EE 382N: Distributed Systems **Fall 2017**

Lecture 3: September 15

Lecturer: Vijay Garg *Scribe: Patrick Sigourney*

3.1 Global Snapshot

Imagine a looking up at the sky and a flock of flying birds. You want to take a *global snapshot* of the sky in order to capture the state of all the birds. You have a camera, but it is only able to photograph a small section of the sky in each image. So you take a bunch of pictures of different parts of the sky in order to capture everything. But as you're taking the pictures, the birds are flying around. How do you capture an accurate snapshot? There are two problems which must be overcome: 1) The double-counting of birds and 2) Missed counting of birds (birds in transit).

3.2 Chandy-Lamport Snapshot Algorithm (1985)

Developed by Leslie Lamport and K. Mani Chandry, the snapshot algorithm assumes interprocess communication occurs reliably, in FIFO (messages sent from A to B arrive in the same order they were sent), and the graph underlying the distributed processes is strongly connected.
The snapshot through the process timelines separates the timelines into "past" and "future", such that for all events which occurred in the "past" happened-before all events which occurred in the "future":

(E, $\rightarrow$) is the set of events in "happened before"
A set G $\in$ E is in a consistent global state if $\forall(e, f) : (f \in G) \wedge (e \rightarrow f) \Rightarrow e \in G$
(tl;dr: If f is in the 'past' and e happened before f, then e is also in the 'past'. This resolves the double-counting problem).

To resolve the missed-counting problem, in addition to knowing the state of each process, we must also know the state of each channel. We need a picture of the sender, the receiver, and what is in transit.

3.2.1 Red and White

The Chandy-Lamport algorithm is such:

1) All processes start as white.
2) Process A takes a local snapshot and turns red.
3) Process A sends a "marker" message on every outbound channel.
4) Process B receives the marker message, takes a local snapshot, turns red, it sends a marker on every outbound channel.
5) etc...

This is enough to capture the process states, but we also need to capture the state of each channel.

There are 4 possible message types, based on the color of the sender and receiver:
WW (white sending to white) - Nothing to do here.
RR (red sending to red) - Nothing to do here.
WR (white sending to red) - This is an "in transit" message and must be recorded.
RW (red sending to white) - Because the first thing a newly-red process does is send a marker to all other directly-connected processes, and because messages are assumed to be FIFO, there should NEVER be a message sent from a red process to a white process. It can't happen.

For the WR messages: These must be recorded by the receiving process to capture the state of the channel at the time of the snapshot.

3.2.2 The Algorithm

$\underline{P_i}$:

```
var Color: white, red initially white
var Closed[k]: array of bools, true once marker has been received from process k
var Channel[k]: array of linked lists of messages received from white process k

def turnRed: if Color == white
    record local state
    Color := red
    Send marker on all outgoing channels

def recvMarker from k:
    if color = white then: turnRed
    closed[k] := true

def recvMessage from k:
    if color == red && closed[k] == false:
            Channel[k].append(message)
```

P_i is done when Color = red and ($\forall$ incoming channels c: closed[c] = True)

Message complexity:
Number of overhead messages = number of channels in the graph
n nodes in completely connected graph = (n^2) messages

If FIFO is not assumed, the marker can also include the number of messages sent while white as a way for the receiver to determine whether all white messages have been received.