Distributed Systems

12 Coordination

June 22/24/29 2009
Gerd Liefländer
System Architecture Group

Outline: Next Lectures

- Coordination Problems
    - Global State
    - Failure Detection
    - Mutual Exclusion
    - Election
    - Multicast
    - Consensus
    - Deadlocks
    - Distributed Transactions

Recommended reading:

Tanenbaum, Ch. 5, 7, Coulouris/Dollimore/Kindberg, Ch. 11, 12, 13

Motivation

- Given an asynchronous DS, i.e. no process has a view of the current global state of the DS
- Need to coordinate the actions of cooperating processes to achieve common goals
  - Failure detection: how to know in an asynchronous network whether my peer is dead or alive?
  - Mutual exclusion: how to guarantee that no two processes will ever get access to a critical section at the same time?
  - Election: how will the system elect a new master in a master-slave based distributed application?
  - Multicast: how to enhance when sending to a group of recipients that
    - ∃ reliability of the multicast (i.e. correct delivery, only once, etc.)
    - ∃ preservation of the order of the messages

Global State

Chandy/Lamport: Distributed Snapshots: Determining Global States of DS
http://research.microsoft.com/users/lamport/pubs/chandy.pdf

Dijkstra: Comments on Chandy/Lamport/Misra Algorithm
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD864.html

Michael L. Powell and David L. Presotto,“PUBLISHING: A Reliable Broadcast Communication Mechanism, *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Oct 83.

Ozalp Babaoglu and Keith Marzullo: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, in **Distributed Systems**, Sape J. Mullender, Addison-Wesley, 1993.

Outline of this Chapter[1]

- Complexities of state detection in DS
- The notion of consistent state
- The distributed snapshot algorithm (Chandy/Lamport)
- Application to detect stable properties and checkpointing
- Another approach for global state recording: publishing

---

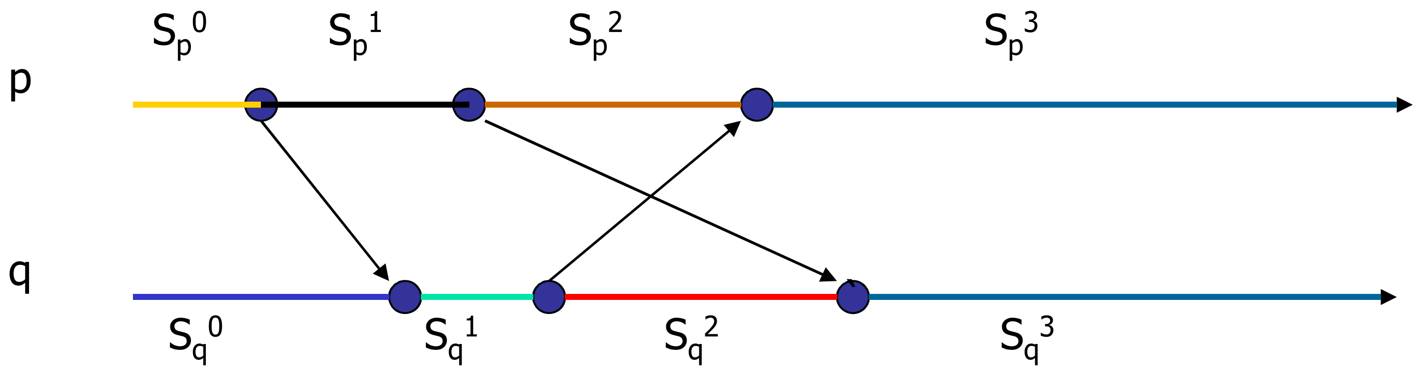
[1] Most slides on Global State are from Sanjeev R. Kulkarni (Princeton Uni)

Model of Computation

- Finite set of processes
- Process send messages on a finite set of unidirectional channels
- Channels are error free, preserve FCFS, and have infinite buffers
- Messages experience arbitrary but finite delays
- Strongly connected network

Model of Computation (cont.)

- A computation is a sequence of events.
- An event is an atomic action that changes the state of a process and at most one channel state that is incident on that channel.



- Arcs indicate a message transfer

Happened Before Relation ⟶

- Events e and e` of the same process.
  - if e happens before e` then e ⟶ e`
- e and e` in two different processes
  - if e = send(m) and e` = recv(m) then e ⟶ e`
- Transitive
  - if e ⟶ e` and e` ⟶ e`` then e ⟶ e``

Determining Global State

- Global State

  “The global state of a distributed computation is the set of local states of all individual processes involved in the computation plus the state of their communication channels.”

More on States

- process state
  - memory state + register state + signal masks + open files + kernel buffers + …

    or
  - application specific info like transactions completed, functions executed etc.
- channel state
  - "Messages in transit" i.e. those messages that have been sent but not yet received

Why to deal with Global States?

- Many problems in distributed computing can be cast as executing some action on reaching a particular state
- e.g.
  - distributed deadlock detection is finding a cycle in the wait for graph.
  - termination detection
  - check pointing
  - some more…..

Snapshot Problem

Suppose computation of a distributed application has become passive on each involved node

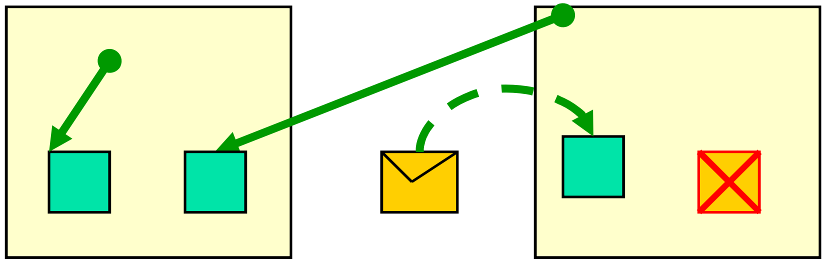
We want to be able to distinguish whether

⇒ a distributed application

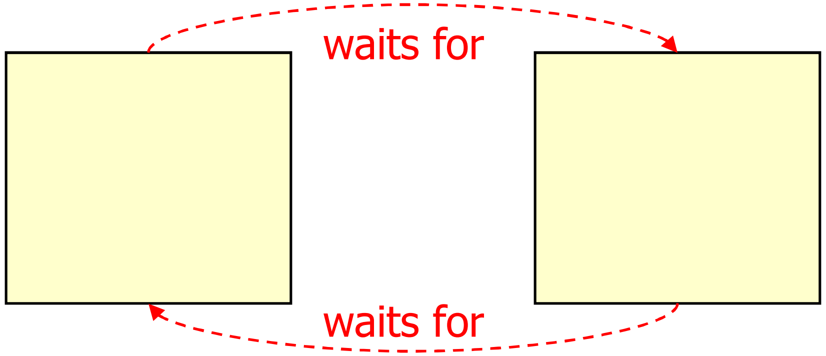
1. is temporarely blocked
2. has "terminated" or
3. is deadlocked

Snapshot Problem

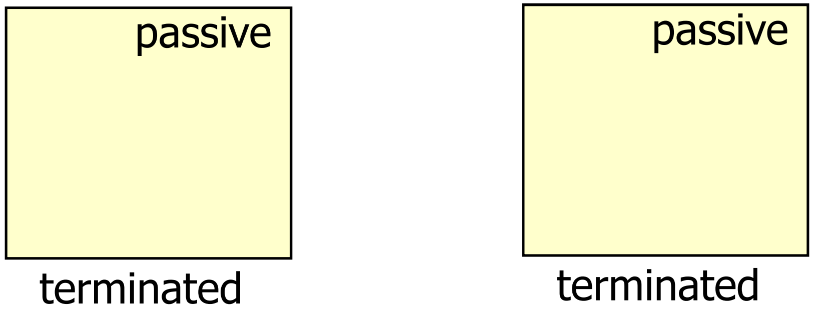
- Garbage collection



- Deadlock



- Termination problem



Why is Global State difficult in DS?

- Distributed state:

  Have to collect information that
  is spread across several machines!!

- Only local knowledge:

  A process in a distributed computation
  might not really know the current states
  of the other processes

Difficulties

- Instantaneous recording not possible
  - No global clock: the distributed recording of local states cannot be synchronized based on time
    - Some local states reflect an outdated state, some reflect the current state
  - Random network delays: no centralized process can initiate the detection

Difficulties due to Non Determinism

- Deterministic Computation
  - At any point in computation there is at most one event that can happen next.

- Non-Deterministic Computation
  - At any point in computation there can be more than one event that can happen next.

Example: Deterministic Computation

- Producer code:

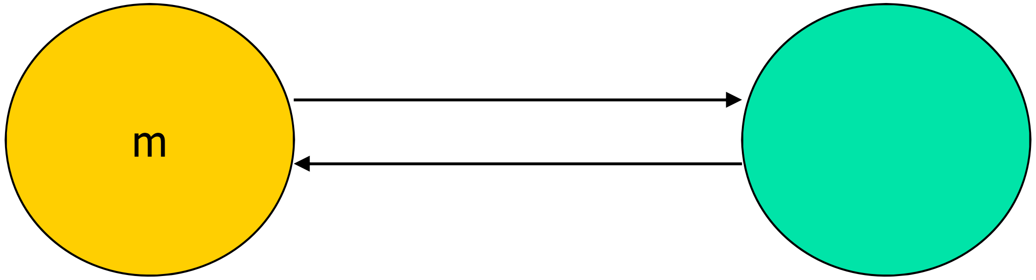
```
while (1)
{
        produce m;
        send m;
        wait for ack;
}
```

- Consumer code:

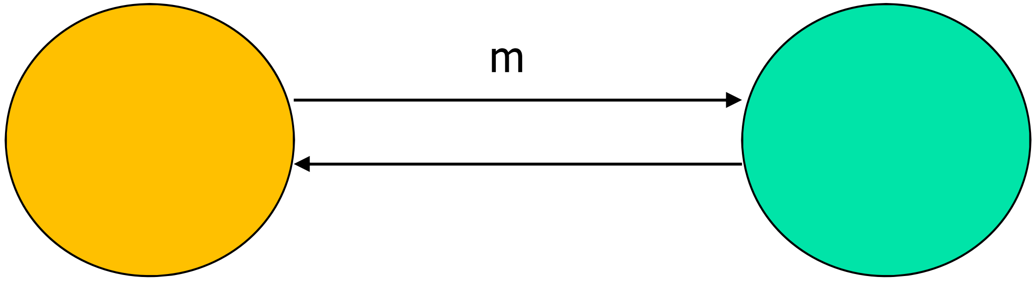
```
while (1)
{
        recv m;
        consume m;
        send ack;
}
```

Very simple solution for a distributed producer consumer problem

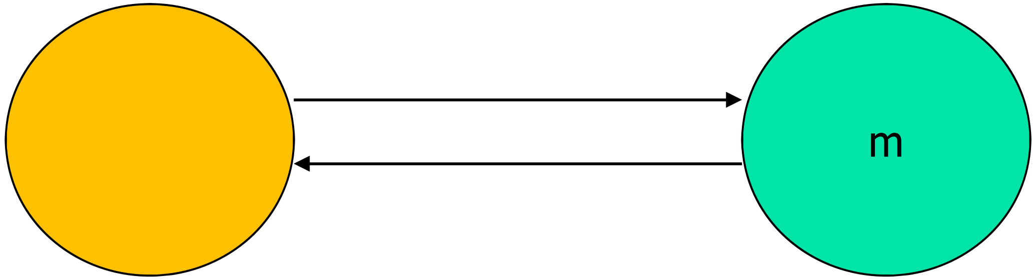
Example: Initial State



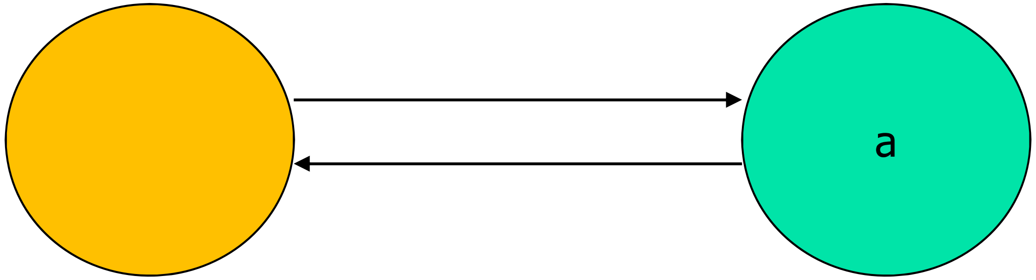
Example: Intermediate State



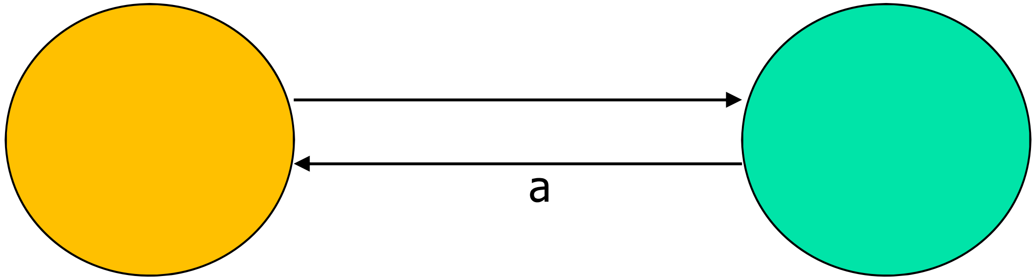
Example



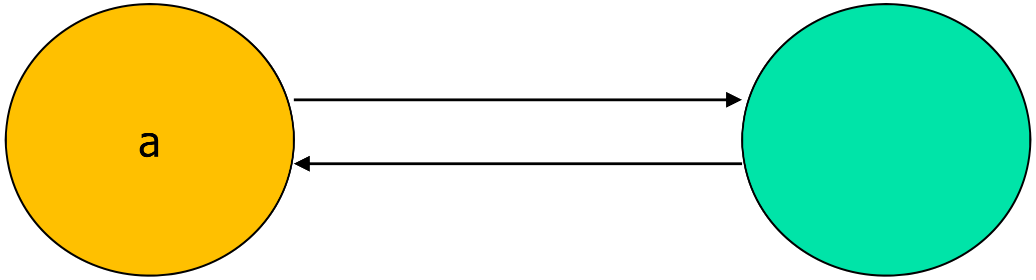
Example



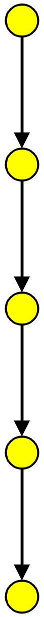
Example: Intermediate State



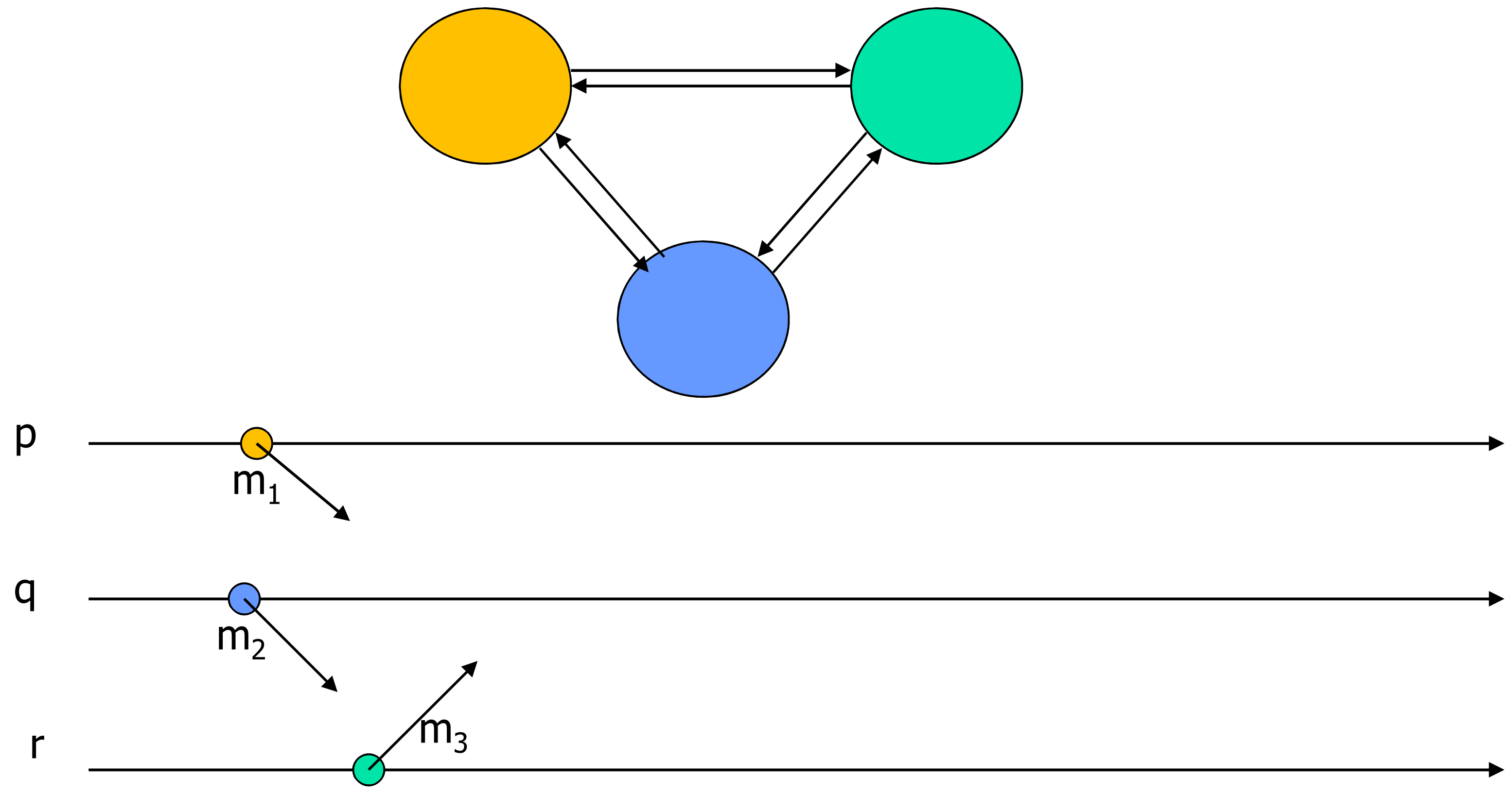
Example: Product m consumed



Deterministic State Diagram

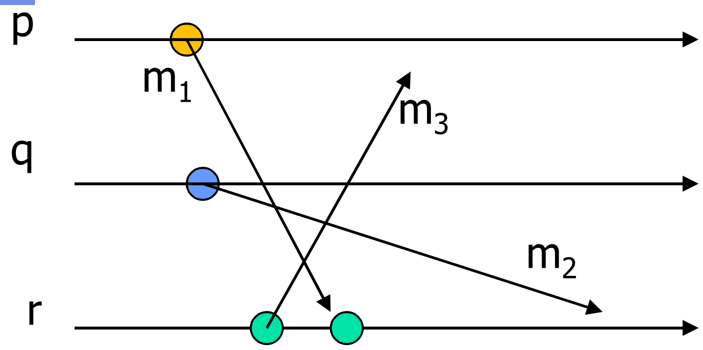


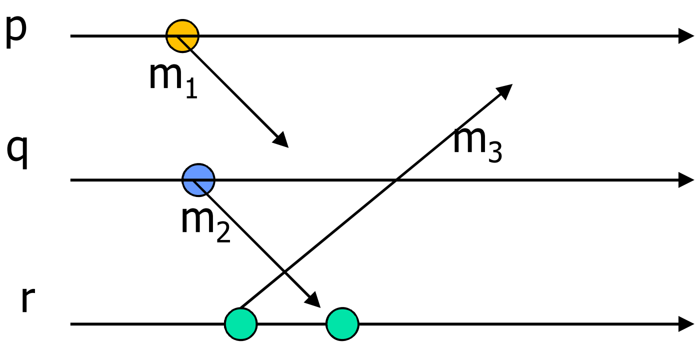
Non-Deterministic Computation

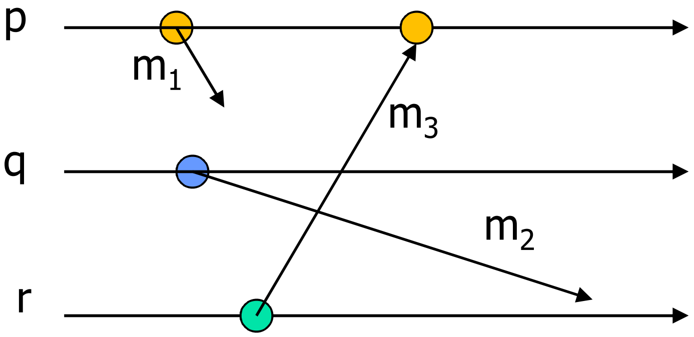


Three processes interacting asynchronously

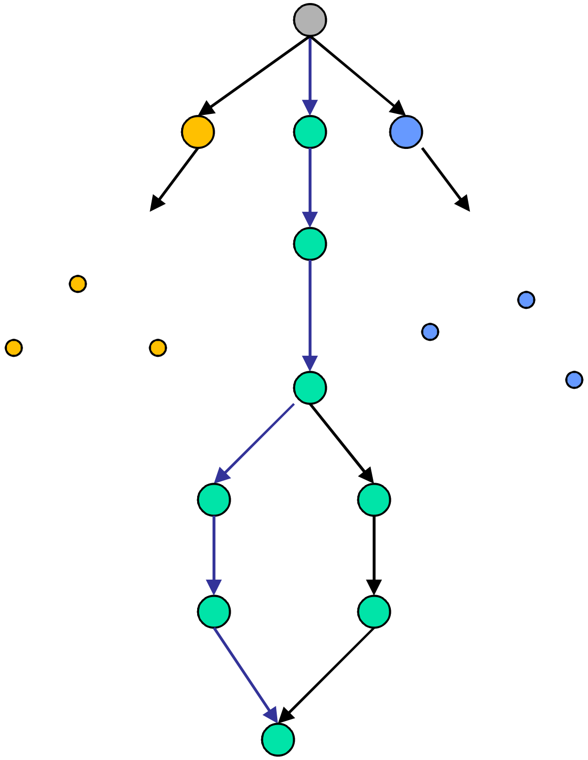
Three Possible Runs







A Non-Deterministic Computation



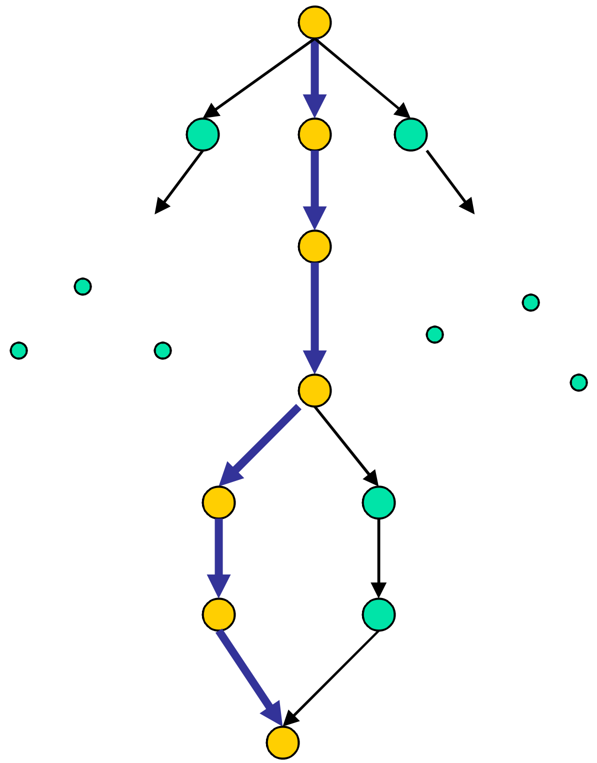
- All these states are feasible

Feasible and Actual States

- Any state that an external observer could have observed is a feasible state

- A state that an external observer did observe is an actual state

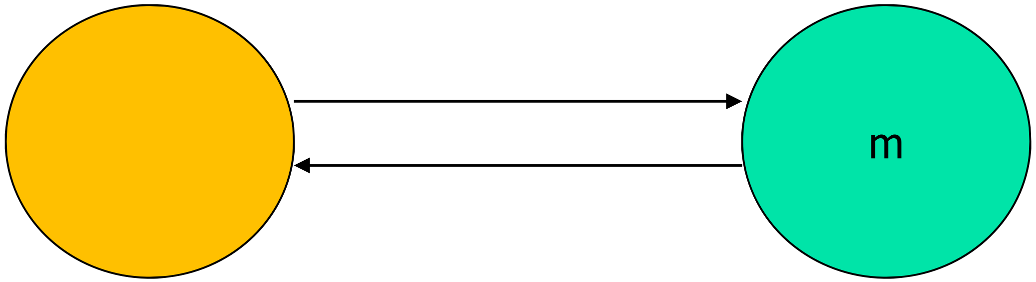
A Non-Deterministic Computation



- Only some states are actual

Non-Determinism

- Deterministic computation
  - A local event would reveal everything about the global state!
  - The process will know other process' state



- Not so for Non-Deterministic computation!

A Naïve Snapshot Algorithm

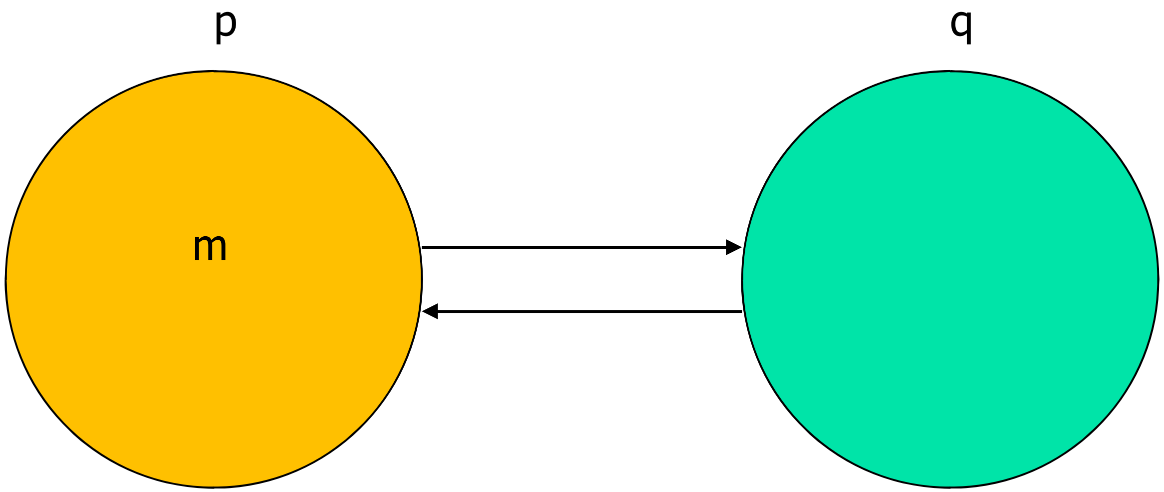
- Processes record their state at any arbitrary point
- A designated process collects these states

\+ So simple!!

