

Assigned: Monday, Mar-14-2011

Due: Monday, Mar-28-2011 (at 5 pm)

Totally-ordered multicast using Lamport's logical clocks

In this homework, you are to implement totally-ordered multicast using Lamport's logical clocks (textbook p244-248, in particular **p248**). The goal is to keep the replicas on the 3 VMs consistent using active replication protocol (p311).

The idea is to have the processes on the VMs share a ball which has a coordinate (x, y) . The processes can access the coordinate information simultaneously. That is, concurrent update operations are allowed on the position of the ball. An operation that occurs at a process is contained in a message that is delivered to other processes as well as the node itself.

Each process has two threads: a worker thread and dispatcher thread. The process generates random coordinate changes $(\Delta x, \Delta y)$ at random intervals and sends them to all the processes including itself. The dispatcher thread waits for updates $(\Delta x, \Delta y)$ from other replicas. Upon receiving a message, it inserts the message in its local queue and multicasts an acknowledgement to the other processes. The local queue containing messages is ordered according to the timestamps of the messages. That is, the earlier the timestamp, the closer the message is to the head of the queue.

Requirements

Your program should be running on the VM instances given to you running on the OpenStack Nova machine (192.168.135.12) in homework 4. As the underlying transport protocol, TCP is to be used to communicate between processes. You will therefore need multithread and socket programming skills. Below are useful links for this homework in Java. For this homework, you can use one of the following languages: C, C++, and Java. If you plan to use other languages, please let us know immediately.

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>

For details on the multithread servers, please refer to page 77 in the textbook.

Processes are assumed to be numbered (P_n) in the order of the last digit of the IP address of the virtual machine that the process is residing in. The process numbers range from 0 to 2. Thus the processes are named P0, P1, and P2.

For example,

P0: 192.168.0.2 (the lowest)

P1: 192.168.0.3

P2: 192.168.0.4 (the highest)

Each process is assumed to have endpoint locations consisting of the IP address and the port. For example, the endpoint of the process P0 would be 192.168.0.2:9999 if the port number is 9999. Also, each process has a list of the endpoints for all the processes (P0, P1, and P2) in the network including itself. To do this, your program should read a file named "info.txt" that contains a list of addresses and connect some of them one by one as described in the deployment phase 1 below. The file name should be EXACTLY the same as "info.txt" in lower case. The lines of the file contain the IP address and port of a process as: [ip address] [port]

The content of the file "info.txt" would be the following:

```
192.168.0.2 9999
192.168.0.3 9999
192.168.0.4 9999
```

You may use the different port number for each process.

The details of the implementation are described below.

Ball

The ball position is specified by coordinate x and y . It is initially located at $(0, 0)$. Coordinate changes (i.e., the "deltas") Δx and Δy are in the range $[-100, 100]$. A process can move the ball by delta $(\Delta x, \Delta y)$.

Update operation: new coordinate $(x', y') = \text{current coordinate } (x, y) + \text{delta } (\Delta x, \Delta y)$

That is, $(x', y') = (x + \Delta x, y + \Delta y)$ where x and y are integer type and the range of Δx and Δy is $[-100, 100]$ including -100 and 100 .

Message

A message contains a type, process id, and payload as shown below. You may add more types if necessary.

- Type: UPDATE, ACK
 - o UPDATE: indicates the coordinate changes $(\Delta x, \Delta y)$
 - o ACK: acknowledges the UPDATE message has been received
- Payload:
 - o If type is UPDATE then the payload contains the process id of the message sender and the coordinate (x, y) of the ball
 - o If type is ACK then the payload contains the process id of the message sender and the timestamp of the message.

Queue

A process has its own local queue to store messages. A message in the queue is inserted in the order of its timestamp. It is removed from the queue only when it is acknowledged by all the other processes.

Clock counter

Processes maintain a Lamport clock at a constant rate (ticks/sec). However, the clock rates are different for each process. The clock increases by the clock rate every second. In addition, it increases by 1 for every event (sending/receiving/delivering message) as in the textbook p246-247. For example, if P0, P1, and P2 have clock counter unit of 2, 3, and 1, respectively, then the clocks tick as follows. Note that the process id is attached to low-order end of time separated by a decimal point to break the tie.

P0's clock: 2 ticks/sec

0.0
2.0
4.0
...

P1's clock: 3 ticks/sec

0.1
3.1
6.1
...

P2's clock: 1 tick/sec

0.2
1.2
2.2
...

Command line

The main program name is Lamport. The program should accept the process id and the number of update operations to be performed.

In the case of Java,

```
java Lamport [Process id] [Number of operations] [clock rate]  
e.g., java Lamport 0 30 4
```

In C/C++,

```
./Lamport [Process id] [Number of operations] [clock rate]  
e.g., ./Lamport 0 30 4
```

If you run 30 update operations for each process, you would end up with 90 ($30 * 3$) update operations performed.

