COP5615 – Distributed Operating Systems Principles – Spring 2012
Project 3

Assigned: March 12, 2012 (Monday)
Due Dates: March 26, 2012 (Local), March 28 (EDGE), 11:55 PM EDT

Programming Language Allowed: Java (only)
Platforms Allowed: thunder/storm/linux-name.cise.ufl.edu (Linux Machines)

Java versions available: Java(TM) SE Runtime Environment (build 1.7.0_04-ea) on thunder/storm/other linux mach.

1. Project Overview:

In our class, we described the Client/Server model and the Remote Procedure Call (RPC) as two very simple concepts and yet the most important notions that characterize a distributed system since they facilitate many aspects of transparency. The Java Remote Method Invocation (RMI) extends RPC to include object model and monitor synchronization (or conversely, we are integrating RPC into the Java object-based system with support of synchronization). This is a very powerful integration, in particular, since system-wide object-based references can be passed in the remote method invocation.

In this project, we will gain some experience of using the Java RMI. The application of it will be the Writer Preference CREW problem with prevention of reader starvation. Recall that we defined writer-preference as: the arriving readers wait until there is no Active Writer (AW) or Waiting Writers (WW). In this version of the CREW problem, readers might starve if writers keep coming in continuously. So in this project, we will relax the writer preference slightly by letting some of the waiting readers to go before the waiting writers if the readers have been waiting for too long. This modification will prevent or alleviate the reader starvation problem while still give writers preference most of the time. We will describe a priority scheme to achieve this in the following section.
RMI is a useful tool for implementing the reader-writer problems in a client/server paradigm. Read and write operations are handled by the server with synchronization supports, and the clients simply call the read and write methods for the shared data. It is much more transparent than using socket communication.

2. A Priority Scheme for the CREW Problem:

We assume that the waiting readers and writers are maintained in two queues: reader queue and writer queue. Each waiting thread has an arrival time and a priority (1 to 4, with 4 as the highest priority to access the shared database). For the writers, they have fixed priority 3, but for the readers, their priorities change dynamically. The initial priority for all the readers is 1. The priority for a reader is increased by 1 each time when the reader yields to a waiting writer which arrived after the reader. If a reader's priority becomes higher than the writers (the reader has priority 4 and writers have priority 3), then we allow the reader to bypass the writers and enters the critical section to begin concurrent read. We will also restrict the concurrent read a little bit. If one reader gets the chance to enter the critical section, only 2 readers behind in the reader queue can inherit the priority of the first reader, that is, only the top 3 readers can enter into the critical section simultaneously. One thing important to mention is that the inherited priority cannot be further inherited. For example, there are 4 readers, R1, R2, R3, and R4 in the reader queue. R1 is at the head of the queue and has priority 4. R4 is at the end of the waiting queue and has priority 1. There are two writers, W2, W3 in the writer-waiting queue. Assuming W1 releases the critical section, then R1 whose priority is 4 is higher than writer priority 3, so R1 should go to critical section first. At the same time R2 and R3 will inherit the priority 4, so R2 and R3 will also enter into the critical section. However, R4 only has priority 1 and cannot inherit the priority of R1, R2, R3, so R4 has to wait until it has priority 4 or there is no writer in the writer queue. What's more, if there is no writer in the writer queue, then all readers in the reader queue can enter the shared read at the same time.

3. RMI Implementation and Helpful Resources:

Java RMI: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html

RMI Registry

Like procedures in RPC that need to be registered and bound, methods in RMI also need to be registered before use. RMI registry is a Naming Service that acts like a broker. RMI servers register their objects with the RMI registry to make them publicly available. RMI clients look up the registry to locate an object they are interested in and then obtain a reference to that object in order to use its remote methods. Of course, servers register only their remote objects, which have remote methods. Details using the registry are given in the next section.

The Remote Interface

Remote methods that will be available to clients over the network should be declared. This is accomplished by defining a remote interface. This interface must extend java.rmi.Remote class and must be defined as public to make it accessible to clients. The body of this interface definition consists of only the declaration of remote methods. The declaration is nothing more than the signatures of methods namely method name, parameters and return type. Each method declaration must also declare java.rmi.RemoteException as the throws part.

