Programming Assignment 3: The Snapshot Algorithm

Due: November 7th, 2019 23:59:59pm

This assignment is worth **11%** of your total score in this course. In this assignment, you will implement the Chandy-Lamport snapshot algorithm we learned in Lecture 8. Unlike assignment 2, 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 is a **group assignment**. Each group should have **exactly two** students. You can choose to work by yourself on this assignment, though no extra credit will be given for working individually.

Once you have formed a group, at least one member in the group **must send the instructor an email listing the names and email addresses of both members of your group by October 31, 2019** by the end of day. If we do not receive your email by this deadline, we will record that you will be working individually. No groups are allowed to be formed after the October 31 deadline.

1 Protocol Buffer Tutorial

For your convenience, we have prepared example code, adapted from the protobuf tutorial[1], in Python, Java, and C++. You can find them at `https://github.com/Yao-Liu-CS457-CS557/protobuf-tutorial`. You should read and try the example code in the language of your choice (one of Python, Java, and C++) before beginning work on the remainder of the assignment.

2 bank.proto

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
$> export PATH=/home/yaoliu/src_code/local/bin:$PATH
$> protoc --python_out=./ bank.proto
```

3 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:

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

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 source branch's name. The branch receiving the message should increase its balance by the amount of money indicated in the message.

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 `t milliseconds`, where `t` is a command line input parameter. It is expected that your program works correctly when `t` is set to any number between 1 and 5000.

If the sender and receiver are run 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 **three** 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. The third input is the maximum interval, in milliseconds, between Transfer messages. For example,

```
$> ./branch branch1 9090 100
```

It is expected that your branch executable will start a new branch called "branch1" which listens on port 9090 for incoming TCP connections. The interval between consecutive transfer operations should be drawn uniformly at random between 0 and 100 milliseconds.

3.1 Command line outputs

When `t` is set to a value greater than or equal to 1000, i.e., Transfer messages are sent less frequently, the branch executable should output meaningful messages, logging the sending and receiving of Transfer messages. It should also output the updated bank balance.

When `t` is smaller than 1000, logging of message transmission and balance should be disabled.

3.2 Controller

Branches rely on a controller to set their initial balances and get notified 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 128.226.114.201 9090
branch2 128.226.114.201 9091
branch3 128.226.114.201 9092
branch4 128.226.114.201 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. You can assume the total balance can be divided by the number of branches with zero remainder.

4 Global Snapshots for Bank Audit

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.

To add snapshot capability to your distributed bank, each branch needs to support the following four types of messages:

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 `send_branch`. 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 return its recorded local and channel states 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 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
```

5 A Few Helpful Notes

5.1 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 3 and snapshot related messages described in Part 4. 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 exist 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.

5.2 Delimiting protobuf messages

Messages a branch process has to handle, e.g., Marker and Transfer, are of different lengths. However, the receiving process does not know what type message is arriving beforehand, and thus does not know how many bytes to expect when reading incoming messages from the socket.

One way to address this is by sending and receiving "delimited" messages - writing and reading the size of the message first then the message itself. You will find this especially useful when increasing the Transfer message frequency to e.g., 1 ms.

In Java, you can use `writeDelimitedTo` and `parseDelimitedFrom`; In C++, you can use functions described in `delimited_message_util.h`[2]. In Python, you have to write the delimiters yourself. Here is an example how to encode/decode a Python integer into/from protobuf varint encoded messages[3].

6 Demo

After the submission deadline, every group will sign up for a demon time slot with our grader, Zheng Li. All group members have to be present during the demo. You will show the capabilities of transferring money and taking global snapshots that your group has implemented. A demo procedure will be distributed before the scheduled demo time.

7 Github classroom

Once you have formed your group, you can access the assignment by first logging into your Github account created with your BU email. Then go to: `https://classroom.github.com/g/SbnEqo4V`.

Github classroom will ask you to create a team name. Please use a team name with both students' BU email IDs. For example, if a group's two students' email IDs are `jdoe@binghamton.edu` and `jsmith@binghamton.edu`, you should create a team name called `jdoe-jsmith`.

Github classroom will automatically create a repository (e.g., if your team name is jdoe-jsmith, the repository will be named cs457-557-f19-pa3-jdoe-jsmith) and import starter code (`bank.proto`) into the repository. This is the repository you will push your code to.

If your partner has already created a repository, you can select your team from a list of existing teams and join it.

The repository created by Github classroom is a private repository. Only the group members, course instructor, and the grader are able to see this repository. Follow the instruction on the Github page to create a new repository on your local directory and link it to the Github repository. Note that `git` is already installed on CS department computers. To use it on your own computer, you must install it first.

To add a file to the next commit, use the `git add` command. To commit your code, use the `git commit` command. Be sure to also push your commit to the Github repository after every commit using the `git push` command (e.g., `git push origin master`).

We expect each repository to have **at least three commits**, with the first one and the last one **more than 48 hours apart**. **Submissions that do not meet the three commits / 48 hours requirement will not be accepted or graded.**

8 How to submit

To submit, make a final commit with the message "final commit", and push your local repository to the private Github repository Github classroom created. Your final commit should contain the following files:

1. Your source code.
2. A `Makefile` to compile your source code into executables.
3. A `Readme` file describing:
    - how to compile and run your code on `remote.cs.binghamton.edu` computers,

[2] `https://github.com/protocolbuffers/protobuf/blob/master/src/google/protobuf/util/delimited_message_util.h`

[3] `https://krpc.github.io/krpc/communication-protocols/tcpip.html`

- the tasks both group members worked on in this assignment,
- completion status of the assignment, e.g., what has been implemented and tested, what has not,
- anything else you want the grader to be aware of while grading your assignment.

4. Two `STATEMENT` files, signed by each group member individually, containing the following statement:

   "I have done this assignment completely on my own and in collaboration with my partner. I have not copied my portion of the assignment, nor have I given the project 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."

After pushing your final commit to the Github repository, please let us know by **submitting your commit hash to myCourses**.

There are two ways to locate your commit hash: You can type `git log` in your local repository, which will output the commit history. Locate the commit you want to use as your final submission, record the hash associated with the commit. The SHA1 hash code used by git should be 40 characters long. For example, the commit in the starter code has a hash value of "0e23b3f2144e4b1540aab67962139bcce0fb602a". Alternatively, you can also go to the Github page of your repository and locate it on the webpage.

It is important that you submit your commit hash to myCourses. This helps us know your submission is ready for grading and which of your commits we should grade. We will not grade your assignment unless you have submitted the commit hash to myCourses before the assignment submission deadline

Your assignment will be graded on the CS Department computers `remote.cs.binghamton.edu`. It is your responsibility to make sure that your code compiles and runs correctly on CS Department computers.