# Assignment 3 (Distributed Snapshots)
# Due: ~~Sunday, March 24th~~ Monday, March 25th, 11:59 PM

## Overview

In this assignment, you will complete the implementation of the Chandy-Lamport algorithm for distributed snapshots.

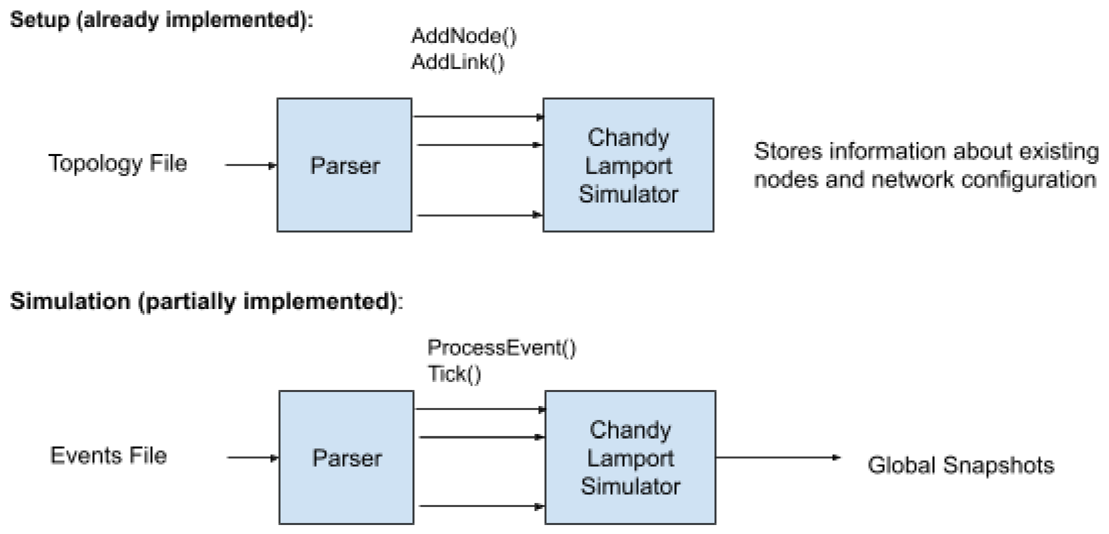
The starting code can be found in this directory. You will complete the implementations of sim.go and node.go

The code is structured as follows:
- sim.go: A discrete-event simulator that constructs a network of nodes, passes events to them and initiates/collects snapshots.
- node.go: A process in the distributed system
- common.go: This has the common snapshot, message and record types. It also has a debug flag.
- logger.go: A logger that records events executed by system (useful for debugging)
- snapshot_test.go: Test cases
- queue.go: Simple FIFO queue interface
- test_data/: Test case inputs and expected results

**Acknowledgement.** This is an *adapted version* of an assignment used in Princenton's COS418 distributed systems course.

## Detailed Description

This assignment will use the same token-based setting used in the lectures.
The following figure shows how the simulator is used at a high level. You will be asked to *complete* the simulator implementation (sim.go), as well as node.go in order for the simulation part to output consistent global snapshots.



**File Description and Examples**

| Topology File | Events file | Expected Snapshot |
|---|---|---|
| The file contains the number of nodes followed by a line for each node with the number of tokens it has, then the existing channels. | Each line in this file can be either:<br>- Token passing event, e.g., send N1 N2 3 tokens<br>- Starting a snapshot at one of the nodes (there can be more than one snapshot running at the same time)<br>- Tick event (This is to advance the clock of the simulator). Between two subsequent ticks, the events are considered concurrent. | This contains the snapshot id, followed by the number of tokens at each node, and the recorded messages.<br><br>If there are multiple snapshots requested, each snapshot will have a file. |
| 3<br>N1 10<br>N2 3<br>N3 0<br>N1 N2<br>N2 N1<br>N1 N3<br>N3 N1<br>N2 N3<br>N3 N2 | send N1 N2 3<br>send N2 N3 2<br>snapshot N2<br>tick<br>send N1 N2 2<br>tick<br>send N1 N2 1<br>tick<br>send N2 N1 1<br>tick 3<br>send N3 N2 1 | 0<br>N1 4<br>N2 1<br>N3 2<br>N1 N2 token(3)<br>N1 N2 token(2)<br>N1 N2 token(1) |

More description of the formats can be found in test_common.go, which also has the parsing logic. You will neither be responsible to handle the parsing nor the output files. You will store the snapshot data in a GlobalSnapshot struct.

**Simulator (sim.go)**

The ChandyLamportSimulator stores information about the topology of the network including the node ids and the existing channels, and passes the events injected via processEvent() from the events file to the corresponding nodes. The events from the file could be either Token Passing or Snapshot start events.

