Distributed Systems

Predicates and Mutual Exclusion

Rik Sarkar
Edinburgh Fall 2014

University of Edinburgh

Where snapshots are not useful: non-stable predicates

- E.g.
  - Was this file opened at some time?
  - Was $x1-x2 < \delta$ ever?
  - Was the antenna accessed for two transmissions at the same time?

  - Non-stable predicates may have happened, but then system state changes..

Non-stable predicates

- Possibly B:
  – B could have happened

- Definitely B:
  – B definitely happened

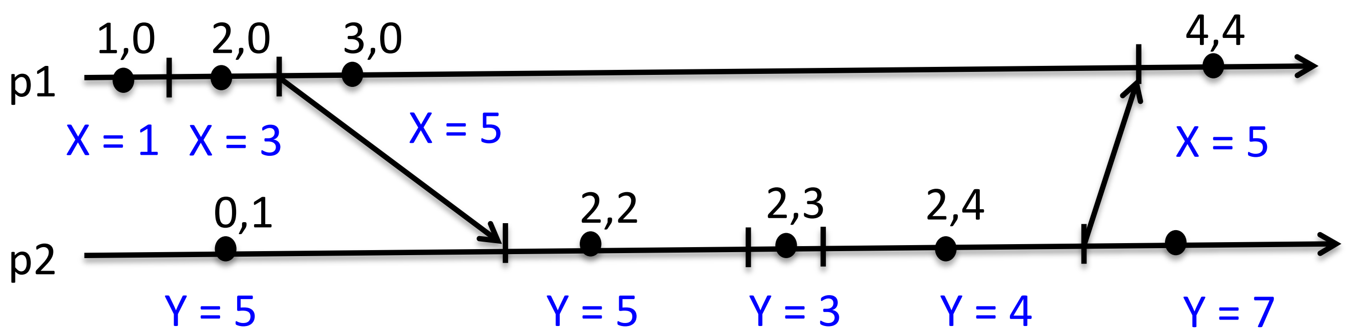
- How can we check for definitely B and possibly B?

Collecting global states

- Each process notes its every state & vector timestamp
  - Sends it to a server for recording
  - Note: we do not need to save every time a state changes: only when it affects the predicates to be checked
    - Assuming we know what predicates will be checked
- The server looks at these and tries to figure out if predicate B was possibly or definitely true

