Chapter 9 – Global Snapshots

* Global Snapshot Algorithms: captures the global state of a distributed system by comparing stable properties in an old snapshot and the current state (impossible to keep track of the **current** state of a distributed system).
* Global state (happened-before model): set of local states such that no two states share a happened-before relationship.
  + Preferred over the time-based model
    - Impossible to determine whether a global state occurs in time-based model without access to perfectly synchronized local clocks
    - Accounts for different execution schedules, allowing for simplification of understanding program properties of more interests (i.e. in time-based model, an execution schedule which doesn’t violate mutual exclusion may violate mutual exclusion in another type of execution schedule)
* **Consistent Global State (state-based distributed computation):** For all processes **i, j** : **G[i] || G[j]** where **G** is the set of local states with exactly one local state for each process.
* **Chandy and Lamport’s Algorithm:**
  + **Receiver** records the state of the channel
  + Channels are unidirectional and obey the FIFO property
  + All processes have a variable **color** which are all initially white and turn red when state is recorded.
  + Rules:
    - Recorded local states are mutually concurrent
    - Mechanism to capture local state of the channels
    - Solution: **marker** message is sent to all outgoing channels from the process being turned to red. Any process which receives the marker must also turn red. Ensures that no process receives a marker from a red marked process through FIFO and proves mutual concurrency with processes
  + **Channel States**:
    - **ww:** messages sent and received before the global snapshot is taken
    - **rr:** messages sent and received after the global snapshot is taken
    - **rw:** sent my red process and received by a white process. This produces an inconsistent global snapshot and **should be checked for**.
    - **wr:** sent by a white process and received by a red process. These are message which have been sent, but are in transit to being received.
  + **Neighbor:** two processes are neighbors if there exists a channel from one of the processes to the other.
  + Recording state of channels:
    - For two processes **Pi** and **Pj**, **Pi** can stop recording messages once it turns red (**“wr” states aren’t possible anymore /** no further white messages sent from **Pi**).
    - When a process turns red, all reachable processes must also turn red.
  + Overhead:
    - A total of **e** “marker” messages must be sent, where **e** represents number of unidirectional channels
    - Potential overhead for combining local snapshots into a global snapshot: have predetermined process receive all local snapshots
  + Algorithm:
    - Var: color: {red, white}; // initially white
    - Var: channels: array[1 .. k] of queues of messages; // initially empty
    - Var: isClosed: array[1 .. k] of Boolean; // initially false
    - Turn\_red():
      * If (color == white):
        + Save\_local\_state;
        + Color = red;
        + Send (“marker”) to all neighbors
    - On receive(“marker”) on channel **j** from channel **i**:
      * If (color == white):
        + Turn\_red();
      * Closed[**i**] = true; // **i** is the message sent from the source channel
    - On receive(“program\_message”) on channel **j** from channel **i:**
      * If (~closed[j] && color == red):
        + Channel[j].append(“program\_message”);
* **Global Snapshots for Non-FIFO Channels:**
  + Can no longer rely on marker to distinguish between red and white channels, since messages can overtake each other (i.e. **a white message may appear after a red message is received**).
    - Solution:
      * Solves **color distinction**: Include color of message in all outgoing messages **except** in sending the marker
      * Solves **non-FIFO problems**: Include in the marker the total number of white messages that will be sent to the receiver. Receiver increments counter every time white message is received, and knows when all white messages are received when counter == count in marker message.
* **Channel Recording by Sender:**
  + Messages in real channels could potentially be lost, so it is advantageous to record the state of the channel from the sender.
  + **Prevention of inconsistent global state:** Ensure no **rw** messages are in channel by setting white process/channel to red before accepting a red message
  + **Recording of channel state:** 
    - Sender records all messages that it sends to all outgoing channels before turning red.
    - Once red, marker is sent to all **incoming channels** (reverse direction as in snapshot algorithms with receiver bias) indicating all messages **it has received so far**, allowing sender to compute state of channel by **removing all messages from its buffer that were indicated by the marker sent.**
    - Reducing burden of message accumulation on sender:
      * Assume control messages (marker and acknowledgements) obey FIFO ordering.
      * Solution: require receiver to send acknowledgements -> when sender has not yet received marker but received the acknowledgement, can remove the message from its buffer.
        + Use sequence numbers to identify message uniquely.
  + Algorithm:
    - Every process is white before recording its state and red after recording its state. A white process **only** sends white messages and red processes **only** send red messages.
    - A white process turns red before accepting a red message or a marker.
    - On turning red, a process sends markers to all incoming channels.
    - A white process acknowledges a white message.
    - A white process records any message sent. On receiving an acknowledgement, the corresponding message is deleted from the process’ buffer.
    - Doesn’t require for application messages to be FIFO.
  + If channels are FIFO:
    - Receiver only needs to record the sequence number of the last message received before turning red.
    - Requires control messages to be FIFO order:
      * Reason: