COP5615

Operating System Principles

Spring 2011

Project 4

Date Assigned: March 15
Date Due: Midnight, March 29(local), April 1(EDGE)

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

In this project we will implement the **Token Based Mutual Exclusion using Suzuki/Kasami Broadcast algorithm (SKB).** For your reference, it is described on pages 137-138 and in Figure 4.21 of the textbook, Distributed Operating Systems and Algorithms by Chow and Johnson. Mutual exclusion ensures that concurrent processes accessing critical shared resources or data are serialized. Suzuki/Kasami's broadcast algorithm can achieve this without being aware of the underlying topology.

To implement this algorithm, each participant should maintain a sequence vector, and all participants share (through message passing) a token, which consists of a token vector and a token queue. Please refer to the textbook for more precise description of the algorithm.

Since each process in the SKB algorithm requests for mutual exclusion by broadcast its request sequence number to the group, we will use the MulticastSocket class in Java. Sine the MulticastSocket class relies on the UDP datagrams (which in turn relies on best-effort connectionless IP), we should expect that the messages could be lost, duplicated or delivered out of order. However, due to the local area network environment we use, these are unlikely to happen. So we will assume that multicast is reliable in this project.

Details of this project are given below.

There will exactly be 5 group members named from 1 to 5. Every member will access the critical section a number of times, which is determined in **system.properties**. Every member will consist of three threads: one for multicasting the request, the second for listening to the request messages, and the third is for delivering/receiving token to/from others (through unicast). Each member should terminate after completing its task (i.e., successfully completion of a certain number of requests). Note that since the messages will be sent to the each member of the group, every group member should also receive its own messages.

How to Send and Receive Multicast Messages

Multicast messages can be sent and received directly between peers by using plain Datagram sockets and packets. But this introduces a lot of overhead to the system since every member has to maintain information about every other group member.

Java provides the **MulticastSocket** class to deal with multicast messages within a group. This class uses the IP Multicast mechanism that is supported by the underlying IP network. IP Multicast mechanism in turn relies on the **Multicast group** that is simply an IP address that falls in to the IP class D (224.0.0.0 - 239.255.255.255) address. So, to multicast a message, a host simply sends its message to the predetermined class D address of the group. To receive a multicast message, a host should listen on a port, which is bound to the same predetermined class D address.

In the application level, we will not deal with the network level details of the Multicast mechanism since Java provides the necessary APIs by hiding all lower network level details.

Common procedure to receive and send multicast messages:

- Determine the multicast group address (from the range given above) and a port number for the multicast socket to be bound.

To receive:

- Create a multicast socket by giving the port number from the previous step to listen on (MulticastSocket())
- Announce interest in the multicast group by joining the group. (joinGroup())

- Receive messages by creating a datagram packet (DatagramPacket()) and calling receive() method.
- Leave the multicast group (leaveGroup())
- Close the socket.

To send:

- Create a multicast socket by giving the predetermined port number. (MulticastSocket())
- Create a datagram packet by specifying the predetermined group address and port number.(DatagramPacket())
- Send messages using send().
- Close the socket.

Network Environment We Will Use

Since we will use the Multicast class of Java and it relies on the underlying network, our network environment must support IP multicast. But because our department network does not support IP multicast in general, we need to limit our environment into a single network, which immediately connects all the hosts we use. We have such a network environment in CISE lab 114. The machines that we can use are:

lin114-01 through lin114-05
lin114-06 through lin114-10
lin113-01 through lin113-05
lin113-06 through lin113-10

Infact, any other machines, which are connected directly to the same network (i.e. there is no router between them) can be used. It may also be practical to test the system first using only one host. After making sure that everything works as expected, a few tests on different machines may be sufficient.

Note on Time-to-Live value

Time-to-Live (TTL) is a parameter that could be specified for an IP datagram. It specifies how many times a given datagram will be forwarded on to the network by the routers. It is also known as the *hop count*. This value is important in the case of multicasts to limit the datagram to be forwarded to a certain part of the network.

Because we will use a single network to test our application, a TTL value of 1 should be sufficient.

Program Specification

We will have two programs:

The first program (**Start.java**) (**Please note that the file name begins with a capital 'S' and NOT small case 's'**) will read the configuration file**, system.properties**, and start all members on remote machines as we did in previous projects using s*sh.* After starting, all the members this program terminates.