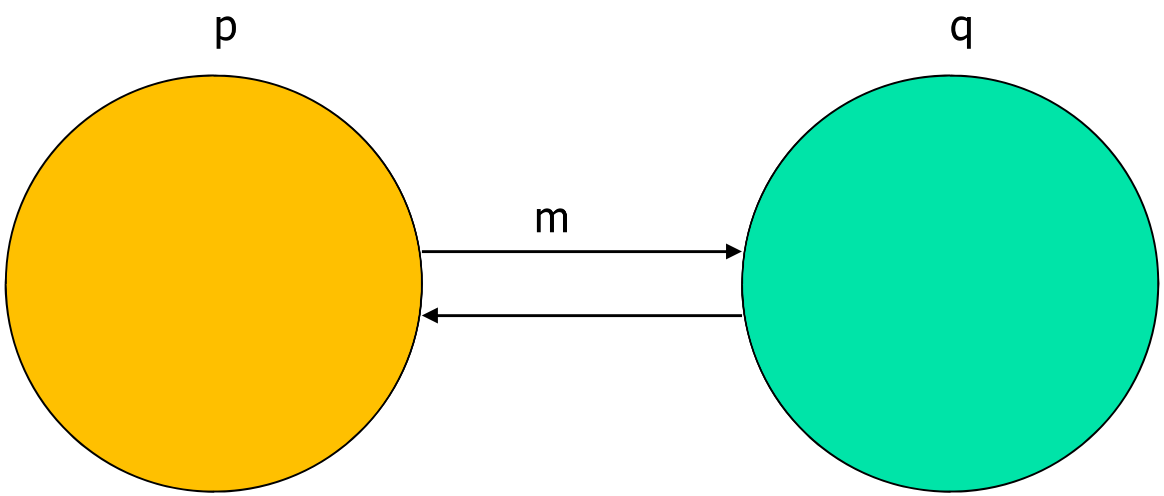
\- Correct??

Example: Producer Consumer

p records its state

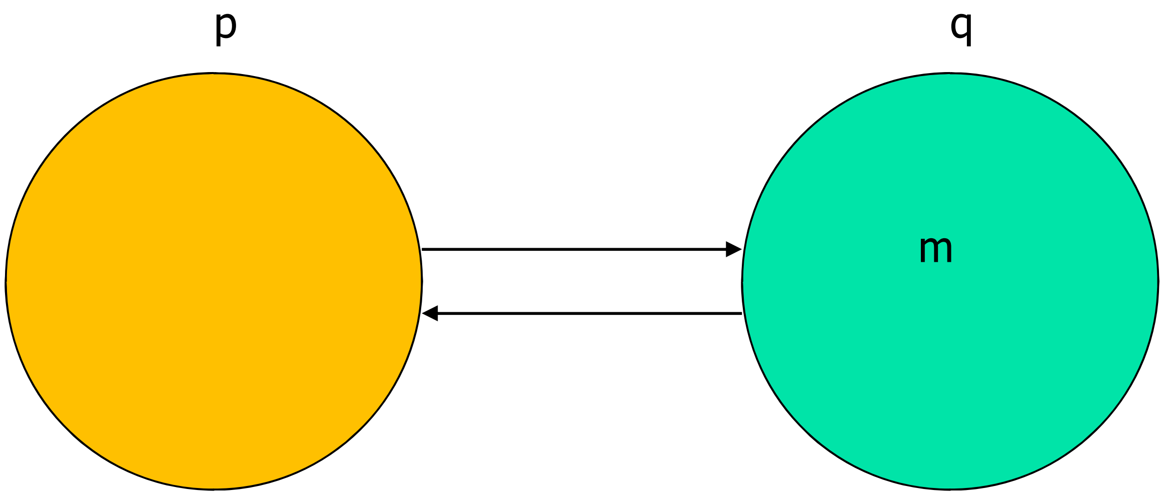


Example

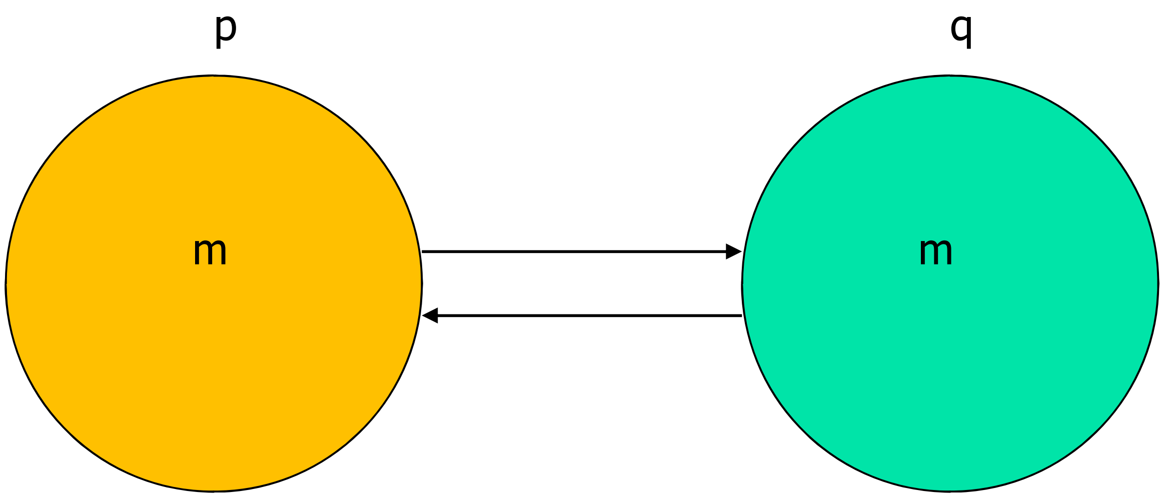


Example

q records its state

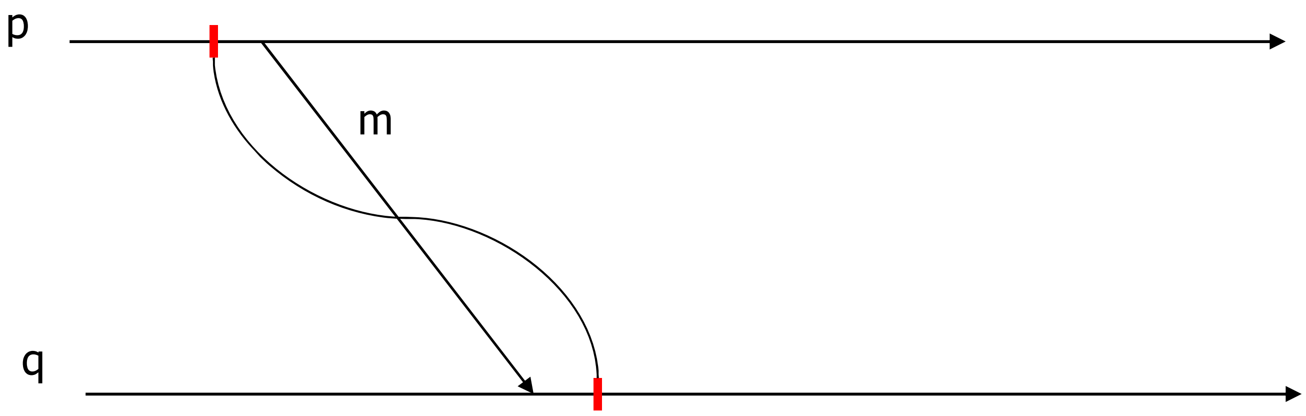


Example: Recorded Global State



Where did we err?

- *What did we do?*



- We recorded inconsistently

Error!!

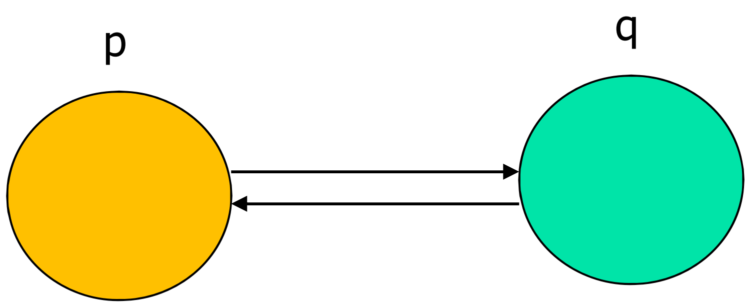
- The sender has no record of the sending
- The receiver has the record of the receipt
- Result:
  - Global state contains record of the receive event but no send event, thus violating the happened before concept
- What we need is something that helps us to determine consistency of local recording

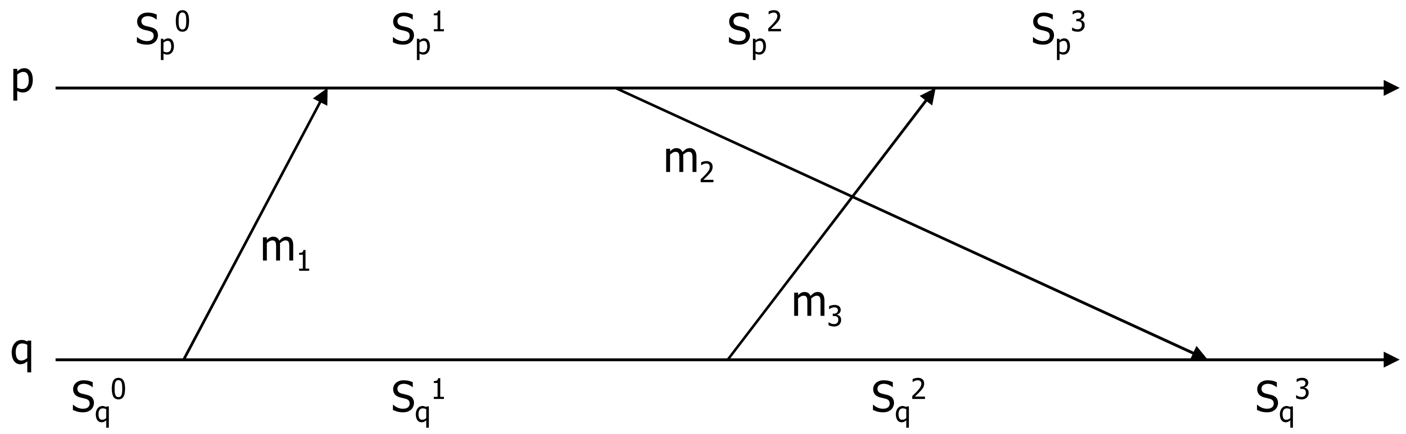
Notion of Consistency

The Notion of Consistency

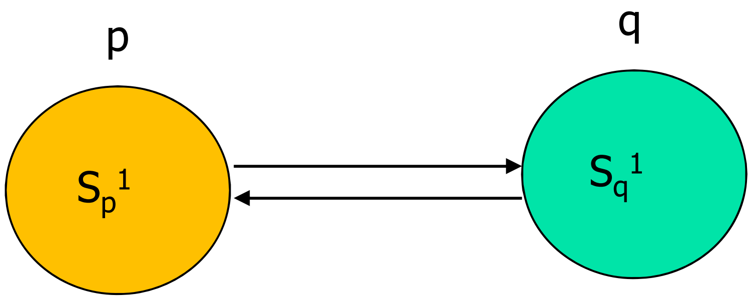
- A global state is consistent if it could have been observed by an external observer
- If e $\longrightarrow$ e` then it is never the case that e` is observed by the external observer and not e
- All feasible states are consistent

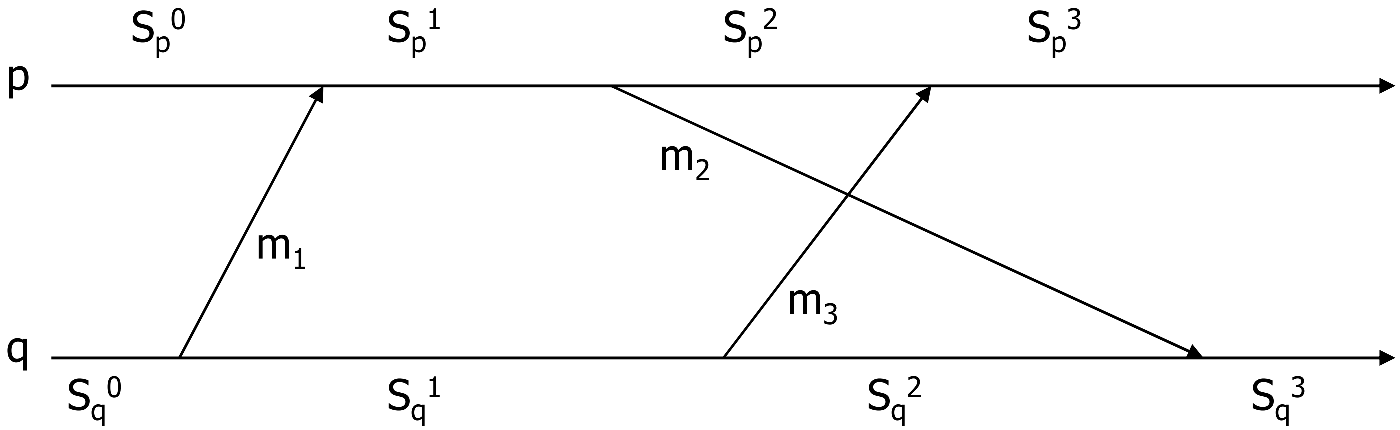
An Example



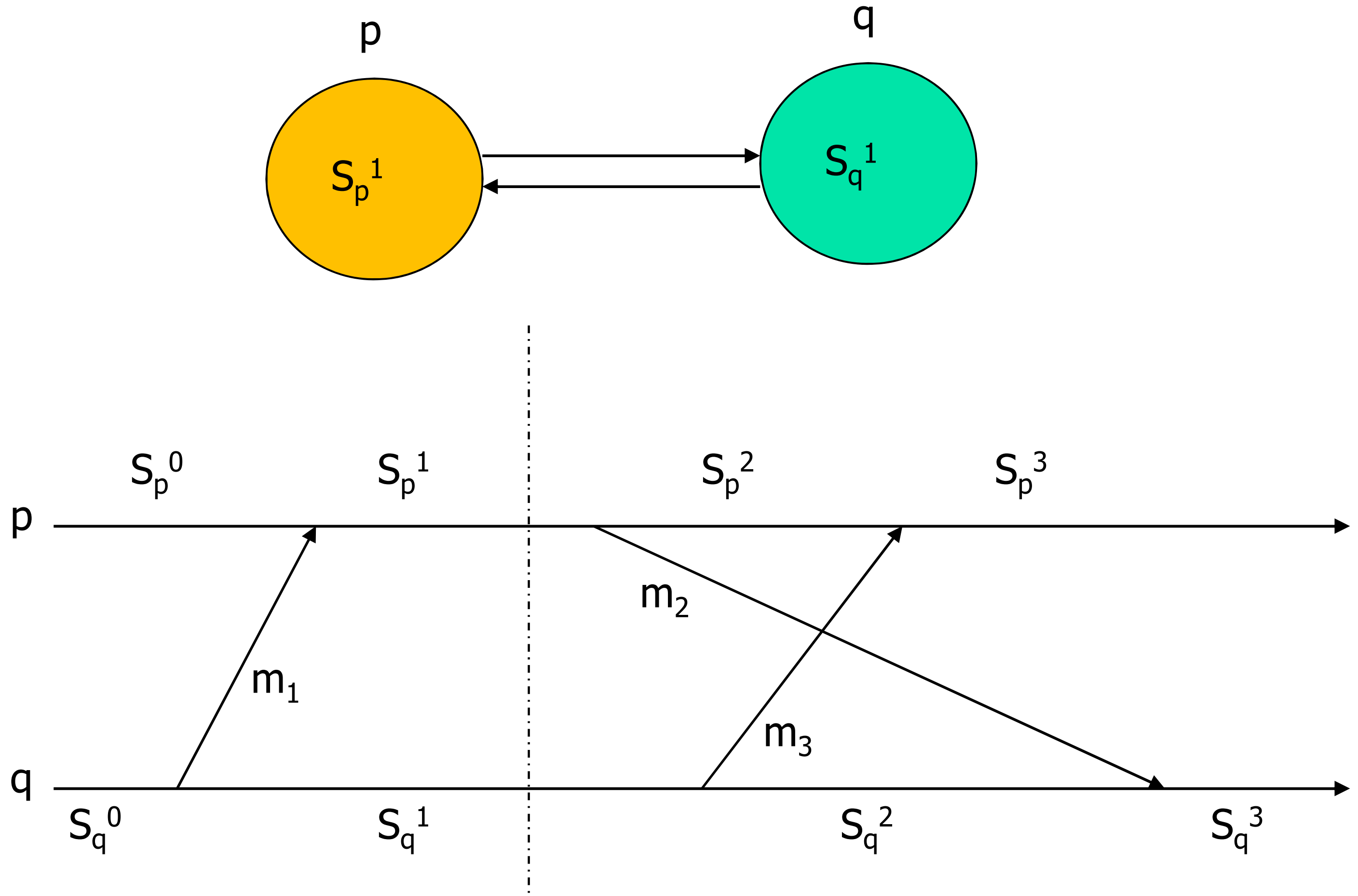


A Consistent State?

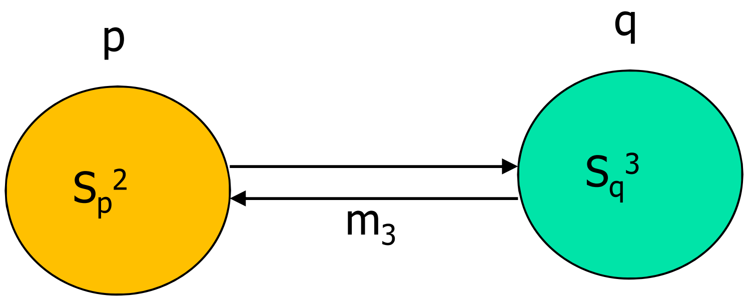


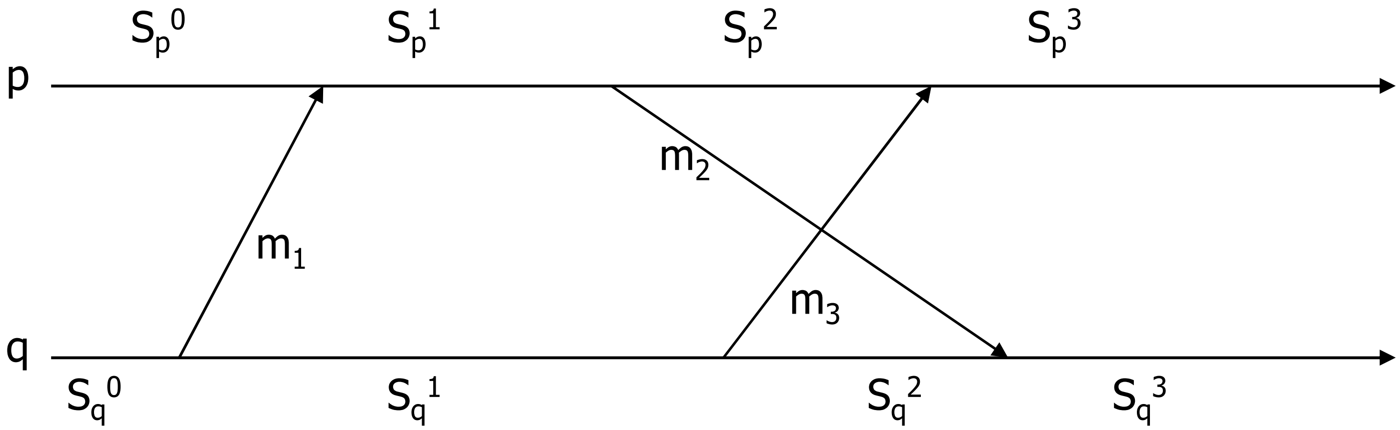


Yes

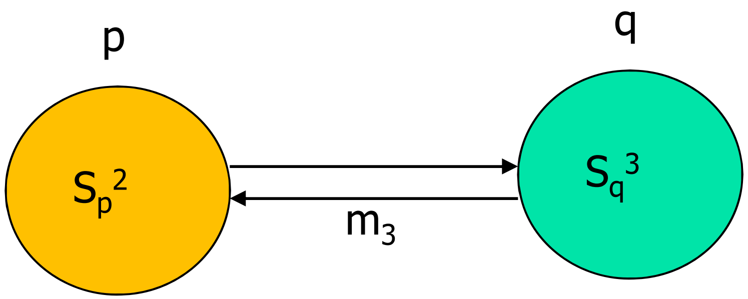


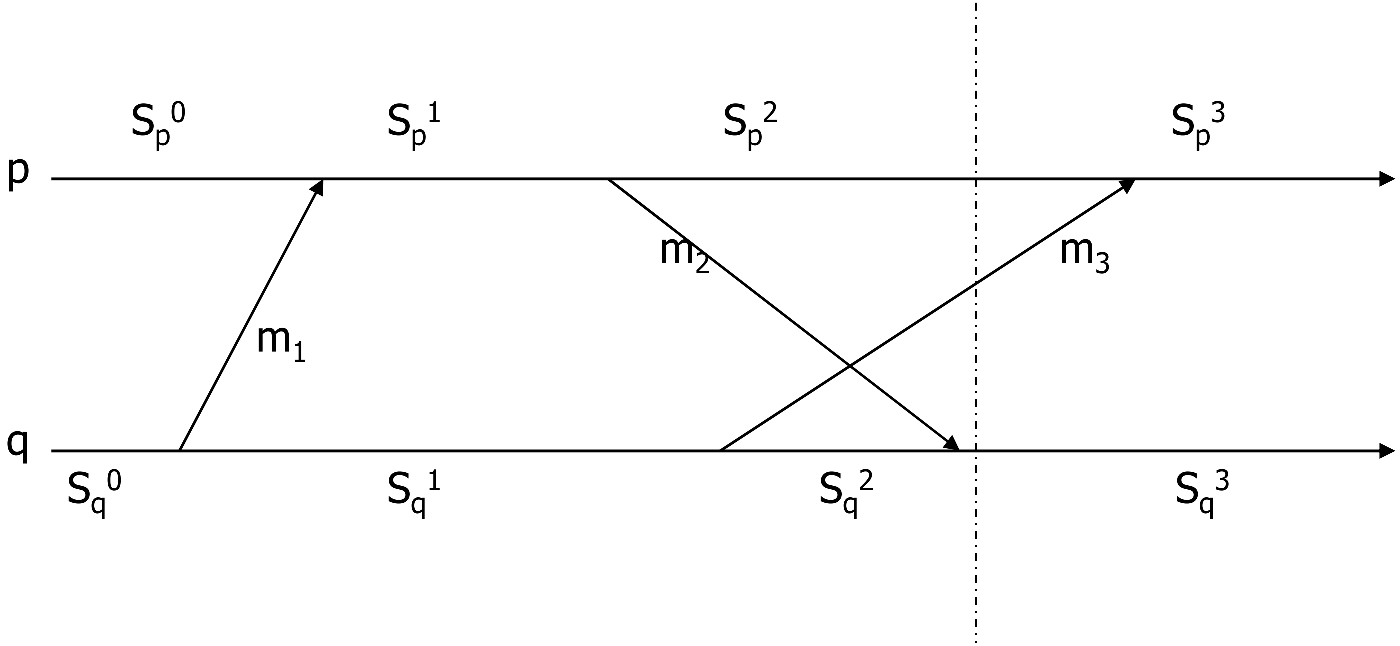
A Consistent State?



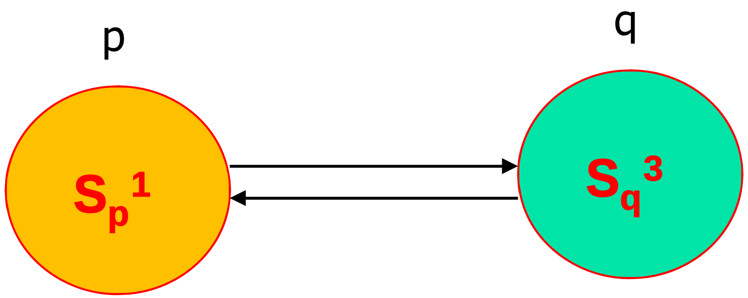


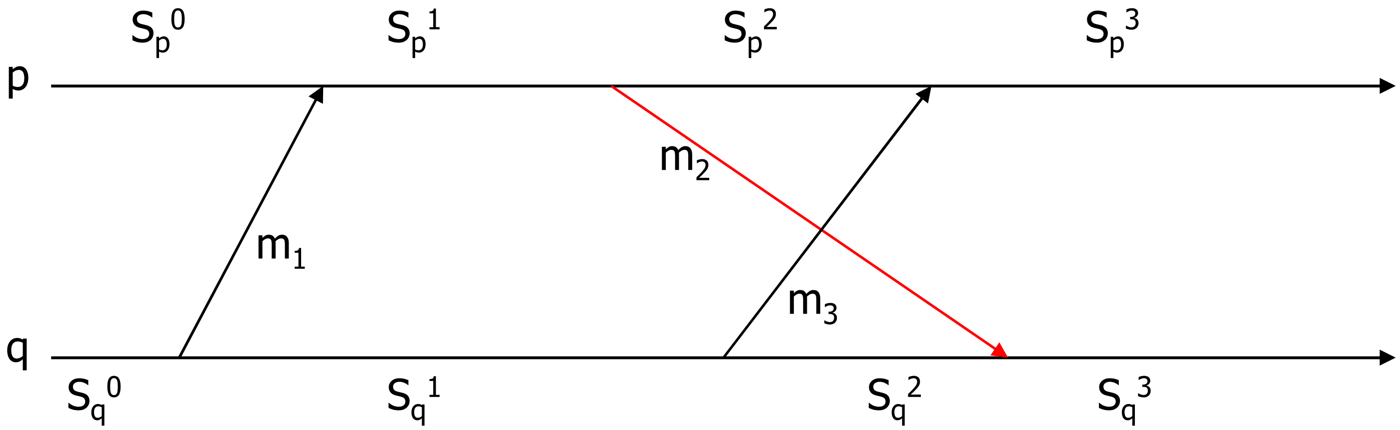
Yes





An Inconsistent State





Why Consistent Global State?

How to combine information from multiple nodes, that the sampling reflects a global consistent state?

Problem:

- Local view is not sufficient
- Global view:
  - We need messages transfers to the other nodes in order to collect their local states
  - Meanwhile these local states can change again

Local History

- N processes P_i, P := $\{P_1, P_2, \dots P_n\}$, for each $P_{i:}$
  - On a separate node n_i
  - Event series = history h_i := $<e_{i,1}, e_{i,2}, \dots >$
  - May be finite or not
- Observing a local history h_i up to event $e_{i,k}$ you get: prefix of history $h_{i,k} := < e_{i,1}, e_{i,2}, \dots , e_{i,k} >$
- Each $e_{i,k}$ is either a local or a communication event
- Process state:
  - State of P_i immediately before $e_{i,k}$ denoted $s_{i,k}$
  - State $s_{i,k}$ records all events included in history $h_{i,k-1}$
  - Hence, $s_{i,0}$ refers to P_i 's initial state

Global History and Global State

- Global history $h := h_1 \cup h_2 \cup \ldots \cup h_{n-1} \cup h_n$
- Similarly we can combine a set of local states to form a global state $S := (s_1, s_2, \ldots s_n)$
- However, which combination of local states is consistent?