Possible states

- Server checks for possible states: consistent cuts for B: x=y



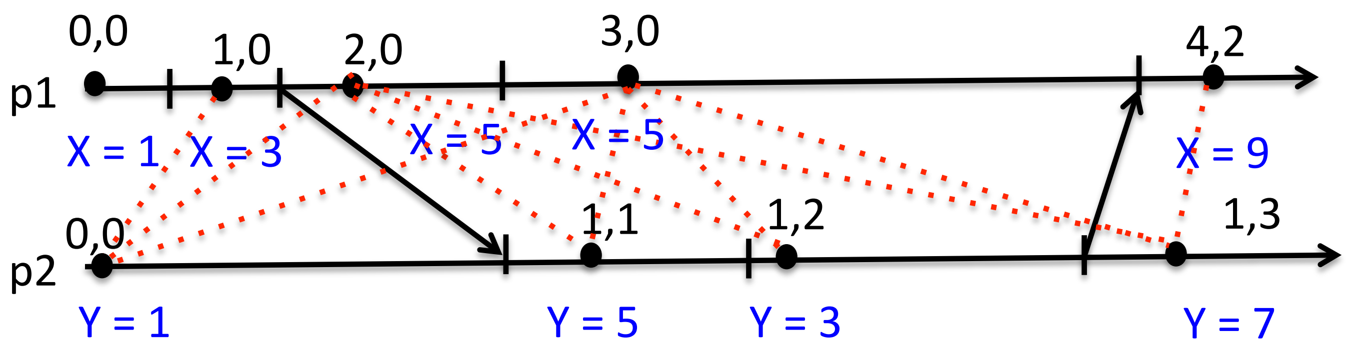
Note on difference with books

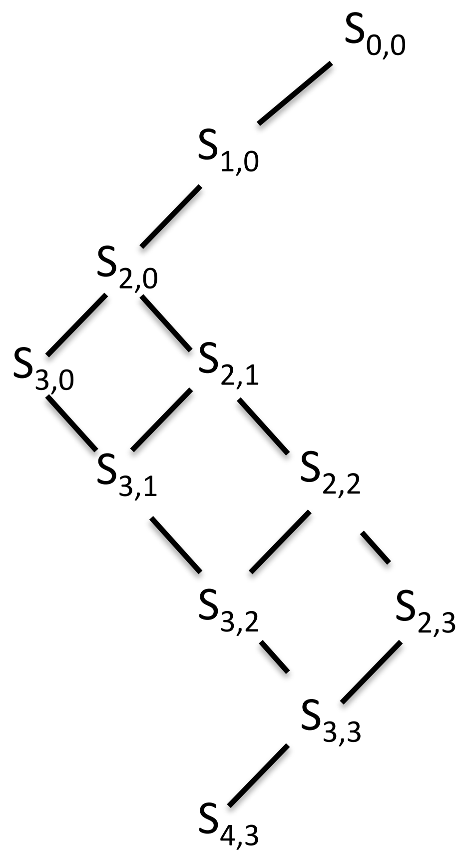
- We are using the following notation that may differ from books
  - The circles are 'states', and bars are 'events'
  - We are concerned with which pairs of *states* form consistent cuts
  - An event's occurrence changes the state of the process
  - We are following the convention that an event carries the label of the state in which it happened i.e. the label of the circle to the left of it.
    - You can see this in the vector clock label carried by the messages
  - Some books follow a different convention that the event (message) carries the label of the state after the event
  - Sometimes the representation of the states are merged with the events
- This does not change any of the fundamental ideas or properties of causality or snapshots
  - But labels in diagrams may look a little different
- In exam, you are allowed to use either convention if you are drawing a diagram. Mention which you are using.

- If a problem explicitly gives a diagram, it will use the convention in the slides, of separating states and events

Possible states

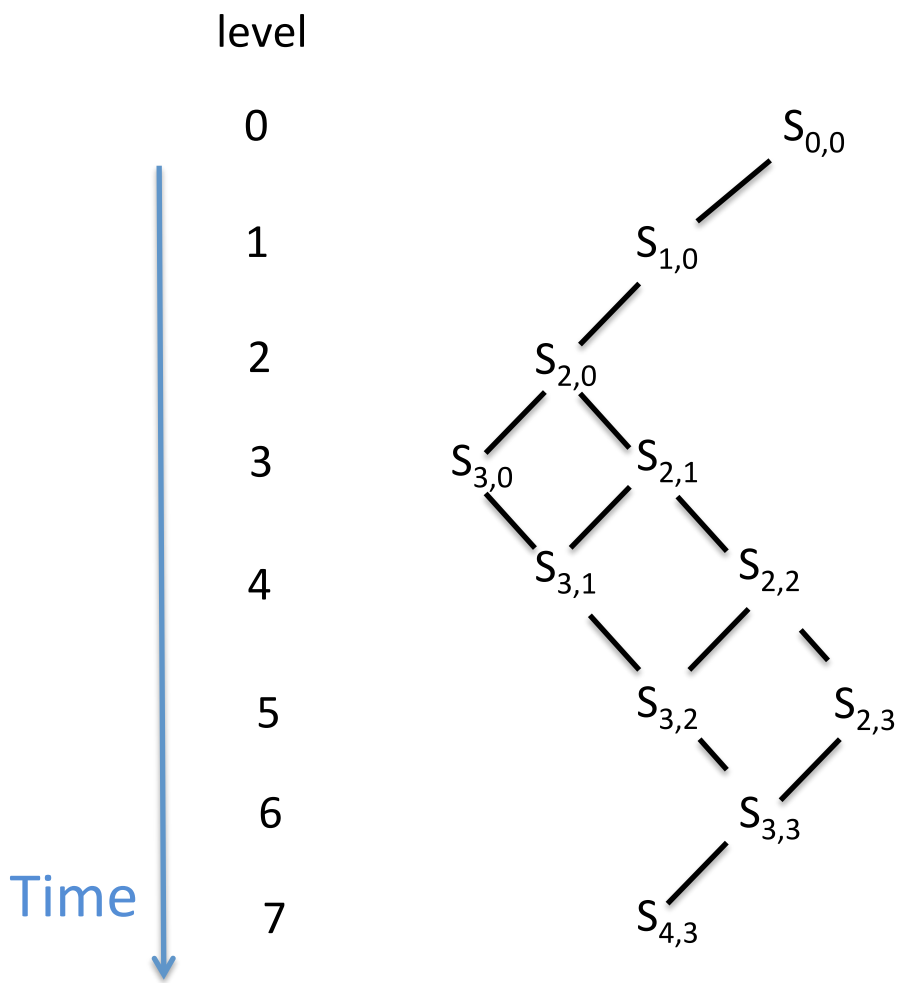
- Server checks for possible states: consistent cuts for B: x=y