Necessary Java packages: *java.rmi.Remote*

Implementation of Remote Interface

After defining the remote interface, you need to implement this interface. Implementing the interface means writing the actual code that will form the bodies of your remote methods. For this purpose, you define a class from which you create your remote objects that will serve the clients. This class must extend *RemoteObject* provided by the *java.rmi.server* package. There is also a subclass *UnicastRemoteObject* which is also provided by the same package that provide sufficient basic functionality for our purpose. When you call its constructor, necessary steps are taken for you so that you will not need to deal with them. An RMI server registers its remote objects by using *bind()* or *rebind()* method of *Naming* class. The latter is preferable even for the first time, since you can run your system as many times as you want without touching the registry.

Necessary Java packages: *java.rmi.**, *java.rmi.server.**

RMI Clients

RMI clients are mostly ordinary Java programs. The only difference is that they use the remote interface. As you now know remote interface declares the remote methods to be used. In addition, the clients need to obtain a reference to the remote object, which includes the methods declared in remote interface. The clients obtain this reference by using *lookup()* method of *Naming* class.

Necessary Java packages: *java.rmi.**

RMI registry

RMI registry can be started on a Unix machine by typing:

```
rmiregistry<port>&
```

<port> is the port number we want the registry to listen for connections from RMI servers and clients. This port number is optional and if not specified, registry is started on default port 1099. A host that provides continuous RMI support should use this port number for convenience and standardization. But **we will specify our own port number** by starting our own RMI registry. This port number should match the one in the **system.properties** file. Note that "&" is to run the registry as a background process. You need to start the rmi service from your java program instead of typing the command manually.

Compiling programs

Compile your start, server and client code as usual.
Note: If the server needs to support clients running on pre-5.0 VMs, then a stub class for the remote object implementation class needs to be regenerated using the **rmic** compiler, and that stub class needs to be made available for clients to download.

Running the System

You need to follow the steps below to run the system:

1 Specify an arbitrary port number that is large enough to avoid collisions with the well-known port numbers. This port number should match the one in the system.properties file. Use *ps* to make sure the rmi registry is running.

2 Start your system with java start

Check the outputs and processes running and repeat this step as many times as you need to test your system. Even after your server thread is completed, the server process, which makes the remote object available, will keep running. To terminate the process, you should use System.exit() method (in the server threads) immediately after all server threads finishes serving the specified number of accesses.

3 Terminate the rmi registry using *kill*.

Note on Registering and Looking up Remote Objects

As we now know, the server needs to register its remote object to the RMI registry to make its own remote methods available to the clients. In turn, clients need to look up the RMI registry to obtain a reference to that object and then to call its remote methods. You must use your CISE login name as the name of the service for registering the remote object to the registry. Similarly the clients must look up the object by giving your CISE login name.

Project Specification

In this project, we will develop a distributed client/server implementation of the Concurrent Readers and Exclusive Writer (CREW) problem.

The implementation will consist of a server that maintains a shared object (an integer variable) for concurrent read and exclusive write accesses and several reader/writer clients on remote machines communicating with the server using rmi. The readers and writers are distributed processes running independently on various CISE machines.

Synchronization in Java is achieved through synchronized keyword. This keyword can be used for any Java object as well as for any method. The mechanism is very similar to the monitor abstraction. If an object is defined as synchronized, that object can be accessed by only one thread at a time. Similarly a Java object can have synchronized methods and hence becomes a monitor. Only one thread can execute an instance of the synchronized method at a time. Entry to the method is protected by a monitor lock around it. If there is any other thread that calls one of the synchronized methods on the same object simultaneously, it cannot continue and therefore is suspended.

In addition we need a mechanism to synchronize access to shared variables. Java provides three methods for this purpose: **wait()**, **notify()**, **notifyAll().** These are similar to the Wait and Signal operations of the monitor approach for process synchronization. Please notice that notify() wakes up only one arbitrary thread and notifyAll() does not guarantee any ordering of the thread requests.

This project requires three programs that run on different machines. These are the server and the client (reader and writer) programs. In addition to these programs we need a configuration file that provides information about the overall system.

