COP5615 – Distributed Operating Systems Principles – Spring 2011
Project 1

Assigned: Jan 20, 2011 (Thursday)
Due Dates: February 03, 2011 (Local), February 06 (EDGE), 11:55 PM EDT

Programming Language Allowed: Java (only)
Platforms Allowed: sand, rain (SunOS Machines) and/or lin113-01 (Linux Machines)

Java versions available: OpenJDK Runtime Environment (build 1.6.0_0-b11) on lin113-01
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_10-b03) on sand/rain

Project Overview:
In the first project, we will develop a very simple distributed application. The underlying motivation will be very similar to SETI@HOME distributed computational model where normal users contribute idle CPU cycles that are utilized to perform a very complex mathematical computation that has been broken down into several simpler computations. In this project you will be asked to develop a distributed client/server model that distributes the task of sorting a large set of numbers to several available idle clients. Task distribution will be done at the server depending on the remote idle clients' CPU power. The aim of this project is to help you learn and understand client/server socket communication as well as multi-threading in Java.

Helpful Resources:
Socket Programming Tutorial: http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
Multi-threading Tutorial: http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

Introduction:
In this project, you are required to develop a server-client program in Java, **including one server and five clients**. The client program will be single-threaded; however, the server program will be multi-threaded so that it can handle multiple clients at the same time. The client program, when started, generates a random number between 1 and 10 that will be used to indicate the hypothetical remote client's relative CPU speed. When the server starts, it will generate a list of 100 random numbers between 1 and 500. It then will distribute this sequence to remote clients in proportion to the CPU speeds reported by the clients. Each client then does the insertion sort (or other sorting algorithm if you like) and reports the results to the server when it finishes. After receiving the results from all clients, the server uses the aggregated results to merge the sorted subset of numbers to get the final sorted sequence. You are required to provide a "**start.java**" that will start the server program as well as all the remote clients. ***Please note that the file name begins with small case 's' and NOT a capital 'S'***. All the communication between server and clients will be done using Java sockets (TCP).

Server Description:
The server program is a multi-threaded program. The server, when started, will create 5 server threads, each communicates with a corresponding client. Each server thread selects a system-assigned free ports via Serversocket for the purpose of establishing a connection to the server from the corresponding client later on. The assigned port number needs to be passed along with the host name to the remote clients from the server. **DO NOT USE PREASSIGNED HARDCODED PORT NUMBER** inside the server or client code. It may lead to grade penalty.

Before the server starts to listen to remote clients, it generates a list of 100 random numbers between 1 and 500 and stores them in an array or another suitable data structure of your choice. You may need some other information in this data structure as well, e.g., state of a number (true/false – true meaning the number has already been assigned to a remote client for processing), client ID (to what client was this number assigned), completion (true/false – whether the processing of the number has been completed). Server also maintains a list of remote client objects containing the remote client's CPU speed rating (lies between 1 and 10).

This data structure, along with the random number list, has to be shared by all child threads in the server side. The server needs to distribute all these numbers to the clients according to their CPU speeds and collect them after being processed at the client sites. Therefore, you are also required to enforce proper synchronization between threads on these shared data-structures. (Please refer to thread concurrency and object synchronization).

To make sure that the remote client communications can be connected to the server successfully at the beginning, you can set 1500 + random(500) in each server thread such that the server can accept multiple remote clients during this period (You can select the clients from the list of machines in our department, either SunOS or Linux). Each thread, corresponding to each client, decides how many numbers to assign to such client, based on the ratio of this remote client's CPU speed over all active remote clients' cumulative CPU speeds. Then the server will assign the numbers to each client via each server thread such that all clients can do the insertion sort in parallel. Each server thread will wait until it receives the sorted list of number from its corresponding client. When all server threads receive the sorted subsequence, the server then does the merge sort to get the final sorted list of numbers.

Client Description:

You must write the client code which takes the remote host and port and client's ID as command line parameters. These parameters should be provided from within the start.java code (in the server side) using Java's Runtime.exec() feature.

When the client first starts, it must pick a number between 1 and 10 that denotes its free CPU speed and it must report this number to the server so that the server can update the clients list appropriately and that in turn will enable it to assign workload to this client proportionally.