Cuts

- Similar to the global state, we can define cuts based on k-prefixes:
- $C := h_{1,c1} \cup h_{2\ ,c2} \cup \ldots \cup h_{n-1,cn-1} \cup h_{n,cn}$
- $h_{1,c1}$ is history up to and including event $e_{1,c1}$
- The cut C corresponds to the state

  $S = (s_{1,c1+1}, s_{2,c2+1}, \ldots s_{n,cn+1})$
- The final events in a cut are its frontier or its border line :

  $BL = \{e_{i,ci} \mid i \in \{1,2, \ldots n\}\}$

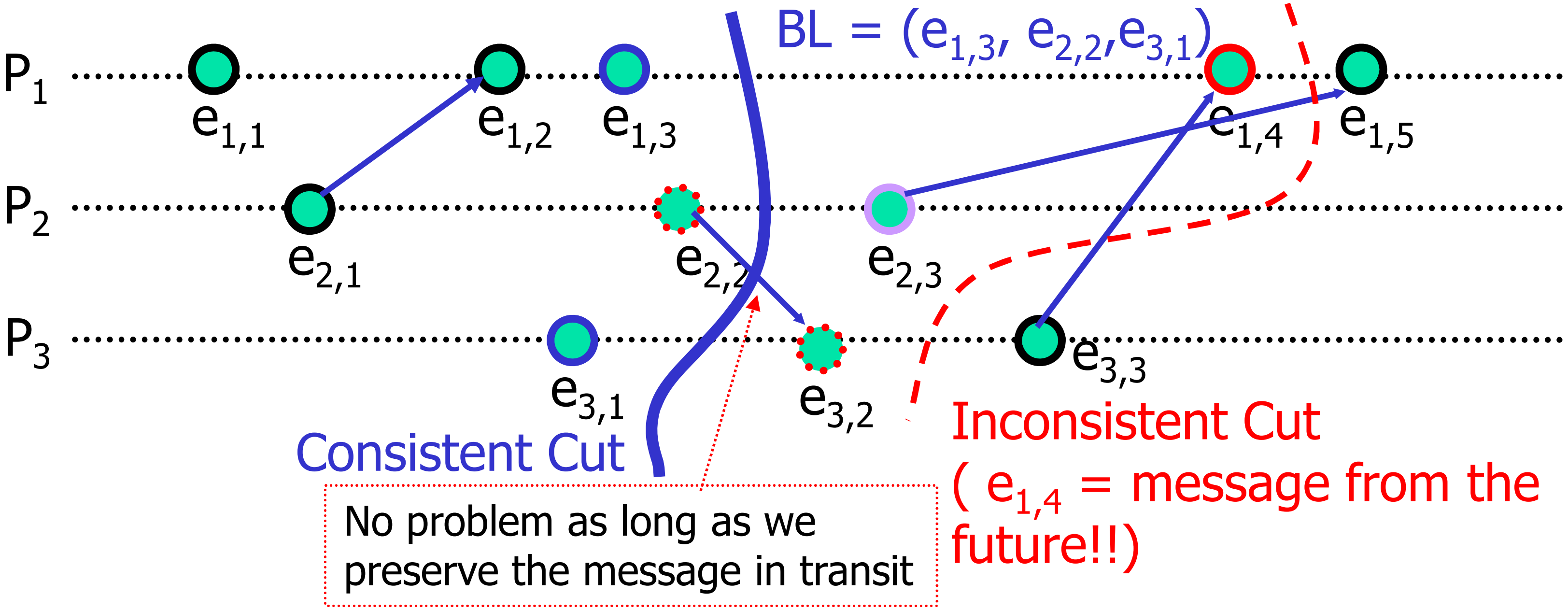
Distributed Snapshots

- Global state of system S:

  $S := (s_{1,c1}, s_{2,c2}, \ldots, s_{n,cn})$

  with the border line:
- $BL := (e_{1,c1}, e_{2,c2}, \ldots, e_{n,cn})$

Events have already happened



Consistent Cut

No problem as long as we preserve the message in transit

Inconsistent Cut ($e_{1,4}$ = message from the future!!)

Consistent Cuts

- We call a cut C consistent iff for all events $e' \in C$: $e \rightarrow e'$ implies $e \in C$

- A global state is consistent if it corresponds to a consistent cut

Remark:

- We can characterize the execution of a system as a sequence of consistent global states

Linearization

- A global history that is consistent with the “happened before” relation is also called a linearization or consistent run

- A linearization only passes through consistent global states

- A state S′ is reachable from state S′ if ∃ a linearization that passes through S and S′

Distr. Snapshot Algorithm (Chandy/Lamport)

Features:

Does not promise us to give us exactly what is there

But gives us consistent state!!

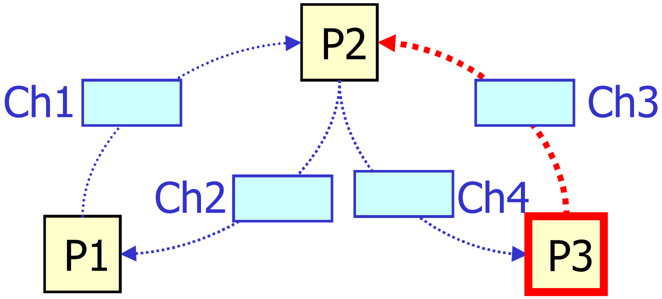
Brief Sketch of the Algorithm

- p sends a marker message along all its outgoing channels after it records its state and before it sends any other messages.
- On receipt of a marker message from input channel c
  - if p has not yet recorded its process state
    - record the local process state
    - state (c) = EMPTY
  - else
    - state (c) = messages received on c since it had recorded its state excluding the marker.

Chandy/Lamport Algorithm[1]

Requirements:

1. No process failures, no message losses
2. Sequence of received messages is the same as sequence of sent messages
3. Bidirectional channels with FCFS property
4. Network is a strongly connected graph
   - From each process there is a connection path to each other process



---

[1] published 1985

Chandy Lamport Algorithm (2)

- Each process can initiate CLA to get a new global state
- 2 types of messages
  - marker messages
  - application messages
- First marker message is for saving local process state
- Next marker messages are for saving the other input channel states

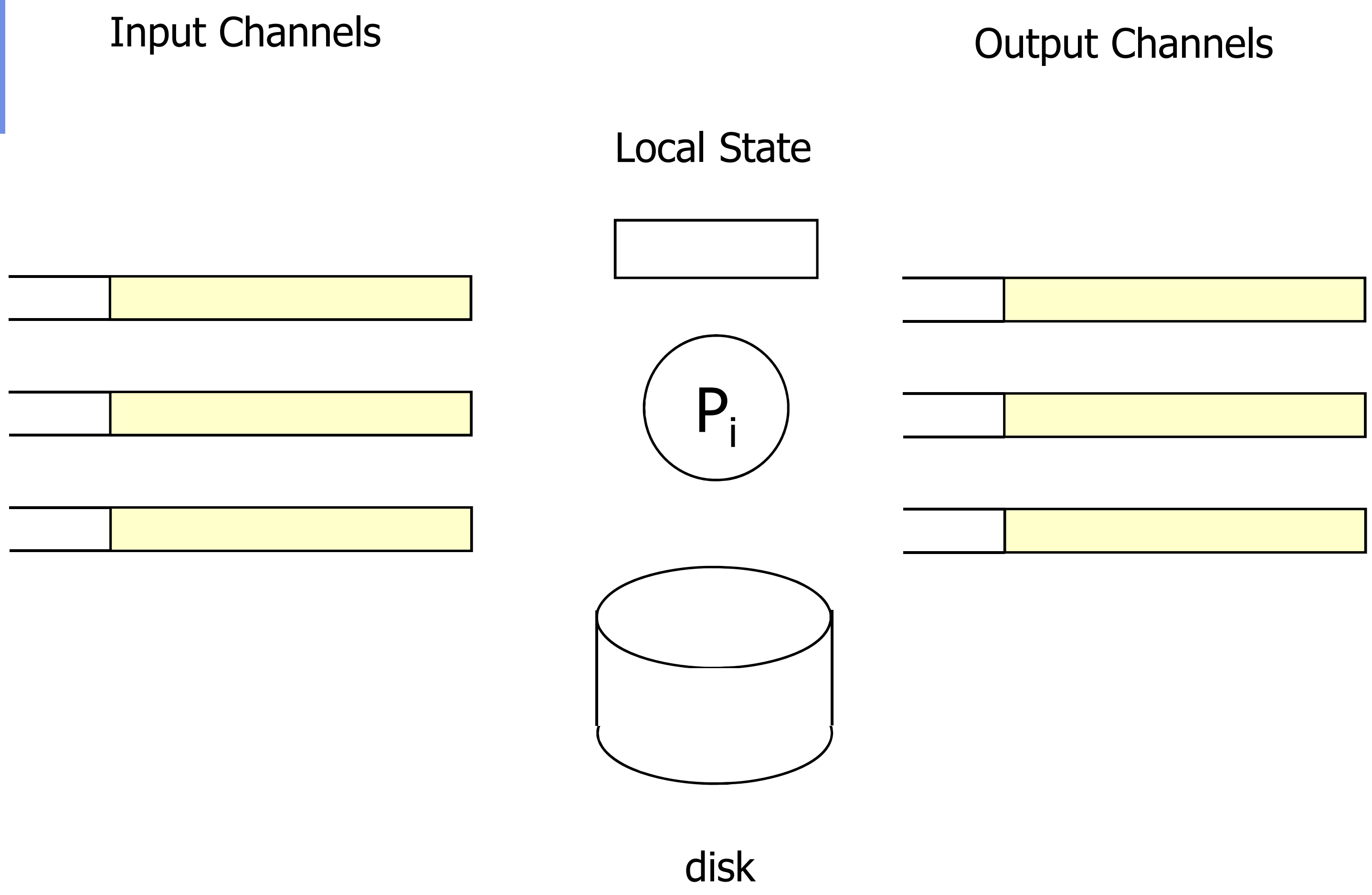
Principle of Operation

- Initially broadcast a marker message that contains a unique snapshot id (e.g. initiator id + sequence #) in order to differ from concurrent snapshot initializations
- Process Q receiving a marker message for the first time from input channel ic:
  - If not yet done, records its local process state
  - Define input channel state ic = EMPTY
  - Q sends the marker message to all its other output channels
  - Continue with the local application process
  - Each received application message is queued in its corresponding message queue

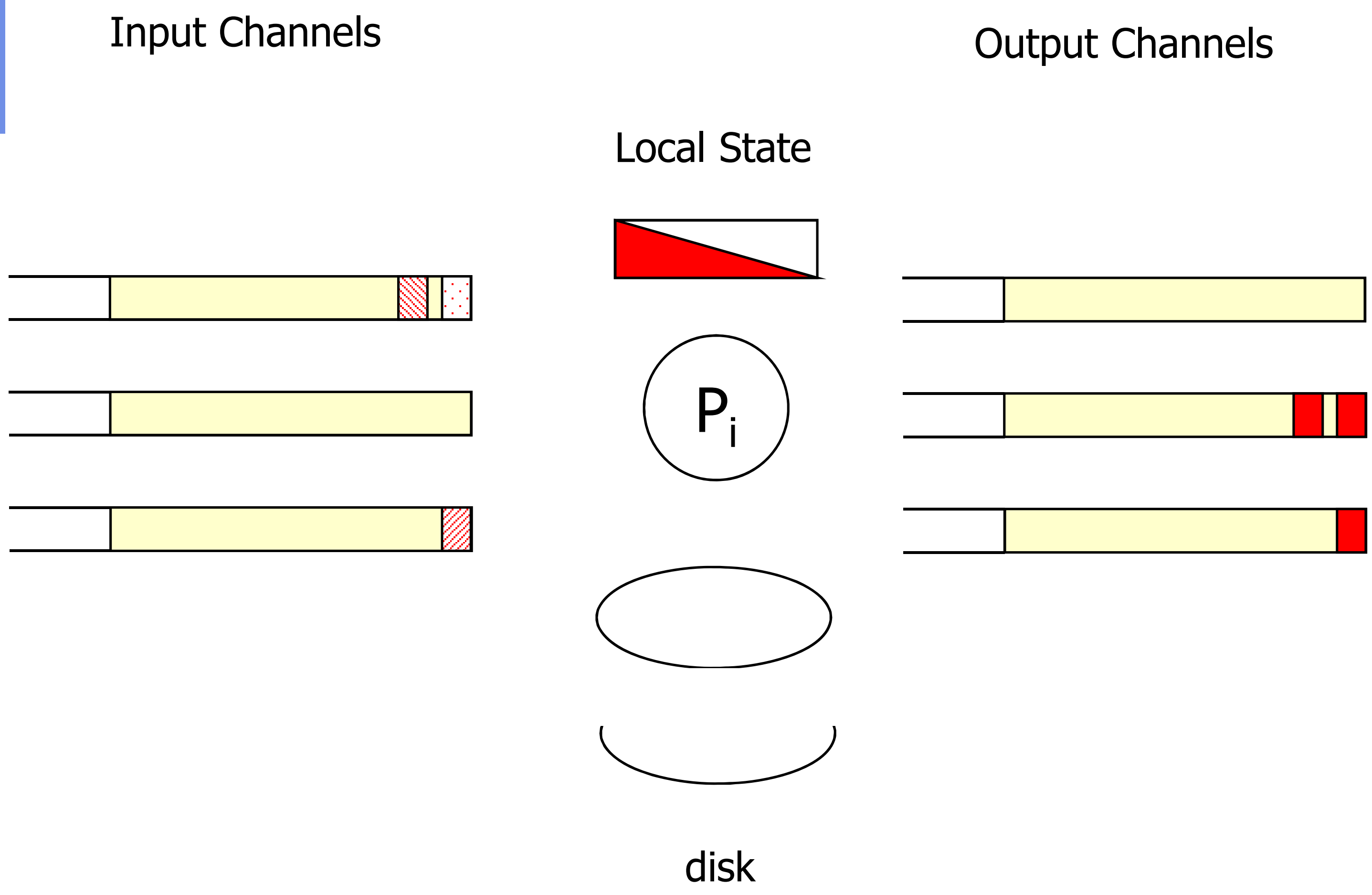
Principle of Operation

- Process Q receiving the marker message at another input channel CH_i
  - Terminates collection of messages at message queue MQ_i
  - Save and records state(CH_i) to local state of Q
  - If all incoming channels of Q have been saved and recorded, send aggregated local state of Q with all its input channels states to the initiator of the CLA

Chandy/Lamport (1)



Chandy/Lamport (2)



Input Channels

Output Channels

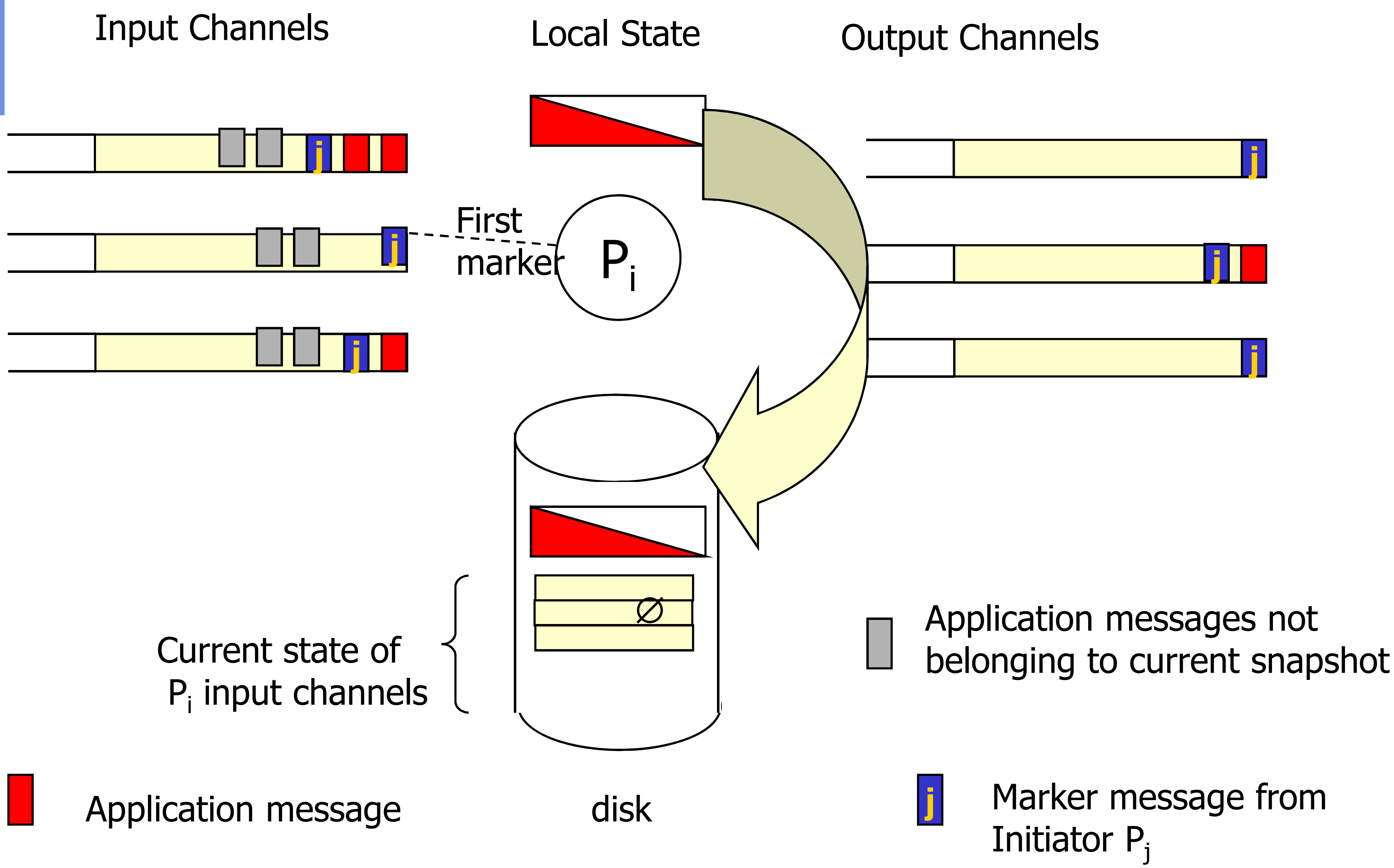
Local State

P_i

disk

🟥 Application messages

Chandy/Lamport (3)



Input Channels

Local State

Output Channels

First marker

P_i

Current state of P_i input channels

∅

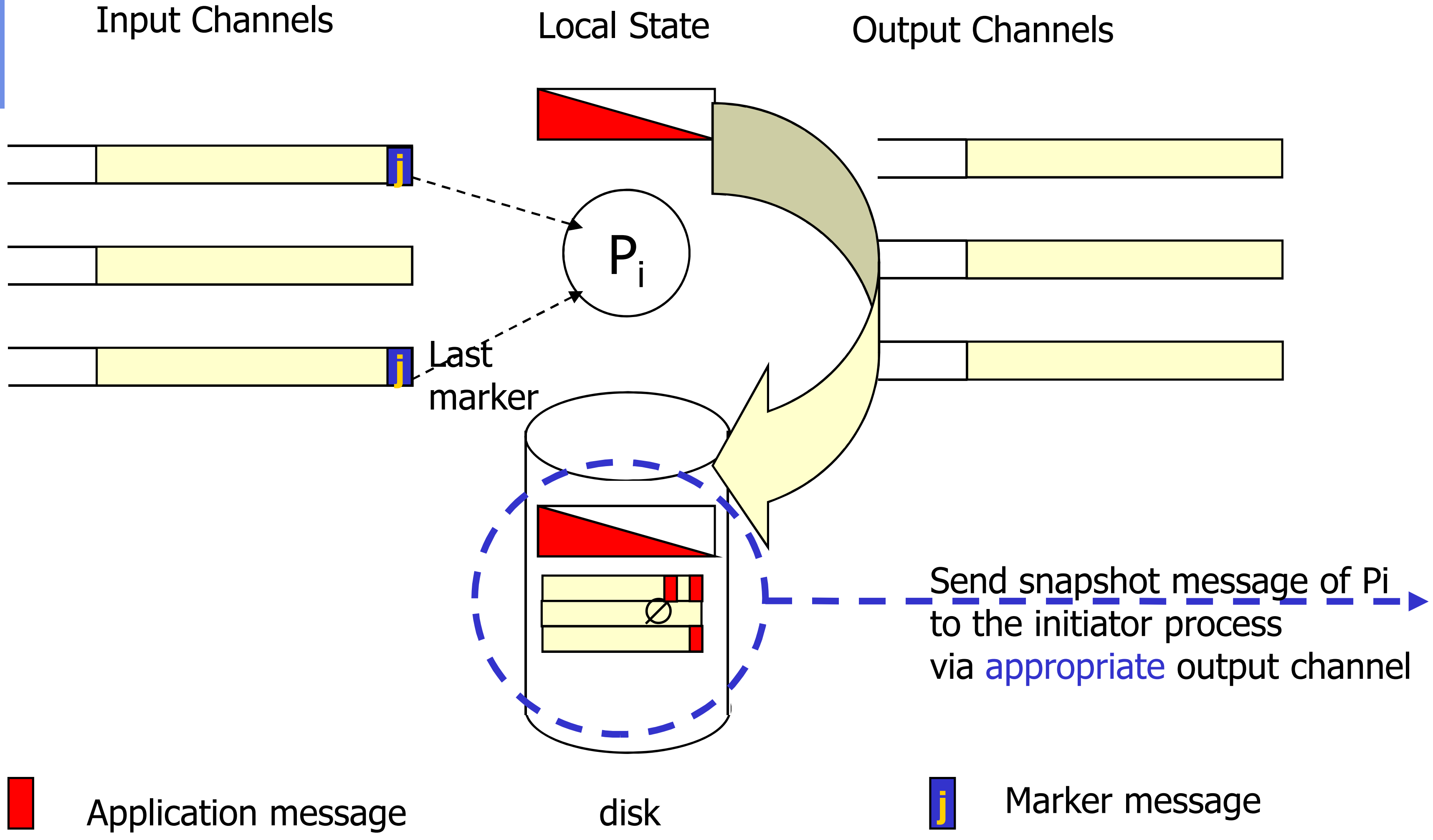
disk

🟥 Application message

▨ Application messages not belonging to current snapshot

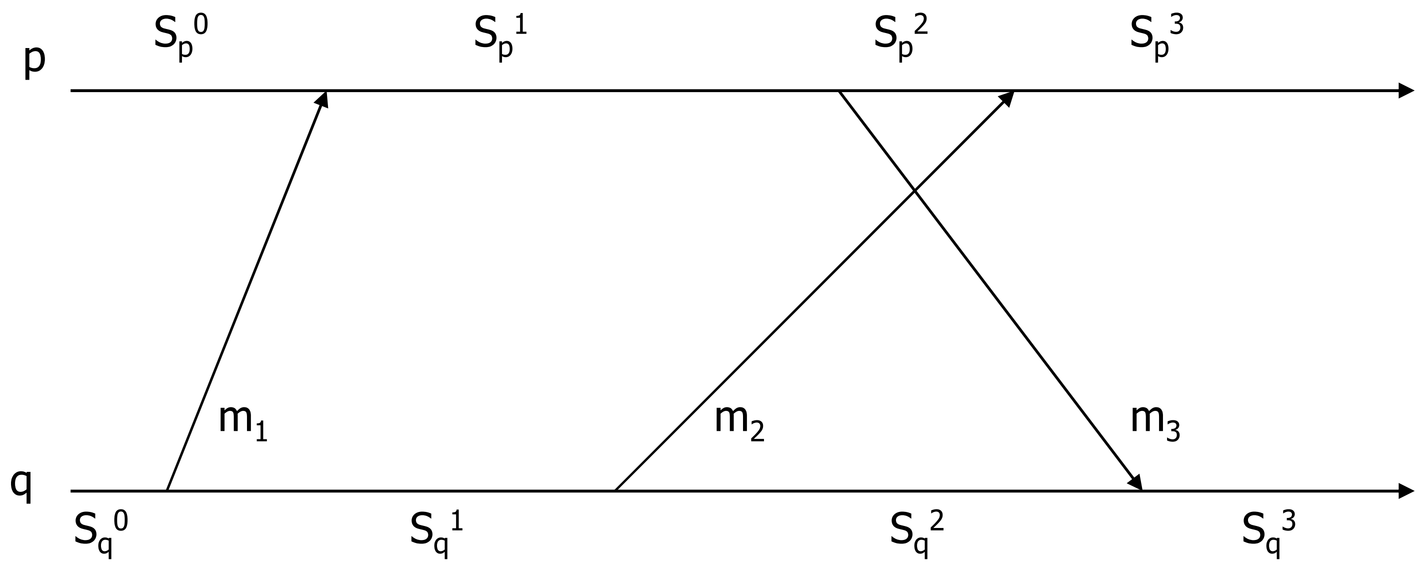
j Marker message from Initiator P_j

Chandy/Lamport (4)



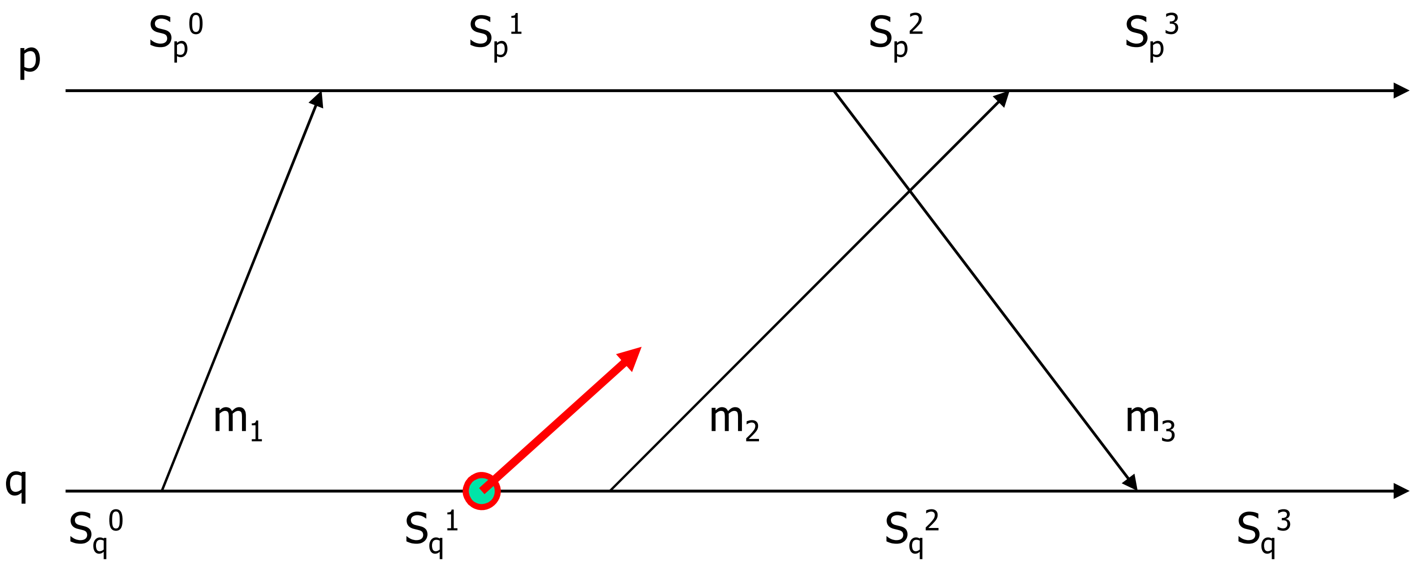
Send snapshot message of Pi to the initiator process via appropriate output channel

