CS 505 Programming Assignment 2
Consensus

Prof. Ananth Grama

Due: March 31, 2011. 11:59pm

1 Reading (Consensus)

Two good references for the Paxos algorithm are:

1. "Paxos Made Simple" by Leslie Lamport. Can be obtained from
   `http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf`
2. "Paxos for System Builders" by Jonathan Kirsch and Yair Amir. Can be obtained from
   `http://www.cs.jhu.edu/~jak/docs/paxos_for_system_builders.pdf`

Feel free to read *either* of them. "Paxos for System Builders" gives a lot of low level details, like the details of (C) data structures used, timeouts etc.

2 Task A: Paxos Consensus Algorithm

Implement the following class Paxos that uses the paxos algorithm for consensus. As you know, Paxos is a simple consensus algorithm where one server is elected leader and the leader coordinates the algorithm by proposing a client update for execution. This class can be used by the leader to run the Paxos algorithm:

```
public class Paxos {
  public class ConsensusDecision {
    public int instance;
    public String proposedValue;
    public String decidedValue;
  }
  public Consensus(List<String> Members);
  //The constructor of this class takes the list of processes in the group.

  public int getNextInstance();
  //This class can be used by multiple threads.
  //Each thread on the leader that uses the class is therefore given a (locally) unique instance identifier

  public ConsensusDecision propose(String proposal, int instance);
  //propose a value for a given consensus instance
  //the decision is returned via ConsensusDecision
  //this is a blocking call
}
```

In this task, you also have to implement a failure detector based on heartbeat messages. For leader election, use a monarchical leader election scheme, where the process with highest rank becomes leader. Assume that there is only one process per physical node, and use lexicographic ordering on the fully qualified host name to define your "rank". For example, if there are 3 processes xinu11.cs.purdue.edu, xinu12.cs.purdue.edu, xinu13 .cs.purdue.edu, then xinu13.cs.purdue.edu becomes the leader. For simplicity, implement the leader election as part of the failure detector class. Your code in this task must be able to handle stopping failures.

```
public class FailureDetector{
  public FailureDetector(List<String> Members);

  public List<String> getAlive();
  //return the list of processes that are alive

  public List<String> getFailed();
  //return the list of processes that have failed

  public boolean isAlive(String hostname);
  //return true if hostname is alive

  public String getLeader();
  //return current Leader
}
```

3 Reading (RMI)

Check out `http://download.oracle.com/javase/tutorial/rmi/overview.html` for an overview of Java RMI.

4 Distributed Lock Manager (DLM)

(Some examples taken from Wikipedia) A distributed lock manager (DLM) provides distributed software applications with a means to synchronize their accesses to shared resources. The DLM uses a generalized concept of a resource, which is some entity to which shared access must be controlled. This can refer to a replicated file, a record, an area of shared memory, or anything else that the application designer chooses. A hierarchy of resources may be defined, so that a number of levels of locking can be implemented. For instance, a hypothetical distributed database might define a resource hierarchy as follows – Database, Table, Record and Field.

Locks are also used in distributed systems for division of responsibilities – e.g., in a topic-based publish/-subscribe system consisting of 10 servers, topics may be abstracted as "locks", and a server that acquires a lock on a topic is responsible for storing messages that are published on that topic.

In distributed systems, the terms *fine-grained lock* and *coarse-grained lock* are overloaded! For the purpose of this assignment, a coarse grained lock is one that is held for a long time, and a fine grained lock is one that is held for a short time. Traditionally (and not in this assignment), the terms "coarse-grained" and "fine-grained" refer to the size of the resource being locked, e.g., a lock on a database is coarse-grained whereas a lock on a row is fine-grained.

When a client acquires a lock from a distributed lock manager (DLM), the DLM usually decides the time for which a client can hold the lock. This time is also called the *lease time* of the lock – intuitively, the client "leases" the lock from the DLM.

Note that the DLM is separate from the shared resource. Hence, in many cases, DLMs provide *advisory locking*. Clients of the DLM are typically assumed to be a set of cooperating processes, which are non-malicious. It is assumed that clients acquire a lock before accessing a shared resource, and clients do not use the resource beyond the lease time. So these locks are different from the locks in Java's AtomicInteger, for example, where the lock acts as a "gatekeeper".

