Machine Problem 2: Simple Object Migration

Due date: To Be Announced

1 Overview

The objective of this machine problem is to implement a simple **object replication infrastructure** that allows objects to be replicated for fault-tolerance purposes. New replica of replicable objects can be created during the lifetime of the object, and existing replica can be terminated. Clients then can invoke methods on replicable objects in a **multi-RPC** fashion. That is, the remote method gets invoked on all replica of the object. The infrastructure should support creation of new replica and termination of existing replica during the lifetime of the object (more precisely, during the lifetime of any replica of the object).

In order to realize the multi-RPC, a **reliable multicast protocol** must be implemented that makes sure that the remote invocation is issued in a totally ordered atomic fashion on all the replica of the object. In addition, a **group management protocol** must handle creation of new replica and termination of existing replica.

1.1 Replicable Objects

We realize the object replication infrastructure by providing **Replicable Objects**, which are remote objects that can create clones of themselves on other hosts. In Java notation, the interface `Replicable` provides the replication functionality, and the class `ReplicableRemote` implements this interface. The critical methods provided by `Replicable` are `replicate()`, which creates a new replica of the object on another host, and `terminate()`, which renders the replica of the object inactive, so that it can be deleted.

```
public interface Replicable extends Remote {

    /* used internally to create replica */
    public boolean accept(String className, String objName)
        throws RemoteException

    /* manage replication */
    public boolean replicate(String newServer, String objName)
        throws RemoteException
    public boolean terminate(String objName) throws RemoteException
}

public class ReplicableRemote extends UnicastRemoteObject
  implements Replicable {
    /** CHANGED: State needs to be passed. **/
    public boolean accept(String className,String objName,Object[] state)
        throws RemoteException {
            /* implement method here */
    }
```

```
    /** DO NOT CHANGE THIS **/
    public boolean replicate(String newServer, String objName)
        throws RemoteException {
            /* implement method here */
    }

    public boolean terminate(String objName) throws RemoteException {
            /* implement method here */
    }

    public void set (Object data){
            /* implement method here */
    }

    public Object get () {
            /* implement method here */
    }
}
```

A note regarding the public method `set()`: Ideally, we would like to be able to derive arbitrary remote classes from `ReplicableRemote`. This would require to support for multi-RPC-style invocation for arbitrary methods, which in turn would require modification of the stub generation or at least of the support libraries used by stubs. We will simplify matters by assuming that our implementation of replicable objects simply manages the state of a single variable in the object, in our case through a simple `get()/set()` interface. Derived application classes then can build on top of this basic capability, for example the following class `BankAccount`:

```
public class BankAccount extends ReplicableRemote implements Account{
    public Integer balance; // This variable can be of any type

    public void setBalance(int value){
        set(value);
    }

    public Integer getBalance(){
        return get();
    }
}
```

A minor problem in the `set()/get()` approach as used above is that the type of the state object must be known by `ReplicableRemote`, which is limiting. A solution to this may be to use a variable of the derived class as state variable. To tell the parent class `ReplicableRemote` which variable carries the state, one can use reflection, maybe in form of a method `setStateVariable(String var_name)`. This makes the implementation more generic.

2 A Software Bus

The underlying protocol to support messaging is a software bus, which implements atomic, totally ordered, group communication. The `ReplicableRemote` class instantiates a SoftwareBus for each replicable object that is created. The idea is that whenever an object is replicated across machines, the SoftwareBus for that object will expand accordingly. More specifically:

- Nodes are assumed to never fail.
- Nodes join and leave the bus during its lifetime.
- Message delivery is atomic, and is totally-ordered.

The software bus is accessible to the replicable objects through the following interface:

```
public interface SoftwareBus {

    public SoftwareBus(String name);
    /* Creates a local access point to the bus specified by the given
       name. If the bus does not yet exist, it is created.
       If the bus exists already, this access point is inserted
       the bus. */

    public Object read();
    /* Blocking read of data from the bus. This method will block
       until is some data available in the SoftwareBus */

    public void write(Object data);
    /* Non blocking write to the bus. The exact definition of nonblocking
       needs to be specified.*/
}
```

3 Realization

We implement the software bus with underlying support for multicast. For the more object-oriented folks among you, the best way is probably to define a class `SoftwareBus`.

3.1 Server Program

A server will start the `rmiregistry()` at a particular port (Have unique port numbers, viz, last 4 digits of your Student ID). It will bind the remote object of `Accept` with the rmiregistry. This object will be responsible for handling the details of Object Replication. If this is first server in the network then this server (i.e. the origin server) will create and bind the application remote object (which extends `ReplicableRemote`) with the rmiregistry. Once the Controller decides to replicate the object, the server will migrate its application object to the new server.

Hence the server program will have two interfaces (in RMI jargon):

1. The Replicable Interface which will have the `accept()`, `replicate()` and `terminate()` remote methods.
2. The `Account` interface which will have `getBalance()` and `setBalance()` remote methods.

3.2 The Client Program

The client will somehow find out which server has a replica of the Application object available. After that it will perform an RMI Lookup to get access to the remote Information object from the server. The client can perform the `get()/set()` methods on the remote object accordingly. The client will also need to know somehow how to get access to the new references of the object in case the current server has crashed or decided to terminate the application. (Note: The client process does not perform an rmiregistry.)

3.3 Integration with ReplicableRemote class

Inside the `ReplicableRemote` class, an instance of the SoftwareBus needs to be created, when the application is initialized. This basically means that at first we will have a SoftwareBus with only one member, the creator of the Bus at Server 1. A Thread responsible for reading the data from the software bus should also be initialized at the same time in the `ReplicableRemote` class.

