Fundamental Distributed Algorithms

(**1**) **What are the snapshot algorithms and what are they used for?**
Before we can introduce *snapshot algorithms*, it is mandatory to introduce what we mean by ***snapshot***:

- A snapshot is, as the name suggests, a "***picture***" of the state of the distributed computation. This picture comprises the state of each process along with the state of each channel. So, a distributed computation could not be completely described by only the processes' states but we need to keep track of the messages that have not yet been processed. (Do you remember that a message changes the state of a process?).

So, a snapshot algorithm is a ***control algorithm*** that just takes a snapshot of the distributed system. Will any snapshot do the work? We'd like to take ***consistent snapshots***, i.e., a snapshot that has been taken just after a *consistent cut*.

(**2**) **Which are the main algorithms for snapshots?**
There two well-known algorithm:

- ***Chandy-Lamport***
- ***Lai-Yang***

Let's present the former. The Chandy-Lamport algorithm claims to take consistent snapshots under two hypothesis:

1. The channels must be FIFO (First In First Out).
2. The graph must be strongly connected (each process p_i must be reachable from another process p_j).

Another important property of the algorithm is that any process can be an *initiator*, i.e., the initiator process doesn't need to have any special property or anything special in general.

```
def snapshot():                          # initiator
     if !marked then:
          marked = true
     foreach channel in out_channels:  # send a special message
          marker = true                # (marker) in order to let
          send(channel, marker)        # the other processes start
                                       # the algorithm
     <take local snapshot, record current channels' states>

def on_message_marker(channel c):
     if !marked then:
          snapshot()
     foreach channel in (in_channels / c):
          <wait to receive marker>
     <terminate algorithm (locally)>
```

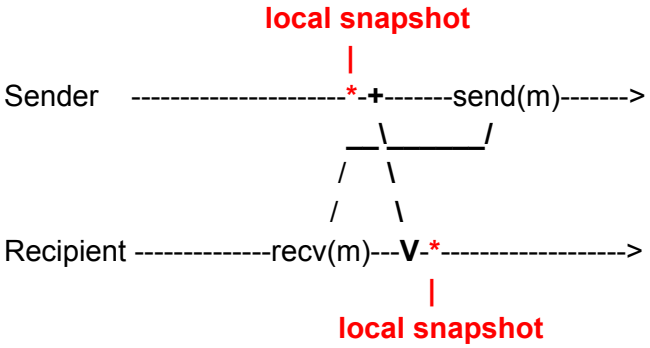
```
def on_message(channel c, msg m):
      if marked and in_channels[c].marked == false:
            in_channels[c].state.append(m)
```

Chandy-Lamport algorithm claims to take consistent snapshots. Is it for real? Let's prove it:

1. To be consistent, for each event b and a subset of events $E = \{e_1, e_2, ..., e_N\}$ such that $e_i \rightarrow b$ $(i = 1, ..., N)$, if b is pre-snapshot then e_i must be pre-snapshot as well.
2. If b and e_i are local events for a process p_j, then it is trivial.
3. Let's consider the non-trivial case in which b = `receive(m)`, and e_i = `send(m)`. If event b occurs in the recipient process p_r before p_r has seen a marker, indicating the initiation of a snapshot, then process p_s must have sent message m before taking its local snapshot. Otherwise, process p_s would have sent the marker before message m. This condition is true if and only if the channel connecting recipient and sender is FIFO, meaning that the order of messages seen by process p_s is exactly the same order of messages seen by the recipient.

The FIFO hypothesis is quite important: without FIFO channels, Chandy-Lamport algorithm could produce non-consistent snapshots. However, such a hypothesis can be quite costly to achieve (even impossible) in practice: how to overcome such a problem? Lai-Yang algorithm:

- The Lai-Yang algorithm is very similar to the Chandy-Lamport algorithm but does not need FIFO channels. However, the Lai-Yang algorithm comes with a price: overhead for each send/receive operation. In particular, Lai-Yang algorithm works just by replacing the moment in time on which a certain message will be received with respect to the moment in time on which the sender process has taken its local snapshot. Let's make an example.
    - Consider non-FIFO channels as well as the following scenario:



      With non-FIFO channels, such a scenario could be possible. receive event occurs pre-snapshot but its relative send event occurs post-snapshot.
- Wrapping each send operation with some information carrying whether the sender process has taken a snapshot, we can delay the corresponding receive in order to respect the relation `send(m) -> recv(m)`.