The second one is the main program that implements the SKB. This program will create three threads, the first one for multicasting messages, the second one for receiving multicast messages, the third one is for receiving the token, delivering it, and passing it to the other member if necessary. This program will accept the necessary parameters as in the previous projects.

The parameters that should be passed to the main program are id number of the members (from 1 to 5), each member can get the unicast port of the other members listening on by reading the system.properties.

Initially, we will assume that the member with id=1 is holding the token.

A Sample configuration for the system.properties:

```
Multicast.address=239.1.2.3

Multicast.port=54325

ClientNum=5

numberOfRequests=3

Client1=lin114-01.cise.ufl.edu

Client1.port=49001

Client2= lin114-02.cise.ufl.edu

Client2.port=49002

Client3= lin114-03.cise.ufl.edu

Client3.port=49003

Client4= lin114-04.cise.ufl.edu

Client4.port=49004

Client5= lin114-05.cise.ufl.edu
```

```
Client5.port=49005
```

Multicast.address and Multicast.port are the address and port of the multicast group respectively. ClientX specifies the address of the computer on which the client X will run. Clientx.port specifies the port number that the specific client will listen on. The token delivery will be realized through this unicast port. Note that there is at most one token in the system, and one thread is enough for receiving/sending token from/to others.

Output

Below is one possible sample output file of the system. You MUST strictly follow this output format. The output file is named **request.log** .

Member ID is the identifier of the node holding the token .

The meaning of **Sequence Vector, Token Vector and Token Queue** is the same as specified in the book.

When each member delivers the token to others, its status should be included in the final output.

```
GROUP MEMBER: 5
# of each Member'sRequst : 3

Member ID  Sequence Vector   Token Vector    Token Queue
=========   ================  =============   = ==========
  1           0 0 1 0 0          0 0 0 0 0         3
  3           0 1 1 0 1          0 0 1 0 0         2,5
  2           1 1 1 0 1          0 1 1 0 0         5,1
  5           1 2 1 1 1          0 1 1 0 1         1,2,4
  1           1 2 2 1 1          1 1 1 0 1         2,4,3
  2           1 2 2 1 1          1 2 1 0 1         4,3
  4           1 2 2 1 2          1 2 1 1 1         3,5
  3           2 2 2 1 2          1 2 2 1 1         5,1
  5           2 3 3 1 2          1 2 2 1 2         1,3,2
  1           2 3 3 1 3          2 2 2 1 2         3,2,5
  3           3 3 3 1 3          2 2 3 1 2         2,5,1
  2           3 3 3 2 3          2 3 3 1 2         5,1,4
  5           3 3 3 2 3          2 3 3 1 3         1,4
  1           3 3 3 2 3          3 3 3 1 3         4
  4           3 3 3 3 3          3 3 3 2 3         4
  4           3 3 3 3 3          3 3 3 3 3         NULL
```

Note that all members run on different computers. One possible way to print the output is to save the status to a string, and when the token is sent to others, the string is appended. Finally, the last member writes the string to the file. Another way can be as follows. Before a member sends the token to others, it writes its status informaiton to the file directly.

Requirements/Reminder

- Always check and kill all run-away processes.
- Please remember the same documentation requirements is valid for all projects.
- There should be 5 members in the group identified as 1, 2,3, 4, and 5.
- The command to start the system must be **java Start**
- The name of the configuration file must be **system.properties** and this file should not be submitted.
- The program prints the output to the file, named log.
- Failure to follow the requirements (including the file and program names and submission procedures) will result in point loss.

Tips

- Between two requests, each member should sleep for a random time in the range of 0-1000ms.
- The time for accessing the object takes a random time also in the range of 0-1000ms.

- Pick random group addresses and port numbers for your tests to make sure you are not receiving others' messages. You may add some unique identifier to the multicast messages (e.g, your login name) and discard any messages, which do not have this identifier.

Submission

Please follow the submission guidelines as in the previous projects and replace **X** with **4** for this project.

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 proj4.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 proj4.tar

tar xvf proj4.tar; rm *.class rm *.log

javac *.java; java start
```

```
#!/bin/sh

cat request.log

cat report.txt
```

Please test your tar file using these scripts before you submit it 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. They should be killed (removed) immediately after each testing. 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.

Reference

http://download.oracle.com/javase/tutorial/networking/datagrams/broadcasting.html