Algorithm in Action



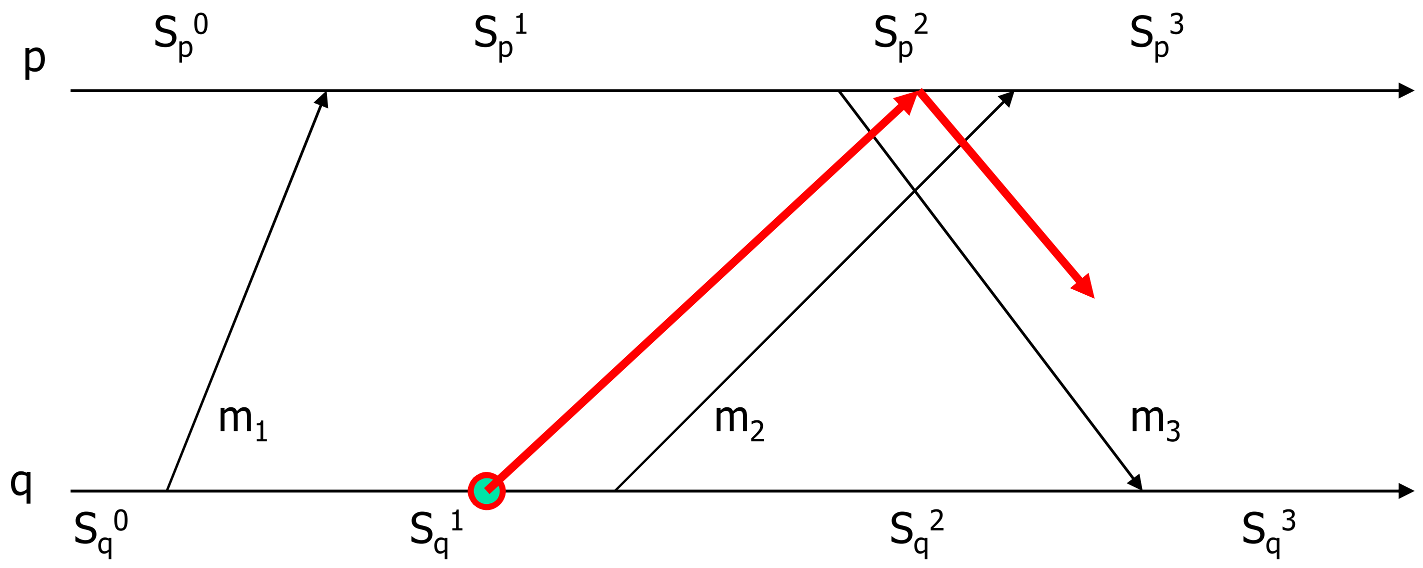
Algorithm in Action

q records state as S_q^1 , sends marker to p



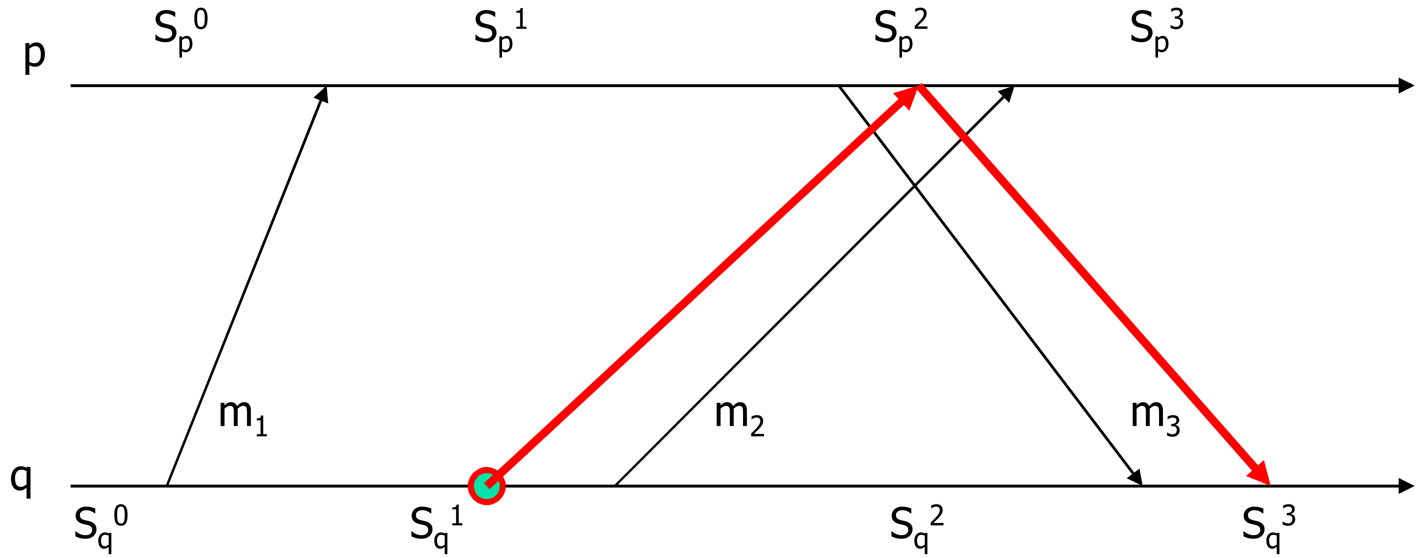
Algorithm in Action

p records state as S_p^2, channel state as empty



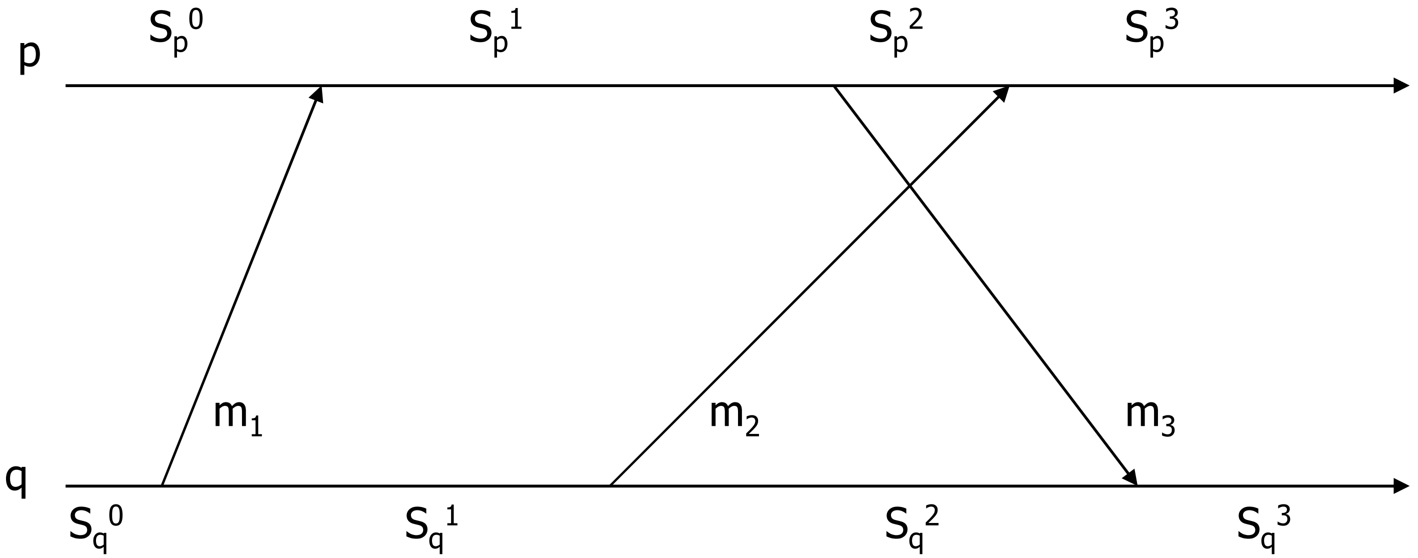
Algorithm in Action

q records channel state as m_3



Algorithm in Action

Recorded Global State = $((S_p^2, S_q^1), (0, m_3))$



Comment: Although application message m2 has been received in the meanwhile, this message does not belong to the global state initiated by q

Properties: Recorded Global State

- If S_i and S_j are the real global state when Lamport's algorithm started and finished respectively and S^* is the state recorded by the algorithm then,
    - S^* is reachable from S_i
    - S_j is reachable from S^*

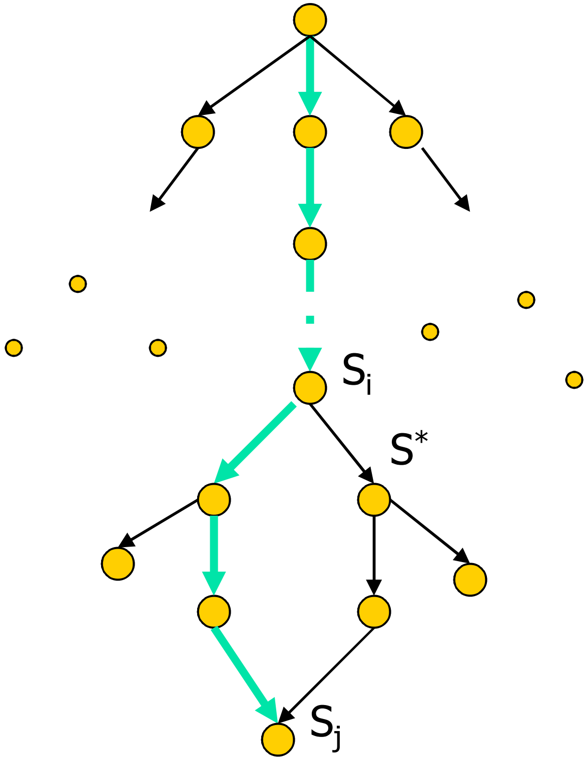
Still what good is it?

- Stable Properties
  - A property SP is called a stable property iff for all states S′ reachable from S

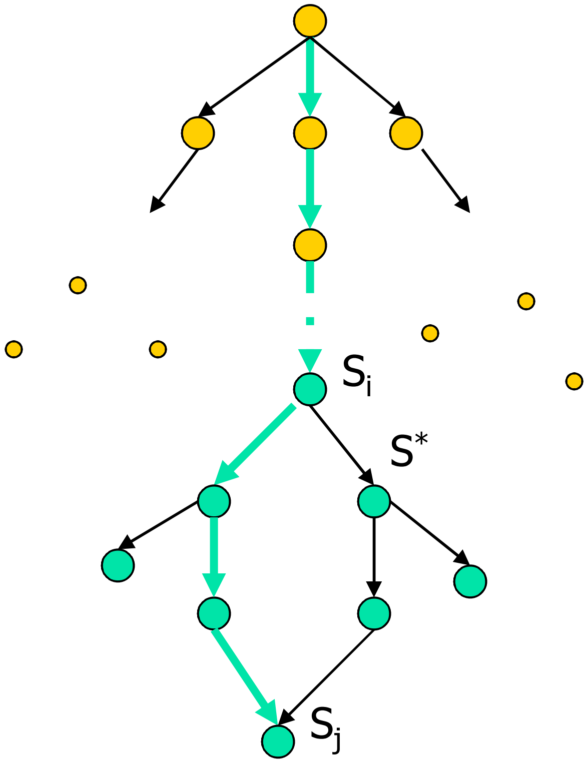
$$SP(s) \rightarrow SP(S')$$

  - eg: deadlock, termination, token loss

Stable Properties



Stable Properties

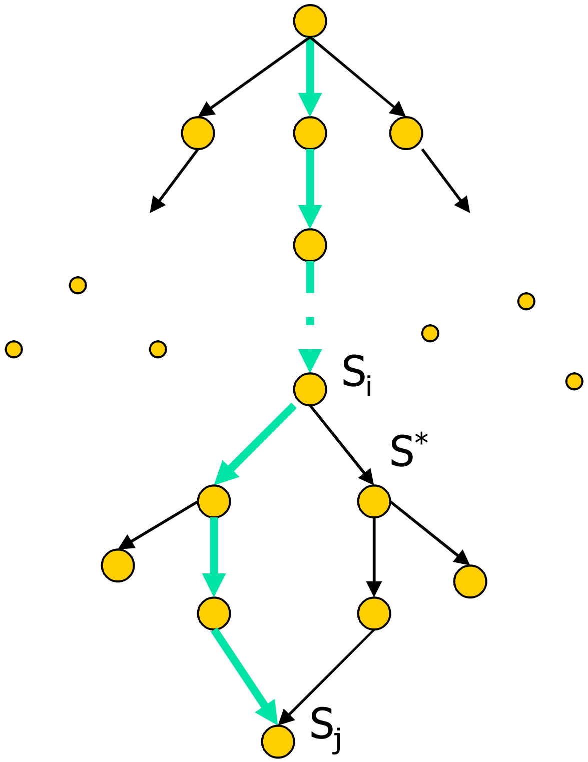


Detection of Stable Properties

```
Outcome = false;
while ( outcome == false )
{
        determine Global State S;
        outcome = SP(S);
}
```

Checkpointing

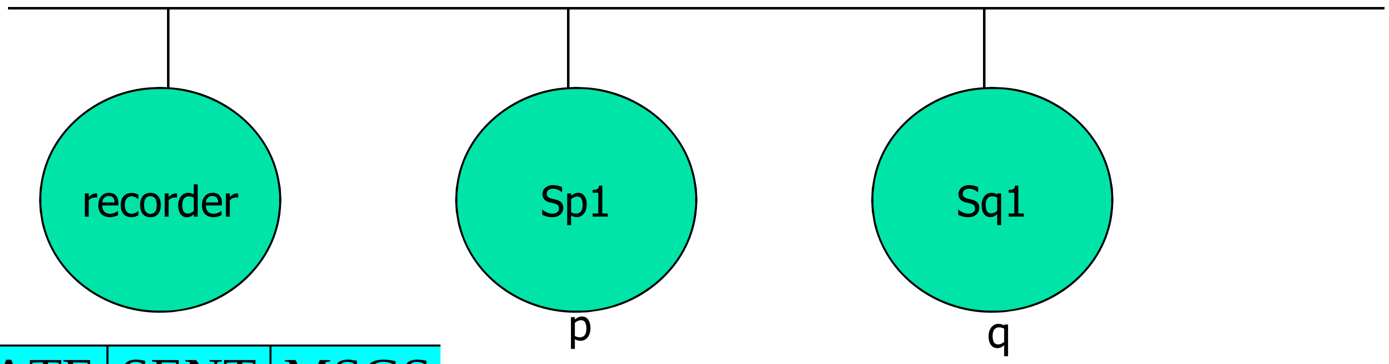
- S* serves as a checkpoint
- On a failure, restart the computation from S*
- Problem!
  - Not able to restore to Sj



Solution: Publishing

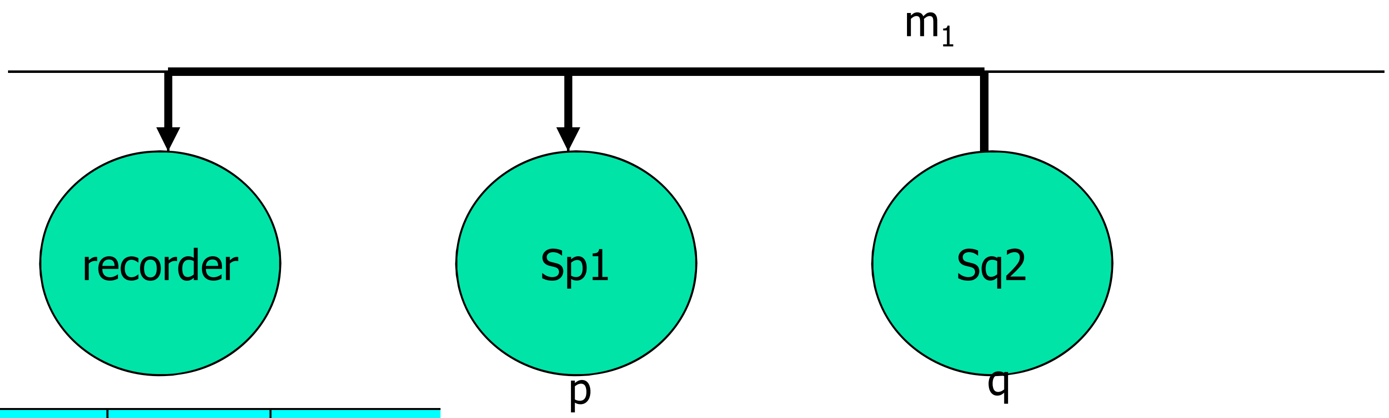
- A Broadcast medium
- A central ***recorder*** process records all the messages received by each process
- Processes record their states at their own time and send it to the recorder

Architecture of Publishing



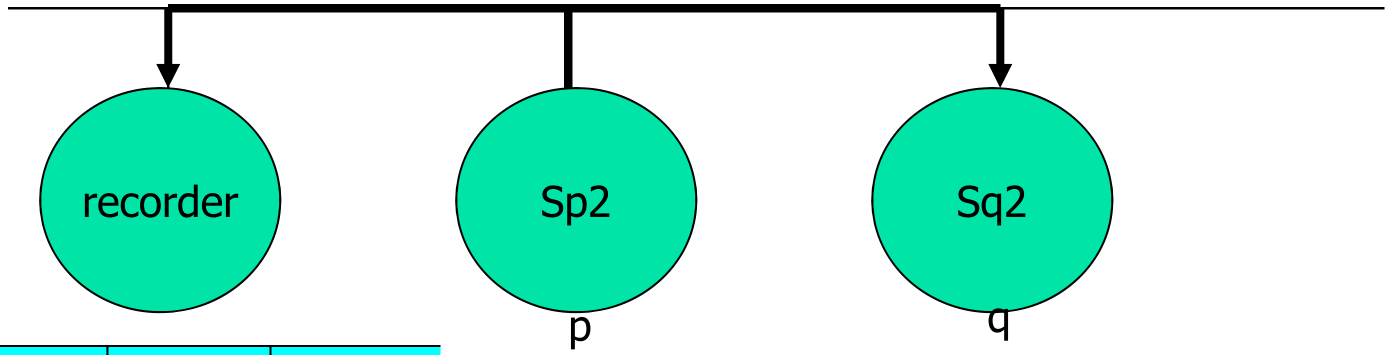
	STATE	SENT ID	MSGS RECD
p	Sp1		
q	Sq1		

q sends the message



	STATE	SENT ID	MSGS RECD
p	Sp1		
q	Sq1	1	

p sends an ack
recorder records m_1



	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Determining Global State

- Recorder can construct global state from
  - Checkpointed States of all processes

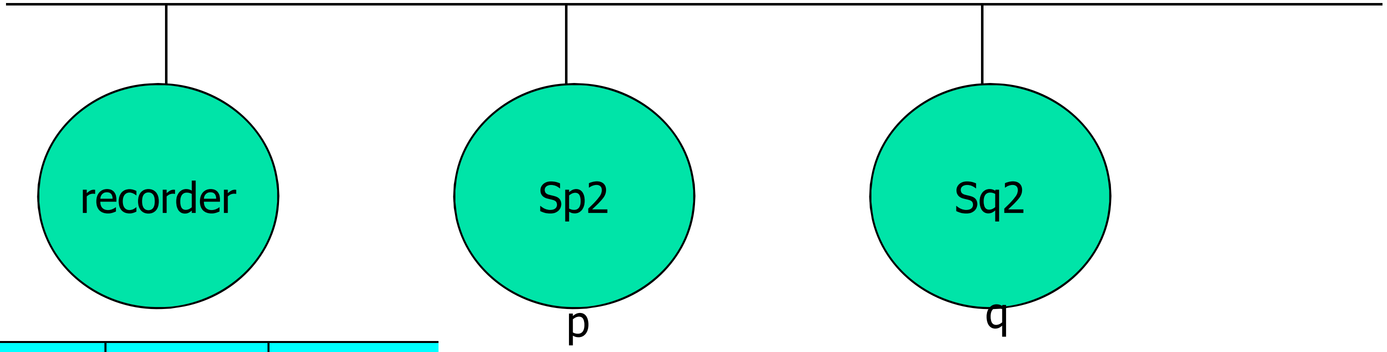
Plus

  - Messages recd since last checkpoint

Problems

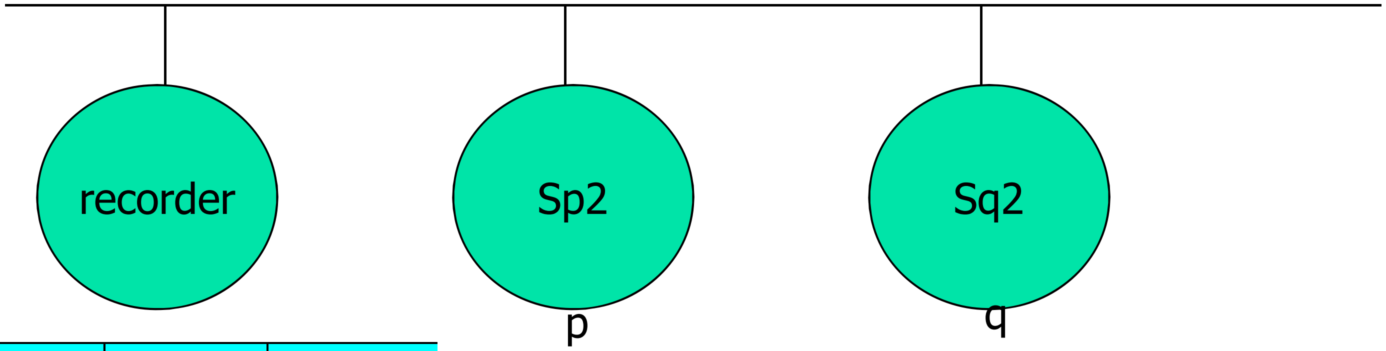
- Publishing keeps track of all messages received by each process
- Expensive!
- Solution
  - recorder takes checkpoint of process p at time t
  - deletes all messages recd by p before t.

p checkpoints



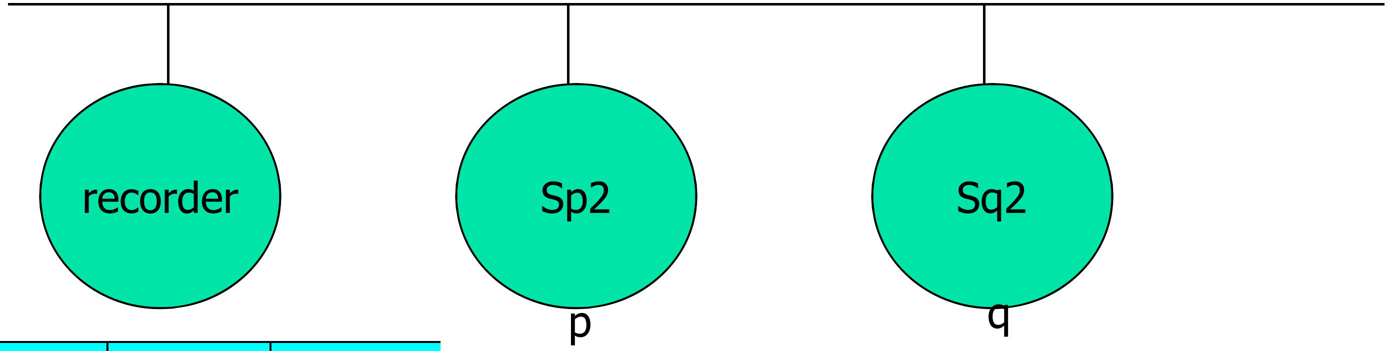
	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Recorder stores Sp2 deletes m_1



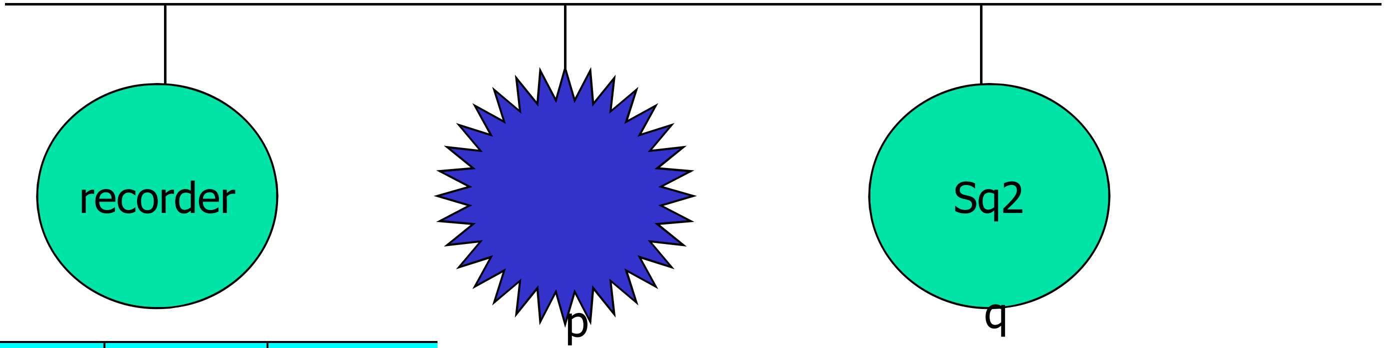
	STATE	SENT ID	MSGS RECD
p	Sp2		
q	Sq1	1	

The initial situation



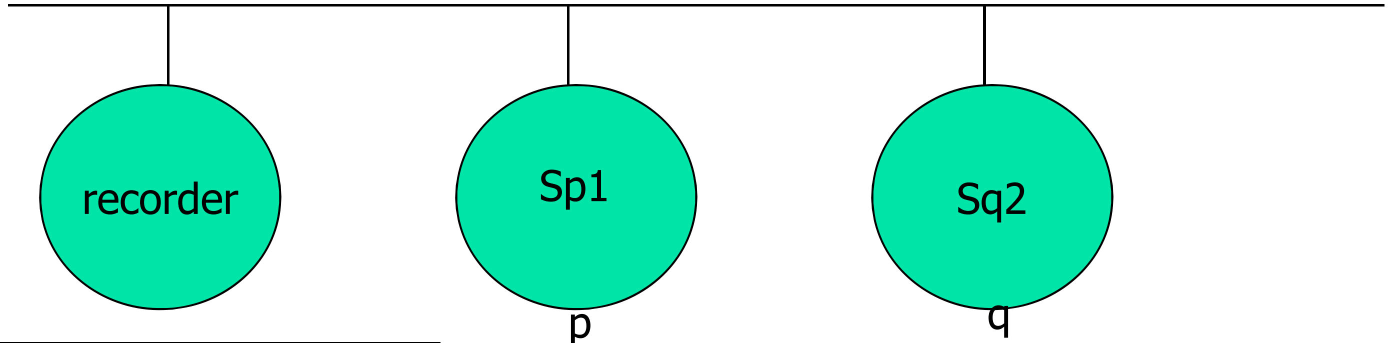
	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Say p crashes



