

Project 2: Taking Global Snapshots of a Distributed Banking Application

Due: November 3rd, 2016 23:59:59pm

In this project, you will capture global snapshots for a distributed banking application. As in Project 1, you will implement this service using the Apache Thrift framework. I have provided a file, `bank.thrift`, that defines several basic operations that can be conducted on branches of a distributed bank.

This project is worth **10%** of your total score. You **must** work **individually** on this project and use **C/C++, Java, or Python** to implement this project.

The project consists of two parts, which are described below.

1 A Distributed Banking Application

In the first part, you will implement a distributed banking application. The distributed bank has multiple branches. Every branch knows about all other branches. Each branch starts with some initial balance. It then randomly selects a branch and send a random amount of money to this branch at unpredictable times.

You must implement the following functions for a branch in a distributed bank:

initBranch this method has two input parameters: the initial balance of a branch and a list of all branches in the distributed bank. Upon receiving this method, a branch will set its initial balance and record the list of all branches.

transferMoney given a `TransferMessage` structure that contains the sending branch's `BranchID`, the amount of money in the message, as well as a `messageId` (more on this later), the receiving branch updates its balance accordingly.

A branch can be both a sender and a receiver. Therefore, you need to implement a method that will be called from a separate thread in your branch to perform the sending operation. Intervals between consecutive sending operations should be drawn uniformly at random between 0 and 5 seconds. A sender can only send positive amount of money. It needs to first decrease its balance, then call the `transferMoney` method on a remote branch. A branch's balance should not become negative. For simplicity, the amount of money should be drawn randomly between 1% and 5% of the branch's initial balance and can only be an integer.

The branch's main program should spawn both a *multi-threaded Thrift server* and also the *thread that sends money from your branch to other branches*. Because the balance of your branch will be read and written by multiple threads, **this variable need to be protected by a mutex or another synchronization method**. In addition, you can assume that neither the branches nor the communication channels will fail. So you don't need to implement anything to guarantee fault tolerant transactions.

Your branch executable should take two command line inputs. The first one is a human-readable name of the branch, e.g., "branch1". The second one specifies the port number the branch runs on.

```
$> ./branch branch1 9090
```

1.1 Controller

In this project, we rely on a controller to set a branch's initial balance and notify every branch of all other branches in the distributed bank. This controller takes two command line inputs: the total amount of money in the distributed bank and a local file that stores the names, IP addresses, and port numbers of all branches.

An example of how the controller program should operate is provided below:

```
$> ./controller 4000 branches.txt
```

The file (`branches.txt`) should contain a list of names, IP addresses, and ports, in the format “<name> <public-ip-address> <port>”, of all of the running branches.

For example, if four branches with names: “branch1”, “branch2”, “branch3”, and “branch4” are running on `remote01.cs.binghamton.edu` port 9090, 9091, 9092, and 9093, then `branches.txt` should contain:

```
branch1 128.226.180.163 9090
branch2 128.226.180.163 9091
branch3 128.226.180.163 9092
branch4 128.226.180.163 9093
```

The controller will distribute the total amount of money evenly among all branches, e.g., in the example above, every branch will receive \$1,000 initial balance. The controller initiates all branches by individually calling the `initBranch` method described above. Note that the initial balance must be integer.

2 Taking Global Snapshots of the Bank

In the second part, you will use the Chandy-Lamport global snapshot algorithm take global snapshots of your bank. In case of the distributed bank, a global snapshot will consist of the local state of each branch (i.e., its balance) and the amount of money in transit on all communication channels. Each branch will be responsible for recording and reporting its own local state (balance) as well as the total money in transit on each of its incoming channels.

For simplicity, in this project, the controller will contact one of the branches to initiate the global snapshot. It does so by making a `initSnapshot` call to the selected branch. The selected branch will then initiate the snapshot by first recording its own local state and send out `Marker` messages to all other branches. After some time (long enough for the snapshot algorithm to finish), the controller makes `retrieveSnapshot` calls to all branches to retrieve their recorded local and channel states. **Note that if the snapshot is correct, the total amount of money in all branches and in transit should equal to the command line argument given to the controller.**

You need to implement the following three functions to add snapshot capability to your distributed bank.

initSnapshot upon receiving this call, a branch records its own local state (balance) and sends out `Marker` messages to all other branches by calling the `Marker` method on them. To identify multiple snapshots, the controller passes in a `snapshotId` to this call, and all the marker messages should include this `snapshotId`.

