COP5615
OPERATING SYSTEM PRINCIPLES
Fall 2007

Project 3

Date Assigned: October 04, 2007
Due Dates: *October 23, 2007* (11:59 pm) for LOCAL students (Sections: 1093, 9333)
       October 27, 2007 (11:59 pm) for EDGE students (all other sections)

Introduction:

For many years the Internet primary communication protocol has been TCP/IP which has a very restricted consistency model which may not be suitable for many real time applications such as live streaming applications. UDP/IP was proposed as a much simpler alternative to TCP/IP with much relaxed consistency model where reliability was left to higher layers. UDP proved very useful to numerous applications including VoIP, streaming video applications to name a few. With more and more household getting connected to the Internet everyday and with people using content rich web applications like YouTube and Flickr, the underlying Internet infrastructure is going to face tremendous resource crunch in near future. IP multicast was proposed in late 1980s as an optimal routing protocol for efficient bandwidth usage for group content distribution managed by the network layer. Additional material on IP multicast can be found on line at this link:

http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/ipmulti.htm

IP multicast is especially well suited for on line group management. In this project you will be asked to implement a distributed group management protocol using IP Multicast / UDP. You are expected to use Java for this project. Below are the couple of on line resources to get you started:

http://www.poplarware.com/udpjava.html
http://www.itee.uq.edu.au/~csse2002/Java%20Documentation/api/java/net/MulticastSocket.html
http://www.itee.uq.edu.au/~csse2002/Java%20Documentation/api/java/net/DatagramSocket.html
http://java.sun.com/j2se/1.4.2/docs/api/java/net/MulticastSocket.html
http://java.sun.com/j2se/1.4.2/docs/api/java/net/DatagramSocket.html#receive

From the earlier projects you already know how to use multi-threading in Java. That concept will be heavily used in this project as well.

Problem Description:

Secure group management is a very prominent research field in the network security community with lots of issues still unresolved. For the purpose of this project we would implement a very naive group management protocol with very little emphasis on security. Although in a real implementation most of the communication between group members would be encrypted, but in this project we would ignore channel security and concentrate on Multicast / UDP aspect.

Manifest constants relevant to this project:

```
Group Member Heart-beat Channel: 239.0.0.1 : 5555 [Group IP : Port]
Group Communication Channel: 239.0.0.2 : 6666
New Member Join Request Channel: 239.0.0.3 : 7777
```

The group management protocol will deal with group leader selection, allowing individuals to join a particular group based on simple majority voting and will deal with a member leaving the group. The leader election algorithm will be simply "*bully algorithm*".

Detailed Description of each group member behavior:

Whenever a process is executed for the first time, it must pick a random port number for its own UDP unicast server process. Continue the random port number selection until your program finds a free port to listen to. On this port number your UDP server would listen to any incoming unicast UDP messages. After successful selection of random port number, this is the flow of behavior for each group member:

1. Send GJOIN group join message to the new member join request channel, this message must also contain the sending process's self unicast IP address and UDP server listening port number as part of the GJOIN message.

2. After step 1, the process must wait for a max of 5 seconds for any reply to its GJOIN message. The reply message could be one of these 3 possibilities: WAIT, DENIED, ACCEPT.
   - If no message is received withing 5 seconds, the process must assume that the group does not exist and assume "group coordinator responsibilities". It must then also subscribe to all the 3 multicast channels and starts to listen on these channels. And it must choose a 16 hex character long random string at the group key.
   - If the reply is WAIT, then the group members are deliberating on its membership join request. The process must wait for the decision. It must wait for a max of 10 seconds. It no response comes, it must restart the process immediately.
   - If the reply is DENIED, the process must sleep for 2000 milliseconds and then retry again.
   - If the reply is ACCEPT then look for the common "group key" that must follow the ACCEPT keyword in the received message.

3. Each group member periodically sends a liveness heartbeat message on the group member heart beat channel with a periodicity of 2 seconds + *alpha*, where *alpha* is a random value between 250 – 750 milliseconds. Each member must listen to these incoming heart beat messages. The heart beat messages also contains the sender's local host time stamp, unicast IP address and the port number. Additionally the group coordinator's heart beat message also contains the current "*group key*". The following sequence of actions are done at each member after receipt of each liveness message:
   - It computes the MD5 hash of (IP address, port) and compares it to the hash value of current group coordinator's (IP address, port) or hash of its own (IP address, port). Which ever value is lower, the corresponding IP address, port number pair value is stored as the group coordinator IP address and port number. In this fashion each group member at any point of time knows the current group's coordinator details. It also

updates the current "group key" value to the value contained in the group coordinator's heartbeat message. If no "group key" is found in the current coordinator's heart beat message then it simply invalidates the current key value from its local entry and waits for the key value in the next heart beat message from the group's coordinator.