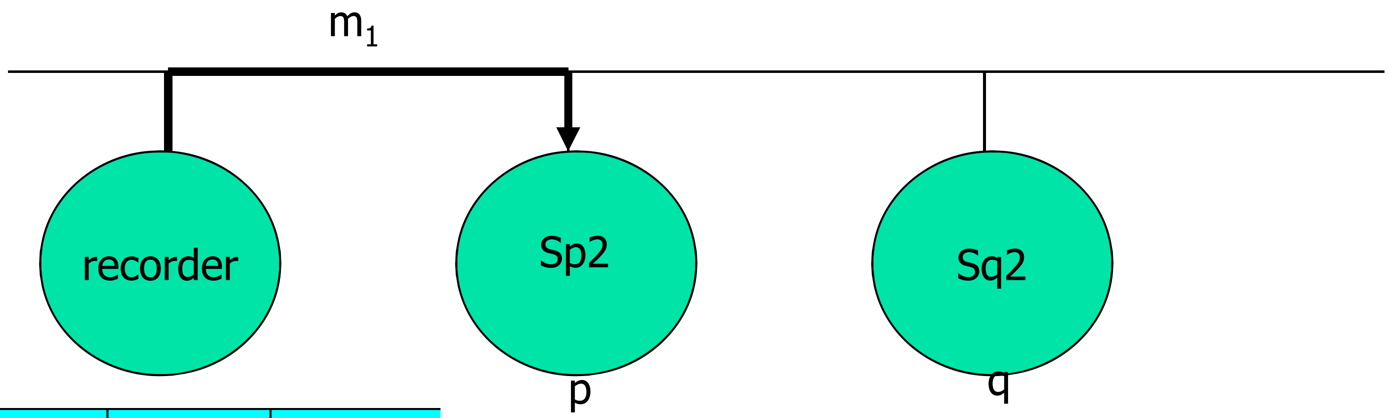
	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Recorder reinstates p to Sp1



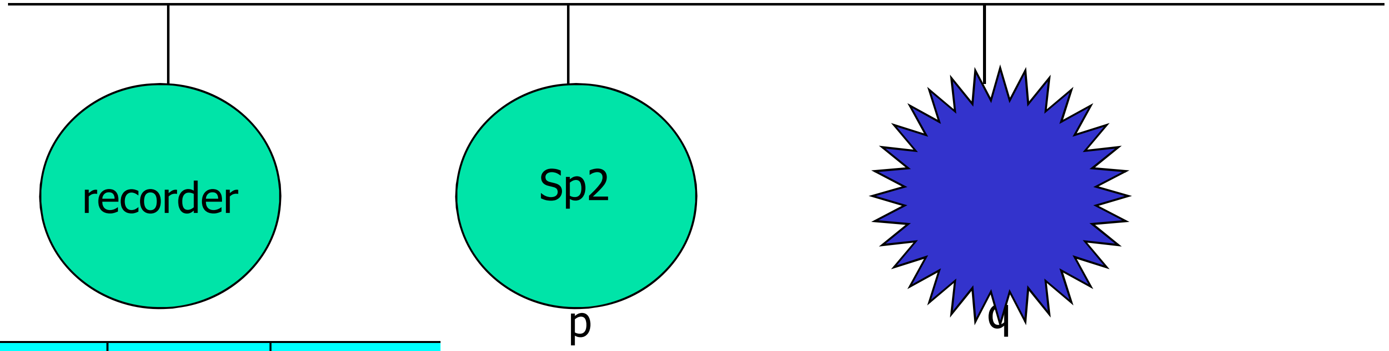
	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Replays back m_1



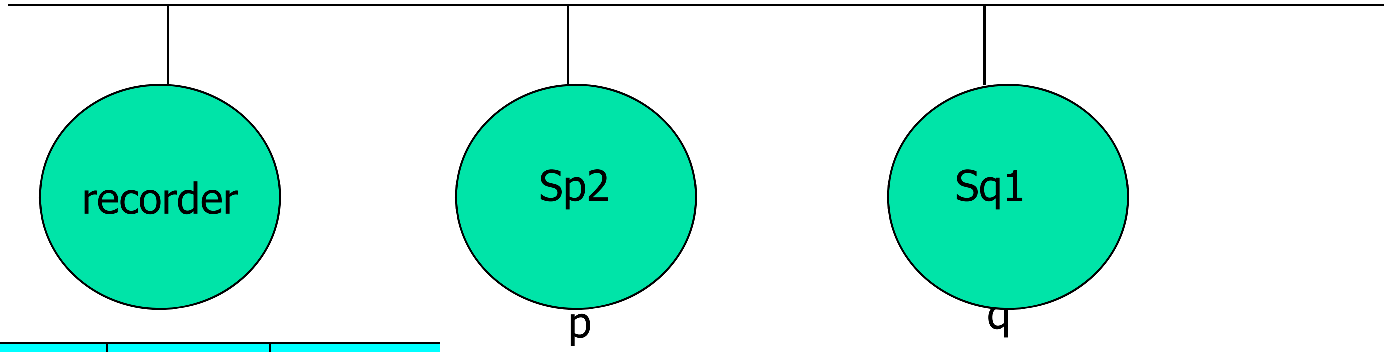
	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

q crashes



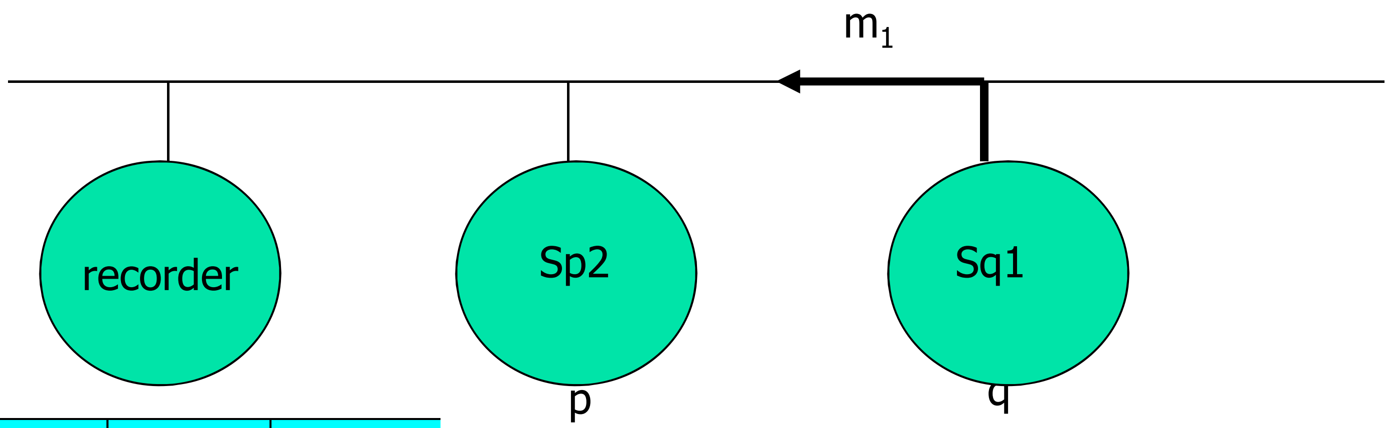
	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Recorder reinstates q to Sq1



	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Ignore m_1



	STATE	SENT ID	MSGS RECD
p	Sp1		m_1
q	Sq1	1	

Comparison

	SNAPSHOT	PUBLISHING
Network	Strongly connected	Need not be
Mode	Distributed	Centralized
Scalability	Yes	No
Restorability	No	Yes

Summary

- Global state detection is difficult in DSs
- Chandy/Lamport’s snapshot algorithm may not give an actual state but is very helpful in detecting stable properties
- Publishing gives an asynchronous way of determining global states but is not realy scalable

Mutual Exclusion

Centralized Algorithm
Decentralized Algorithm
Token Ring Algorithm
Distributed Algorithm

Mutual Exclusion in Local OS

Well known problem in multitasking OSes, e.g.

- access to shared memory, e.g.
  - Buffers
  - Global variables …
- access to shared resources
- access to shared data

- ∃ various centralized mechanisms to ensure mutual exclusion, e.g.
  - Semaphores
  - Monitors
  - Spin locks

Requirements: Mutual Exclusion

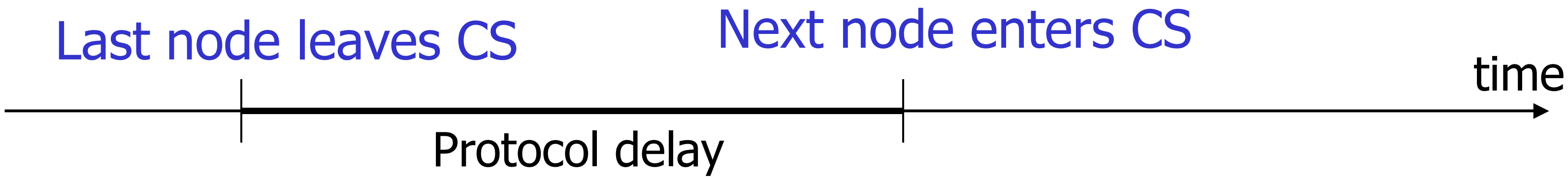
No Starvation
No deadlock

Requirements for a valid solution:

1. Safety: At most one process allowed to be in the CS
2. Liveliness (bounded Waiting): Each competitor must enter or exit its CS after some finite waiting time
3. Fair Ordering: Waiting in front of a CS is handled according to FCFS
4. Progress: Length on RS does not influence the protocol in front of a CS
5. Portability: Hard to achieve in a DS
6. Fault tolerance: We assume that messages are delivered correctly, e.g. only once and after some finite delay

Performance Criteria

- Number of needed messages per critical section CS, minimal n_m
- Protocol delay (to evaluate who is the next) per CS, minimal d



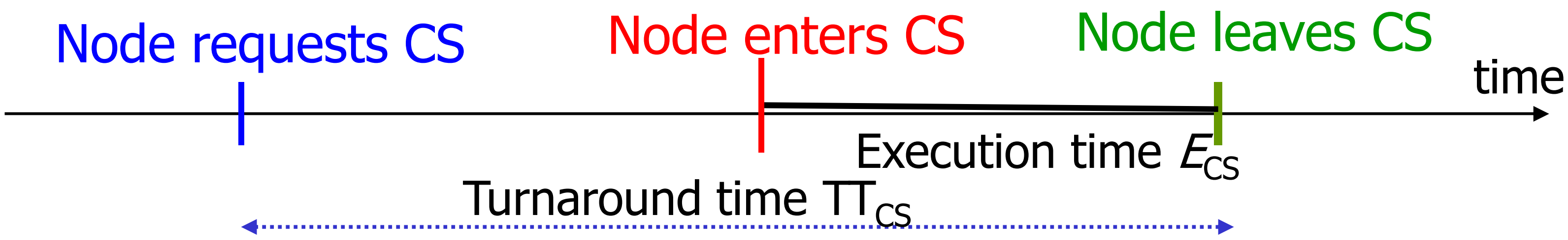
Last node leaves CS

Next node enters CS

time

Protocol delay

- Turnaround time TT_{CS}, time interval between requesting to enter a CS and leaving the CS, minimal TT_{CS}



Node requests CS

Node enters CS

Node leaves CS

time

Execution time E_{CS}

Turnaround time TT_{CS}

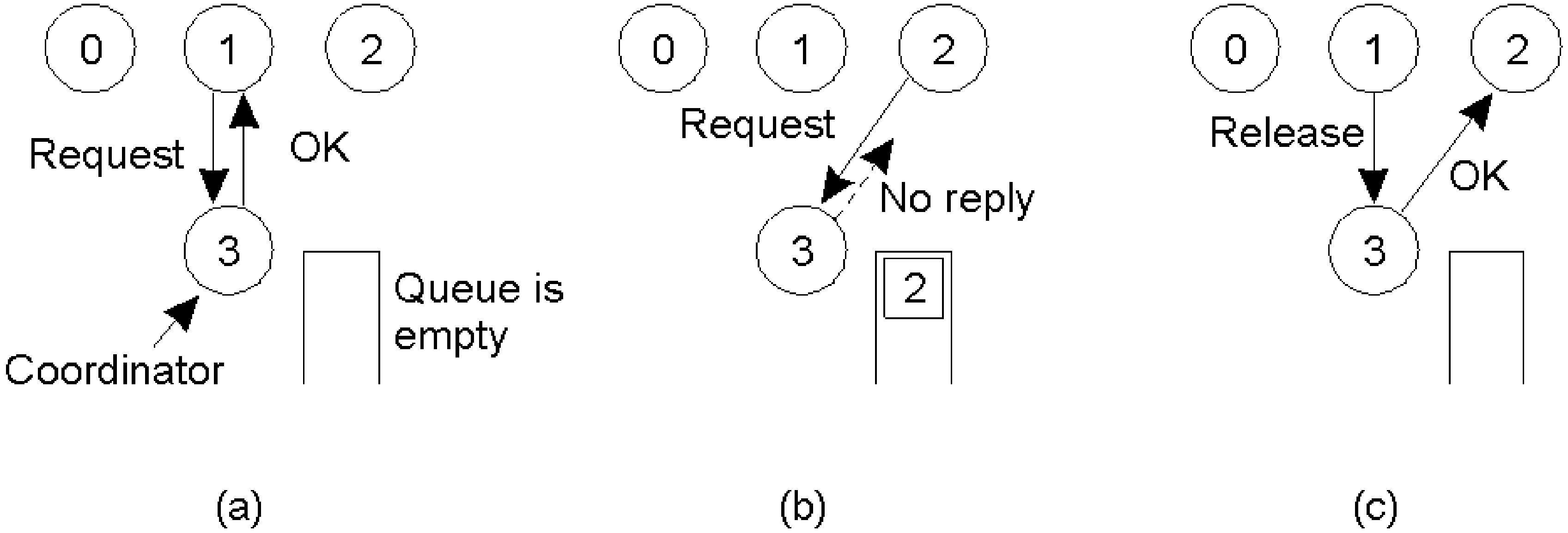
- Throughput TP_{CS}, # passing a CS per time unit (maximize TP_{CS})
  $TP_{CS} = 1/(d + E_{CS})$

Centralized Lock Manager

Centralized Lock Manager CLM

- A specific process CLM per critical region is designated to be the lock manager for all competing application clients
- CLM controls accesses to CR using a grant token representing permission to enter
- To enter its CS, a client sends a request message to the CLM awaiting a positive answer from the CLM
- If no client has the token, CLM replies immediately with the grant token. Otherwise CLM queues this request
- Leaving the CS the client sends the grant token back to the CLM

A Centralized Algorithm

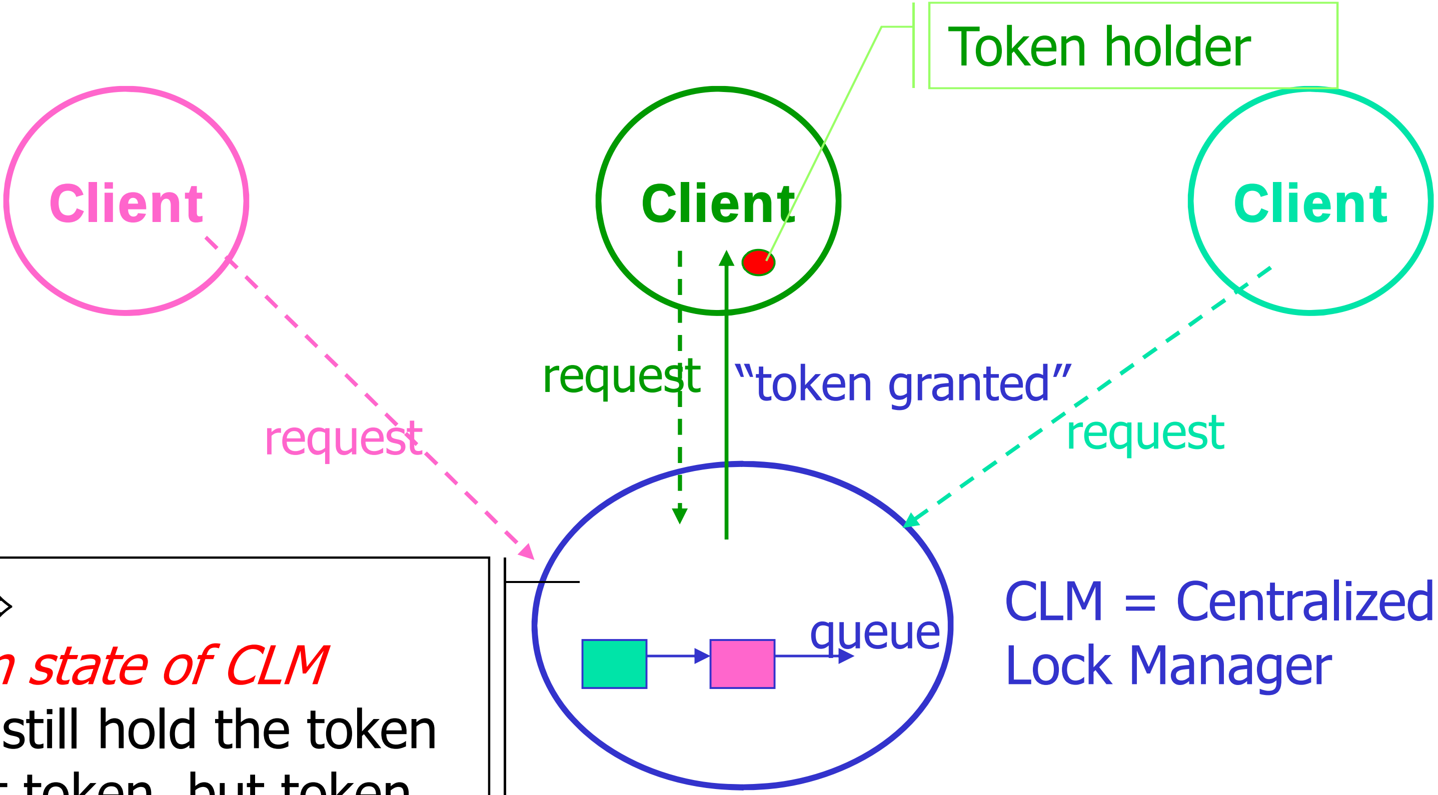


a) P1 asks CLM (P3) for permission to enter its CR ⇒ granted

b) P2 asks permission to enter same CR. CLM does not reply.

c) When P1 exits its CR, it notifies CLM that grants access to P2

Problems with Centralized Locking?



CLM = Centralized Lock Manager

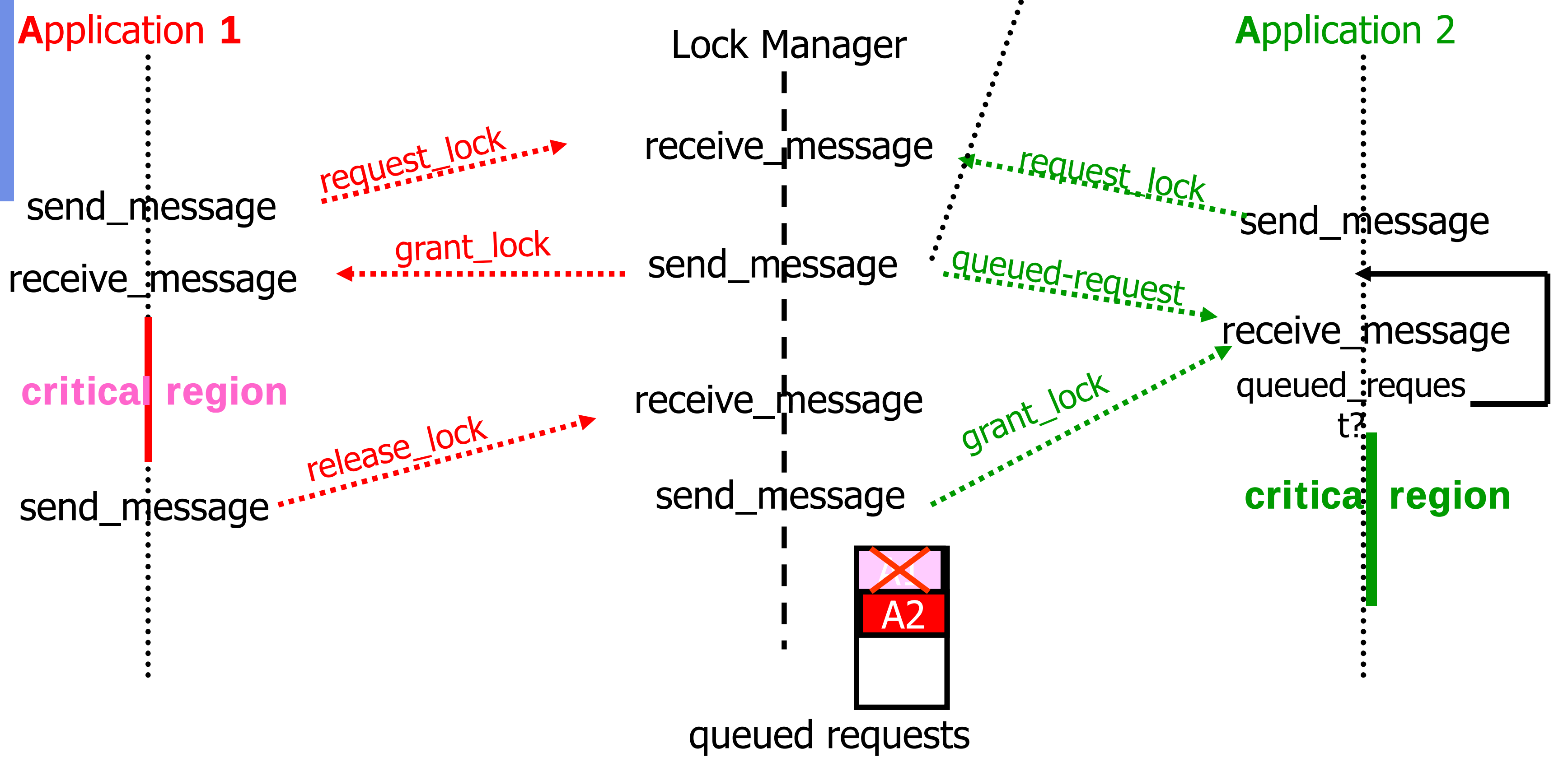
If CLM crashes ⇒

uncertain state of CLM

1. A client might still hold the token
2. Client has sent token, but token was not yet received at CLM
3. The CLM has the token
4. *How long would you wait, before electing a new CLM?*

Centralized Lock Manager

Queued message is optional
Benefits?



Application 1

Lock Manager

Application 2

receive_message

request_lock

request_lock

send_message

send_message

grant_lock

send_message

receive_message

queued-request

receive_message

queued_request?

critical region

receive_message

release_lock

grant_lock

send_message

send_message

critical region

A1

A2

queued requests

Disadvantages:

- single point of failure
- potential bottleneck

Summary on CLM

- Easy to implement
- *Scalability? Bottleneck?*
- Safety fulfilled
- Liveliness fulfilled
- Fair ordering not fulfilled: Without additional requirements concerning the network, request are not served in FCFS order
  - Adding logical time stamps per request might *improve* the situation, but still does not solve fair ordering
- Progress is fulfilled
- Fault tolerance: CLM might fail ⇒
  - Elect a new CLM (see election algorithms)

Performance Properties of CLM

- Per CS you need at least 3 messages
  1. Request from client to enter
  2. Reply from CLM that client can enter
  3. Notification from client that it has left CS

⇒ Turnaround time of CS is augmented by at least $3\ \Delta d + t_{CLM}$ if

  - Δd is the message transfer time
  - t_{CLM} is average execution time of CLM

What is the maximal delay in front of a CS?

Decentralized Algorithm

Lin’s Voting Algorithm in DHT DS.

