The Second Mandatory Programming Assignment

Replicated Bank Account

IN5020/IN9020 Autumn 2025

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 Java RMI to simulate a message distribution server.

Scope

For this assignment you have to use the Java RMI to build a replicated banking system. The system architecture will consist of (a) the standard Message Delivery server and (b) multiple bank servers that you need to develop and link with the Message Delivery server.

The application only needs to support a single bank account with the sequentially consistent replication semantics. Each running instance of the bank server will represent a replica of this account.

Technical features

The bank (server) file must be run with 4 arguments:

<server address> is the address (ip:port) of the Message Delivery server that the bank server should connect to.

<account name> stands for the name of the account: "groupXX". All bank server that a student group runs should use the same account name.

<number of replicas> is the number of bank server that will be initially deployed for <account name>. The application should handle the dynamic addition of new replicas and tolerate departure of individual bank server (due to leaves or crashes) and continue operation when it occurs. All bank servers 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 bank servers in different machines as in practice, some problems may manifest themselves when the number of replicas is at least 3.

[currency file name]. This file describes the trading rate from <Currency> to USD. Each line represents a type of currency. The unit is 1.

[file name] (optional). If the optional argument of **[file name]** is not present, the bank server will interactively accept commands from the user through a command line. If [file name] is present, then the bank server will perform batch processing of commands that it will read from **[file name]** every T seconds and exit. T is a random float number between 0.5-1.5s.

Message Delivery server execution should be as follows:

1. The Message Delivery server allows the connection from bank servers.
2. When receiving a message (transaction list) from bank servers, it must immediately broadcast that message to all bank servers (parallelly) and wait for ACK, before sending the next message. If after 2 seconds and does not receive an ACK from a server, resend the message. If after 5 seconds and still does not receive an ACK from that server, mark that server as FAIL, and remove it from server list.
3. You must use Java RMI to build this connection.

Bank server execution should be as follows:

1. The bank server should create a connection to a Message Delivery server.
2. The bank server should initialize the balance on the account to 0.0.
3. The bank server should initialize the List<Transaction> **executed_list** and Collection <Transaction> **outstanding_collection** as empty and an **order_counter** and **outstanding_counter** to 0, where Transaction is defined as follow:

```
Class Transaction{

        String command;
        String uniqueId;
}
```

4. The bank server should join a group whose name is <account name>.
5. The bank server should wait until it detects that all <number of replicas> bank servers 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 bank server 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.
6. If the bank server is deployed without [file name], it should open a command line and wait for user commands. If [file name] is present, then the bank server 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 bank server:

1. **getQuickBalance <Currency>**
   This command causes the bank server to check and print the balance when converting all the amounts to <Currency> on the account right away, without synchronizing with any previously issued transactions.

2. **getSyncedBalance <Currency>**
   This command causes the bank server to print the synchronized state of the account after applying all of the outstanding transactions in **outstanding_collection**.

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

4. **addInterest <Currency> <percent>**

   This command causes the balance of <Currency> to increase by <percent> percent of the current value. In other words, the balance of <Currency> should be multiplied by (1 + <percent>/100). This update should be performed on all the replicas in the group. If <Currency> is not specified, make change on all types of currency.

5. **getHistory**
   This command causes the bank server to print the list of recent transactions (deposit and addInterest operations with timestamp of execution), sorted by the order in which the transactions were applied, plus outstanding transactions not applied yet. “Recent” is defined as all transactions since the last emptying of the list.

6. **checkTxStatus <Transaction.unique_id>**
   This command returns the status of a deposit or addInterest transaction to show if it has been applied yet.

7. **cleanHistory**
   This command causes the bank server to empty the list of recent transactions.

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

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

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

After a deposit or addInterest command is invoked locally, a new Transaction object is created. The object will then be set information as follow:

```
Transaction.command = "<Command name> <argument value>"; (Ex: "deposit NOK 500")

Transaction.unique_id = "<Bank server_instance_name> <outstanding_counter>"
```

The transaction will then be appended to Collection<Transaction> **outstanding_collection** and **outstanding_counter** will be increased by 1.

