Project 1: Consistent Distributed Snapshot

- Antony Yun (EID: any283, CSID: antony)
- Justin Wei (Auditing)

Implementation Details

This project implements the Chandy-Lamport distributed snapshot algorithm applied to the toy problem of a distributed banking system.

The main restriction on our system was that each process can only communicate through a series of channels and a select few bookkeeping commands between the master process. In order to fulfill this requirement, we decided to use Python multiprocessing to spawn multiple processes. We considered using multiprocessing's abstractions for message passing, but found it difficult to pass messages between arbitrary child processes, so we settled on Linux FIFOs (named pipes).

Our implementation faithfully mirrors the specification, so we won't describe the documented logic in detail. However, the following is an overview of how we specifically approached the problem:

- We have separate classes defined for the master, observer, and node in `master.py`, `observer.py`, and `node.py` respectively.
- Invoking the program creates the master, which is responsible for creating a new process for the observer and for each node.
- The master creates pipes between each pair of processes in the system. When a new node is created, it sends a special message outside of the spec to update each other process's membership list.
- The master forwards all of the commands listed in the spec to the appropriate process. This is done by sending `pickle`'d strings through the `master-{process}` pipe. The master blocks by waiting for an `ack` message on each command. We use a simplified version of the API in the spec for these messages.
- The master introduces almost no logic for the snapshot algorithm, with the following exceptions:
  - `ReceiveAll` - master sends `ReceiveAll` messages round robin to each node until all nodes report empty channels.
  - `BeginSnapshot` - master first tells the observer to send a snapshot message, then issues a `Receive observer` command to the target node.
  - `CollectState` - master first tells the observer to initiate collecting state. The master and observe then proceed in lock step: the observer send a message to a node and ack's the master, the master sends a `Receive` to that node and waits for the node to ack, the observer blocks until the node replies, and this repeats for each node.

- All channels between nodes are non-blocking, as prescribed in the specification.

Chandy-Lamport Algorithm

Our implementation of the Chandy-Lamport Algorithm operates as follows:

- The observer sends a snapshot token (SS) to the starting node.
- Whenever a node receives a SS, it atomically performs one of the following steps:
  - If it has yet to receive another SS:
    - The node records its balance as its current state.
    - The node broadcasts a SS to all other nodes in the system.
  - If the node is already in recording mode:
    - The node records the state of the channel as the sum of all messages received since entering recording mode.

Usage

Our implementation requires Python >= 3.6. Python should be invoked with the `-u` flag to disable output buffering when piping stdout to other commands.

We ran each test using the following command:

```
cat tests/0.test | python3 -u master.py | diff tests/1.out -
```

Tests

Our code passes each of the three sample test cases provided in the spec (`0.test`, `1.test`, `2.test`). We added one additional test in `3.test`. This test is pretty simple, but overall it tests a few new possible interleavings of normal messages with snapshot messages (using 3 nodes instead of 2), and it also tests taking consecutive snapshots. This test helped us identify a last minute bugs with resetting snapshot state, and with `ReceiveAll` only flushing each node once, rather than until they're all empty.