“A Practical Distributed Mutual Exclusion Protocol in Dynamic P2P Systems”

Study of your one

Decentralized Mutual Exclusion

- Principle: n lock manager per CS (resource), i.e. the resources are replicated and each replica has its own lock manager

- A client can only access a resource if the majority of the n lock managers have sent a grant reply

- Each lock manager responds ”immediately” to a client’s request with grant or deny

- A client receiving a deny will retry again soon after

- When a lock manager crashes, it will recover quickly, but will have forgotten about permission it had granted in the past

Decentralized Mutual Exclusion

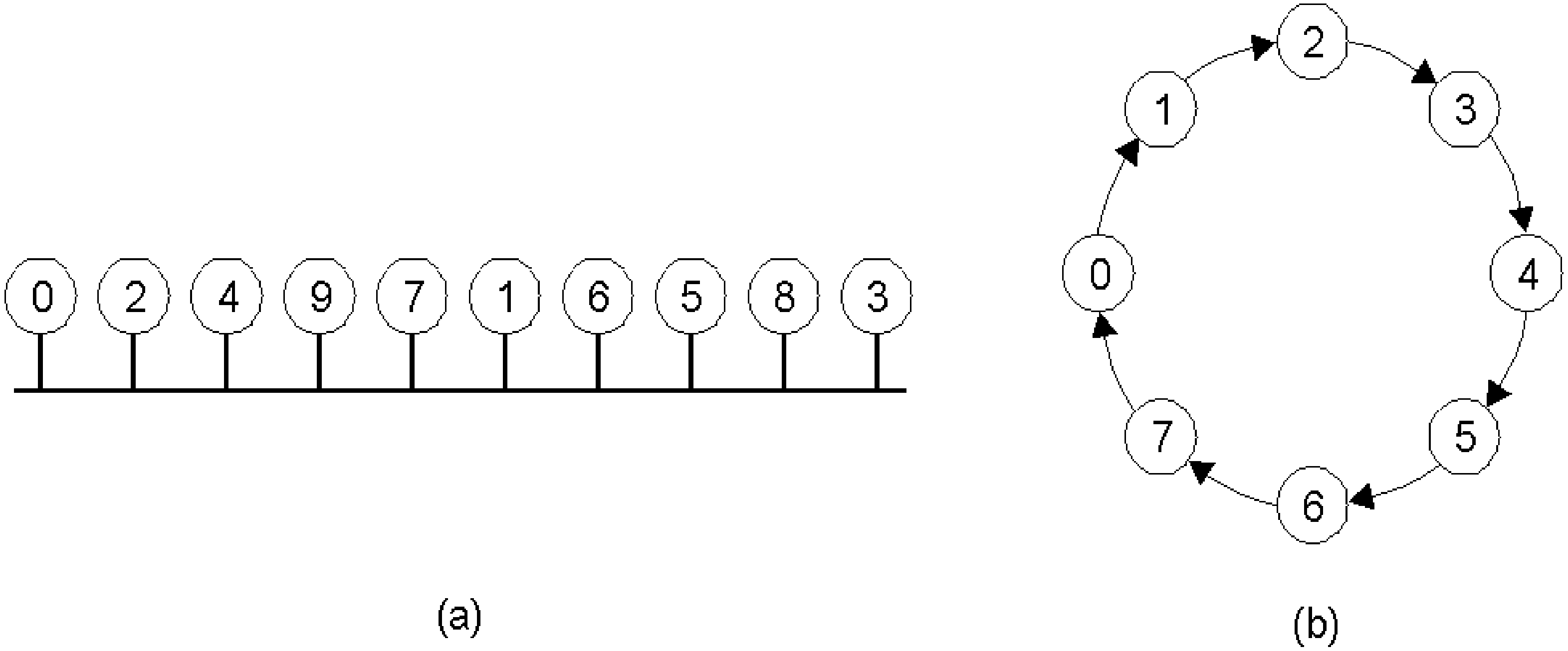
- Lin et al. showed that it is quite robust
- However, under heavy load, i.e. high concurrency in front of the CS (resources) no client will get the majority of the n lock managers, thus resulting in a poor performance

Algorithms based on Logical Structures

Token Ring

Tree Structured

Token Ring Algorithm



a) A group of processes on a network à la Ethernet

b) A logical ring (constructed in software)

Token-Passing Mutual Exclusion

The token-passing algorithm:

- A process can enter its CS iff it is the *current owner* of the access token
- When leaving its CS, the owner of the access token sends this token to its immediate successor

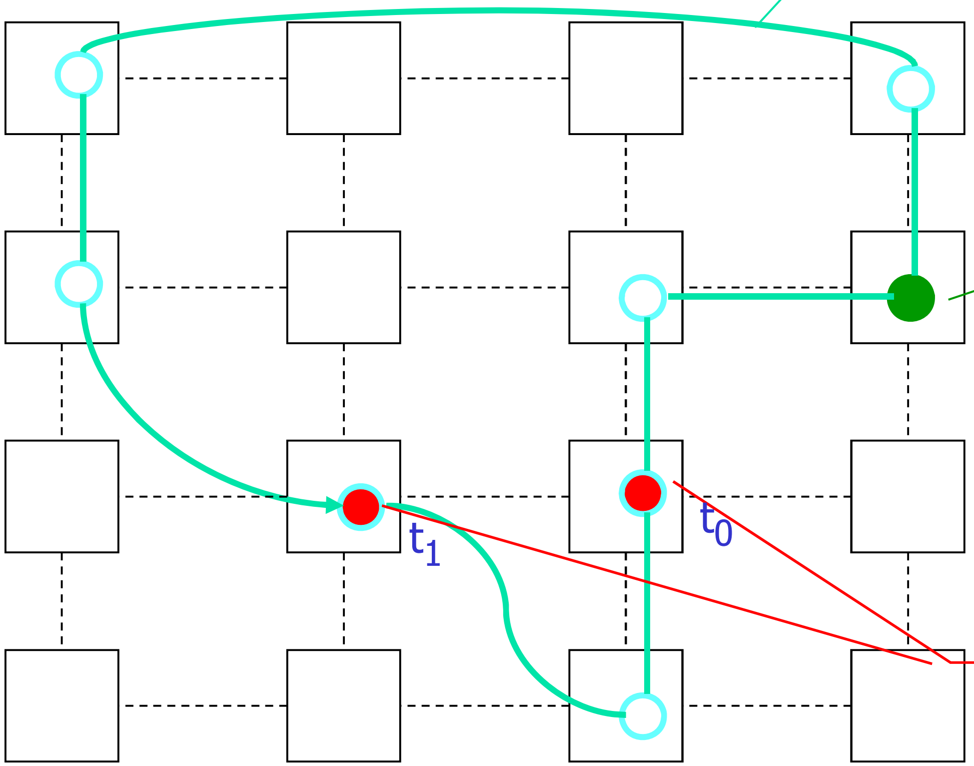
Observation:

In times when no participant wants to enter its CS, nevertheless the access token is circulating within the logical ring *reducing the bandwidth of the network*

⇒ **overhead**

Standard Token Algorithm

Given a lattice of nodes:

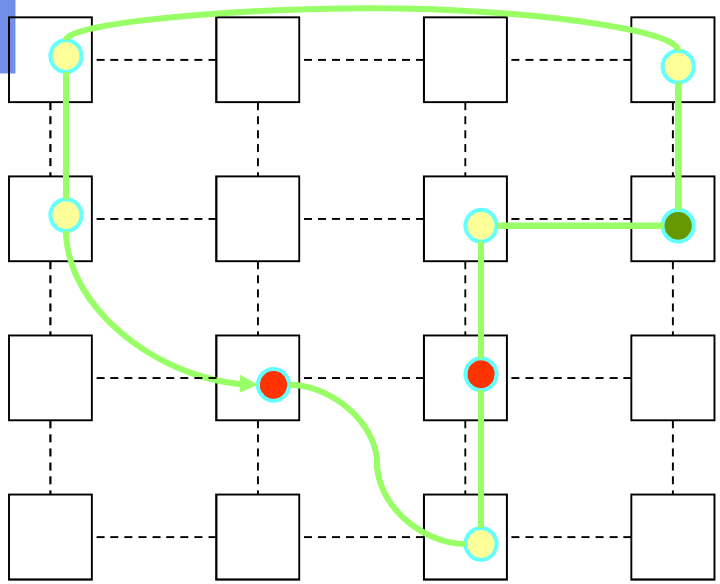


Logical Ring

Current Token Holder

Processes waiting in front of their critical sections CS ⇒ request are not served according to FCFS

Analysis of Token Based Exclusion



Check out the list of requirements:

1. Safety, *yes*, due to ***unique token***, only token holder may enter its CS
2. Liveliness, *yes*, as long as logical ring has a finite number of nodes
3. Sequence order, ***no***, TLM may change the internal order of the waiting requests
4. Fault tolerance?
   - splitting of the logical ring and you might be lost.
   - losing the token

Problems with Token-Algorithm

1. *How to distinguish if the token has been lost or if it is used very long?*

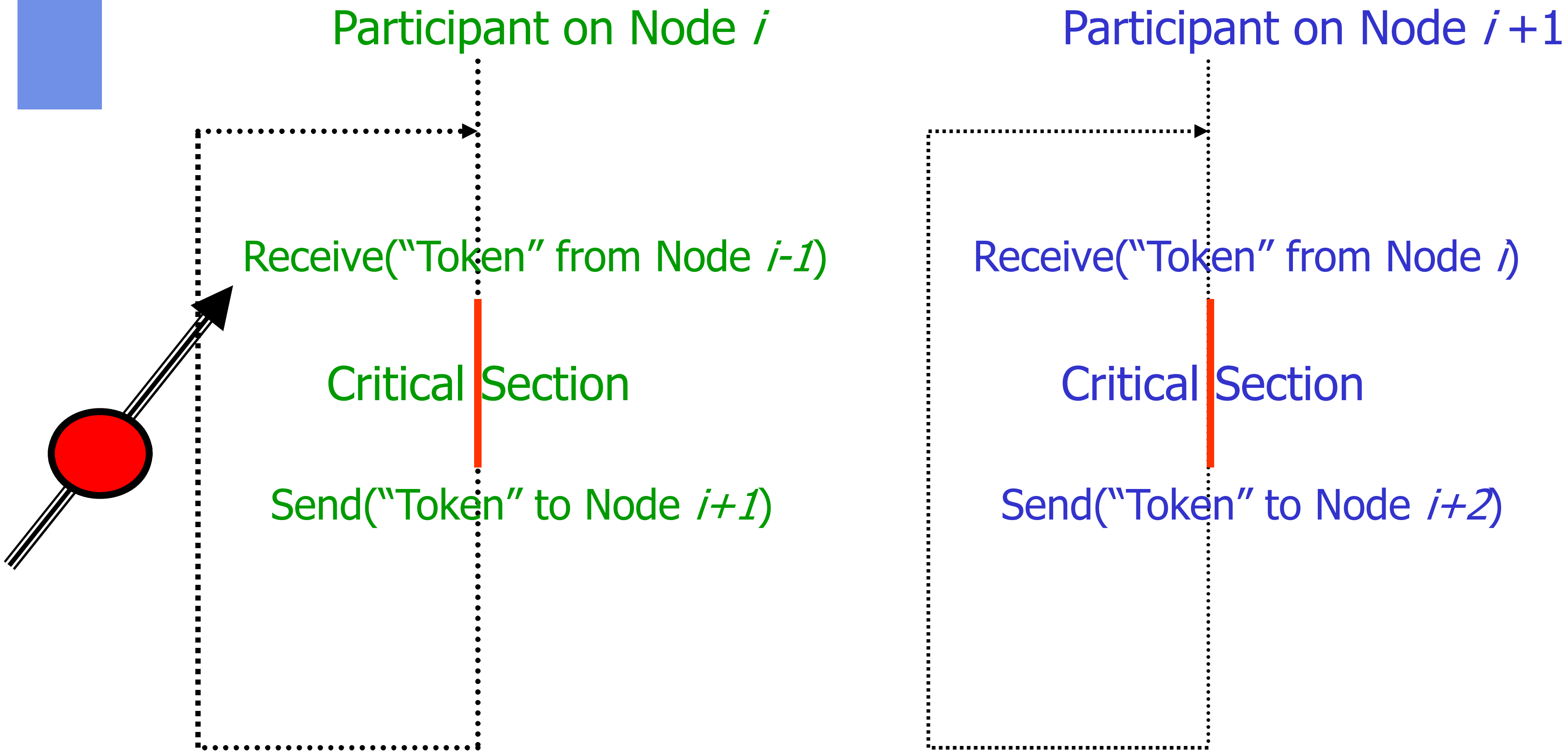
2. *What happens if token-holder crashes for some time and recovers later on?*

3. *How to maintain a logical ring if a participant drops out (voluntarily or by failure) of the system?*

4. *How to identify and add new participants?*

5. Ring imposes an average delay of N/2 hops ⇒ limiting scalability

Implementation Issues



Participant on Node *i*

Receive(“Token” from Node *i-1*)

Critical Section

Send(“Token” to Node *i+1*)

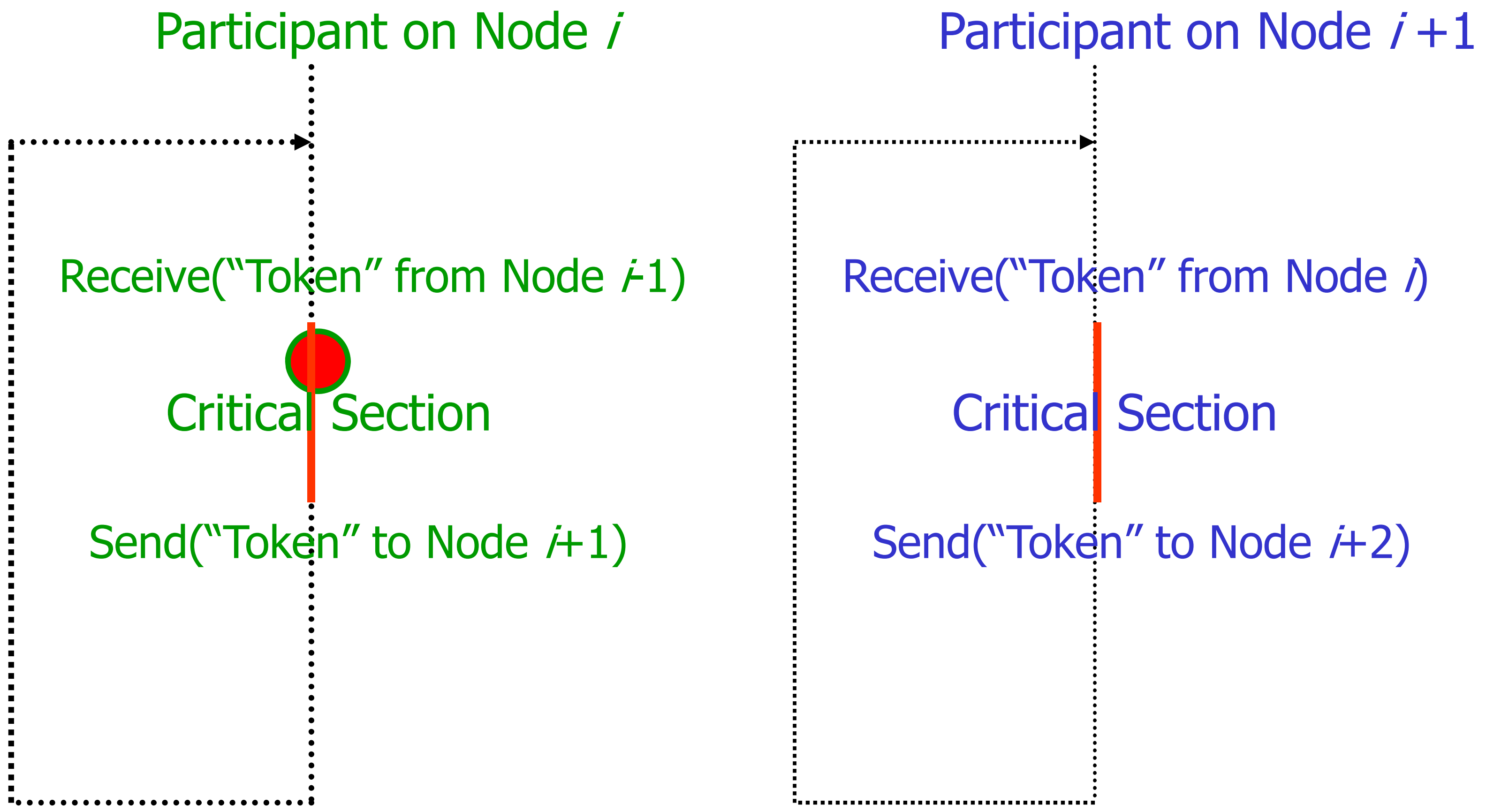
Participant on Node *i* +1

Receive(“Token” from Node *i*)

Critical Section

Send(“Token” to Node *i+2*)

Implementation Issues



Participant on Node *i*

Receive("Token" from Node *i*-1)

Critical Section

Send("Token" to Node *i*+1)

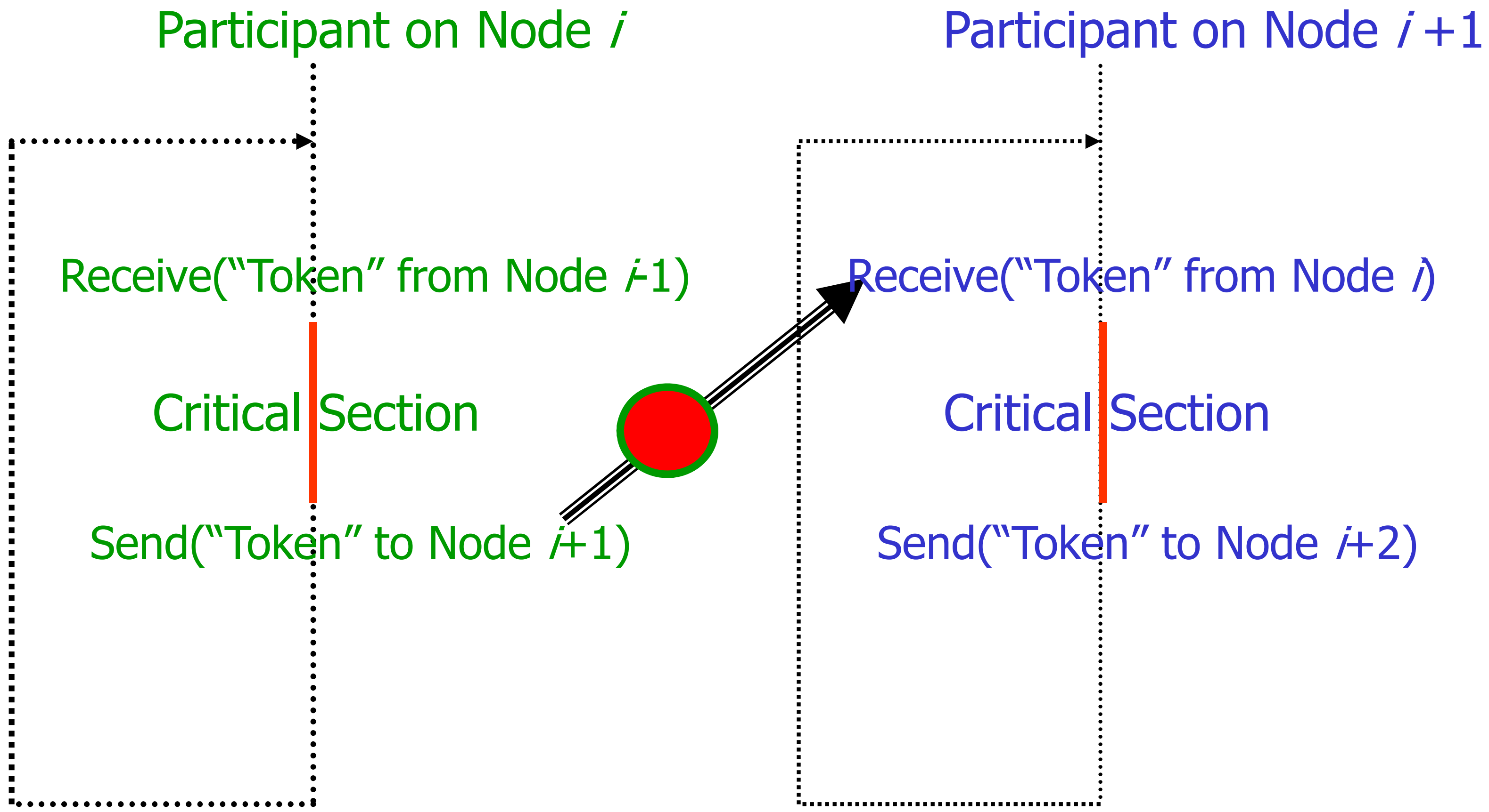
Participant on Node *i* +1

Receive("Token" from Node *i*)

Critical Section

Send("Token" to Node *i*+2)

Implementation Issues



Participant on Node *i*

Participant on Node *i* +1

Receive("Token" from Node *i*-1)

Critical Section

Send("Token" to Node *i*+1)

Receive("Token" from Node *i*)

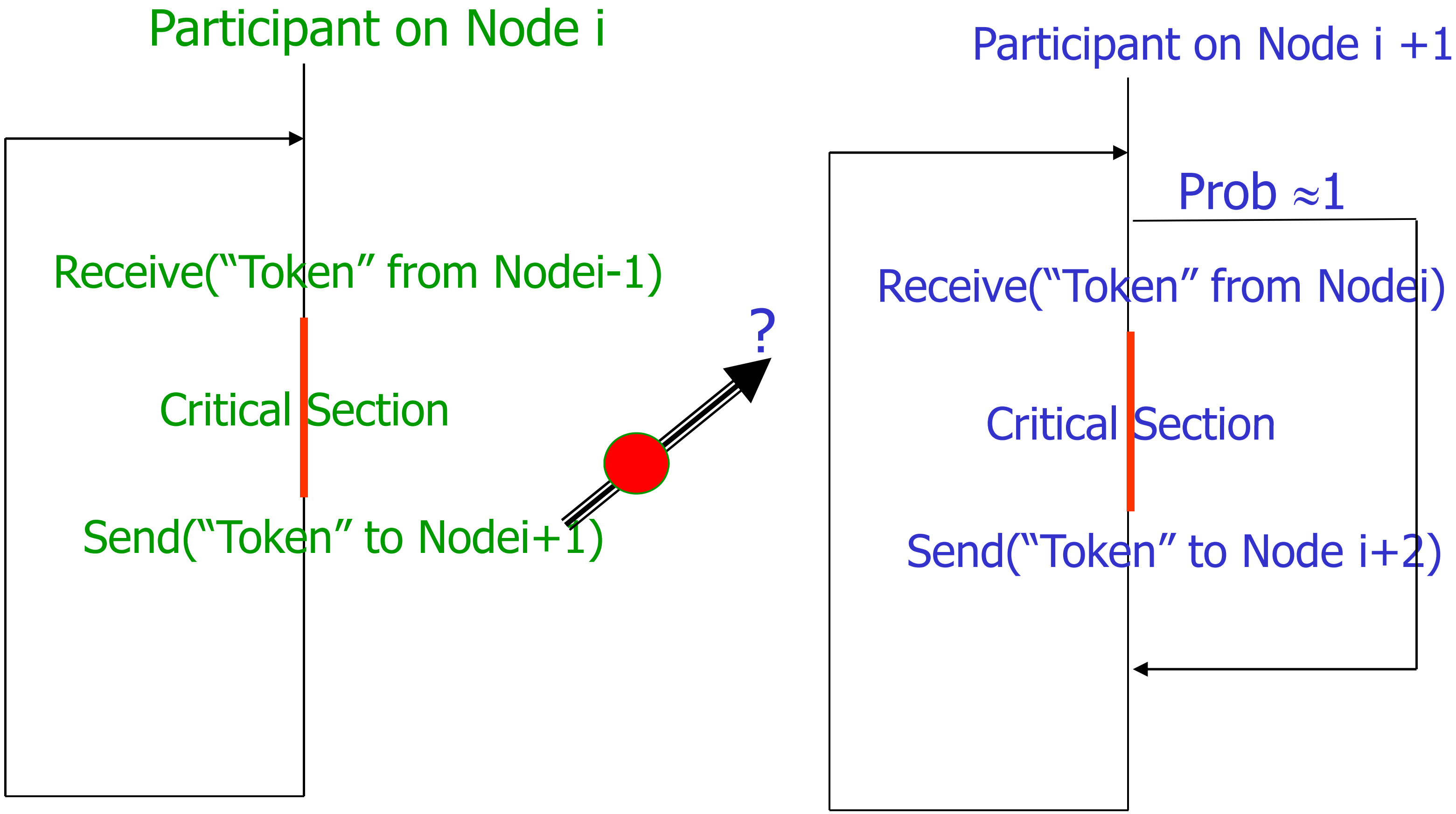
Critical Section

Send("Token" to Node *i*+2)

Question:

What may happen if you try to give token to immediate successor?

Implementation Issuess



Participant on Node i

Receive(“Token” from Nodei-1)

Critical Section

Send(“Token” to Nodei+1)

?

Participant on Node i +1

Prob ≈1

Receive(“Token” from Nodei)

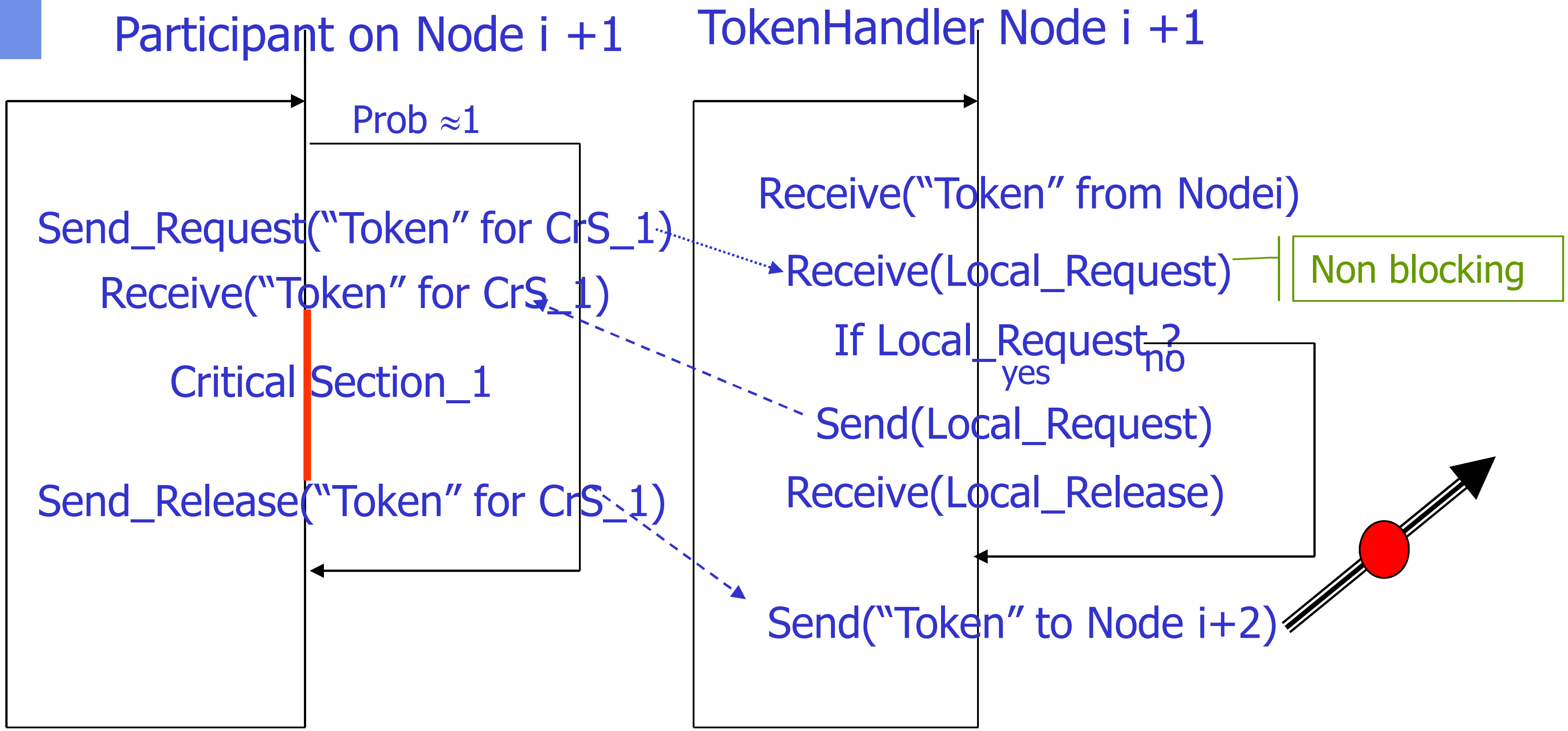
Critical Section

Send(“Token” to Node i+2)

Question: How to solve this problem as a system architect?

Implementation of a System Architect

A token-handler-thread per application and critical section



Participant on Node i +1

TokenHandler Node i +1

Prob ≈1

Send_Request("Token" for CrS_1)

Receive("Token" for CrS_1)

Critical Section_1

Send_Release("Token" for CrS_1)

Receive("Token" from Nodei)

Receive(Local_Request)

Non blocking

If Local_Request ?

yes

no

Send(Local_Request)

Receive(Local_Release)

Send("Token" to Node i+2)

Performance of Token Ring Alg.