5 Task B: Replicated Distributed Lock Manager

In this task, you will implement a Replicated DLM for *coarse-grained advisory* locks. A lock has two states – ACQUIRED and RELEASED. The simplest DLM is a single server that stores the state of each lock, and atomically updates their states. But, such a DLM is not fault-tolerant. Hence, in this task, you will implement a replicated DLM consisting of n servers, where the state of each lock is replicated on n servers. To keep the state of each lock consistent on each replica, the Paxos consensus algorithm is used. So, you can assume that the n servers elect a leader L among themselves, and a client acquires a lock and releases it by communicating with L. L uses the Paxos consensus algorithm to propose changes to the state of a lock on the replicas.

Here's the interface for the replicated DLM in this task.

```
public class Lock {
  public String lockName; //name of the lock
  public int leaseTime; //the lease time is decided by the DLM
}
public interface LockServer extends Remote {
  public boolean createLock(String lock) throws RemoteException;
  //This method is used by the "resource" to define locks
  //For example, a database service may use this method to define its locks

  public List<String> getAllLocks() throws RemoteException;
  //Returns a list of currently defined locks

  public Lock acquireLock(String lock, int minTime) throws RemoteException;
  // This is used to request a lock for a minimum lease time
  // The server grants the lock only if it can lease the lock atleast for minTime
  // The actual time for which the server leases the lock is returned in the leaseTime variable of Lock
  // Your code should throw a RemoteException if the call fails for any reason
  // Your code should set the Message field of RemoteException properly
  // The server may choose to queue acquireLock requests from clients
  // If lock l was last held by client c1, and both c1 and c2 request it, the server may choose to grant it to c2.
  // The server has to run an instance of the Paxos protocol before returning an acquired lock.

  public boolean releaseLock(String lock) throws RemoteException;
  // self explanatory
  // The server has to run an instance of the Paxos protocol before returning true.

  public Lock renewLock(String Lock, int renewalTime) throws RemoteException;
  // This method is similar to acquiring the lock again
  // It is upto the server to grant this request
}

class LockClient() {
  public boolean createLock(String lock);
  public List<String> getAllLocks();
  public Lock acquireLock(String lock, int leaseTime);
  public boolean releaseLock(String lock);
  public Lock renewLock(String lock, int renewalTime);
  public boolean isValid(String lock);
}
```

You can assume that the communication delay between the client and the server is bounded. You will also implement a LockClient class which applications can use to acquire and release locks. The methods in the LockClient class except isValid will invoke (RMI calls) the corresponding methods in a remote LockServer object to create, acquire and release locks. The application can renew a lock by using the corresponding method in

LockClient. Assume that a lock can be acquired only in one mode.

Note that this task makes simplifying assumptions. In particular, we assume that the communication delay between the server and client is bounded by d. Hence, when the client receives a lock from a server with a lease time l, it is safe to use the lock for $l - d$. Furthermore, we assume that the locks are coarse grained, i.e. $l >> d$. Remember that the locks are advisory locks, so you may assume that an application using LockClient doesn't use a lock when its lease has expired. The LockClient class provides an isValid() method to check the validity of a lock.

Use the Paxos algorithm implemented in Task A for consensus. You can reuse the failure detection and leader election components from Task A. Your code should handle stopping failures.

6 Reading

How to implement a scalable distributed lock service, with a hierarchy of locks (a lock can have several levels of sub-locks), each of which can be acquired in different modes (read, write, etc.)?

Read "The Chubby Lock Service for Loosely-Coupled Distributed Systems" by Mike Burrows from `http://labs.google.com/papers/chubby.html`.

7 Bonus Credit

If you program Task A and Task B to handle stopping failures with the possibility of recovery (of servers), then you get bonus credit. "Paxos for System Builders" gives you a good idea of how to handle stopping failures with the possibility of recovery. You can use your home directory for storage – you can assume that it provides persistent storage.

8 Submissions

Put all the files in your submission into a directory, and name it after (one of) the team members, i.e., if your name is John Smith, the directory name should be JohnSmith. Create two sub-directories, TaskA and TaskB. Please submit a README containing the names and emails of the both the team members. Please also submit a brief writeup of your design, either as a plaintext or as a pdf file. You have to use turnin to submit this assignment: turnin −c cs505 −p Programming2 <directory containing your files>

9 Resources

You can use any set of CS Linux machines, e.g., {sslab00−sslab24, xinu00−xinu21,mc01−mc16}@cs.purdue.edu.