The Second Mandatory Programming Assignment

Replicated Bank Account

IN5020/IN9020 Autumn 2023

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]
```

<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. 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 (if ifi machines are used) 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.

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 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 client should join a group whose name is <account name>.
5. 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.
6. 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. **getQuickBalance**
   This command causes the client to check and print the balance on the account right away, without synchronizing with any previously issued transactions.

2. **getSyncedBalance**
   This command causes the client to print the synchronized state of the account after applying all of the outstanding transactions in **outstanding_collection**.

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

4. **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.

5. **getHistory**
   This command causes the client to print the list of recent transactions (deposit and addInterest operations), 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 client 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 client to do nothing for <duration> seconds. It is **only useful in a batch file**.

10. **exit**
    This command causes the client to exit. Alternatively, the client 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 500")

Transaction.unique_id = "<Client_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 client 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 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 client have been applied. Therefore, getSyncedBalance offers stronger consistency compared to getQuickBalance, at the expense of longer delays. In the example above, getSyncedBalance 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, in addition to the possibility of deadlock because of never having an empty outstanding_collection, there will be the possibility of getting a wrong response. Let us consider another example with the following order of commands:

```
deposit 100

getSyncedBalance

deposit 50
```

In this example, the naive implementation may produce a wrong answer for the getSyncedBalance command. Why?

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 100", "addInterest 10");
```

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

order_counter=5;
```

getHistory result:

executed_list

```
3. deposit 100

4. addInterest 10
```

outstanding_collection

```
deposit -50

addInterest 5
```

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, getHistory, cleanHistory, memberInfo and balance operations.

Note that: balance of the account can be negative.

Spread at UiO:

Running Spread at IFI:

Try ssh to linux.ifi.uio.no with your username, and run the Spread daemon from linux and connect to it from your client.

Development Tools:

The Spread toolkit for group communication.

1. http://www.spread.org/

In case of the following error: “fatal error: 'spu_system.h' file not found”, contact the TA for instructions!

2. 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, and explaining the differences between the two requested implementations for getSyncedBalance command. Additionally, your documentation should explain how you distributed the workload among your group.
  - Output Results
  - Ready to deploy jar file

On the day of submission or no later than 3 days after deadline:

Present your assignment to the TA by explaining your design.

Please Note:

- All submitted files will be checked for code plagiarism!

Submission Deadline: 23:59 on October 13th, 2023