The transactions in **outstanding_collection** are broadcast to the group of instances every 10 seconds so that the outstanding_collection will be ordered in a consistent view across the replicas.

Once a **deposit or addInterest** transaction is ordered, the transaction's command is executed and the **order_counter** is increased by 1 (Each transaction with unique_id can be ordered and executed once). Next, the transaction will then be appended to List<Transaction> **executed_list** and the corresponding transaction in **outstanding_collection** will be removed. The **order_counter** and **outstanding_counter** are not reset when cleanHistory is applied.

Note that getQuickBalance issued after a deposit or addInterest by the same bank server can be performed before the previously issued deposit or addInterest transaction has been applied. This occurs because deposit or addInterest need to be propagated to the other replicas first and they are only applied when the totally ordered broadcast message has been received. For example, assume that you have the following order of commands:

```
deposit NOK 100

addInterest 10

getQuickBalance
```

getSyncedBalance

The getQuickBalance command may be executed before applying deposit and addInterest, in that case the result will be 0.

To address this, we add the getSyncedBalance command, which is not executed immediately but instead, it waits until all outstanding transactions previously issued by the same bank server have been applied. Therefore, getSyncedBalance offers stronger consistency compared to getQuickBalance, at the expense of longer delays. In the example above, getSyncedBalance NOK will return 110.

A naive implementation for handling the getSyncedBalance command is that you execute the getSyncedBalance command only when the outstanding_collection is empty. In such a case, there exists possibility of deadlock because of never having an empty outstanding_collection.

For solving the problem of the above implementation, you need to propose another approach. A simple correct implementation can be appending the getSyncedBalance command to the outstanding_collection, similarly to how the deposit and addInterest commands are handled. However, there are two differences in handling the getSyncedBalance command. First, when the getSyncedBalance transaction is received by the replicas, it is executed only by the replica that had sent the command. The second difference is that when the getSyncedBalance transaction is executed, it is removed from the outstanding_collection but it is not appended to the executed_list.

You have to implement both of the mentioned ideas (the naive implementation and the correct one), observe the differences in the response for the getSyncedBalance command, and explain the reasons.

In the getHistory command, the executed_list printed out should start from order_counter - executed_list.size meanwhile there is no order number for the outstanding_collection. Ex:

executed_list=(“deposit NOK 100”, “addInterest 10”);

outstanding_collection=(“deposit NOK -50”, “addInterest NOK 5”);

order_counter=5;

getHistory result:

executed_list

3. <timestamp 1> deposit NOK 100

4. <timestamp 2> addInterest NOK 10

outstanding_collection

```
deposit NOK -50

addInterest NOK 5
```

If the bank server 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 (which do not fail) go through the same sequence of changes and must end up with the same balance value**. The execution should satisfy sequential consistency with respect to the deposit, addInterest, getHistory, cleanHistory, memberInfo and balance operations.

Note1: balance of the account can be negative.

Note2: no need to consider the happened-before relationship of every single command between bank servers. Only consider the consistent view.

Note3: both Message Delivery and Bank are servers, meaning that the connection between them are bi-directional (Message Delivery server can call APIs from Bank servers and vice versa)

Deliverables:

Via the Devilry system.

Must have:

- A compressed file (zip) containing all your source code and documents:
  - The source code must be well commented.
  - The documentation can be a simple help-me file explaining how to run your application, and explaining the differences between the two requested implementations for getSyncedBalance command. Additionally, your documentation must explain how you distributed the workload among your group.
  - Output Results (the output log from replica 1 – with executed timestamp in each line)
  - Ready to deploy jar file

Note: The Input file provided is just an example. The final Input Files for grading will be posted on Wednesday 8th Oct 2025.

Please Note:

- All submitted files will be checked for code plagiarism!

Submission Deadline: 23:59 on Thursday 9th, 2025