EEL 6935 Spring 2013 (Distributed Computing) **Homework 5**

Assigned: Thursday, March-21-2013

Due: Friday, April-18-2013 (at 5 pm)

Reliable multicast with JGroups toolkit

You are to implement a stock exchange system using a middleware called JGroups (http://jgroups.org) instead of implementing group communication primitives. The goal is to handle process failures by means of reliable multicasting using virtual synchrony (textbook page 348-355). Virtual synchrony was used to develop the New York and Swiss Stock Exchange fault-tolerance architecture, the French Air Traffic Control System, the US Navy AEGIS system, IBM's Business Process replication architecture for WebSphere, and Microsoft's Windows Clustering architecture for Windows Longhorn enterprise servers [1]. For this homework, the stock exchange system will be implemented, replicated and maintained to be consistent on your 3 VMs. You will handle multiple clients, multiple shared objects, and multiple replicated servers in the presence of process failures.

Goals:

1. Becoming familiar with group communication
2. Dealing with multiple clients and multiple shared distributed objects
3. Designing and implementing a distributed application using existing middleware
4. Handling process failures by means of using virtual synchrony

[1] http://en.wikipedia.org/wiki/Virtual_synchrony

Unless otherwise stated, the same specification in homework 4 is applied to this homework.

Since JGroups is written in Java we will take into account only Java for this homework.

Below are useful links for this homework. Java 1.6 or later: http://java.com

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

JGroups

JGroups is a toolkit for the development of distributed Java applications based on group communication. We will use the latest version 3.2.7.

The jar file is available here:

http://sourceforge.net/projects/javagroups/files/JGroups/3.2.7.Final/jgroups-3.2.7.Final.jar/download

The source code is also available here:

https://github.com/belaban/JGroups

User's guide including tutorial and manual:

http://jgroups.org/ug.html

Sample application

You can start writing a simple text-based chat application (SimpleChat) in the JGroups tutorial (http://www.jgroups.org/tutorial-3.x/html/index.html) and extend it to the stock server.

Protocol stack

The most notable feature of JGroups is the flexible use of protocol stack. This allows you to modify the underlying protocol stack as needed. The default stack is defined in file "udp.xml". You may edit the file to make use of virtual synchrony and ordering of message delivery. The sample protocol stack files are under the directory "/conf" of the jgroups source code. If you want to use your own protocol stack file, the file name should be "protocol.xml". If you don't provide it, the default file "udp.xml" will be used.

Virtual synchrony

Please refer to the section "5.7. Flushing: making sure every node in the cluster received a message" in the manual (http://www.jgroups.org/manual-3.x/html/index.html)

Message ordering

Please refer to the section "7.11.1.SEQUENCER" of the manual.

State transfer

When a process joins or recovers, the application state needs to be transferred from other processes to that process. If the state information is too large to transfer, you may want to use “STREAM_STATE_TRANSFER”. Since the state information won’t be that large in this homework, “STATE_TRANSFER” will be enough. However, its size should be bounded. For more details, refer to the section “2.6. Extra credits: maintaining shared cluster state” in the JGroups tutorial.

Group name

A group has a unique name by which it is identified and located. A group name is assigned to each of you and you are required to use that name for your group. Your group name is “*Your_name*”.

Implementation

The stock exchange server acts as a broker. It accepts requests from multiple clients simultaneously and matches buy and sell orders. When it finds out a match (buy and sell), the server commits the actual stock trade. You do not need to implement the client though you may need it to test the server. However, the server MUST accept the requests in the format specified as below. The main components of the system include stock, request, order, trade, and account.

The server maintains:

- Stock list
- Account data for clients

A stock contains:

- Name: company name such as Google and Amazon
- Symbol: stock market ticker symbol such as “GOOG” and “AMZN”
- Shares: total number of shares. Initially 10000
- Price: the price of a share (natural number). Initially $100. If the price becomes below 1 the stock is delisted.
- Order lists: sell list (increasing price order) and buy list (decreasing price order)
  Example sell list: 100, 110, 150 …
  buy list: 200, 150, 100 …

Stock information is retrieved from a file called “index.properties” which can be easily loaded using Java Properties class (http://en.wikipedia.org/wiki/.properties).

<symbol>	<name>
GOOG	Google
AMZN	Amazon
AAPL	Apple
…	

You will be given the file “index.properties” which is copied from the NASDAQ-100 Index (http://dynamic.nasdaq.com/dynamic/nasdaq100_activity.stm).

Request

A request submitted by a client is a string and has the following properties.

- action: sell or buy
- format: <client id> <buy|sell> <stock symbol> (<price> <shares>)
- buy: C<client id> buy <stock symbol> <price> <shares>
- sell: C<client id> sell <stock symbol> <price> <shares>
- example: `C2 BUY GOOG 105 10`

The request strings to be used for testing will be uppercase (capital). Thus, the example request should be "C1 BUY GOOG 105 10", and NOT "c1 buy goog 105 10". If you're worried about your program not working with the string, you can simply use equalsIgnoreCase(), toLowerCase() or toUpperCase() function of String class.

The format is important because your program will be tested by the requests.

Order

For simplicity, there will only be a limit order. Also, a transaction is to be completed only in 10 shares. An order is defined by an action (buy or sell) and a price. For a limit order, the wanted prices of a seller and a buyer must be matched. For example, if a seller submits a limit order of $105 then it should wait until the buying price higher than or equal to $105 appears. Likewise, if a buyer submits a limit order of $105 it should wait until the selling price lower than or equal to $105 appears.

Trade

The account information of the buyer and seller are updated when there is a match. For simplicity, the actual trade price is set to the seller’s wanted price. The balance of the seller increases by <the trade

price * shares> and the balance of the buyer decreases by the same amount. The current market price is updated to the price of the trade.

For the sake of simplicity, there are no trades with companies and clients are allowed to have negative shares (-1, -2, -3 …). Therefore, the initial trades would be simpler than trading with companies.

Account

Account information is created when the first server starts. It contains a unique client id assigned by the server and balance (initially $10000). It also maintains a list of stocks owned by the client with the shares. After the trades start, the account balance is calculated using the following formula,

Total balance = current cash balance + stock1 current value of stock1 + stock2*current value of stock2+…*

Handling failures

We only consider a process failure (crash) which can be simulated by abruptly pressing "ctrl + c". When a process crashes, it is considered as leaving the group and is detected by others. This is because there is no difference between crashing and leaving from the process point of view. When the failed process recovers, it obtains the most current information of the group from other processes so that the state can be up-to-date. While testing, the processes will be turned on and off in a random fashion.

Server port number

Your program should accept the port number that clients can access. If the port number is unavailable the program prints an error message and terminates. Again, use the first four digits of your UFID as your program port number

Communication between a server and a client

Please use the following codes for communication between the server and clients since your program MUST accept the requests in Java String class.

Client code to send a request to a server:

```
try {
        sock = new Socket(addr, port);
        oos = new ObjectOutputStream(sock.getOutputStream());

        String order = "C2 buy GOOG 105 10"; // example order string
        oos.writeObject(order);
        oos.flush();
        oos.close();
        sock.close();
} catch(…){…}
```

Server code to receive a request from a client:

```
Socket clientsock = null;
while(true) {
        try {
                clientsock = serversock.accept();
                ObjectInputStream ois = new ObjectInputStream(clientsock.getInputStream());

                String request = (String)ois.readObject();
                … // handle request
                ois.close();
                clientsock.close();
        } catch (…) {…}
}
```

Command line

Compile your program:

```
javac –cp .:jgroups-3.2.7.Final.jar VSynchrony.java

Java –cp .:jgroups-3.2.7.Final.jar VSynchrony <process id> <number of clients> <server port number>
```

Deployment

When the server process starts, it loads the stock information from “index.propreties” file into the stock list and joins the group. Note that the processes can enter the group in any order. The first server (the one that joins the group first) creates a certain number of accounts for the clients and waits for incoming requests from the clients. The first server also maintains the account information in which an account is identified by a unique client id starting from 1 and increasing by 1 for each client (1, 2, 3 …). When the subsequent servers join the group, they receive the current state information from the group.

Note that only the first server creates the client information. That is, the subsequent servers do not create new clients and just receive the existing client information from the group.

A client submits a request along with its id. Since TCP is used to communicate with clients, the server needs to create a TCP socket for each request. When the server receives a request, it sends the request to the group.

