockClients(4) ! AskLease("file2") // client4 try to acquire the lease for file2, but file2 is currently used by client1
Thread.sleep(20)
lockClients(0) ! AskLease("file1") // client0 try to acquire the lease for file1, but actually it already have the lease
Thread.sleep(20)
lockClients(2) ! AskLease("file1") // client2 try to acquire the lease for file1, but file1 is currently used by client 0
Thread.sleep(20)
lockClients(0) ! ReleaseLease("file1") // client0 try to release a lease it owns
Thread.sleep(20)
lockClients(3) ! AskLease("file1") // client3 try to acquire the lease for file1, which is just released by client 0
Thread.sleep(20)
lockClients(1) ! ReleaseLease("file2") // client1 try to release the lease for file2
```

And here is the log after this operation.



In the log, the lines without an indent are reminders for applications and the lines with an indent at the beginning is what happens between the lock client and the lock server. Application X works on client X and server is represented as client -1 for coding convenience.

From the log we can see that all logs are correct for the test cases we applied. The yellow line is an intentional reminder, because we tried to release a lease which does not belong to the requested client. Therefore this proves that the basic operation of our lock service is correct.

Network Partition:

In second test, we focus on testing the behavior of the lock service if there is network partition on one client. We are trying to prove that the server can tolerate a brief transient of T period without being damaged using an example of two clients contending one file.

What we are trying to simulate in this test is that client0 is disconnected with holding lease for file1. During its disconnection, client1 keeps trying to acquire the lease for file1. We try to see whether the server unilaterally seizes the lease and assigns to the other requester after previous holder's expiration. Here is the test case we use,

```
// test for network partition
lockClients(0) ! AskLease("file1") // lock client 0 gets the lease for file1 as usual
Thread.sleep(50)
lockServer ! Disconnect(0, 10000)   // lock client 0 gets disconnected with the lease for file1
Thread.sleep(5000)
lockClients(1) ! AskLease("file1") // lock client 1 try to get the lease for file1
Thread.sleep(5000)
lockClients(1) ! AskLease("file1") // lock client 1 try to get the lease for file1
Thread.sleep(1000) // client0's lease on file1 expired
lockClients(1) ! AskLease("file1") // lock client 1 try to get the lease for file1
system.shutdown()
}
```

in this test case, client0's file expires before it recovers from the disconnection, and the lease is supposed to be occupied so that we could test whether server works well

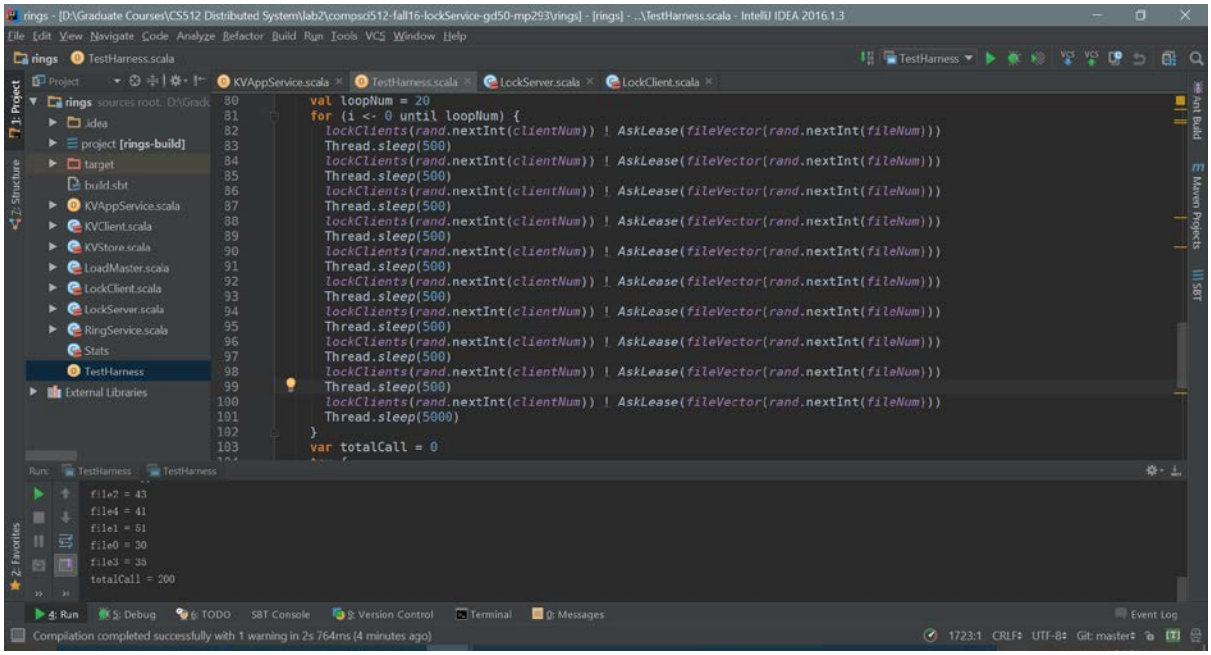
This is the log for the test above,

```
37:56 : Application 0 asks for file1 lease
37:56 :    client 0 ask file1 lease which is on client -1
37:56 :    client 0 acquire file1 lease success
37:56 : Application 0 asks for file1 lease success
37:56 :    client 0 disconnected!
38:01 : Application 1 asks for file1 lease
38:01 :    client 1 ask file1 lease which is on client 0
38:01 :    Server reclaim file1 on client 0 fail
38:01 :    client 1 acquire file1 lease failed
38:01 : Application 1 asks for file1 lease fail
38:04 :    client 0 auto renew file1 lease
38:06 : Application 1 asks for file1 lease
38:06 :    client 1 ask file1 lease which is on client 0
38:06 :    client 1 acquire file1 lease success
38:06 : Application 1 asks for file1 lease success
38:07 : Application 1 asks for file1 lease
38:07 : Application 1 find file1 lease in Cache
38:07 : Application 1 asks for file1 lease success
38:09 :    client 0 timeout: renew file1's lease
```

we can see from the log that after client0's disconnection, the lease is still with it before the expiration. After expiration, the lease is successfully assigned to client1, this proves the liveness of our lock service.

Random Test:

For random test case, we combined **ReleaseLease** into **AskLease**, so that the application could proactively release its lease after a random time. To verify the correctness (leases are mutual exclusive), we remove disconnect commands. Because disconnect could cause data loss, which destroys my test result. The test principle is that, when an application gets (askLease) a lease, we add one to this lease's counter. If the whole system is mutual exclusive, the total lease requested times would be equal to the lease's counter number.



The test case executes 10 random ask requests for 20 times and calculates leases' counters. In the diagram above, we could see these five files are executed 43, 41, 51, 30 and 35 times, respectively. We add these 5 values together and get 200, which is equal to our total request times.