Programming Assignment 2: The Snapshot Algorithm

Due: April 10th, 2018 23:59:59pm

In this assignment, you will implement the Chandy-Lamport snapshot algorithm we learned in Lecture 8. Unlike assignment 1, we will not use an existing RPC framework. Instead, we will use Google's protocol buffer for marshalling and unmarshalling messages and use TCP sockets for sending and receiving these messages.

This assignment is worth **13.3%** of your total score. You **must** work **by yourself** on this assignment and use **Python, C/C++, or Java** to implement this programming assignment.

The assignment consists of three parts, which are described below.

1 Protocol Buffer

I have provided a file, `bank.proto`, that defines the messages to be transmitted among processes in protocol buffer. You can use the protocol compiler, `protoc`, to compile this file and use the auto-generated code for marshalling and unmarshalling messages.

To use `protoc` installed on CS department computers, you need to first set the environment variable PATH. For example:

```
$> bash
$> export PATH=/home/vchaska1/protobuf/bin:$PATH
$> protoc --python_out=./ bank.proto
```

For your convenience, we have prepared example code, adapted from the protobuf tutorial[1], in C++, Java, and Python. You can find them at `https://github.com/vipulchaskar/protobuf-tutorial`. You should read the example code in the language of your choice (one of Java, C++, and Python) before beginning work on the remainder of the assignment.

2 A Distributed Banking Application

You will first implement a distributed banking application. The distributed bank has multiple branches. Every branch knows about all other branches. TCP connections are setup between all pairs of branches. Each branch starts with an initial balance. The branch then randomly selects another destination branch and sends a random amount of money to this destination branch at unpredictable times.

Each branch must handle the following two types of messages in a distributed bank:

InitBranch this messages contains two pieces of information: the initial balance of a branch and a list of all branches (including itself) in the distributed bank. Upon receiving this message, a branch will set its initial balance and record the list of all branches.

Transfer this message indicates that a remote, source branch is transferring money to the current, target branch. The message contains an integer representing the amount of money being transferred and the remote, source branch's name. The branch receiving the message should increase its balance by the amount of money indicated in the message.

[1] `https://developers.google.com/protocol-buffers/`

Every branch is both a sender and a receiver. A sender can only send positive amount of money. It needs to first decrease its balance, then send out a message containing the amount of money to 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. Intervals between consecutive sending operations should be drawn uniformly at random between 0 and 5 seconds.

If you choose to implement the sender and receiver in different threads, you have to protect the balance of your branch using **a mutex or another synchronization method**. In addition, you can assume that neither the branches nor the communication channels will fail.

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
```

It is expected that your branch executable will start a new branch called "branch1" which listens on port 9090 for incoming TCP connections.

2.1 Controller

In this assignment, we rely on a controller to set a branch's initial balance and notify every branch of all 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.

3 Taking Global Snapshots of the Bank

In this 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 contain both 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 assignment, **the controller will contact one of the branches to initiate the global snapshot**. It does so by sending a message indicating the `InitSnapshot` operation 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 sends `RetrieveSnapshot` messages to all branches to retrieve their recorded local and channel states.

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.

Each branch needs to support the following four types of messages to add snapshot capability to your distributed bank.

InitSnapshot upon receiving this message, a branch records its own local state (balance) and sends out `Marker` messages to all other branches. To identify multiple snapshots, the controller includes a `snapshot_id` to this initSnapshot message, and all the Marker messages should include this `snapshot_id` as well.

Marker every Marker message includes a `snapshot_id` and the sending branch's name `branch_name`. Upon receiving this message, the receiving branch does the following:

1. If this is the first `Marker` message with the `snapshot_id` the receiving branch has seen, 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).

   Note that to ensure the correctness of the algorithm, it is important that no `Transfer` messages can be sent out until all the necessary `Marker` messages have been sent out.
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 the controller sends retrieveSnapshot messages to all branches to collect snapshots. This message will contain the `snapshot_id` that uniquely identifies a snapshot. A receiving branch should its recorded local and channel states and return them to the caller (i.e., the controller) by sending a returnSnapshot message (next).

ReturnSnapshot a branch returns the controller its captured local snapshot in this message. This message should include the snapshot_id, captured local state, as well as all incoming channel states.

The controller should be fully automated. It periodically sends the `InitSnapshot` message with monotonically increasing `snapshot_id` on a randomly selected branch and **outputs to the console the aggregated global snapshot retrieved from all branches in the correct format**. 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. Here is an example controller output:

```
snapshot_id: 10
branch1: 1000, branch2->branch1: 10, branch3->branch1: 0
branch2: 1000, branch1->branch2: 0, branch3->branch2: 15
branch3: 960, branch->branch3: 15, branch2->branch3: 0
```

3.1 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 assignment, from "branch1" to "branch2" is one communication channel. From "branch2" to "branch1" is another channel.

In order to ensure FIFO message delivery, in this assignment, we use TCP as the transport layer protocol for branch communications – both banking messages described in Part 2 and snap-shot related messages described in Part 3. TCP ensures reliability and FIFO message delivery. Because TCP is full duplex, allowing messages to transmit in both directions, there are two ways to setup branch communications:

1. We can use TCP in the half duplex manner, setting up **two TCP connections between every pair of branches**. In this way, suppose there exists 4 branches, we will setup a total of 12 TCP connections, with each branch handling 3 incoming connections and 3 outgoing connections.

   If you choose to implement the assignment in this way, you need to make sure that you do not mix up the use of these connections. For example, if a connection is designated to be an incoming connection for branch1, then branch1 should never use this connection for sending outgoing messages. Otherwise, the FIFO nature of communication channels will be violated.

2. Or, we can take advantage of the full duplex nature of TCP and have each branch should set up **exactly one TCP connection** with every other branches in the distributed bank. In this way, given 4 branches, we will set up a total of 6 TCP connections.

You can use either design. Bottom line is the FIFO property of communication channels can never be voilated.

4 How to submit

To submit the assignment, you should first create a directory whose name is "your BU email ID"`-p2`. For example, if your email ID is `jdoe@binghamton.edu`, you should create a directory called `jdoe-p2`. 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 2.)

3. A `Readme` file briefly 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, sample input/output, and any other information you want us to be aware of while grading.

4. A `STATEMENT` file, containing the following statement followed by the student's full name:

   "I have done this assignment completely on my own. I have not copied it, nor have I given my solution to anyone else. I understand that if I am involved in plagiarism or cheating I will have to sign an official form that I have cheated and that this form will be stored in my official university record. I also understand that I will receive a grade of **0** for the involved assignment and my grade will be reduced by one level (e.g., from A to A- or from B+ to B) for my first offense, and that I will receive a grade of **"F" for the course** for any additional offense of any kind."

Compress the directory (e.g., `tar czvf jdoe-p2.tgz jdoe-p2`) and submit the tarball (in `tar.gz` or `tgz` format) to myCourses. Again, if your email ID is `jdoe@binghamton.edu`, you should name your submission: `jdoe-p2.tar.gz` or `jdoe-p2.tgz`. Failure to use the right naming scheme will result in a 5% point deduction.

Your assignment 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.

Your assignment must be your original work. We will use MOSS[2] to detect plagiarism in the projects.

[2] `https://theory.stanford.edu/~aiken/moss/`