The simulator starts at time t = 0. When a tick is parsed from the events file, the clock is advanced by 1 step (or n steps if “tick n” was found). All events that happen between time t and time t + 1 are considered concurrent. Events that happen at time t strictly happen before events that happen at t + 1 or later.

When a start snapshot event is found in a file, the simulator is responsible for

- Initiating the snapshot process (by calling the corresponding method in node.go on a specific node)
- Collecting the snapshot state from the nodes after the process has terminated.

You are required to complete the implementation of three relevant methods:

- **StartSnapshot**: This method starts the snapshot process at the given node.
- **NotifyCompletedSnapshot**: This method is called by each node when a snapshot with the given id completes at the node.
- **CollectSnapshot**: This method collects and merges snapshot state from all the nodes. This function blocks until the snapshot process has completed on all nodes.

**Node (node.go)**

You are required to complete the implementation of two methods:

- **HandlePacket**: This is a callback for when a message is received on a node. This should process both token pass messages and marker messages. When the snapshot algorithm completes on this node, this function should notify the simulator by calling `sim.NotifyCompletedSnapshot`.
- **StartSnapshot**(snapshotId int): This starts a ChandyLamport snapshot algorithm on this node with the given snapshotId. This should be called once per node for a specific id.

**Notes**

1. You can add more fields in the ChandyLamportSim and Node structs as needed. You may also add your own structs and methods (but only in node.go and sim.go).
   **Hint:** Think about the state that each node needs to keep track of and what the simulator needs to keep track of in order to produce a consistent snapshot. Also, recall that there are multiple snapshots that can be running at the same time. This is what you should plan first before writing the methods above.

2. Some key data structures are defined in common.go. Going through them will help while exploring the code.

```
// The output of the Chandy Lamport algorithm
type GlobalSnapshot struct {
    id       int            // snapshot id
    tokenMap map[string]int // key = node ID, value = num tokens
    messages []*MsgSnapshot // snapshots of all messages recorded across all
links
}
```

```
// This should be used to store a message recorded in the link src->dest
during the snapshot
type MsgSnapshot struct {
    src     string
    dest    string
    message Message
}

// This type represents the content of messages exchanged between nodes during
the simulation. A wrapped version that contains the sender and the receiver
can be found in SendMsgEvent
type Message struct {
    isMarker bool // if true, the message is a marker, if false, the message
is a token transfer
    data     int  // if the message is a marker message, data carries the
snapshot id, else data carries the number of tokens transferred
}

// An event that represents the sending of a message.
// This is expected to be queued in link.msgQueue
type SendMsgEvent struct {
    src     string
    dest    string
    message Message
    // Note: The message will be received by the node at or after this time
step. This value is generated by the simulator at random. See the assignment
document for mode detail.
    receiveTime int
}
```

3. The links between the nodes deliver the SendMsgEvents in a FIFO manner. Each SendMsgEvent injected in a link between two nodes has a receiveTime that is set by the simulator at random, however, this delivery time is not exact. The message will be received by the receiving node at or after this time.

   Let's see how this works through an example.

   Suppose that a message $m_1$ was sent from A to B at time t, and another message $m_2$ was sent from A to B at time t + 1 (or later). The simulator will generate a receive time $r_1$ for $m_1$ and receive time $r_2$ for $m_2$ through this method:

```
// This method is used within SendTokens() and SendToNeighbors
() in node.go
func (sim *ChandyLamportSim) GetReceiveTime() int {
      return sim.time + 1 + rand.Intn(maxDelay)
}
```

How are the messages going to be delivered to B then?

- First, as $m_1$ was sent before $m_2$, then in all cases $m_1$ will be delivered before $m_2$ even if $r_2 < r_1$.
- At most one message will be delivered to a node at any time step.
- When will a message be delivered? To deliver $m_1$ to the recipient node, it needs to be the head of the queue and $r_1$ needs to be less than or equal to the current simulator clock. If the $m_2$ is the next in line, it will be delivered at a later step, even if it satisfies the same condition.

In short, the generated receive times are not meant to be exact. They are to introduce some random behavior during the simulation.

The above detail won't much affect the code you write for the assignment. This detail is provided to help you while reading the code and debugging, if needed.

4. The random behavior across all the program will be the same when you run the tests as the seed is fixed.

5. Some aspects in your implementation will need to be thread-safe, as snapshot *collection* from nodes runs in separate threads. See the threads in readEventFile() in test_common.go.
   It is not allowed to use sync.map in this assignment. If you want your maps to be thread safe use a proper locking mechanism.

**Testing code**

-
  ```
  go test -run . -v -count=1
  ```
  The above command runs all tests once

  You can change the command to run an individual test as you learned in assignment 2. You can also change the count value to repeat the test several times to make sure your implementation is thread-safe.

**Submission**

- The submission form will be posted to the MS team of the course.
- Only sim.go and node.go will be delivered. Don't edit any other code.