Starting remote processes

You may want to use the following Java code to start processes on remote machines.

```
Runtime.getRuntime().exec("ssh -i " + key + " " + username + "@" + host + " cd " + path + " ; java  
Lamport " + arguments);
```

Deployment

Phase 1: Connection setup

All the processes on the VMs are started. You will need to have the processes wait until the other processes are up. When a process starts, it establishes TCP connections to the processes with lower process ID.

In this case,

P0 starts

P1 starts and connects to P0

P2 starts and connects to P0 and P1.

Phase 2: Running phase

Once all the processes are running and connected to each other, each process starts moving the ball by randomly generating changes (Δx , Δy) of the ball's position (x , y) at every interval period. The interval is chosen also randomly between 0 to 1000 milliseconds (1 second) excluding 0. A process sends the update (Δx , Δy) to all the processes including itself. It also accepts incoming messages from other processes and performs the operations in the messages. Upon receiving an UPDATE message, the process multicasts acknowledgement (ACK) messages to others. The process is allowed to update the position of its ball ONLY after it receives acknowledgements from all the processes. Note that the order of updates performed could be different from what is actually done in real-time, but all the processes see the updates in the same order (total order). Also, since there are local objects shared by multiple threads, you need to consider synchronization of accesses to the objects within a process.

Phase 3: Termination phase

After the number of iterations given, all the processes are terminated gracefully. Note that a process terminates only after all the messages are delivered and its local queue is empty. Also, the process is allowed to terminate after it confirms that others have finished their job (no more messages to send AND have delivered all the messages). In order to do this, they need to exchange finishing messages.

Termination conditions

1. All the messages are delivered.
2. Local queue is empty
3. Other processes finish

Logging

Your program is required to create a log file in which all the events regarding connections and messages are written with the timestamps. The log files are named log0, log1, and log2 with the number being the corresponding process id.

Process connection

When a process is connected to other process, write something like the following into the log file:
P0 is connected to P1 (192.168.0.3).

When a process is connected from other process, write something like the following into the log file:
P1 is connected from P0 (192.168.0.2).

When a process has finished its job, write something like the following into the log file:

P2 finished.
P0 finished.
P1 finished.

When all the processes have finished, write something like the following into the log file:

All finished. P2 is terminating...

Message

When a message is delivered, the ball moves to a new location and this event is written into a line in the log file. The line has three columns which represent the local time, process id of the message sender, timestamp, and the current coordinate of the ball separated by a white space as shown below.

column1 column2 column3

[localtime] [operation number : local counter] description of the event ...

[month/day hour:minute:second] [OPnum : Cclock] description of the event ...

where operation number and local counter start from 1.

The exact format is:

[MM/DD hh:mm:ss] [OPnum : Cclock] Ball moved to (x,y) by (Δx , Δy).

An example of the log file of the process P1 would be:

```
[P0]192.168.0.2:9999
[P1]192.168.0.3:10000
[P2]192.168.0.4:10001

P1(192.168.0.3) is listening on port 10000 ...

[ 03/09 14:34:39 ] P1 is connected to P0 (192.168.0.2:9999).
Waiting for all to be connected...

[ 03/09 14:34:41 ] P1 is connected from P2 (192.168.0.4).
All connected.

[ 03/10 02:18:32 ] [ OP1 : C9 ] Ball moved to (42,28) by (42,28).
[ 03/10 02:18:32 ] [ OP2 : C11 ] Ball moved to (-31,81) by (-73,53).
[ 03/10 02:18:33 ] [ OP3 : C12 ] Ball moved to (-27,16) by (4,-65).
...
[ 03/10 02:18:38 ] P2 finished.
[ 03/10 02:18:38 ] P0 finished.
[ 03/10 02:18:38 ] P1 finished.
[ 03/10 02:18:39 ] All finished. P1 is terminating...
```

IMPORTANT: Your log file format **MUST** be the same as the above format. Otherwise, you will not get credit.

Submission (March 28th 2011, before 5 pm):

You are required to provide the following in a single file in your private forum.

- readme.txt file that describes the program structure such as files, classes, and significant methods

- report.txt file that shows what you learned and the difficulties you had in developing the program and how you solved them
- makefile to compile the program. We will type only “make” to compile the program as done in homework 2.
- program source code files. DO NOT include binary files (.class, .obj, .out, .exe...). That will lead to grade penalty.

IMPORTANT: ALL the above files should be tarred into **ONE FILE** named as **login-hw5.tar** where **login** should be replaced by **your login name** on the course website. If you submit more than one file, you will be penalized.

Please use the following commands when you submit the files.

Create a directory *login_hw5*.
mkdir *login_hw5*

Copy all the required files into the directory *login_hw5*.
cp file1 file2 file3 ... *login_hw5*

tar cf *login_hw5.tar login_hw5/**

If you submit files with extension other than tar (such as tar.gz, rar, and zip), points will be deducted from your grade.