- It also maintains the last refresh received from the current group coordinator. If (current cycle time – last received time stamp of group coordinator) > 5 seconds, then it sets itself as the current group coordinator. This is done only at moments when this member is sending its own liveness message on the channel. It then follows all the steps that a new group coordinator process follows as described earlier.

4. The group coordinator job is to *additionally* listen on the new member join request channel. Once a new GJOIN request comes, it must immediately send a WAIT reply to the requesting process's unicast address. These are the steps it does after sending out the WAIT reply,

- Send out an ELECT message on the group communication channel. It must wait for 5 seconds for any or all replies to come in from the existing group members. The ELECT message also contains the MD5 hash of the requesting client's IP address and the port number so that the group members can distinguish between different processes membership request.
- It must listen to all group member's vote that comes on its UDP unicast channel. The reply message has the format ELECT YES or ELECT NO from the group members. The reply also contains the same MD5 hash that was sent out in the ELECT message by the coordinator. After the 5 seconds vote timer expires, it must tally the vote and based on simple majority principle reply back either ACCEPT or DENIED unicast message to the process pending membership.
- If the unicast message reply is a ACCEPT message then it also contains the group 16 character hex key.

5. Group coordinator also manages any members' leave group requests. This message is received at the group communication channel and the format of the message is LGROUP. It also contains the sending process's IP address ad the unicast port number. At this moment the group coordinator invalidates the "group key" value and computes a new value which it distributes to all current group members as part of its periodic heart beat messages on the group channel.

ADDITIONAL NOTES

Each message sent on the *group communication channel* by any group member must begin with the "group key" as part of the message. If any message is received on the group communication channel that does not contain "group key" as the first word of the message, it must be ignored by all the members. All message keywords like LGROUP are in all upper case. Additional fields in the message ex. IP addresses, ports etc must be separated by just a single blank space character. The only exception to the above rule is for the process termination condition which is described later.

Each group member must maintain how long it has been up and running. Once a process joins the group successfully, it starts a timer of 10 seconds periodicity. After each timer expiration it randomly decides to continue group participation or leave the group. If the process leaves the group it must send out a LGROUP message on the group communication channel, stop listening on all the subscribed multicast channels, wait randomly [500 – 750 milliseconds] and then restart the whole cycle again.

Project Structure

You project must contain a java file named "start.java". Inside "start.java" you must start 6 additional processes on remote machines using the following java command:

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

You must select 6 different hosts on CISE network. You may use (but are not restricted to):

```
sun114-11.cise.ufl.edu
sun114-12.cise.ufl.edu
sun114-13.cise.ufl.edu
sun114-14.cise.ufl.edu
sun114-15.cise.ufl.edu
sun114-16.cise.ufl.edu
```

These 6 remote processes will act as the participating group members that will take part in group management using the structure described above. You must specify 6 different log file names for each of these remote processes to write their output in addition to their console outputs. The individual log files must be named as process1.log, process2.log and so on.

NOTE: *Make sure there is 5 seconds delay between execution of successive remote processes for the first time from within the "start.java" program.*

Each of your log file entries must strictly adhere to the log format specified below. Also your project must run using the script provided later.

Project Termination Trigger

This you can hard code in your project implementation. The trigger will be pressing of key 'q'. Send a multicast exit message on the group communication channel that looks like -

"EXIT *your_random_hardcoded_verification_sequence*".

Each of your own processes should verify **your_random_hardcoded_verification_sequence** in the received EXIT messages on the group communication channel. If it matches then gracefully terminate your processes and the exit the whole program. If **your_random_hardcoded_verification_sequence** does not match then simply ignore the exit message.

The reason for doing this to make your code immune to interference from your fellow students testing their code simultaneously when you are testing your own code.

Also you must justify to yourself that even if several students might be testing their code at the same time, and using the same multicast channels, it would not interfere your execution. In fact there might be situations where lots of processes from independent students could gel into a large group which is OK. Also you must convince yourself that the above protocol in the long run is self stabilizing which should be a major goal of any distributed algorithm in principle.