Lattice of global states (consistent cuts)

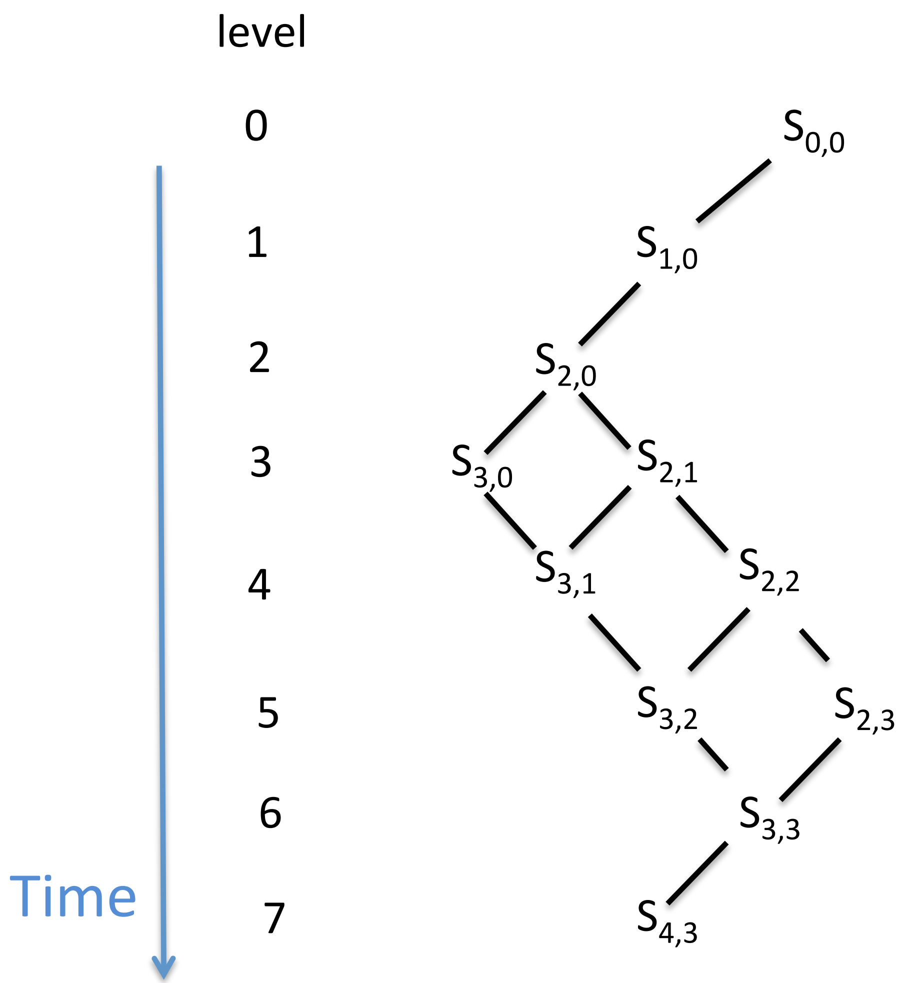
- Any downward path from Initial state to final state is a valid execution
  - A possible sequence of states that could have existed



Lattice of global states (consistent cuts)