As the object gets replicated on Server 2 using the `replicate()` method, the software bus needs to expand on its own. So this would mean calling a method (using java reflection) that is responsible for creating the SoftwareBus instance for the replicable object on Server 2, in the `accept` method of `ReplicableRemote`. The initialization of the SoftwareBus should take care of group management properties, viz, the pointers to the next and previous servers as explained below.

In order to write data into the SoftwareBus, inside the `set(Object data)` method of the ReplicableRemote class, we call `bus.write(data)` and return. The `write()` method of the Bus should take care of how the data moves around the software bus in an atomic and totally ordered way.

The read thread started when an object is created (or replicated for that matter) is responsible for reading the data from the software bus and storing it locally in the state-carrying variable.

Remember: The `ReplicableRemote` class is completely independent of the functionalities of the SoftwareBus, and viceversa. All the `ReplicableRemote` class has is a reference to the SoftwareBus, and access to its public methods. While the replica of the replicable object are copies of the same remote object, the local instances of the SoftwareBus are local objects, and not replicated!

3.4 Multicast-based Software Bus

The implementation of the multicast-based software bus uses IP multicast (remember, IP multicast is unreliable, just as UDP). The algorithm used for this implementation is a very much simplified version of Brian Whetten's algorithm discussed in class.

The members of the group are, again, arranged in a logical ring. One member carries a token. Whenever the sender has a message to send, it multicasts it to the group using IP multicast. Each message carries an identifier of the sender (e.g. IP address and process id) and a sender-specific sequence number. (The token holder will include this information into the ACK in order for the sender to know which message the ACK is for.) When the token holder receives a message, it multicasts an ACK, which in turn contains *(i)* the current timestamp of the token holder and *(ii)* the identifier information from the sender (sender identification and sequence number). The timestamp in the ACK defines the ordering of how messages are delivered at the receivers: The members deliver the messages in increasing order of timestamps. After sending out the ACK, the token holder increases the local timestamp value, and passes the token to the successor in the logical

ring. The new owner of the token will then take care of acknowledging the next message sent to the group.

The token is transferred reliably from member to member after every message acknowledgment, e.g. by using TCP. This means that, assuming that nodes do not fail, we don't have to worry about lost or duplicated tokens.

A basic invariant of the algorithm is that a process can only accept a token with timestamp t if it has received all messages with timestamps less than t. The new token holder can accept the token only after having recovered all the missing messages. So, if the last message received by the new token holder had timestamp $t - k$, the new token holder has to ask for the last $k - 1$ messages to be forwarded to him. The best member to ask for any missing messages is the previous token holder, who is guaranteed to have received all $t - 1$ messages. Let's look at what can go wrong, and how this algorithm takes care of it:

1. The sender has missed a message before sending the current message: It will get an ACK with a timestamp that is higher than what it expected. (Every member of the group keeps track of what the current timestamp is, since every member keeps receiving ACKs). The sender can then start recovering by asking for a copy of the missing message(s) from its predecessor in the ring.

2. Some other member misses a message: This case is similar to the sender missing a message. The missing message is detected by comparing timestamps in the ACKs.

3. The current token holder does not receive the message, or the ACK gets lost on the way to the sender: The sender times out and retransmits. The token holder acks the message. If it had acked it before, and has forwarded the token in the meantime, it simply keeps acknowledging.

4 A Few Implementation Issues

4.1 Circulating the Token in the Multicast-Based Version

The token has to be handed down to the successor in the ring in a reliable way. The best way to do this is to use either TCP directly. It is probably beneficial to include the request and forwarding of missing messages into this token-forwarding protocol.

4.2 Managing the Group

The group membership can change as replica of objects get created or deleted, and **local copies** of the software bus are created or destroyed accordingly. This needs to be implemented. The groups should be managed in a fully distributed way. A solution is to keep the information about the first member of the group in every member, and have each member have a **pointer** (ip address and the port number for token management, for example) to the predecessor member and the successor member in the ring. Particular attention must be made that group operations are properly ordered. For example, new members only get messages that are sent "after" the member joins. You may want to make membership operations synch-ordered with respect to message delivery.

4.3 Multicast

As underlying communication substrates we use TCP and Multicast IP. The latter allows us to group processes into a IP multicast host group. Multicast host groups are identified by their IP

class-D address. Whenever a process wants to join a particular host group, it binds a socket to the address of that particular group. Also, feel free to follow the Resources link on the course homepage to the MBONE (Multicast IP Backbone) homepage, which has lots of interesting stuff.

Note: I am not sure at this point if multiple processes on the same machine can bind to the same port for IP multicast. Probably the best way is to be conservative and limit yourself to at most one "local copy" of the same software bus per machine.

4.4 Test Application

We will provide you with a test application that relies on the interface defined above. Multiple clients will then access objects that are derived from `ReplicableRemote`, and a control application will then control the replication of these objects by using the interface provided by `Replicable`.

5 What to Hand In

The running system will consist of the SoftwareBus library, and ReplicableRemote components.

Again, you will hear more regarding the details on this.

In order to compare the relative performance of the two implementations, you will perform several suites of measurements.

First, we want to know how long it takes to replicate an object, or how long it takes to terminate a replica. In order to determine this, you will measure the latency incurred by a call to `replicate()` and to `terminate()`.

In order to determine **jitter** introduced by the SoftwareBus protocol, you will instrument client and/or server code appropriately to log the inter-arrival times of messages. Maybe a client would periodically send a pair of `set()`/`get()` commands, with the `get()` command being delayed. At that point you can measure the latency distribution of the `get()` command to assess how much disturbance is added by the protocol implementation. The result will be a histogram for the multicast-based application.

You will gather statistics about he round-trip time of the message with increasing numbers of receivers in the two buses.