**EEL 6935 Spring 15 (Distributed Computing)** **Homework 6**
**Assigned: Tuesday, Mar-17-2015**
**Due: Tuesday, Mar-24-2015 before 5 pm**

# Distributed control of currency exchange value using ZooKeeper API

In this homework, you will implement the totally-ordered multicast using ZooKeeper. The goal is to keep the replicas on 3 VMs consistent using active replication protocol, which is the same as Homework 5.

The idea is to have the processes on the VMs maintain a consistent currency value as specified by the pair of rates (sell rate, buy rate). The processes can access the currency value simultaneously. As the same time, concurrent update operations are allowed on the value of currency. 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 VM runs a ZooKeeper instance, maintained in the ***replicated mode***.

Each process has two threads: a worker thread and a dispatcher thread. The worker thread generates random rates changes (Δx, Δy) at random intervals and sends them to the ZooKeeper server. The dispatcher thread waits for (or pulls) updates (Δx, Δy) from the ZooKeeper server. The ZooKeeper inserts the message into its message queue when receiving a new one and deletes the message from its message queue when all processes have obtained the message copy.

## Requirements

Your program should be running on the VM instances given to you. As the underlying transport protocol, TCP is to be used to communicate between processes. You might need the below knowledge:

Running Replicated ZooKeeper: http://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html
Programming with ZooKeeper: http://zookeeper.apache.org/doc/r3.4.2/zookeeperTutorial.html

The reset is the same as Homework 5.

Processes 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,
P**0**: ece-acis-dcXX**1 (the lowest)**
P**1**: ece-acis-dcXX**2**
P**2**: ece-acis-dcXX**3 (the highest)**

Processes communicate with each other via the ZooKeeper service so you might not need the "info.txt" as used in Homework 5.

## Currency

The currency value is specified by a pair of rates (sell rate, buy rate). It is initially set to (100, 100). Rate changes (i.e., the "deltas") Δx and Δy are in the range [-80, 80]. A process can update the rate by delta (Δx, Δy). Update operation: new pair of rates (x', y') = current pair of rates (x, y) + delta (Δx, Δy). That is, (x', y') = (x + Δx, y + Δy) where x and y are integer type and the range of Δx and Δy is [-80, 80] including -80 and 80. (For simplicity, we allow negative rates in this homework.)

**Message**
You will need the UPDATE message in this homework. You might also need the ACK message to notify the ZooKeeper server the successful delivery (this is not required). You may add more types if necessary.

- Type: UPDATE
  - UPDATE: indicates the rate changes ($\Delta x$, $\Delta y$)
- Payload:
  - If type is UPDATE then the payload contains the process id of the message sender and the pair of rates (x, y) of the currency

**Queue**
The message queue is maintained and ordered by the ZooKeeper server.

**Clock counter**
Processes still 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 pages 246-247. For example, if P0, P1, and P2 have clock counter unit of 2, 3, and 1, respectively, then the clocks tick as follows. The process id is attached to low-order end of time separated by a decimal point to break the tie. ***Please note that the totally-ordered multicasting mechanism in this homework is ensured by the ZooKeeper service.***

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 TotalOrderZK. The program should accept the process id and the number of update operations to be performed.

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

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

## Deployment

**Phase 1: Connection setup**
All the processes on the VMs are connected with the ZooKeeper server.

**Phase 2: Running phase**
Each process starts update the currency value by randomly generating rate changes (Δx, Δy) of the value pair (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 the ZooKeeper server. It also obtains incoming messages from the ZooKeeper and performs the operations in the messages. Upon receiving an UPDATE message, the process might send an acknowledgement (ACK) message to the ZooKeeper server (this is not required).

**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.

Termination conditions:

- ✓ All the messages in the ZooKeeper Server are delivered.

## 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 the ZooKeeper server, write something like the following into the log file:

*P0 is connected to ZooKeeper (192.168.0.3).*

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 currency is set to a new value pair 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 rates of currency 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] [OP*num* : C*clock*] description of the event, *where operation number and local counter start from 1.*

The exact format is:
[ MM/DD hh:mm:ss ] [OP*num* : C*clock* ] Currency value is set to (x,y) by (Δx, Δy).

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

## Submission (March 24 2015, before 5 pm):

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

- readme.txt file that describes the program structure such as files, classes, and significant methods
- makefile to compile the program. We will type only "make" to compile the program as done in previous homework.

- 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 ***yourfirstname_yourlastname_*****hw6.tar**. If you submit more than one file, you will be penalized.

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

**Submission Policy:**

- Do NOT include binary files. Use the file names as specified above. Incorrect submission formats will lead to a grade reduction.
- You will be given access to three virtual machines. Your program will be tested on a virtual machine with exactly the same configuration as your virtual machine. Make sure to test your program before submission.
- All submissions are expected by the deadline specified in the homework assignment. Grade is automatically reduced by 25% for every late day.
- Make sure you test your submitted code using the tar file you submitted on the virtual machine used for course homework. If untar, make, compile or any other command needed to execute your program do not work, your homework grade will be zero.