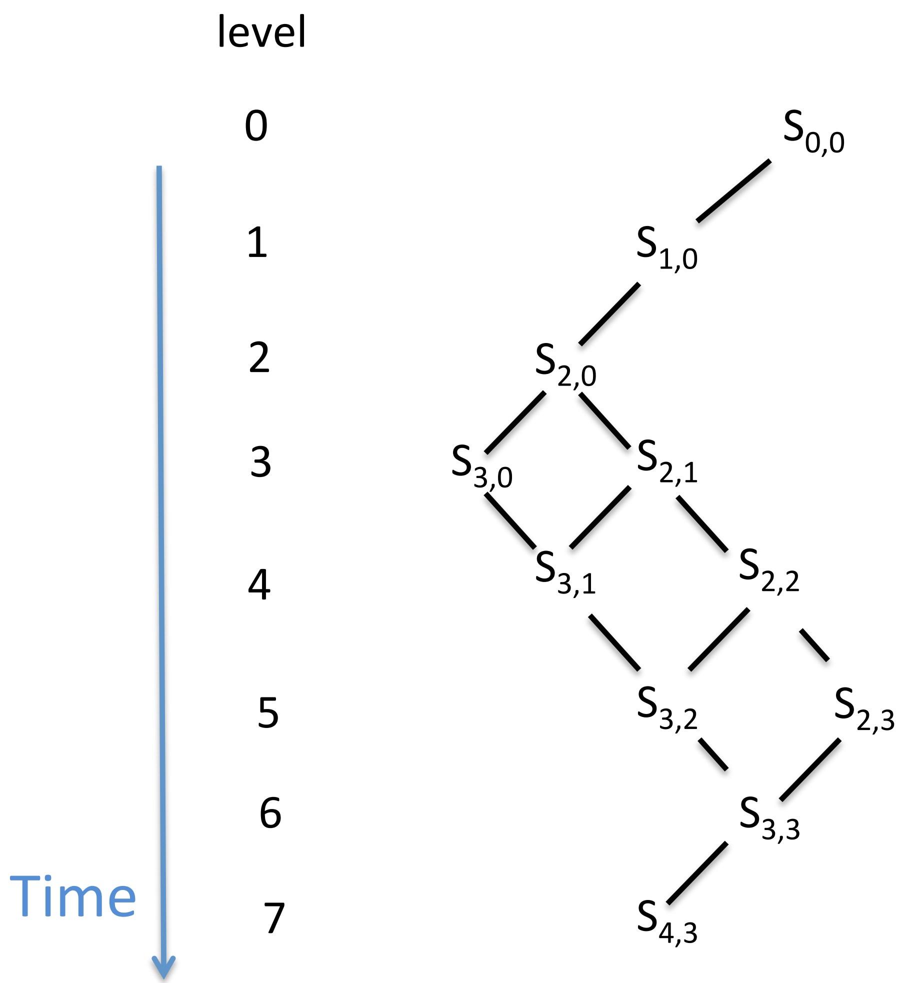
- Possibly B:
  - B occurs on at least one downward path

- Definitely B
  - B occurs on all downward paths



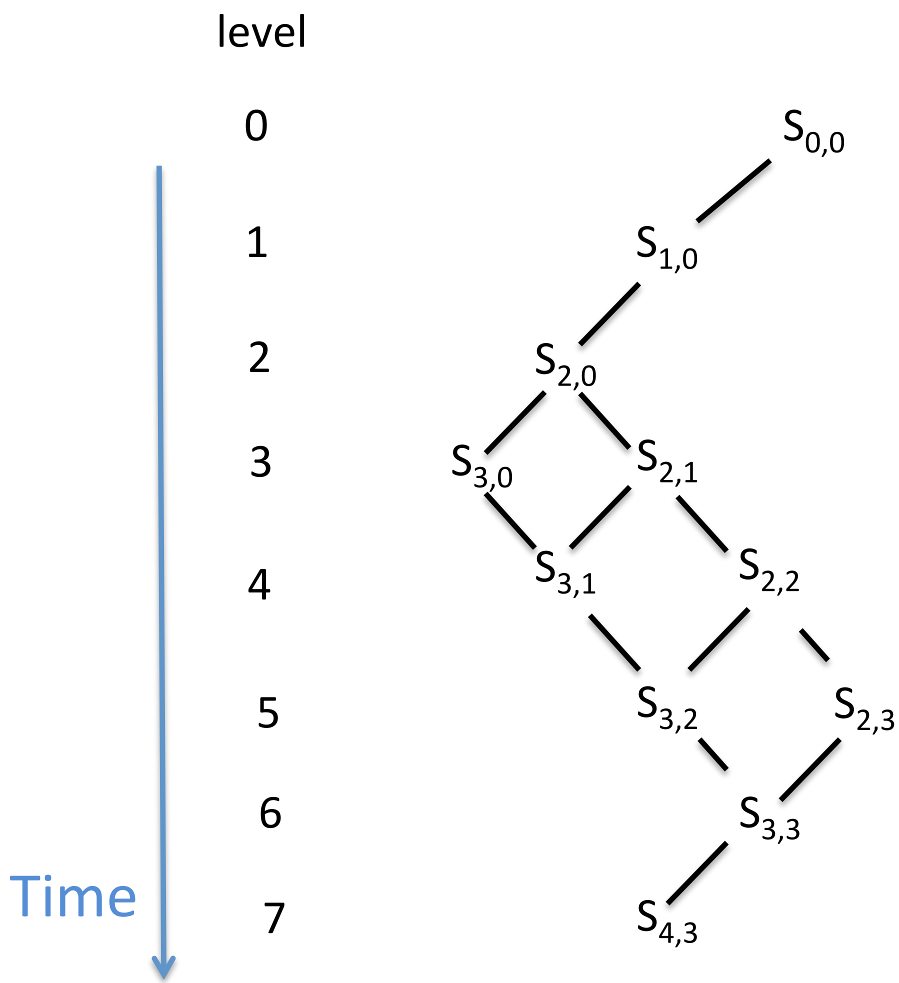
Lattice of global states (consistent cuts)

