Distributed Systems (CSE431) Monsoon 2016

Assignment-2

Due Date: 17th September, 11:55 pm

Task : Write a program to implement Ricart and Agrawala's mutual exclusion algorithm among a group of processes (RPC servers).

You will have to submit a script (script.sh) and an executable (myProg) that work as follows:

script.sh -n <N> -o <outputFile>

This will start N instances of myProg on the local machine. Each instance of the program is invoked as follows:

myProg -i <myId> -n <N> -o <outputFile>
myId can be an integer from 0 to N-1

Each instance of the program does the following:

1. Register with portmapper or with NamingServer
    a. If you are using Java, use "myProg-<myId>" to register with the NamingServer
    b. If you are using C, use svc_register() to register the server. Use versionNumber=<myId> to register different servers with portmapper
2. Find handles to all servers
    a. In Java, invoke Naming.lookup() or equivalent for each name
    b. In C, invoke clnt_create()
    c. Depending on the order in which the servers are started, the first invocations of these calls might fail. So, you will need to sleep and retry to ensure that you have all handles available
3. Write a loop that acquires a lock; updates counter in output file; releases the lock; sleep for a random amount of time from 1 to 2N seconds; until the counter reaches 100. Do the following after acquiring the lock:

a. Read the last line in the output file and find the counter value
    i. If file is empty or non-existent, the counter is 0
b. Each line in the output file is of the following format:
    i. **<counter>:<myId>:<clock value of lock request>**
    ii. **Use generalised clocks.** Hence, the clock value would like {(0, 4), (1, 3),...} and so on where 4 represents the clock from program instance 0, 3 represents the clock from program instance 1, etc
c. Increment the counter, and write the new line in the above format

4. Once the counter reaches 100, print a message and wait for the user to press Ctrl-C

The loop looks like this :

```
while (i < 100) {
        lock();
        /* (Critical Section )
            -read the output file and read the last record of the above format
            -read the counter
            - appends a new record <counter++>:<MyId>:<clock value of lock request>
         */
        unlock();
        sleep();
        i++;
}
```

All instances of the program should be killed when the script is killed by typing Ctrl-C.

You will end up defining the following RPCs:

1. **LockRequest**(byte [ ]): request sent when a node wants to acquire a lock
2. **UnlockRequest**(byte [ ]): request sent by a node to a peer when it releases a lock
3. **SendAckRequest** (byte [ ]): request sent by a node to acknowledge a peer's lock request

Here byte[] : serialized form of the protobuf message you have defined.

You will implement the lock() and unlock() methods.

lock()

- Construct the LockRequest() message by incrementing the portion of clock belong to <myId>
- Insert lock request in queue in sorted order of the clock
- Send the **LockRequest**() message to all peers including self
    - You can choose to bypass the LockRequest() while sending to self
    - The Java and SUN RPC provide blocking semantics for RPC calls. Hence, the peer needs to respond to the LockRequest() immediately and send the ack later
        - You can piggyback the ack on the response to the lock request if the ack is to be sent immediately
- Wait until request is at the head, and all acks are received
    - This can be N or N-1 acks depending on self responds to a lock request

unlock()

- Remove lock request from head of local queue
    - You can verify that the head is infact a lock request from self to ensure that there are no bugs in your code
- Send **UnlockRequest**() to all peers so that the peers can remove the request at the head
    - If the queue has 2 lock requests L1 and L2 with L1 at the head. It is possible that a node sees the unlock for L2 before the unlock for L1
    - While processing the UnlockRequest() always compare the clock on the lock request and remove the appropriate request from the queue
- Trigger **SendAckRequest**() to peers, if required
    - Again, peer has to acknowledge the SendAckRequest() immediately
    - You can avoid this by piggybacking the ack as part of the UnlockRequest() itself

Add the following printf statements in code to help us evaluate:

- You can redirect these statements to myProg.<myId>.out
- One printf while sending LockRequest() to a peer. This should include the peer's id, sender's id and the clock of the lock request and my current clock
- One printf on seeing a LockRequest() from peer. This should include the peer's id, clock of the lock request, my current lock, and my id
- One printf on sending a SendAckRequest(). This should include my id, peer's id, clock of the corresponding lock request
- One printf on seeing a SendAckRequest(). This should include my id, sender's id, clock of lock request that is being ack'd
- If you are going to piggyback the SendAckRequest() as part of the LockResponse(), add appropriate printf statements similar to those for sending and receiving the SendAckRequest()
- Similar printf statements for sending and receiving the UnlockRequest()
- One printf statement when a lock() request is granted. This should include the clock of the lock request that is granted
- One printf statement when unlock() is invoked. This should invoke the clock of the lock request that's being released

You will have to figure out the following and implement the same:

1. A comparison function on the generalised clock to come up with a total ordering on the messages
2. Ensure that the messages are processed in the order in which they are received. Use TCP to ensure that messages are received in the correct order
    a. One easy of way of doing this is to do the entire message handling in a critical section
    b. See if you can improve on this and improve the performance
3. Define protobuf messages , serialized forms of which will be used while making RPCs

Submission Format:

- Put all the code in a folder named “Assignment2”
- Name the wrapper script “script.sh” which invokes everything based on the flag values.
- Submit as archive “Assignment2.tar.gz”

Note:

- Languages allowed: Java/C++/Python
- Use google protobuf for communication.
- Include the proto file that you have created to define messages in the submission.
- Plagiarism will be penalised.