- Suppose your logical token ring consists of p processes on p different nodes
- Per CS you need ***at least*** 2 messages
  1. Token passing message from immediate predecessor
  2. Token passing message to immediate successor

⇒ Minimal turnaround time of CS is increased by $2\ \Delta d$

  - Δd is the message transfer time

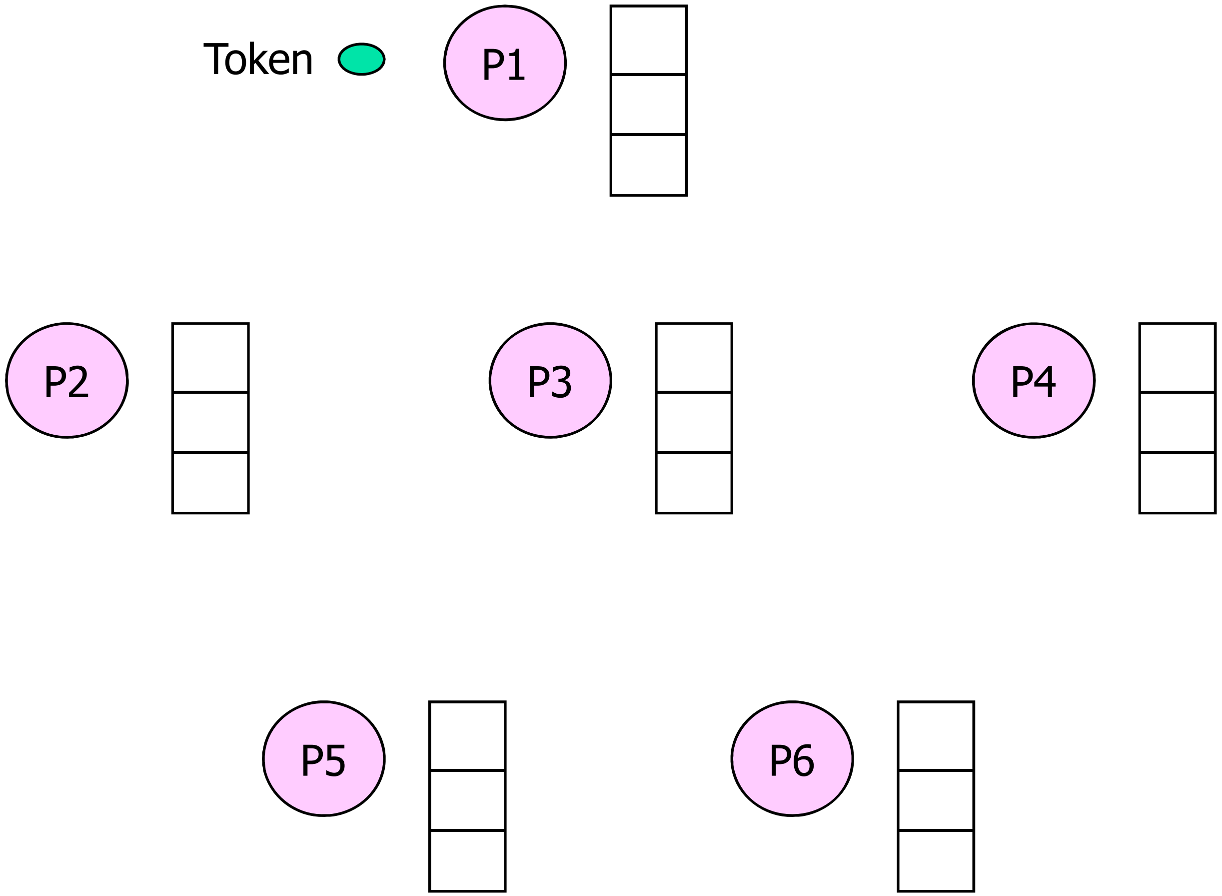
Average and maximal turn around times?

What about the requirements for a valid solution?

Tree Based Token Algorithm

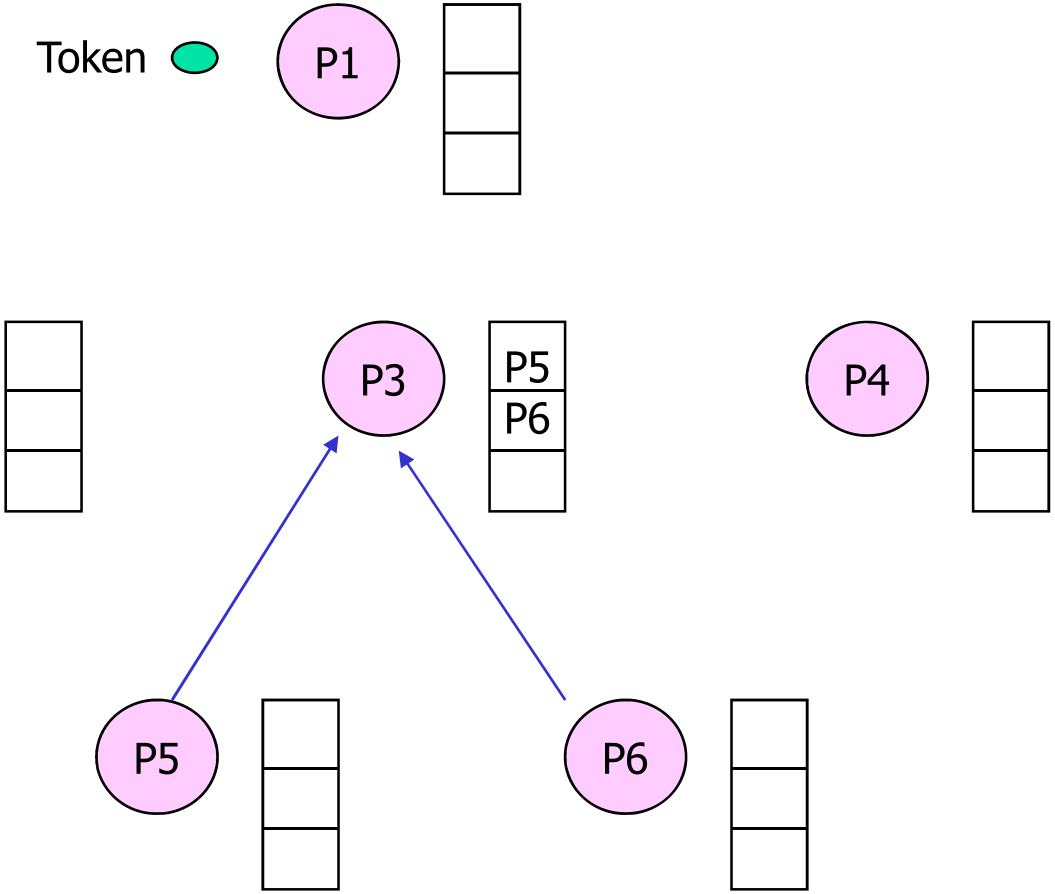
- Set of processes can be structured as a rooted tree
- Each node has a list for storing processes that want to enter their critical sections
- Initially all request lists are empty and the root contains the *grant token*
- Lower nodes send their requests to the immediate predecessors

Tree Based Mutual Exclusion (1)

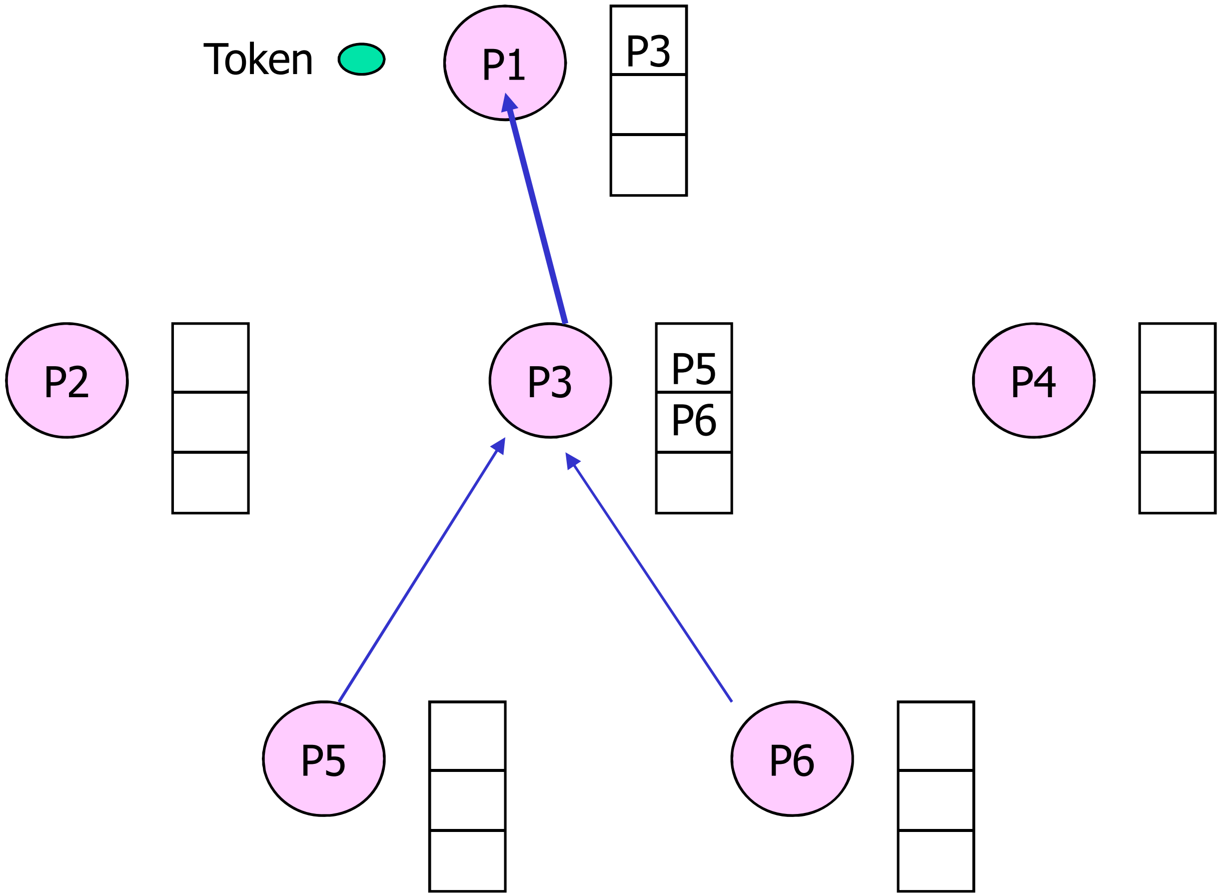


Initially root P1 is the token holder

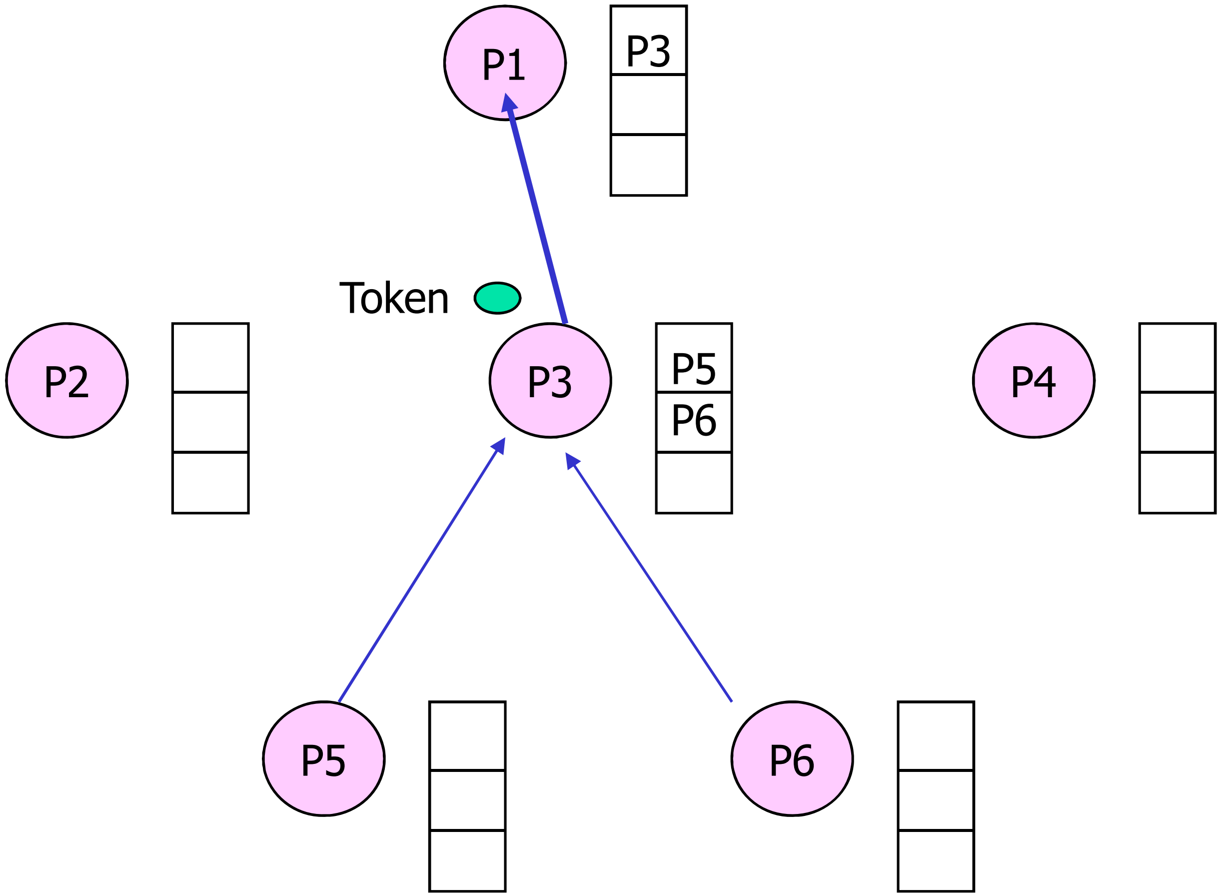
Tree Based Mutual Exclusion (2)



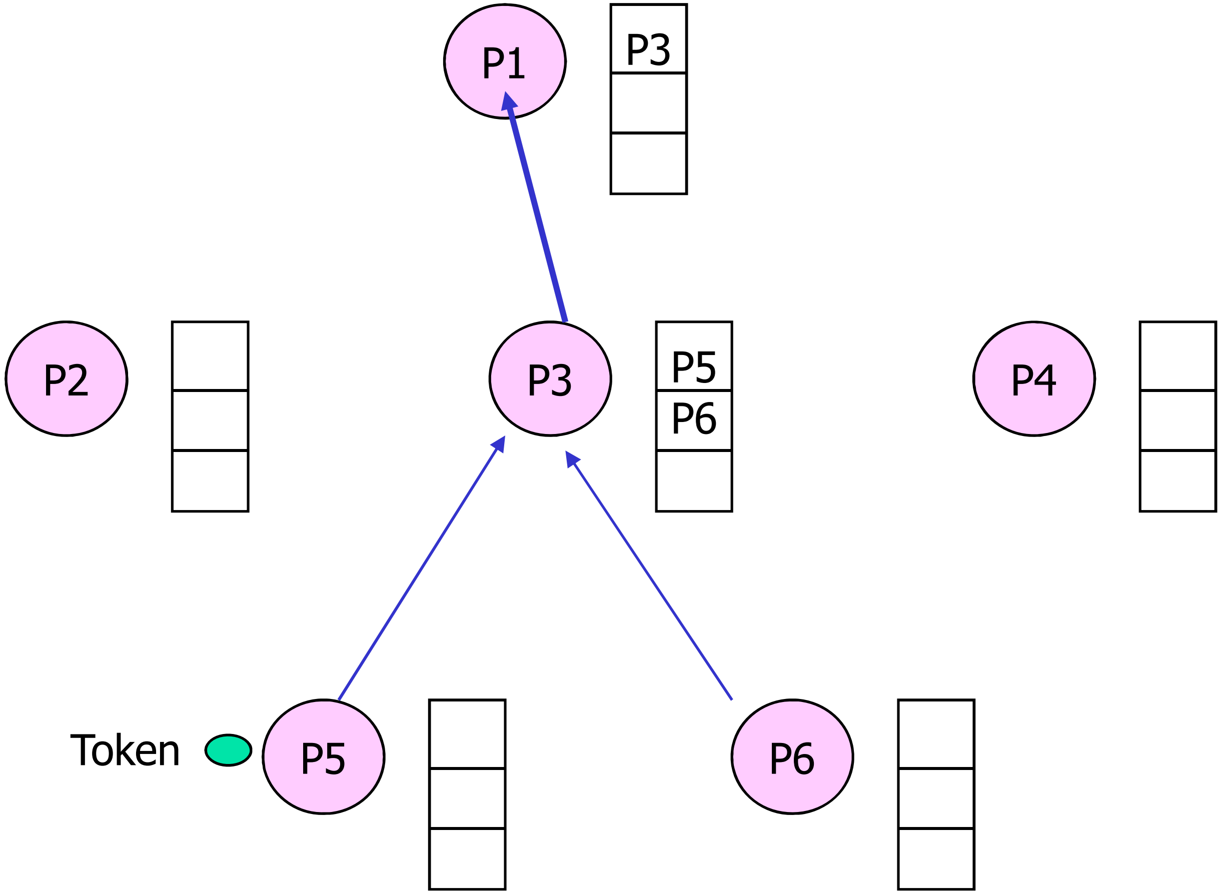
Tree Based Mutual Exclusion (3)



Tree Based Mutual Exclusion (4)



Tree Based Mutual Exclusion (5)



Finally P5 can use the token to enter its critical section
Releasing the token is almost as easy, but …

Performance of Tree Based Token?

- Analyze in the tutorial
- *How to implement an as fair solution as possible avoiding unbounded waiting of sub-trees*
- Problem: P3 in the example has no knowledge what's going on in the other sub-trees
- *Where to collect needed information about the requests*

Distributed Mutual Exclusion

Ricard Agrawala

Maekava

Distributed Lock Managers

Two distinct solutions:

- Ricart/Agrawala consensus algorithm
  - *All competitors* have to agree upon the process that is allowed to enter its CS
  - Algorithm needs logical clocks
  - Ricart, G.; Agrawala, A.: “An optimal Algorithm for Mutual Exclusion in Computer Networks”, C.ACM, 1981
- Maekawa’s voting algorithm
  - *Sufficient processes* have to vote for one competitor before it can enter its CS
  - M. Maekawa. "*A Square-root(N) Algorithm for Mutual Exclusion in Decentralized Systems*". ACM Transactions on Computer Systems, May 1985.

Distributed Lock Managers

Assumptions:

- N Processes have unique numeric identifiers
    - They maintain totally ordered Lamport times
    - All processes have communication channels to all other processes
- Reliable communication based on multicast
    - Process requesting access multicasts its request to all other N-1 processes
    - Process may only enter its CS when all other N-1 processes have replied an acknowledge message
- No node failures

Process States

- *Released,* i.e. process doesn‵t need its CS at the moment
- *Wanted*, i.e. process wants to enter its CS
- *Held*, i.e. process is in its CS

Ricart Agrawala Algorithm

```
enter():
   state := WANTED;
   Multicast request to all peers;
   T := request's Lamport timestamp;
   Wait until (N - 1) responses are received;
   state := HELD;

On receipt of a request <T(i),  P(i)> at P(j), j≠i:
   if( state == HELD or (state == WANTED and
         (T, P(j)) < (T(i), P(i)) )   {
      Queue request without replying;
   } else {
      Reply to P(i);
   }

release():
   state := RELEASED;
   Respond to queued requests;
```

Distributed Lock Manager (DLM)

Three message types (2 are required, 1 is optional)

- Request_Message
- *Queued_Message* ←
- Grant_Message

Request Message

- A process wishing to enter its CS either
  - *multicasts* or
  - *sends (n-1) times individually*

  an according *request message* to all processes competing for the critical region

- Each request message contains a "*Lamport timestamp*" and the *PID of the requester* ⇒ *∃ total ordering*

Queued Message

This type of message is only *optional* and is sent by recipients of the request message whenever the request cannot be granted immediately, i.e.

- recipient itself is currently in its CS or
- recipient had initiated an earlier request

Remark: This message type eases to find out whether ∃ *suspected dead participants*

Grant Message

Sent to a requesting process from all participants in two circumstances:

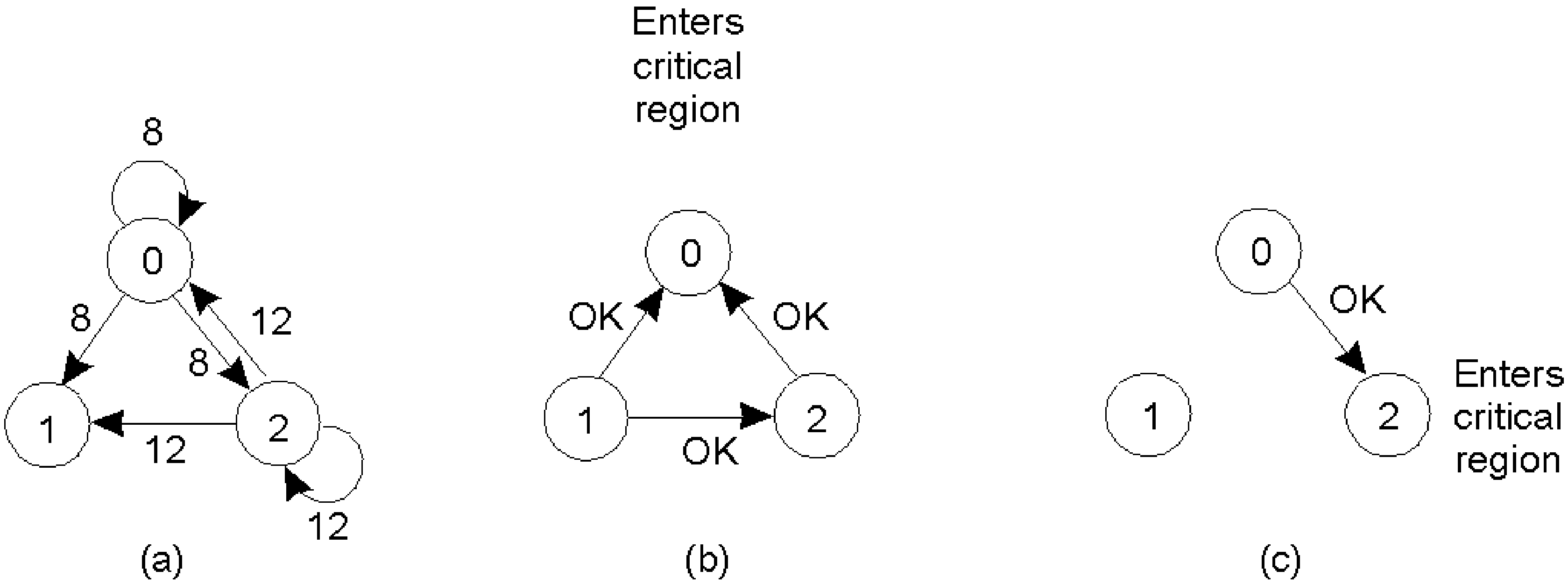
- recipient is not in *its CS* and has *no earlier request*
- if recipient is in its CS
  - first, it queues the request
  - Later on when it leaves its CS it will send the grant message to the requester

Release Message

Having released the resource this message is sent to *all participants* with a queued request-message.

- Another example for Java's **`notify_all()`**
- *Why is it not sufficient to notify just one of the waiting participants?*

Ricart-Agrawala Algorithm



a) 2 processes enter same CR at the same moment.

b) Process 0 has the *lowest timestamp*, so it wins.

c) When process 0 is done, it sends an OK also, process 2 can now enter the critical region.

Analysis of Ricart/Agrawala

- No tokens anymore
- Cooperative voting to determine sequence of CSs
- Does not rely on an interconnection media offering ordered messages
- Serialization based on logical time stamps (*total ordering*)
- If client wants to enter CS it asks all others for *permission* and proceeds if *all* others have agreed
- If a client C gets a permission request from another client C’ and if C is not interested in its CS, C returns permission immediately to the requester C’.

Correctness Conditions (1)

All nodes behave identically, thus we just have to regard node x

After voting, 3 groups of requests can be distinguished:

1. known at node x with time stamp less than C_x
2. known at x with a time stamp greater than C_x
3. those being still unknown at node x

Correctness Conditions (2)

During this voting, marks may change according to the following conditions:

Condition 1: Requests of group 1 have to be served or they have to take a time stamp greater than C_x

Condition 2: Requests of group 2 may not get a time stamp smaller than C_x

Condition 3: Request of group 3 must have time stamps greater than C_x

Two Phases of Voting Algorithm

1. Participants at node *i* willing to enter their CS send request messages e_i to all other participants, where *ei* contains the actual *Lamport time* L_i of node *i*. (After each send, node *i* increments its counter C_i).

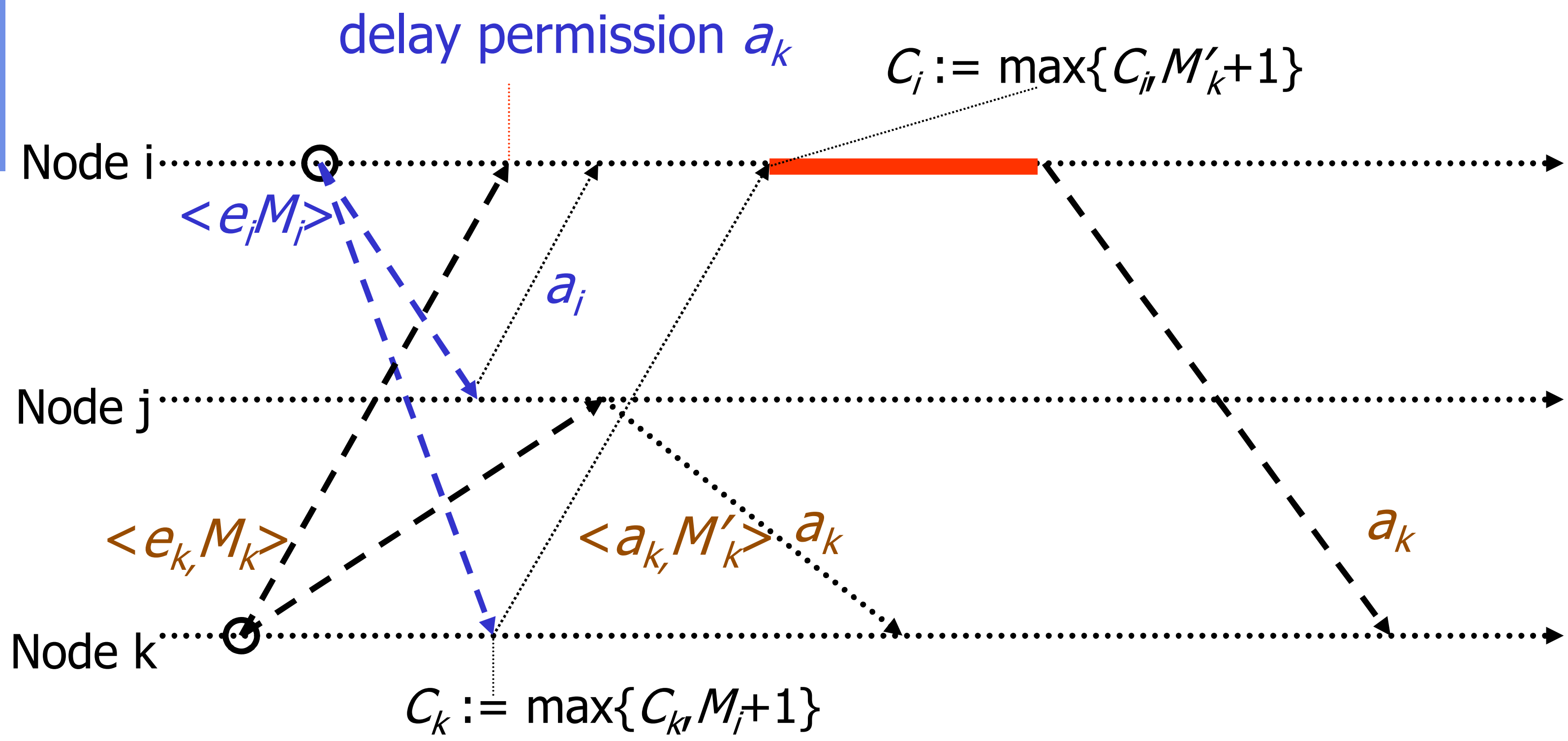
Delay a bit

2. All other participants return permission messages a_i. Node *x* replies to a request message e_i as soon as all older requests (received at earlier Lamport times) are completed.