- How do you compute possibly and definitely B?



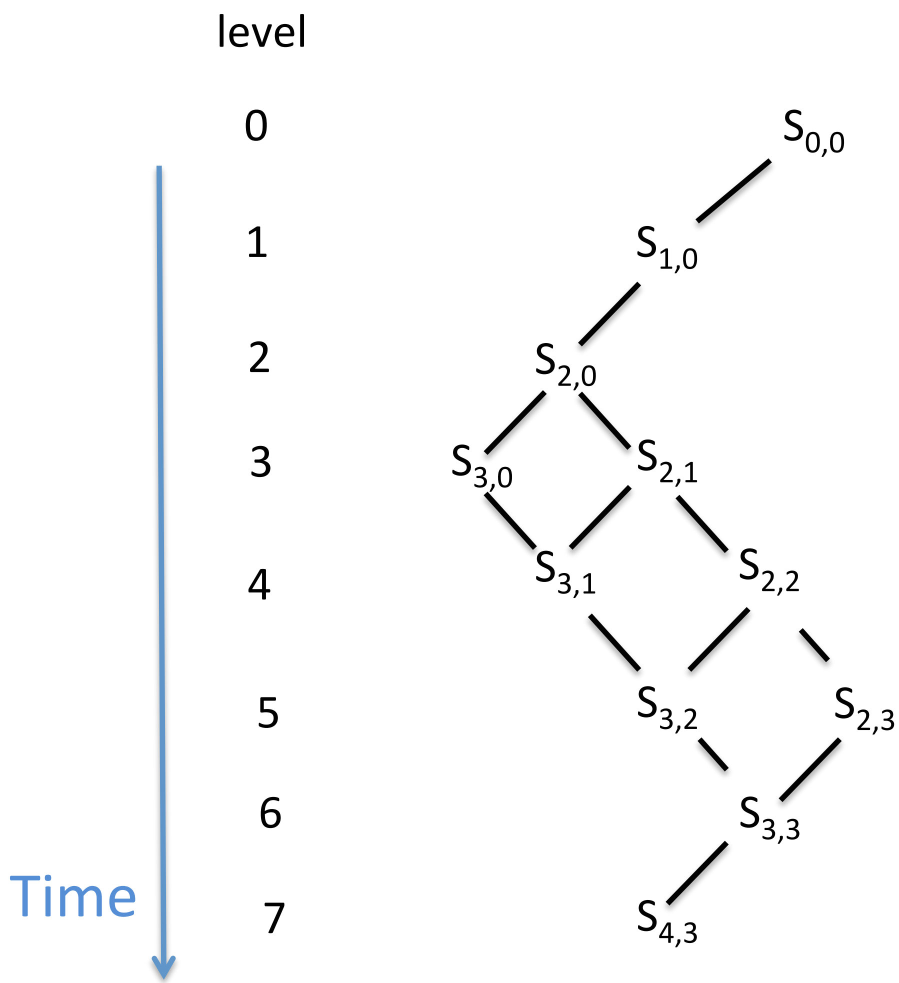
Lattice of global states (consistent cuts)