The client's functionality is very simple. It gets assigned numbers from the server and sorts the numbers using insertion sort (or other sorting algorithm) and returns the sorted subsequence back to the server. Then it will send a BYE message indicating completion and then terminates *cleanly*.

Clients must also output everything on the console and to a log file with same name as the client ID (we use ID1, ID2, ID3 etc … for client IDs). This is to verify the client's behavior by TAs later on when they grade your project. For example, if the client's ID is **ID1**, then everything that this client outputs on the console must also be written into the file with filename ***ID1.log***.

What does start.java do?

Here is a list of actions that must be performed inside start.java.

1. Create a Serversocket without specifying port number
2. Query Serversocket to find out what port number it was started on?
3. Query Serversocket to find out the server's host name.
   - Make sure it is not localhost or 127.0.0.1 or 0.0.0.0 as in that case remote client's connection will fail
4. Start the Serversocket thread so that it can start accepting remote client connections.
5. Use Runtime.exec() to start 5 remote clients on 5 different CISE systems for clients, you may select from this list below
   - `lin114-01.cise.ufl.edu`
   - `lin114-02.cise.ufl.edu`
   - `lin114-03.cise.ufl.edu`
   - `lin114-04.cise.ufl.edu`
   - `lin114-05.cise.ufl.edu`
   - `lin114-06.cise.ufl.edu`

   Remember these are Linux machines, if you need to select 5 SunOS machines, then select appropriate Linux machines. Check CISE help pages on-line to find out more.
6. Make sure that you provide correct runtime arguments as they will act as command line parameters for the remote clients in server code.
7. You must query the runtime object to get input stream object so that you can display what remote processes output in the console, otherwise you cannot see anything that these remote processes output only using System.out.print() on the console.

Additional instructions:

```
String path = System.getProperty("user.dir"); // get current directory of the user
Runtime.getRuntime().exec("ssh " + host + " ; cd " + path + " ; java Site " + ... arguments ... );
```

This is one way to start remote processes in Java. It provides very limited functionality. Please refer to the Java tutorial mentioned above under helpful resources to find more elegant way of doing remote execution.

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 project1.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 proj1.tar
tar xvf proj1.tar; rm *.class
rm *.log
javac *.java; java start
```

```
#!/bin/sh
cat ID1.log
cat ID2.log
cat ID3.log
cat ID4.log
cat ID5.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 up 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:	80%
Elegant Report:	10%
Nonexistence of runaway processes upon termination:	10%
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.

Sample Outputs:

Given below are a few lines from sample output by server and remote clients log files. Your code implementation should try to follow the output structure as closely as possible to the sample provided here. Minor reasonable deviations are of course allowed.

Start.java Sample Output: [non continuous and partial output]

```
Starting Server thread … [OK]
Server has been started on port 18263!
Host name: 128.227.248.169

Starting Remote Process: Client ID1 on sun114-01
Status: SUCCESS! Parameters Passed: 128.227.248.169 18263 ID1
.
.
.

Server: Accepting a new connection. Added to socket list!
Server: Accepting a new connection. Added to socket list!
ClientHandlerThread (ID1): Received ID1 CPU Speed: 8 [Updated
Records]
ClientHandlerThread (ID3): Received ID3 CPU Speed: 3 [Updated
Records]
ClientHandlerThread (ID1): Assigning 23, 43, 12 to Client ID1
ClientHandlerThread (ID1): Received sorted subseq from Client
ID1 (12,23,43)
.
.
.
ClientHandlerThread (ID1): Client ID1 terminates successfully!
OK!
.
.
.
Server: Received all sorted subseq
Server: Doing merge sort ...
.
.
.
---------------
The final sorted sequence:
...,12,...,23,...,43...
```

Client Side Sample Output: [ID1.log]

```
Processing Command Line Arguments:
Host: 128.227.248.169
Port: 18263
ClientID: ID1
---------------
Starting remote connection: SUCCESS!
Sent: CPU Speed 8
RECV: 23, 43, 12
Doing insertion sort ...
Sent: 12, 23, 43
BYE
Terminating Connection! OK!
```