No malicious clients are assumed. However, you should assume that multiple clients submit requests to the servers simultaneously. Also, they are able to submit requests to any server at any moment. That is,a client can place an order at server S1, but the next order can be at server S2. This can happen also in reality if the client moves from one place to another.

Logging

The server processes are required to print the stock information every time it receives an order as well as when the view changes. Since JGroups provides logging the view changes by default, you only need to take care of printing the orders. For each order, the order information, the current market price, the buy and sell lists of the corresponding stock, and the list of the account information are displayed as shown below. Note that the stock information is the one that is updated after the transaction has been completed.

```
---------------------------------------------
Order #: C<client id> <action> <symbol> <price>
Price: <current market price>
Buys: [C1 BUY GOOG 105 10]
Sells: []
[C<client id> <balance of the client>, …]
---------------------------------------------------
```

An example printing is as follows.

```
-------------------------------------------------------------------------
GMS:   address=localhost-51045,   cluster=your_name-group,   physical
address=fe80:0:0:0:16:3eff:fe39:9632:54650
-------------------------------------------------------------------
** view: [localhost-51045|0] [localhost-51045]
-------------------------------------------------------------------
Order 0: C1 BUY GOOG 105 10
Price: 100
Buys: [C1 BUY GOOG 105 10]
Sells: []
```

```
[C0 10000, C1 10000, C2 10000, C3 10000, C4 10000, C5 10000, C6 10000, C7 10000, C8
10000, C9 10000, C10 10000]
-------------------------------------------------------------------
Order 1: C2 BUY AMZN 100 10
Price: 100
Buys: [C1 BUY GOOG 105 10] [C2 BUY AMZN 100 10]
Sells: []
[C0 10000, C1 10000, C2 10000, C3 10000, C4 10000, C5 10000, C6 10000, C7 10000, C8
10000, C9 10000, C10 10000]
-------------------------------------------------------------------
Order 1: C3 SELL AMZN 90 10
Price: 90
Buys: [C1 BUY GOOG 105 10]
Sells: []
[C0 10000, C1 10000, C2 10000, C3 10000, C4 10000, C5 10000, C6 10000, C7 10000, C8
10000, C9 10000, C10 10000]

-------------------------------------------------------------------

… <omitted >

-------------------------------------------------------------------
** view: [localhost-51045|1] [localhost-51045, localhost-27498]
-------------------------------------------------------------------

… <omitted>

-------------------------------------------------------------------
Order 17: C8 BUY AAPL 115 10
Price: 110
Buys: [C2 BUY AAPL 100 10]
Sells: [C5 SELL AAPL 110 10]
[C0 10000, C1 9905, C2 10000, C3 10095, C4 10115, C5 10000, C6 9895, C7 10105, C8 9885,
C9 10000, C10 10000]
--------------------------------------------------------------------
```

Submission (April 18th 2013, before 5 pm):
Report

Create and run an example sequence of the orders. Plot the corresponding graph with the x-axis representing time and the y-axis representing price of each stock. You also need to answer the following questions.

1. What are the distributed shared objects in your program?
2. Is your program Byzantine failure safe? Why or why not? If not, what needs to be done?
3. Is virtual synchrony in JGroups scalable? Why or why not?

4. In order to achieve consistency between the processes, did you need additional distributed locking protocols? Why or why not?

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

- "readme.txt" file that describes the program structure such as files, classes, and significant methods
- report file (txt/pdf/doc format)
- "makefile" to compile the program. We will type only "make" to compile the program as done in past homeworks.
- program source code files (only server part). Client program is not required. DO NOT include binary files (.class, .obj, .out, .exe...). That will lead to grade penalty.
- "protocol.xml" file, if any

You do not need to provide the file "index.properties".

IMPORTANT: **ALL** the above files should be tarred into **ONE FILE** named as *your_name-hw5.tar* where *your_name* should be replaced by your 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 your_name_hw5.

```
mkdir your_name_hw5
```

Copy all the required files into the directory your_name_hw5.

```
cp file1 file2 file3 … your_name_hw5

tar cf your_name_hw5.tar your_name_hw5/*
```

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

Grading guideline

Report:	10%
Correct file format/compilation:	10%
Correct implementation/running:	80%