- Possibly B:
  - B occurs on at least one downward path
- Do a BFS from start state
  - If there is one state with B true, then possibly B is true



Lattice of global states (consistent cuts)

- Definitely B
  - B occurs on all downward paths
- Do a BFS from start state
  - Do not visit nodes with B: true
  - If BFS reaches final state and B is false in final state then Definitely B is false
  - Else Definitely B is true



What is the computational complexity?

What is the computational complexity?

- Possibly exponential in number of processes
- Problem is NP-complete

- Observation: more messages reduces complexity!

Mutual exclusion

Ref: CDK, VG

- Multiple processes should not use the same resource at once
  - Eg. Print to the same printer
  - Transmit/receive using the same antenna
  - Update the same database table
- Critical section (CS): the part of code that uses the restricted resource
- Mutual exclusion : restrict access to critical section to at most one process at one time

Properties in ME

- Safety: Two processes should not use critical section simultaneously

Properties in ME

- Safety: Two processes should not use critical section simultaneously

- Liveness: Every live request for CS is eventually granted
- Fairness: Requests must be granted in the order they are made (upto logical time)

Distributed Vs Centralized Mutex

- On a single computer, OS can manage access to a shared variable
- On a distributed system, we have to use messages

Assumption

- There is only one resource in question
- In reality there can be more, but for now, let us focus on just one
- All channels are FIFO

Central server algorithm

- There is a server or coordinator
  - Holds a "token" for the resource
- Other processes send token request to the server
- Server puts incoming requests in a queue
- Sends token to first process in queue
- Process returns token when done
- Server sends to next process

Central server algorithm

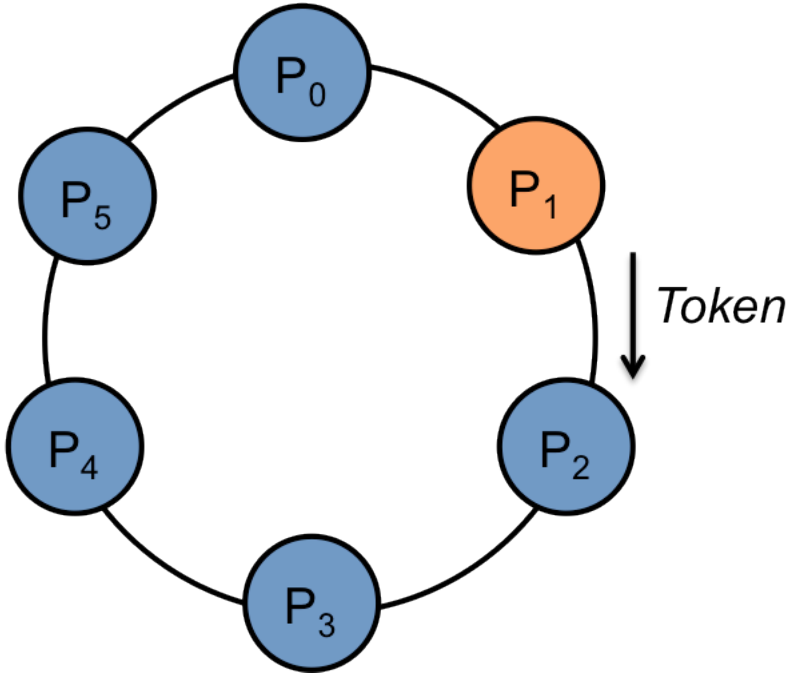
- What are the advantages and disadvantages?

