The Second Mandatory Programming Assignment
Part 1:

Replicated Bank Account

INF5040/INF9040 Autumn 2015

Objective

- Develop a distributed application that models a replicated bank account.
- The implementation should follow the "replicated state machine" paradigm on top of group communication.
- Using the Spread Toolkit

Scope

Spread is an open source toolkit that provides a high performance messaging service that is resilient to faults across local and wide area networks. For this assignment you have to use the Spread toolkit to build a replicated banking system. The system architecture will consist of (a) the standard Spread server and (b) a client that you need to develop and link with the Spread library. The application only needs to support a single bank account with the sequentially consistent replication semantics. Each running instance of the client will represent a replica of this account.

Technical features

The synopsis for running the client should be:

- `java accountReplica <server address> <account name> <number of replicas> [file name]` for java, or

<server address> is the address of the Spread server that the client should connect to. Spread is installed in all ifi machines and several Spread servers can be constantly running. The address of at least one Spread server at ifi will be given to you by the teaching assistant. When deploying clients on University machines, you can use those servers instead of running your own ones. If you want to test your client outside the University, you should run your own Spread server and the clients should connect to it.

<account name> stands for the name of the account. In order to avoid name clashes between different groups, each group of students should choose a unique name when running the client. A good tactic would be to use an actual person name or id as part of the account name. All clients that a student group runs should use the same account name.

<number of replicas> is the number of clients that will be initially deployed for <account name>. The application should handle the dynamic addition of new replicas and also tolerate departure of individual clients (due to leaves or crashes) and continue operation when it occurs. All clients can

be deployed on the same machine or different machines; it will not affect the behavior of the application. It may be a good idea to test the application with at least three clients in different machines as in practice, some problems may manifest themselves when the number of replicas is at least 3.

If the optional argument of **[file name]** is not present, the client will interactively accept commands from the user through a command line. If [file name] is present, then the client will perform batch processing of commands that it will read from [file name] and exit.

You can also develop your application using a graphical user interface if you are not comfortable with the command line.
Client execution should be as follows:

1. The client should create a connection to a Spread server.
2. The client should initialize the balance on the account to 0.0.
3. The client should join a group whose name is <account name>.
4. The client should wait until it detects that all <number of replicas> clients have joined the group. To this end, it should receive and analyze messages about membership changes.
    a. All initial replicas will start with the same state: balance = 0.0. After that, the client should handle new joins by setting the state of the new replica, and the state should be consistent across all the replicas: the balance of all replicas should be the same.
5. If the client is deployed without [file name], it should open a command line and wait for user commands. If [file name] is present, then the client will perform batch processing of commands that it will read from [file name] and exit.

The following commands should be accepted and supported by the client:

1. **balance**
   This command causes the client to print the current balance on the account.

2. **deposit <amount>**
   This command causes the balance to increase by <amount>. This increase should be performed on all the replicas in the group.

3. **addinterest <percent>**
   This command causes the balance to increase by <percent> percent of the current value. In other words, the balance should be multiplied by (1 + <percent>/100). This update should be performed on all the replicas in the group.

4. **exchange <from> <to>**
   Exchanges the currency from e.g. NOK to USD. Support these currencies only: NOK, USD and EUR. Look at the "Currency table" down below for exchange rates.

5. **memberinfo**
   Returns the names of the current participants in the group, and prints it to the screen.

6. **sleep <duration>**
   This command causes the client to do nothing for <duration> seconds. It is **only useful in a batch file**.

7. **exit**
   This command causes the client to exit. Alternatively, the client process can be just killed by the means of the operating system.

If the client is deployed with [file name], the batch file should just contain a single command on each line.

As a reminder, the "replicated state machine" paradigm dictates that **all the replicas that do not fail go through the same sequence of changes and end up with the same balance value**. The execution should satisfy sequential consistency with respect to the deposit, addinterest, exchange, memberinfo and balance operations.

Spread at UiO:

Running Spread at IFI:
Try ssh to rubin.ifi.uio.no with your username, and run the Spread daemon from rubin and connect to it from your client.

Development Tools:

1. *Integrated Development Environment*: Eclipse IDE for **Java EE** Developers: http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/junor
2. The Spread toolkit for group communication.
   a. http://www.spread.org/
   b. http://www.cnds.jhu.edu/.

Deliverables:

Via the Devilry system.

- A compressed file (zip) containing all your source code and documentation:
  - The source code should be well commented.
  - The documentation can be a simple help-me file explaining how to run your application.

Deadline: 23:59 on 7th November, 2016

Currency table:

	NOK	USD	EUR
NOK		0.12 USD	0.10 EUR
USD	7.87 NOK		0.83 EUR
EUR	8.55 NOK	1.00 USD	

You change from a currency in a row, to a currency a given column.

Example:

We want to change 100 USD to EUR.

We take a look at row 2 (USD), then at the EUR column and see 0.83 EUR, which means that we get 0.83 EUR for each dollar we have.

Thus, we have 100 * 0.83 = 83 EUR.

Likewise, in row 1 (NOK) column USD, it says 0.12 USD. That means we get 0.12 USD for each NOK we have. 100 NOK is therefore 100 * 0.12 = 12 USD.