Machine Problem 1: Simple Object Migration

Due date: To Be Announced

1 Overview

Remote invocation systems like Java/RMI allow for distributed computation by providing clients access to objects across machine boundaries. In its basic form, Java/RMI has the limitation that remote objects are bound to their well-known location, and cannot move to another host. This is a problem, for example, when a server with remote objects needs to be taken off the system because of maintenance or as response to security threads: What to do with remote objects on this server?

The objective of this machine problem is to build an object migration system, which allows remote objects in Java/RMI to transparently migrate from one server to the other during their lifetime. We call remote objects that can be migrated Migratable Objects.

Whenever a server (bank server, for example) needs to be taken off the system, any migratable objects (account objects, for example) on the server can be forced-off the server beforehand, without affecting clients (ATM machines, for example) that are accessing those objects. These clients continue transparently accessing the remote objects, but now on the new server.

2 Issues in Object Migration

In order to realize migratable objects we must know (1) how to effectively move the object from the origin server to the new server, and (2) how to transparently update the references from the client to the migrating object. A number of solutions can be found for each point, each providing a different level of transparency.

2.1 Moving the Object

When an object is migrated, its state must be moved from the origin server to the new server. A very simple (and crude!) way to move an object is to create a new object on the new server, and copy the data from the old object to the new object. (Note: This is acceptable for this project.) A number of issues need to be addressed.

- **References to non-movable objects.** Not all the data of a migratable object can be freely moved. If a migratable objects refers to an open file, for example, the file cannot be simply moved to the new server. Instead, the file has to stay at the origin server, and a proxy file handle is moved to the new server. The implementation of the object moving should be able to detect references to non-movable objects and appropriately (and transparently!) handle them. (Note: Bonus points will be given to teams who can handle migratable objects with references to non-movable objects.)
- **Moving objects by passing references.** The basic implementation approach of moving objects by explicitly copying their data is rather crude. It would be of course much nicer to actually move the object, instead of its state only. This would mean that we would need to modify the stubs generated by rmic (the Java/RMI stub generator) or to modify rmic itself. (Note: Bonus points will be given to teams who provide solutions that solve this.)

- **Threaded migratable objects.** When an object must be moved that has threads, the threads must be stopped, the stacks of the threads must be moved, and the threads must be re-started at the new server. This most likely requires modifications to the Java Virtual Machine. (Note: For teams who consider themselves severely underchallenged, this would be something worthy considering in return for bonus points.)
- **Monitoring of object migration.** Also it will be a better demonstration if some kind of GUI is available. (Note: No bonus points here.)

2.2 Updating the References to the Object

After the remote object has moved to a new server, its reference needs to be changed at the clients. A very crude way of doing this would be for the origin server to perform some sort of callback to the all the clients that are accessing the object. However such a scheme would not make the servers stateless. A number of other mechanisms are possible:

- **Change the Object reference in the Registry.** This would mean that the origin server somehow manages to change the entry of the remote object in the registry to point to the new reference
- **Perform Lookup on Exceptions.** The client will catch the exception when the remote object no more exists on the origin server, and resolve the new reference to the migrating remote object.

3 Realization of Migratable Objects

The objective of this machine problem is to build a **Migratable Object Library**, which involves creating the implementation of the following.

```
public interface Migratable extends Remote {
  public Object accept(String className, String objName, Object[] state}
    throws RemoteException;

  public Object migrate(String newServer, String objName)
     throws RemoteException;
}
```

where

- `className` is the name of the actual class whose object will be migratable
- `objName` is the name of the object registered in the rmi registry
- `state` is an array of variables that the migrating object contains.
- `newServer` is the name and port no of the new server where the object should to migrate

```
public class MigratableRemote extends UnicastRemoteObject implements
Migratable {
    public Object accept(String className, String objName, Object[] state)
       throws RemoteException{} /* Implement the method here */

    public void migrate (String newServer, String objName)
       throws RemoteException{} /* Implement the method here */

    /* This remote method will be called whenever the object
       is supposed to be migrated. It is the responsibility
       of this method to get state of the current object to
       be migrated to the newServer. After the object is
       migrated, it needs to unbind the objName from
       the rmi registry.
     */
}
```

The mechanism about how a new client will get details of the server holding the object at present, or, how the origin server gets details about a new server, is left open. Also the implementation of nested references (i.e. when the object has moved across many servers) is left open.

4 Detail Design

4.1 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 Migration. If this is first server in the network then this server (i.e. the origin server) will create and bind the `BankAccount` remote object (which extends `MigratableRemote`) with the rmiregistry.

After some time (for example, after the server has received a number requests from clients, the server will migrate its Information object to the new server and obtain the new Information Object pointer.

After successful migration, the current server will unbind the Information Object from the rmiregistry and destroy the object. Hence the server program will have two interfaces (in RMI jargon):

1. The Migratable Interface which will have the accept() and migrate() remote method
2. The Information Interface which will have getData() and setData() remote methods.

4.2 Client Program

The client will somehow find out which server has the Information 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 `getData()` and `setData()` methods on the remote object accordingly. The client will also need to know somehow how to get access to the new references of the migrating object. (Note: The client process does not perform an rmiregistry.)

5 Test Application

The test application is provided on the website. In order to migrate the objects, you should have a controller that decides which object to move and when. A sample code of the controller is as follows

```
class Controller{
  public static void main(String[] args) {
    Migratable server1 = (Migratable)Naming.lookup(url+"Account");
    /* Move the object here*/
    server1 = server1.migrate("cs-sun02:2000", "Account");
    Thread.sleep(5000); // i.e. 5 sec
    server1 = server1.migrate("cs-sun03:2000", "Account");
  }
}
```

Note: The Library should be generic enough for any application to be run. It should NOT be application specific. Have a look at `java.lang.reflect` package for details about this.

Also, the controller is an independent entity. Neither the client nor the server have any idea about its existence. As a result, the controller cannot be a remote object for the client or the server. Its job is just to migrate an object.

Also this controller code is just an example. You are free to write your own controller that may take commands from the shell, if necessary.

6 What to Hand In

The running system will consist of the Object Migration library server programs and the test client program. After the test application is provided, a demonstration would be expected of the system. As platform, you should be using the SUN workstations, develop the program in Java.

6.1 Design

Before you start hacking away, plot down a design document. The result should be a system level design document, which you hand in along with the source code. Do not get carried away with it, but make sure it convinces the reader that you know how to attack the problem. List and describe the components of the system: Server Program, Client Program etc.

In case you have implemented something special (like the bonus points implementation), kindly highlight that part in the document. Also if you feel you have done something differently, please mention it in a eye-catching way.

6.2 Hard Copy of Portions of Source code

Hand in a hard copy of all the code you created. Do not excessively waste trees! Try to condense the printouts, e.g. with `enscript -2r <file>`. The code should be easy to read (read: well-commented!). The grader reserves the right to deduct points for code that he considers undecipherable.