Central server algorithm

- Advantages
  - Simple
  - Constant complexity per message
- Disadvantages
  - Central point of failure
  - Central bottleneck
  - Does not preserve order in asynchronous systems
  - Server must be selected/elected

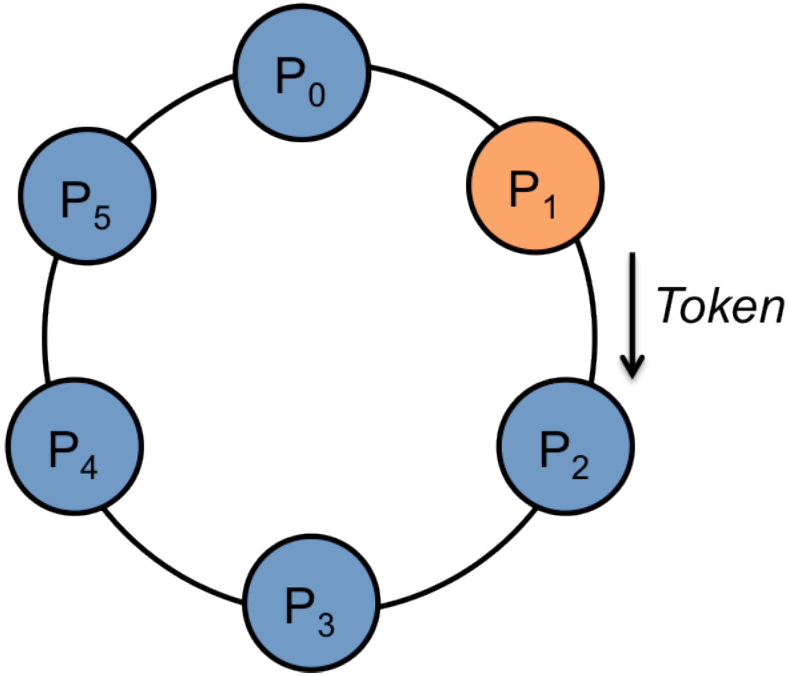
Token ring algorithm

- Processes are arranged in a ring
- The token is continuously passed in one direction
- A process on reciving token:
  - If it does not need CS, passes token to next one
  - If it needs CS, it holds token, executes CS and then passes token



Token ring algorithm

- Observe:
  - Processes do not need to be in an actual ring
  - Each process just needs to know the next process and have a method to send it a message



Token ring

- Problems:

Token ring

- Problems:
  - Not in-order
  - Long delay in getting token
    - Upto n-1
  - One failure breaks the ring
  - Passes token around even when there are no requests

Lamport's algorithm

- Every node i has a queue q_i of requests
  - Keeps requests sorted by logical timestamps
- Process i sends CS request:
  - Timestamped REQUEST (tsi, i) to all processes
  - Enters (tsi,i) to its own queue q_i
- Process j receives REQUEST (tsi,i)
  - Send timestamped REPLY to i
  - Enter (tsi,i) to q_j

Lamport's Algorithm

- Process i enters CS if
  - (tsi,i) is at head of its own queue
  - It has received REPLY from all processes
- To release CS
  - Process i sends RELEASE message to all
- On receiving RELEASE, process j
  - Removes (tsi,i) from q_j

Observations

- Requests granted in order consistent with happened before
- 3(n-1) messages per CS

Ricart and Agrawala's algorithm

- Main modification:
  - Node j does not send a REPLY if j has a request with timestamp lower than i's request
  - j simply delays the REPLY until its RELEASE message

Ricart-Agrawala's algorithm

- Process i sends CS request:
  - Timestamped REQUEST (tsi, i) to all processes
- Process j receives REQUEST (tsi,i)
  - If j has no outstanding request of its own earlier than (tsi,i) or is not executing CS
    - Send timestamped REPLY to i
    - Enter (tsi,i) to q_j
  - Else keep (tsi,i) pending

Ricart-Agrawala's algorithm

- Process i enters CS if
  - It has received REPLY from all processes
- To release CS
  - Sends REPLY message to pending processes

Ricart-Agrawala's algorithm

- Has no queues at processes
- The queue is maintained distributedly across all processes through timestamps and delayed replies
- Uses 2(n-1) messages

Maekawa's Quorum algorithm

- Idea: instead of getting permission from all processes, get permission from only a subset of processes
- For each process i, we have a voting set (quorum) V_i
  - For all i,j: $V_i \cap V_j \neq \varnothing$
  - For all i, $i \in V_i$
  - Voting sets are same size, each node is part of same number of sets

Maekawa's Quorum algorithm

- Idea:
  - Arrange nodes in a square grid
  - Quorum for node i:
    - All nodes in same row or same column as i
  - Any two quorums intersect

- Complexity?

- Complexity per CS: $O(\sqrt{n})$