Helpful Resources:

Java RMI: http://java.sun.com/javase/6/docs/technotes/guides/rmi/index.html
Multi-threading Tutorial: http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html
Java Synchronization Tutorial: http://www.java-samples.com/showtutorial.php?tutorialid=306

System Configuration:

The system configuration shown below is in the file system.properties

```
Rmiregistry.port=1099
RW.server=thunder.cise.ufl.edu
RW.numberOfAccesses=3
RW.numberOfReaders=4
RW.reader1=lin114-01.cise.ufl.edu
RW.reader2=lin114-02.cise.ufl.edu
RW.reader3=lin114-03.cise.ufl.edu
RW.reader4=lin114-04.cise.ufl.edu
RW.numberOfWriters=4
RW.writer5=lin114-05.cise.ufl.edu
RW.writer6=lin114-06.cise.ufl.edu
RW.writer7=lin114-07.cise.ufl.edu
RW.writer8=lin114-08.cise.ufl.edu
RW.reader1.opTime=200
RW.reader1.sleepTime=300
RW.reader2.opTime=200
RW.reader2.sleepTime=400
RW.reader3.opTime=200
RW.reader3.sleepTime=500
RW.reader4.opTime=200
RW.reader4.sleepTime=600
RW.writer5.opTime=300
RW.writer5.sleepTime=250
RW.writer6.opTime=300
RW.writer6.sleepTime=350
RW.writer7.opTime=300
RW.writer7.sleepTime=450
RW.writer8.opTime=300
RW.writer8.sleepTime=550
```

RW.server is the address of the host on which the server runs. RW.reader1/RW.writer1 is the address of the host on which the reader/writer with ID 1 runs, and so on. RW.numberOfReaders is the number of readers, RW.numberOfWriters is the number of writers, RW.numberOfAccesses indicates how many times a reader/writer should read/write the values in shared object. RW.opTime is an operation time that each reading and writing take. RW.sleepTime is a sleep time for readers/writers between two requests. Each reader and writer should sleep for RW.sleepTime before making a next request. Time unit for RW.opTime and RW.sleepTime is ms.

The pseudo code to describe this process is:

```
for every access {sleep(sleepTime);  waiting to enter into critical section; sleep (opTime); leaving the critical section}.
```

DO NOT USE PREASSIGNED HARDCODED PORT NUMBER inside the server or client code. It may lead to grade penalty.

Note that the above is only a sample configuration file. We will use different configurations (different values, but the configuration format is exactly the same) to grade your programs, so you need to do the same for testing.

Server Description:

The name of the program we will run is "**Start.java**". **Please note that the file name begins with capital case 'S'**. This program is responsible for starting the server on RW.server specified in system.properties and the clients on remote machines. First it should create a thread, which will run the server code. Then, it should start clients on remote machines. Therefore, the server will be running in the background as a thread. The server thread accepts remote requests from clients (readers and writers) and spawns a new thread for each client. This is done implicitly because we are using RMI in this project. If multiple clients invoke the same method simultaneously, then multiple instances of the method start executing in different threads. These threads represent the remote clients and act on their behalves. This means our server is non-blocking and is able to service concurrent requests without waiting for any reading or writing process to be completed.

This program will read the configuration file mentioned above and start up the system accordingly.
The server thread needs to maintain the number of readers and writers served and when all the clients are done, should terminate gracefully.

Reader and Writer Client Description:
Clients are started on remote machines as processes. Therefore they will run in the background. Readers send their request type to the server as read and receive a packet that contains the value of the shared object. Writers send their request type as write and also the value (its ID) to be written to the shared object. The initial value of the shared object is assumed to be

-1.

Output Format:
The server should print out its actions in the following format:

```
 ServiceSequence    ObjectValue      AccessedBy     OptTime//this line is included in the output file
       1                -1              R3            300-500
       2                -1              R1            400-800
       3                -1              R2               …
       4                -1              R1               …
     …
       7                -1              R3               …
       8                -1              R2               …
       9                4               W4           1000-1500
      10                5               W5               …
     …
      16                4               R3               …
      17                4               R2               …
      18                6               W6               …
     …
      24                8               W8               …
```

Note: the all the values are separated by a tabl('\t').
Service Sequence is the sequence number that reader/writer accesses the shared object, *Object Value* is the value now saved in the object, *AccessedBy* is the ID of the readers or the ID of the writer updating the shared object.
The operation time (Opt Time)'s format is begin time-end time. The begin time is the Unix time at the beginning of the current access minus the Unix time when the program starts. The end time is the Unix time at the end of the current access minus the Unix time when the program starts. So the endtime-begintime should equal OpTime. The service sequence is ordered in ascending order of beginning Unix time.
For example, your program is launched at Unix time 1327260100599(2:22 PM, 2012-1-22). Now we have the reader R4, it begins its first read at Unix time 1327260100899, and it finished its first read at Unix time 1327260101099. So the Opt time for the first access of the R4 is 300-500. (300=1327260100899-1327260100599, 500=1327260101099-1327260100599).
The output file(log file) should be named 'crew.log'.

How to start processes on remote machines

To start a process on a remote machine we use the remote login facility ssh in Unix systems. It accepts the host (computer) name as a parameter and commands to execute separated by semicolons.
To execute the ssh command from a Java program, you should use the exec() method of the Runtime class.
First, we need to get the current directory of the user (all the files necessary for this project should be in the same directory, and start.java will be run in this directory), as follows:

```
String path = System.getProperty("user.dir");    // get current directory of the user
```

Second, we invoke exec() method as in the following:

```
Runtime.getRuntime().exec("ssh " + host + " cd " + path + " ; java Site " + ... arguments ... );
```

Here ssh is the command to start a process that will run on the host giving by the program variable host. Immediately after the host name you need to specify the cd command to make the current working directory of the new remote process be the same as the starting program. Hence the output of the remote processes will all be directed to the same directory where we originally started the system. The path is another program variable, which specifies the full path name of the directory where we invoked the start.java. Note that you should use this path as exactly returned by the Java method. You need not append anything to this string.

After the semicolon you should specify the name of the program to run on the remote machine, which is specified as Site.java in the above example. Following the program name, of course, you need to specify the necessary arguments to the program.

Additional instructions:

You are required to submit a 1 page TXT report. Name it **report.txt**. Note the small case 'r' and **NOT** capital 'R'. Your report must contain a brief description telling us how you developed your code, what difficulties you faced and what you learned. Remember this has to be a simple text file and not a MSWORD or PDF file. It is recommended that you use wordpad.exe in windows or vi or gedit or pico/nano applications in UNIX/Linux to develop your report. Avoid using notepad.exe.

You should tar all you java code files including the txt report in a single tar file using this command on Linux/SunOS:

```
tar cvf project3.tar <file list to compress and add to archive>
```

We will use the following scripts to test and execute your project:

```
#!/bin/sh
mv ?*.tar proj3.tar
tar xvf proj3.tar; rm *.class
rm *.log
javac *.java; java start
----------------------------------------
#!/bin/sh
cat *.log
cat report.txt
```

Please test your tar file using these scripts before you submit on line. **If your code fails to execute with these scripts, we will initially assign a score of 0 pending resolution.**

Note on Runaway Processes:

Your threads should terminate gracefully. While testing your programs run-away processes might exist. However these should be killed frequently. Since the department has a policy on this matter, your access to the department machines might be restricted if you do not clean these processes.

To check your processes running on a Unix:	`ps -u <your-username>`
To kill all Java processes easily in Unix, type:	`skill java`
To check runaway processes on remote hosts:	`ssh <host-name> ps -u <your-username>`
And to clean:	`ssh <host-name> skill java`

Grading Criteria:

Correct Implementation/Graceful Termination:	85%
Elegant Report:	5%
Nonexistence of runaway processes upon termination:	5%
Obeying the described output format exactly	*5%*
Total	100%

Late Submission Policy:

Late submissions are not encouraged but we rather have you submit a working version than submit a version that is broken or worse not submit at all.

Every late submission will be penalized 15% for each day late for up to a maximum of 5 days from the due date.