Sample log output format:

```
Process Details: [host: sun114-11.cise.ufl.edu port: 54233]
LOG FILE NAME: PROCESS1.LOG
SENT: GJOIN on channel [239.0.0.3:7777] [SYSTEM TIMESTAMP 1188414357]
RECV: WAIT at [sun114-11.cise.ufl.edu port: 54233]
RECV: ACCEPT at [sun114-11.cise.ufl.edu port: 54233] GROUPKEY: A54CD6513FE0BD91
RECV: HEARTBEAT on channel [239.0.0.1:5555] from [sun114-13.cise.ufl.edu port: 54263]
GROUP-COORDINATOR [sun114-14.cise.ufl.edu port: 41623]
SENT: SELF HEARTBEAT on channel [239.0.0.1:5555]
RECV: HEARTBEAT on channel [239.0.0.1:5555] from [sun114-15.cise.ufl.edu port: 53152]
RECV: ELECT 28305d456f24668bc967d6cc57d3640e on channel [239.0.0.2:6666]
SENT: ELECT NO 28305d456f24668bc967d6cc57d3640e to [sun114-14.cise.ufl.edu port: 41623]
SENT: A54CD6513FE0BD91 LGROUP on channel [239.0.0.2:6666]
****EXIT TRIGGER RECEIVED****
****EXIT TRIGGER VERIFICATION SUCCESSFUL****
****SENDING EXIT NOTIFICATION TO ALL RUNNING THREADS****
****SELF TERMINATING NOW****
```

This is what your log files must look like for each processes. The order of messages is not important but the format is. Some missing details in the above sample log must in principle follow the similar format structure.

Failure to follow the log format and failure of your submitted code execution using the provided script will result in points docked from your total project grade.

Report:

You are to provide a file named **report.txt** that should include:
- Problem definition, proposed solution and how it works
- How you designed and implemented the system
- What are the results you got (in plain English, discuss the output of your system, do not copy & paste the program output)
- All the bugs or problems known, any missing items and limitations of your implementation – IF ANY. (honesty deserves additional points)
- Any additional sections you see necessary

Please note that your reports MUST consist of your own sentences, if you have to copy anything from anywhere you have to quote it and provide reference.

Also a perfect program does not necessarily deserve full points if it is not complemented with a good report. A good report is a brief one that help its reader to understand the system thoroughly from the problem definition through the limitations.

Submission Guidelines:

1. Only submit your Java code files, do not include .class files in your submissions
2. Use **tar cvf proj3.tar <*file list*>** to tar your submission.
3. Include a 1 - 3 page(s) project documentation in .txt format. Name it **report.txt**
4. You must submit the tar file after logging into your UF-IBA account. Select *Project 3* from the drop down menu and select the appropriate tar file on your local computer before submitting.
5. UF-IBA allows file uploads up to 1 Mb in size, make sure your tar file does not exceed the size.
6. Please preserve the submission confirmation generated by UF-IBA after successful upload.

Execution Requirements:

1. Your Java file containing main() should be named ***start.java***.
2. Your program must compile and execute properly on CISE **sand** and **rain** (Solaris) machines. Your projects will **<u>not</u>** be tested on any other machines.
3. Execution must begin with command: **java start**
4. All your code files must reside at the root level of the project directory and it is advised not to use java ***package*** primitive at the start of your code.
5. You will assume all your code files will be expanded at the **root level directory** containing no other subdirectory.
6. We will use these script to test your project:

```
#!/bin/sh
tar xvf proj1.tar; rm *.class
rm *.log
javac *.java; java start
---------------------------------------
#!/bin/sh
cat process1.log
cat process2.log
cat process3.log
cat process4.log
cat process5.log
cat process6.log
cat report.txt
```

Grading Criteria:

Correct Implementation/Execution/Graceful Termination/Nonexistence of Run-away processes	*75%*
Report	*20%*
Stylish Code/Readability/Comments/Structure and Readability of Output	*5%*
TOTAL:	*100%*

Java Resources and note on MD5:

You may find tutorials and sample code on general Java programming, Threads and TCP/IP programming in Java by following the link http://java.sun.com and searching for specific information you need. You may use any 3rd party MD5 Java implementation provided you credit the source in your project report. Alternatively on Solaris machines you can use the system command "***openssl md5*** <argument>" to compute the MD5 checksum of the argument. Use man pages to get additional info.

Note on Run-away 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`