# Project 1: Consistent Distributed Snapshot

## 1. Introduction

In this project, you will implement a crude distributed banking system. It will provide the functions to transfer the money between all participating nodes and to take a global snapshot. You will build an observer that makes use of Chandy-Lamport global snapshot algorithm to take a consistent-global snapshot of all participating nodes which contains both the individual state of each node and the individual state of each communication channel.

### 1.1 Logistics

This project is due on March 11. You should work in groups of 2-3 people.

## 2. Technical Details

The implementation can be written in any language or languages you like. The two requirements are that any number of nodes can be joined to the system and all processes should be able to run concurrently. Other than that, the rest is up to you.

## 3. Correctness

Part of your grade will be a correctness component. We will benchmark your implementation and you will receive points based on whether the results of a global snapshot produced by your system are correct or not. This section is worth 70 points out of the total 100. Points will be awarded as follows:

- 35 points for correctly capturing the individual state of each node
- 35 points for correctly capturing the  individual state of each channel

We will use the median of five runs using different testing cases.

## 4. Grading

There will be a total of 100 points for this assignment. The split-up will be as follows:
10 points - Submission compiles, and master correctly responds to all commands
70 points - correctness
20 points - description of what you have done, how you implemented the Chandy-Lamport algorithm, etc.

# 5. Implementation & Testing

In implementing the system, there are a few assumptions and requirements you should make sure.

Assumptions

- There are 1 master, 1 observer, and N participating nodes in the system.
- There are two unidirectional communication channels between each ordered node pair (Ni → Nj, Nj → Ni).
- The messages in channels are FIFO ordered, but there's no ordering guarantee about the messages between channels.
- There is no failure in participating nodes.
- All messages are intactly delivered without being duplicated.
- There are two kinds of messages: SnapshotToken, or Transfer. SnapshotToken is used in the Chandy-Lamport algorithm, while Transfer is used for transferring money between nodes.

Requirements

- The local tasks in a node should not be interrupted by a snapshot request.
- Each node must be able to locally store its own state.
- The global state is collected in a distributed manner.
- Any participating node can start the snapshot process.

## 5.1 The Master Program

To help you test your implementation and assist with the evaluation, each submission must include a master program which will provide a programmatic interface with the distributed banking system. The master program will keep track of and will also send command messages to all nodes and an observer. More specifically, the master program will read a sequence of newline delineated commands from standard input ending with EOF which will interact with the distributed banking system and, when instructed to, will display output from the playlist to standard out. We will be using this master program to test your implementation, not to trick you up on API edge cases. Your Makefile for the project should compile this to an executable named "master". If your implementation does not require a Makefile, turn in an empty file. The API for the master program is defined in Section 5.2. All ids mentioned in the API are in the same namespace; for example, if there are 5 participating nodes, the ids 0-4 would be handed out.

## 5.2 Master Program API Specification

| Command | Summary |
|---|---|
| **StartMaster** | This command creates a master node and an observer while making them connected to each other. The master node receives every request from a user and intercepts it to an associated node. It keeps track of the global state history of each node in the system. |
| **KillAll** | This command will kill all processes involved in the system including a master, an observer, and all participating nodes. An impatient user, change of plans, or unforeseen bug in your code may cause some number of processes in your system to crash while leaving others up and running. To allow us to move smoothly between different test cases, please provide this command to clean up such orphaned processes. |
| **CreateNode** [node id] [initial amount of money] | This command creates a node and its channels. It will connect this node to all other nodes in the system, including a master node and an observer. Messages in each channel are FIFO ordered but, there's no ordering guarantee between channels. The newly created node is initialized to have a specified amount of money. |
| **Send** [sender node id] [receiver node id] [Amount of money] | This command will tell a node to send the specified message (amount of money) to a receiver node. This command should block until the sender's message is enqueued to its outgoing channel associated with the specified receiver. If receiving the requests sending the amount of money over the current budget, a sender node should ignore that message and should print out the error message described below. |
| **Receive** [receiver node id] [sender node id] (Note that the sender node id is optional.) | This command will tell a node to retrieve the head-of-queue message from a randomly picked incoming channel, or from the one |

| | |
|---|---|
| | associated with the specified sender. This command should block until the receiver starts the associated computation. Except for messages from the master, nodes are not allowed to receive enqueued messages without this command in order to simulate message delay and reordering between channels. This command will print out which message a node received in the format described below in 5.3. |
| **ReceiveAll** | This command randomly picks one of the channels that are not empty and makes the corresponding node retrieve a message from the head of the queue. These procedures are performed repeatedly and should be blocked until all channels become empty.<br><br>This is equivalent to the Master repeatedly calling Receive on all nodes with non-empty incoming channels.<br><br>The purpose of RecieveAll is to drain channels of messages caused by the Chandy-Lamport algorithm. |
| **BeginSnapshot** [node id] | This command will tell an observer to send the message “Take a snapshot” to the specified node. Once finishing enqueuing the snapshot message, it should immediately return. It should print out the node id receiving the snapshot message in the format described below in 5.3. |
| **CollectState** | This command will tell an observer to collect an individual state of each node and an individual state of each channel. This command should block until the observer finishes the associated computation. |
| **PrintSnapshot** | This command will print out the collected global state in the format described below in 5.3. |

## 5.3 Print Format

For a “Send” command, if the node doesn’t have enough budget for sending the requested amount of money, it should print out “ERR_SEND”.

The “Receive” command instructs the process to print out which message the receiver node retrieved. This should be printed in the format:

[sender node id] [message (“Transfer” or “SnapshotToken”)] [amount of money]

If the type of message is the token message for a snapshot, the amount of money should be printed in “-1”.

The “BeginSnapshot” command instructs the process to print out the node id retrieving the snapshot message. This should be printed in the format:

Started by Node [id]

The “PrintSnapshot” command instructs the process to print out the collected global state including the individual state of each node and each channel. This should be printed in the format:

---Node states
node 1 = <state>
node 2 = <state>
...
---Channel states
channel (1 → 2) = <state>
channel (2 → 1) = <state>
...

**Example:**
$ ./master
StartMaster
CreateNode 1 1000
CreateNode 2 500
Send 1 2 300
Send 2 1 100
Send 1 2 400
Receive 2 1
1 Transfer 300
BeginSnapshot 1
Started by Node 1
Receive 2
1 Transfer 400
Receive 2

1 SnapshotToken -1
ReceiveAll
CollectState
PrintSnapshot
---Node states
node 1 = 300
node 2 = 1100
---Channel states
channel (1 → 2) = 0
channel (2 → 1) = 100
Send 1 2 5000
ERR_SEND

# 6. What to Turn In

One member of your group should submit everything to Canvas as a zip file.

1. Source code for your implementation (master, observer, and nodes)
2. A Makefile that compiles your implementation on the CS machines (can be empty)
3. A README file that details your implementation, your names, UT EIDs, and UTCS ids, a description of your protocol, a description of your tests, instructions on how to use your system, and any other information you think is relevant to the grading of your submission.