Lai-Yang algorithm is the following:

```
def snapshot():
      marked = true
      <take local snapshot, record channels' states>

def wrapper_send(channel out, msg m):
      if marked then:
            msg.color = red       # msg post-snapshot
      else
            msg.color = green     # msg pre-snapshot

      send(out, msg)
      log(out, msg)

def wrapper_receive(channel in, msg m):
      if msg.color == red then:  # msg is post-snapshot w.r.t.
            snapshot()           # sender point of view. It must
                                 # be post-snapshot also for
                                 # recipient process.
      receive(in, msg)
      log(in, msg)
```

---

(3) **How does the spanning tree algorithm work?**

Before we start describing how the spanning tree protocol works, we should recall some properties about spanning trees:

1. For any connected (and undirected) graph G (n nodes, E edges), exists at least one spanning tree.
2. A spanning tree for a graph G has n nodes and n - 1 edges.
3. A spanning tree is maximally acyclic, i.e., it is a graph with the maximum number of edges without a loop. Adding one edge always creates a loop.
4. A spanning tree is minimally connected, i.e.,, removing an edge the spanning tree becomes disconnected.

To build a spanning tree in a distributed way, it is sufficient that each process keeps information about its parent and a list of children. The spanning tree algorithm is based on the following idea: An initiator (which will later become the root) starts the algorithm by sending a fwd message to a list of processes. Once the fwd messages arrive at the leaves, the flow of messages inverts its direction and starts to come back. Using bkw messages, the algorithm starts to establish the relations between parents and children.

```
def start_spanning():
      parent = getpid()
      children = {}
      expected = n_neighbors
      foreach p in neighbors:
            send(p, fwd)

def on_forward(proc sender):
      if parent == null then:
            parent = sender
            children = {}
            expected = n_neighbors - 1
            if expected == 0 then:                    # dead end
                  send(sender, bkw({getpid()}))
            else
                  foreach p in neighbors/sender:
                        send(p, fwd)
      else
            send(sender, bkd(empty))

def on_backword(proc sender, msg m):
      expected -= 1:
      children.append(m.list)
      if expected == 0 then:
            my_pid = getpid()
            if parent != my_pid then:
                  send(parent, bkd(list(my_pid)))
            else
                  <termination>
```

(**4.1**) **How does mutual exclusion work in distributed systems?**

What do we mean by mutual exclusion? A shared resource can be granted to at maximum one process at any time: we need to ensure that only one process at any time is executing a ***critical section***. To model the mutual exclusion scenario, a process can be:

1. OUT_CS, a process which is not executing any operation belonging to the critical section.
2. REQ_CS, a process awaiting to be granted to start executing the critical section.
3. IN_CS, a process executing the critical section.

There are three different types of mutual exclusion algorithms:

- In ***individual permissions***, each process in the system grants permission on its own behalf.

- In ***token-based***, a process can execute the critical section only if it holds the unique token corresponding to the requested resource.
- In ***arbiter-permissions***, an external process acts as a controller for resource requests, resolving conflicts between two or more requesting processes.

Historically, any mutual exclusion algorithm adheres to a protocol:

1. Before entering the critical section, a process needs to execute the `req_access()` primitive.
2. Immediately after the process has finished to execute the critical section, it needs to execute the `rel_access()` primitive.

`req_access()` protocol is in charge of managing the request to access the CS. In a distributed scenario, such requests need to be orderdered not by a normal timestamp but using a different mechanism: logical clocks such as Lamport timestamps. Why so? Because, in order to ensure that a request will not "jump the queue", we need to ensure a precise order of arrival. Such order could be ensured by means of logical clocks such as Lamport timestamps.

Recall, Lamport timestamps have the property of "***clock consistency***":

$$e \rightarrow b \Rightarrow T_{LP}(e) < T_{LP}(b)$$

So, possible conflicts on different `req_access()` can be resolved using a logical clock and a level of privilege:

- Consider two requests coming from processes p_i , p_j. Process p_i made the request at ts_i , process p_j made the request at ts_j .
- Request (p_i, ts_i) comes before (p_j, ts_j) iff $(ts_i < ts_j) \vee (ts_i = ts_j \wedge i < j)$.

(**4.2**) **How does the Ricart-Agrawala work?**

As already said before, a mutual exclusion algorithm has to adhere to a precise protocol:

- `req_access()` and
- `rel_access()`.

```
def req_access():
      state = REQ_CS              # keep track of the process' state
      expected = n - 1            # the process needs to receive n-1
                                  # OKs to start executing CS
      req.pid = getpid()
      req.ts = my_clock + 1       # Lamport timestamp

      foreach p in process:
            send(p, req)
      <wait until #expected OKs arrives>
      state = IN_CS
      <start executing CS>
```

```
def rel_access():
      state = OUT_CS
      foreach p in queued_processes:
            send(p, “OK”)
      queued_processes = {}            # empty queued processes

def on_request_msg(proc sender, msg):
      my_clock = max(msg.ts, my_clock)
      if state == REQ_CS and priority(msg) == true then:
            queued_processes.append(sender)
      else
            send(sender, “OK”)

def on_OK(proc sender, msg):
      expected -= 1
      if expected == 0 then:
            <awake process and start executing CS>
```

`priority()` is just a function that computes the priority between two processes, according to (p_i, ts_i) comes before (p_j, ts_j) iff $(ts_i < ts_j) \vee (ts_i = ts_j \wedge i < j)$.