Marker given the sending branch's `BranchID`, `snapshotId`, and `messageId`, the receiving branch does the following:

1. if this is the first `Marker` message with the `snapshotId`, the receiving branch records its own local state (balance), records the state of the incoming channel from the sender to itself as empty, immediately starts recording on other incoming channels, and sends out `Marker` messages to all of its outgoing channels (i.e., all branches except itself).
2. otherwise, the receiving branch records the state of the incoming channel as the sequence of money transfers that arrived between when it recorded its local state and when it received the `Marker`.

retrieveSnapshot given the `snapshotId` that uniquely identifies a snapshot, a branch retrieves its recorded local and channel states and return them to the caller (i.e., the controller).

The controller should be fully automated. It periodically calls the `initSnapshot` method with monotonically increasing `snapshotId` on a randomly selected branch and **outputs to the screen the aggregated global snapshot retrieved from all branches in a human-readable manner**. In addition, the snapshot taken by branches needs to be identified by their names: e.g., “branch1” to represent branch1’s local state, and “branch2->branch1” to represent the channel state.

2.1 FIFO message delivery

The correctness of the Chandy-Lamport snapshot algorithm relies on FIFO message delivery of all communication channels among all branches (processes). A communication channel is a one way connection between two branches. For example, in this project, from “branch1” to “branch2” is one communication channel. From “branch2” to “branch1” is another channel. In order to ensure FIFO message delivery, we can include a monotonically increasing `messageId` in all messages transferred in a channel.

For instance, suppose that “branch2” receives a `transferMoney` message or a `Marker` message from “branch1” with `messageId` 5, but the last `messageId` that it observed from “branch1” was 3. In this case, “branch2” would be able to infer that it is missing the message with `messageId` 4 from “branch1”. When “branch2” sees that it is missing a message, it will pause its processing of the message with `messageId` 5 and wait for the message with `messageId` 4 to first arrive and then be processed.

The sender branch must maintain a `last used messageId` for **each** of its outgoing communication channels. In this project, two types of messages are transferred among branches (processes): `transferMoney` and `Marker` messages. Before sending a `transferMoney` or `Marker` message, the sender must increment the `last used messageId` of appropriate outgoing channel and include this `messageId` in the request. In addition, since each branch sends `transferMoney` and `Marker` messages in different threads, you must also use a **mutex** to protect each `last used messageId` variable for each outgoing communication channel.

For each incoming channel, a `last seen messageId` should be maintained at each branch. On receipt of a new message, you can use a **condition variable** to wait for all prior messages of this incoming channel with lower `messageId` values to be processed. After completing message processing on the channel, you should increment the `last seen messageId`.

3 How to submit

To submit the project, you should first create a directory whose name is “your BU email ID”-project2. For example, if your email ID is `jdoue@binghamton.edu`, you should create a directory called `jdoue-project2`. You should put the following files into this directory:

1. Your source code.
2. A `Makefile` to compile your source code into two executables, which should be named `branch` and `controller`. (It is okay if these executables are bash scripts that call the Java interpreter, as long as the command line arguments follow the format described in Part 1.)
3. A `Readme` file describing how to compile and run your code on `remote.cs.binghamton.edu` computers, the programming language you are using, a brief description of your implementation, and sample input/output.

Compress the directory (e.g., `tar czvf jdoue-project2.tgz jdoue-project2`) and submit the tarball (in `tar.gz` or `tgz` format) to Blackboard. Again, if your email ID is `jdoue@binghamton.edu`, you should

name your submission: `jdoo-project2.tar.gz` or `jdoo-project2.tgz`. Failure to use the right naming scheme will result in a 5% point deduction.

Your project will be graded on the CS Department REMOTE computers `remote.cs.binghamton.edu`. If you use external libraries that do not exist on the REMOTE computers in your code, you should also include them in your directory and correctly set the environment variables using absolute path in the Makefile. If your code does not compile on or cannot correctly run on the CS computers, you will receive zero points.

Your project must be your original work. We will use MOSS¹ to detect plagiarism in the projects.

¹<https://theory.stanford.edu/~aiken/moss/>