Lab2 Report

yy189 yy191

Data Struture:

We mainly use a queue to record the incoming request for a lock which is held by other clients.

Mechanism:

LoadMaster sends commands to applications and *lockServer*. An application's command includes: "acquire lock" and "release lock". The *lockServer*'s the one and only command is "emulate failure".

1. Application's acquire:

When an application is told to acquire a lock, it will generate a random Int from [0, number of locks]. In this project, the "name" of the named locks is of Int type. For example, number zero *namedLock*'s name is 0, number one *namedLock*'s name is 1, etc. Then the application tells it's *lockClient* to aquire that *namedLock*. The *LockClient* will first check if that *namedLock* is in-cache (whether its *lockSet* contains such *namedLock*). If it's not in-cache, the *LockClient* will tell *LockServer* it wants to aquire that lock.

The *LockServer* has an array of class *LockQueue* and with length *numLocks*. We call the array *lockQueue.* member variables of class *LockQueue*:

var holder: Int // the application that holds the namedLock
var requester: mutable.Queue[Int] // applications acquring the namedLock
var timeLastAcquire // the time when the lock was granted to the holder

"*holder*" is initialized with -1. So, upon receiving an acquire request from some application X, the *LockServer* will first check the array to see whether the acquired lock is held by some other application Y. If (lockQueue(namedLock).holder == -1), it means no application is holding that lock. The lockServer will update holder to *appID*, *timelastAcquire* to system's current time and tell the application *AcquireSucceed* with parameter "*namedLock*" and "*currentTime* + T". "*curentTime* + T" (*validUntil*) tells the application the expiration time of this *namedLock*. So, when an application receives the message *AcquireSucceed*, it will put *namedLock* into its lockSet. And, it will set a scheduler of "validUntil - system's current time" milliseconds to delete the *namedLock*, which means after that period of time, the *namedLock* will expire.

Or, upon receiving an acquire request for a *namedLock* from some application X, we will process the top requester, the head, of that *namedLock*'s queue. If there's no request in the queue, the head will be application X. Otherwise, we enqueue application X.

in this project, we only emulate the failure of the communication path between *app0*， which is the application whose *appID* equals 0, and *lockServer.*

If there is no failure in the system or the holder is not *app0*, we recall the acquired *namedLock* from the holder. When receiving the message *Recall*, the application will delete the lock from its *lockSet*. And such behavior will not have conflict with "deleting the lock when lease expires". Since in the scheduler, we set that the program will check whether the *lockSet* contains the lock to be deleted. If not, the lock must have

been recalled and we do nothing at this time. Other than doing recall, the *lockServer* will also update the *lockQueue* as before and dequeue the head, which will be used next time.

If the failure takes place (*failure == 1 && holder == app0*), we check the *waitUntil* parameter. The definition of *waitUntil* time is: *lockQueue(namedLock).timeLastAcquire + T - currentTime*, which means how much time is left before expiration. If *waitUntil < 0*, it means the *namedLock* already has expired. We send success notification to the head, do updates and dequeue the new head.

Else if *waitUntil >= 0,* we set a scheduler of *waitUntil* milliseconds. In the scheduler, do the same thing above. There is a flag here. If flag ==0, it means we have never been in such case (fail but need to wait), we can set the scheduler ensuring the head (we assume app1) that we will give it the lock after *waitUntil* time. We set the flag to 1 once we entered in the clause and set the flag back to 0 in the scheduler. So if there is only *app1* in the *lockQueue*, and *app2* sends an acquiring request for the lock held by the failed app0, it will only be pushed into the *lockQueue* and will not be processed until app1 is given the lock after *waitUntil* time and another acquiring request from app3 comes.

2. Failure emulation:

To verify whether the failure emulation functions properly, we define a *numMessage* for *app0*. If there is no failure and the *lockServer* receives a message from *app0*, *numMessage++*. If the failure occurs, no message from *app0* will be received. So, we print the *numMessage* when failure begins. In the scheduler, we set the system to print the *numMessage* when failure ends. If the emulation is correct, two numbers should be the same.

3. main flow:

Similar to *rings*, the *LockServer* and every application has specified number of bursts (commands). When an actor receives that number of bursts, that actor will be deactivated. When all are deactivated, the program ends.

4. Experiments:

The following are experiment results:

1) Acquire & Acquire Succeed:

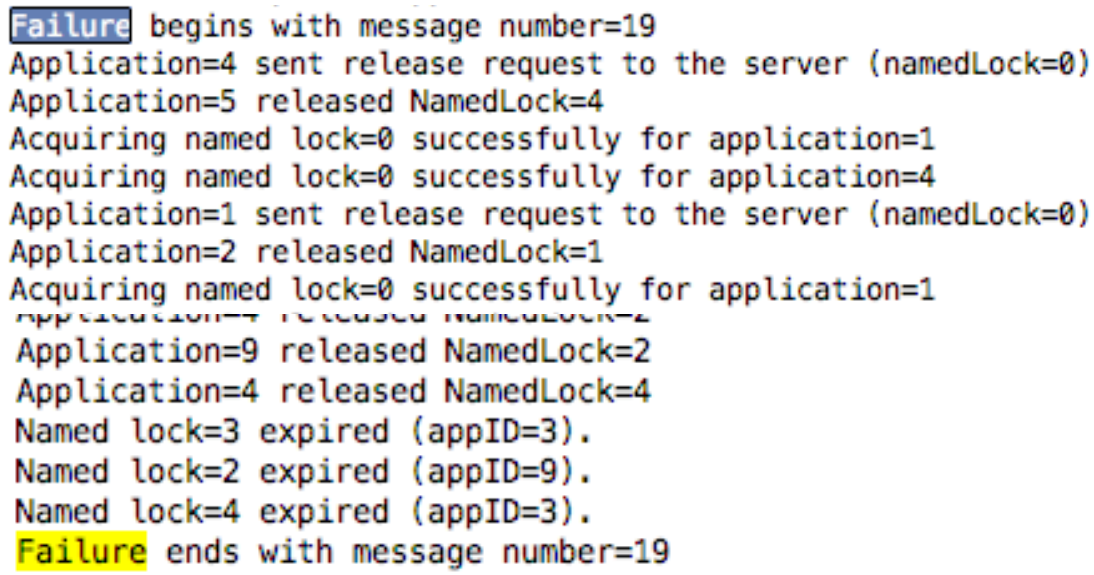
```
Application=3 sent acquire request to the server (namedLock=1)
Application=9 sent acquire request to the server (namedLock=0)
Acquiring named lock=4 successfully for application=0
Acquiring named lock=1 successfully for application=3
```

2) Lease Expire

```
Acquiring named lock=1 successfully for application=0
Acquiring named lock=2 successfully for application=4
Named lock=1 expired (appID=0).
```

3) Failure

```
Failure begins with message number=2
Named lock=3 expired (appID=5).
Named lock=1 expired (appID=8).
Failure ends with message number=2
```



When Failure begins, the number of messages sent by *app0* that have been received by server equals to the number of messages sent by *app0* received by server. Therefore, the server has ignored all messages from *app0* during the failure.