$C_x := \max\{C_x, C_i + 1\}$

<u>Result:</u> If all permission messages have arrived at node *i*,
the corresponding requester may enter its critical section.

Example of the Voting Algorithm



Suppose: $M_i < M_k \Rightarrow$ the request message M_i has a smaller time stamp than M_k, we have to delay the answer for the request message e_k in node *i* !

Summary

- Instead of a single point of failure in the centralized solution, now each node is supposed ***not to fail***
- We need an efficient multi-cast and/or a group management
- In practice rarely used

Analysis of Mutual Exclusion Alg.

Algorithm	#messages per CS	Delay d	Response time **if CS is free**	Potential Problems
Centralized	3	2T*	2T + E**	Crash of central node
Decentralized	3mk	2m		Starvation, low efficiency
Standard Token	1 …∞	(0 … n-1)*T	(0,n-1)*T + E	Loss of token, Crash of node
Ricard-Agrawala	2(n-1)	2(n-1)*T	2(n-1)T + E	Crash of any node

* T: Message Transfer Time

** E: Execution Time of CS

Quorum based Algorithms

Maekawa Quorum Voting

Motivation

- Major drawback of Ricard/Agrawala is its scalability problem, because every other member of the critical region has to agree before any P can enter its CS
- Each P when about to leave its CS has to sent the release message to its N-1 partners
- Furthermore, despite the message transfers overhead reliability is even less than in the centralized solution
- Goal: Solution with fewer partners accepting a current request for entering a CS

Maekawa’s Voting Approach

Observation:

- to get access, not all processes have to agree
- suffices to split set of processes up into subsets (voting sets) that overlap
- suffices that there is consensus within every subset

Model:

- processes p1, .., p_N
- voting sets $\mathbf{V_1}$, .., $\mathbf{V_N}$ chosen such that $\forall$ i, k and for some integer M:
  - $p_i \in V_i$
  - $\mathbf{V_i} \cap \mathbf{V_k} \neq \{\}$ (some overlap in every voting set)
  - $| V_i | = K$ (fairness: all voting sets have equal size)
  - each process p_k, is contained in **M voting sets**

Maekawa’s CS-Protocol

Protocol:

- to obtain entry to CS, p_i sends *request messages* to all K-1 members of its voting set V_i
- cannot enter until all K-1 replies received
- when leaving CS, send ***release messages*** to all members of V_i
- when receiving request message
  - if state = HELD or already replied (voted) since last request
    - then queue request
    - else immediately send reply
- when receiving release message
  - remove request at head of queue and send reply

Voting Algorithm (Maekawa)

```
On initialization
   state := RELEASED;
   voted := FALSE;
For pi to enter the critical section
   state := WANTED;
   Multicast request to all processes in Vi – {pi};
   Wait until (number of replies received = (K – 1));
   state := HELD;
On receipt of a request from pi at pj (i ≠ j)
   if (state = HELD or voted = TRUE)
   then
        queue request from pi without replying;
   else
        send reply to pi;
        voted := TRUE;
   end if
```

Voting Algorithm (Maekawa)

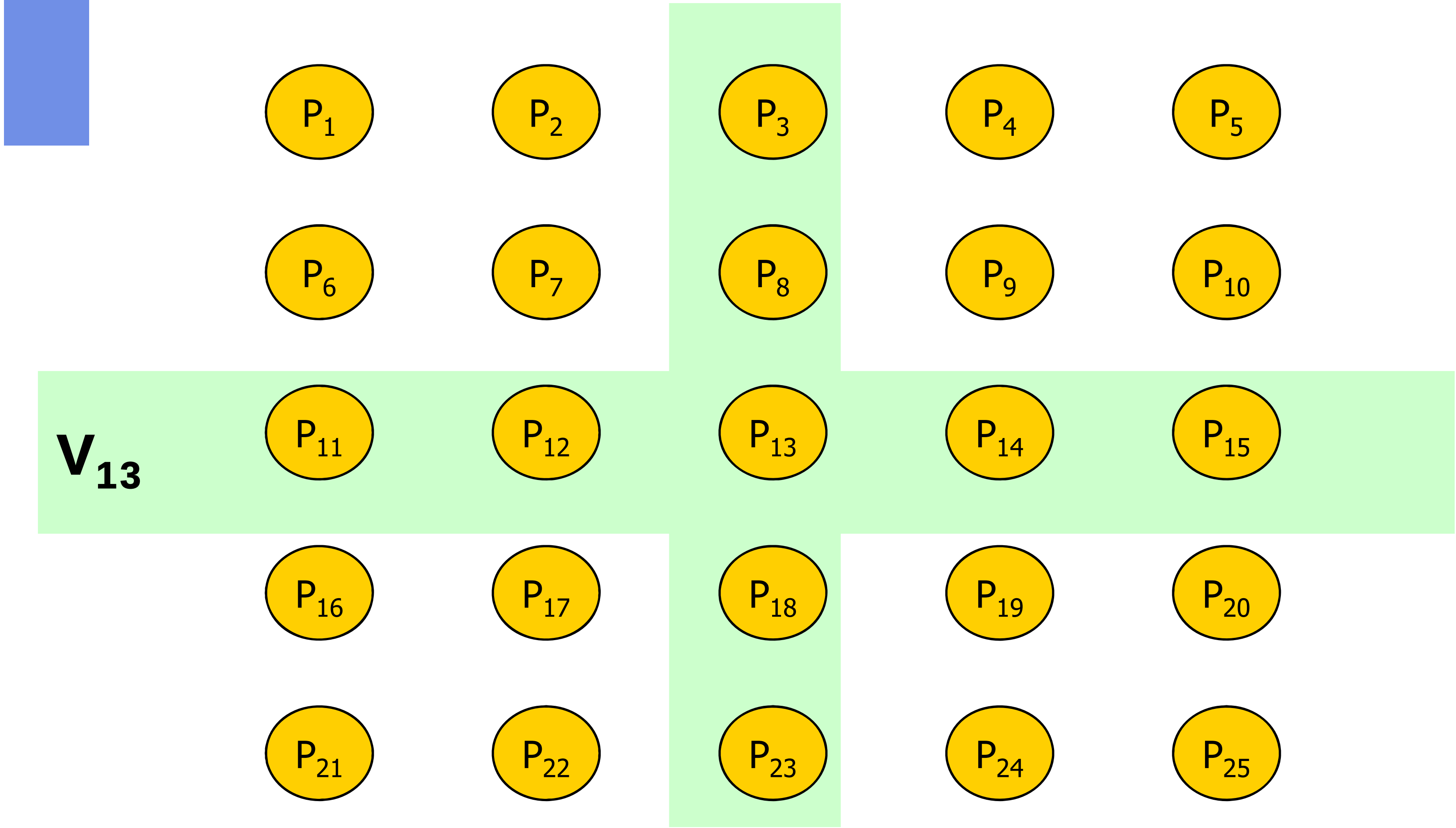
```
For pi to exit the critical section
   state := RELEASED;
   Multicast release to all processes in Vi – {pi};
On receipt of a release from pi at pj (i ≠ j)
   if (queue of requests is non-empty)
   then
        remove head of queue – from pk, say;
        send reply to pk;
        voted := TRUE;
   else
        voted := FALSE;
   end if
```

Each process only needs grants from all its potential voters

Maekawa’s Properties

- Optimization goal: minimize K while achieving mutul exclusion
  - Can be shown to be reached when K~√(N) and M=K
  - optimal voting sets: nontrivial to calculate
    - approximation: derive V_i so that | V_i | ~ 2* √(N)
      - place processes in a **√N x √N matrix**
      - let V_i the union of the row and column containing p_i

Quorum Example (Grid Scheme)



Properties of Maekawa

- Satisfies mutual exclusion
  - if possible for two processes to enter critical section, then processes in the non-empty intersection of their voting sets would have both granted access
  - impossible, since all processes make at most one vote after receiving request
- However, deadlocks are possible
  - consider three processes with
    - V1 = {p1, p2}, V2 = {p2, p 3}, V3 = {p3, p1}
  - possible to construct cyclic wait graph
    - p1 replies to p2, but queues request from p3
    - p2 replies to p3, but queues request from p1
    - p3 replies to p1, but queues request from p2

Variations

- Maekawa’s algorithm can be modified to ensure absence of deadlocks
  - use of logical clocks
  - processes queue requests in happened-before order
  - means that ME3 is also satisfied
- Performance
  - bandwidth utilization
    - 2 $\sqrt{N}$ per entry, $\sqrt{N}$ per exit, total 3 $\sqrt{N}$ is better than Ricart and Agrawala for N>4
  - client delay
    - same as for Ricart and Agrawala
  - synchronization delay
    - round-trip time instead of single-message transmission time in Ricart and Agrawala

Comments on Fault Tolerance

- None of these algorithms tolerates message loss
- Ring-algorithms can not tolerate single crash failure
- Maekawa’s algorithm can tolerate some crash failure
  - if process is in a voting set not required, rest of the system not affected
- Central-Server: tolerates crash failure of node that has neither requested access nor is currently in the critical section
- Ricart and Agrawala algorithm can be modified to tolerate crash failures by the assumption that a failed process grants all requests immediately
  - requires reliable failure detector

Election

Traditional Election

Elections in Wireless Environments

Elections in Large-Scale Systems

When Elections?

- Necessary when
  - System is booted in order to instantiate a
    - centralized coordinator for system activities
    - centralized monitor to watch system’s state
  - At run-time when a serial server
    - fails or
    - retires

Election Algorithms

- Some distributed applications need *one specific centralized process* (task), acting as a
    - Coordinator, e.g.
        - for centralized mutual exclusion manager
    - Monitor
    - Collector
    - …
- Via election algorithms you can establish a *new coordinator* -if the *old one has crashed*
- You need an *agreement* on the new coordinator

Election

An election should fulfill the following ***requirements***:

- ***E_0: Correctness:*** Only *one process* will be elected
- ***E_1: Safety:*** each process p_i has the attribute
  - *$elected_i = null$* or
  - *$elected_i =$* ***P***,

  whereby ***P*** is the *live process* with *highest id* at the end of the current election
- ***E_2: Liveness:*** each process p_i eventually will have the attribute *$elected_i \neq null$*

Election Algorithms

Suppose, your centralized lock manager has crashed.
How to do elect a new one in a DS?

∃ two major election algorithms, both are based upon:

- each process/node has a *unique process/node number* (i.e. there is a total ordering of all processes/nodes)
- live process with *highest process number* of all active processes is the current (will b the next) *coordinator*
- after a crash the *restarting former process* (eventually the previous coordinator) is put back to the set of active processes and the ***election is restarted again***

Election in a Logical Ring

Assumptions:

- Processes (+nodes) have unique identifiers
- Each process can communicate with *all live successors* on the ring
- Processes can fail (stop responding to its environment); this failure can be detected

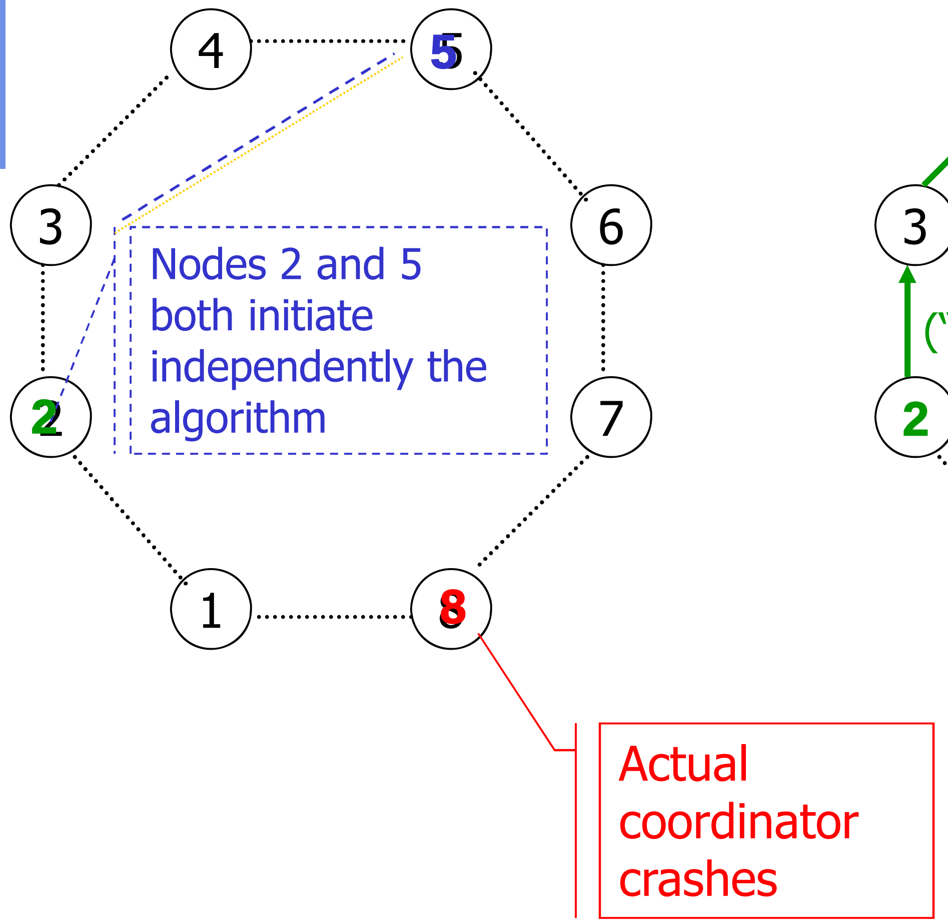
Ring Algorithm (Le Lann, 1977)

- Each process/node N_i knows all its successors, i.e. the complete logical ring
- 2 types of messages are used:
  - *election e*: to elect the new coordinator
  - *coordinator c:* to introduce coordinator to the nodes
- Algorithm is initiated by any node N_i suspecting that the current coordinator no longer works
- N_i send a message *e* with its node number *i* to its immediate successor N_{i+1}
- If this immediate successor N_{i+1} does not answer, it is assumed that N_{i+1} has crashed and the *e* is sent to N_{i+2}, …

Ring Algorithm

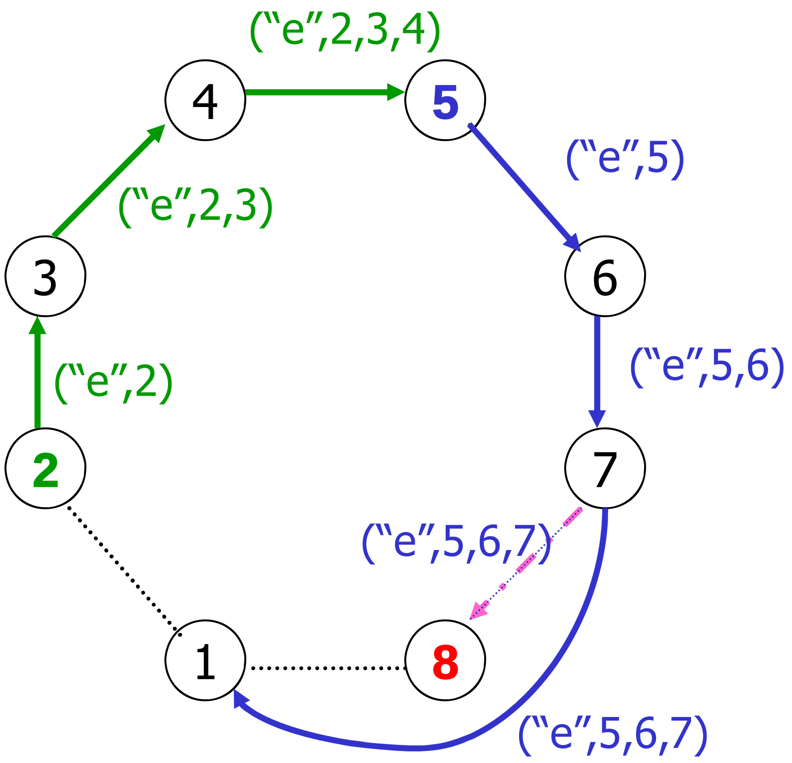
- N_i receives an *e/c-message* with a list of node numbers:
  - If an e-message does not contain its process/node number i, N_i adds it to the list, sends e-message to N_{i+1}
  - If an e-message contains its node number i, this e-message has circled the ring of all active nodes. The highest process/node number in the list is the new coordinator and N_i converts e-message into a c-message
  - If its an c-message, N_j keeps in mind the node with the highest number in that list being the new coordinator
  - If a c-message has circled once, it's deleted
- After having restarted a crashed node you can use an "*inquiry"-message*, circling once around the ring

Ring Algorithm

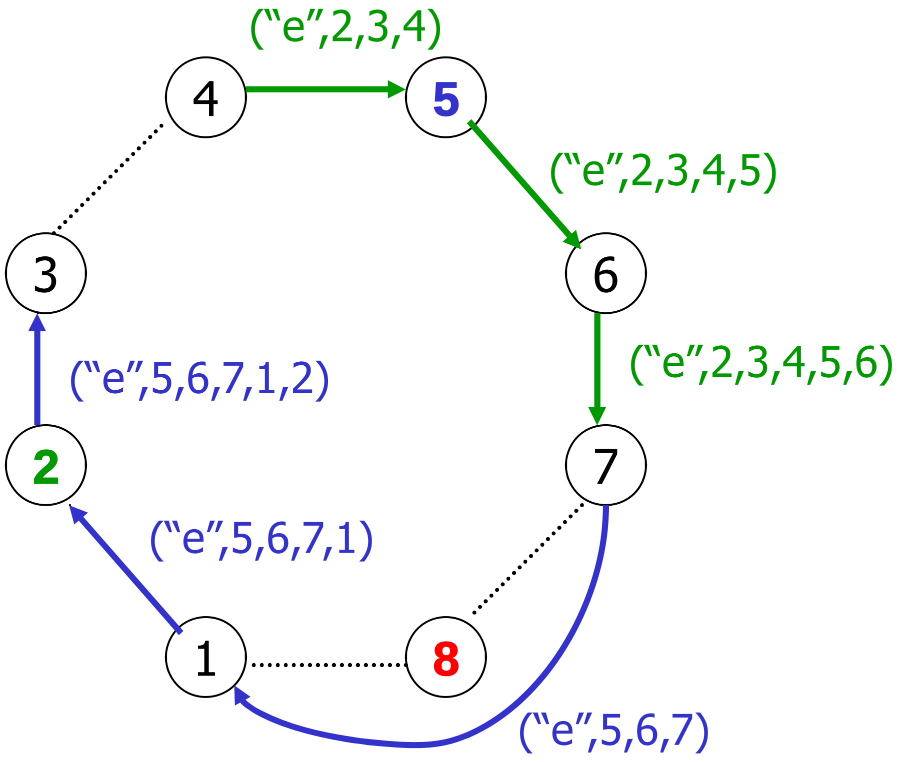


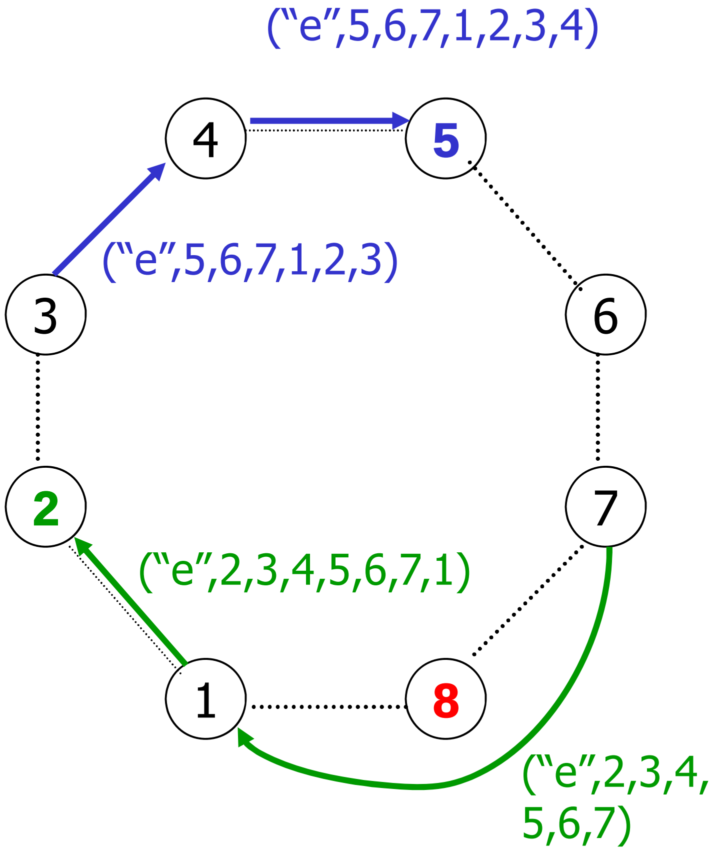
Nodes 2 and 5 both initiate independently the algorithm

Actual coordinator crashes



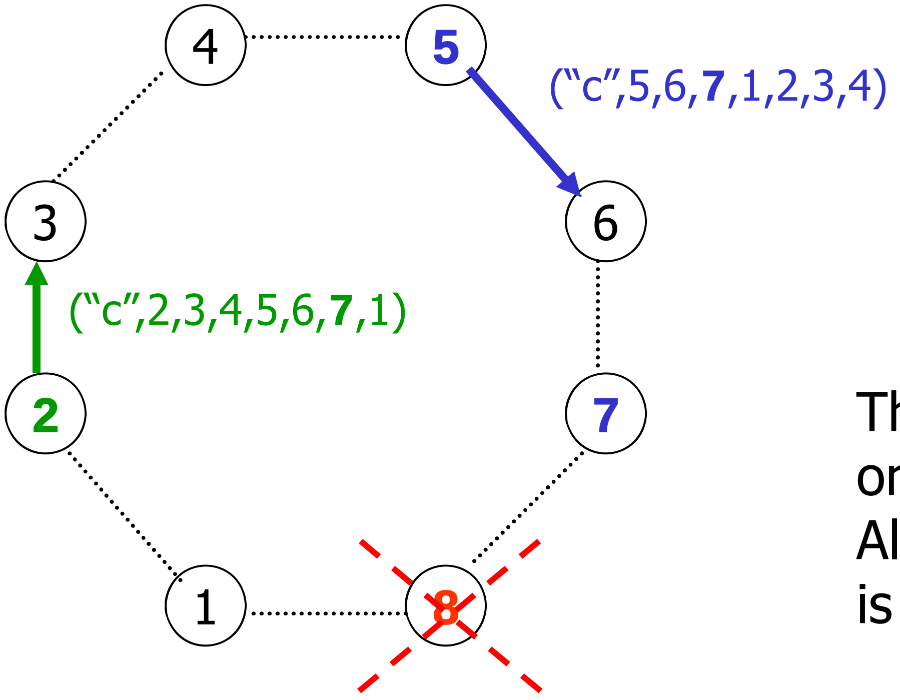
Ring Algorithm





Both e-messages circled once around the ring of all active nodes

Ring Algorithm



This coordinator-message circles once around the logical-ring,
All nodes know that **7**
is the new coordinator

Improved Ring Algorithm

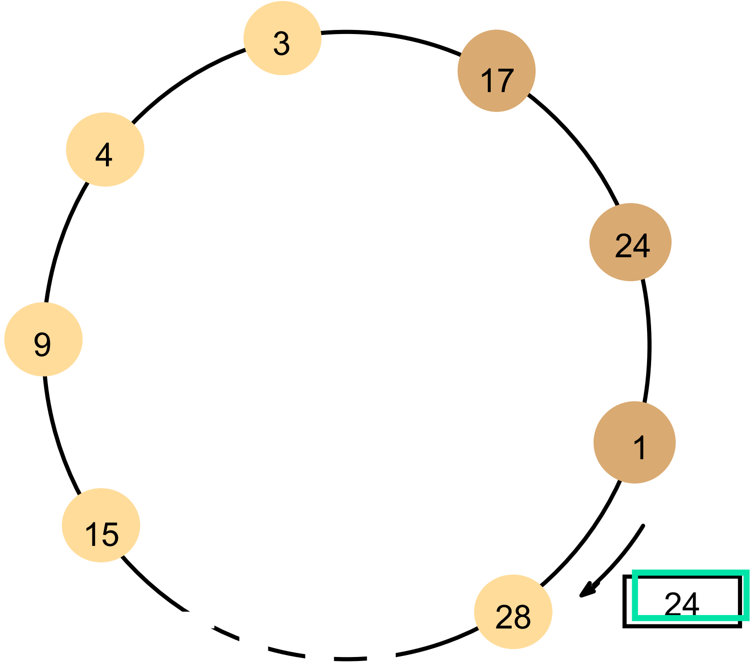
Assumptions:

- Processes do not know each others PID
- all nodes communicate on a uni-directional ring structure, i.e. only with its successor
- all processes have unique integer id
- asynchronous, reliable system

Improved Ring Algorithm

- Initially, all processes marked "*non-participant*"
- To start election, process place **election message** with own identifier on ring and marks itself "*participant*"
- upon receipt of election message, compare received identifier with own
    - if received id greater than own id, forward message to neighbor
    - if received id smaller than own id,
        - if own status is "*non-participant*", then substitute own id in election message and forward on ring
        - otherwise, do not forward message (already "*participant*")
    - if received id is identical to own id
        - this process's id must be greatest and it becomes elected
        - marks own status as "*non-participant*"
        - sends out **coordinator message**
- when receiving coordinator message
    - mark own status as "*non-participant*"
    - set attribute elected_i appropriately and forward coordinator message

Improved Ring Algorithm[1]



Process has 2 possible states:

- ***participating***
- ***not participating***

Initially each p = not participating

Election message only contains PID of maximal passed process

Receiving process compares PID in election message with its own PID:

```
If (state = non participating and
ownPID > e(PID)) then
   { e(PID)=ownPID
       state = participating}
else …
```

Note: The election was started by process 17.
Highest process identifier encountered so far is 24.
Participant processes are shown darkened

[1] Chang-Roberts 1979

Analysis: Improved Ring Election

- Properties
  - E_0 is satisfied, only one new coordinator
  - E_1 satisfied, since all identifiers are compared
  - E_2 follows from reliable communication property
- Performance
  - at worst 2N-1 messages for electing the left-hand neighbor
  - another N coordinator messages
- Failures
  - tolerates no failures

Election by Bullying

Assumptions:

- Network is *synchronous*
- Nodes can crash, crashes will be detected *reliably*
- *Fully connected* network, no message loss
- Crash failures only
- Nodes have unique identifiers and know ids of all other nodes (else broadcast)

Bully Algorithm[1]

Goal: Find live node with the highest number, choose it as coordinator and tell this all other nodes

Start: Algorithm may start at any node, having recognized that previous coordinator is no longer responding.

Message types:

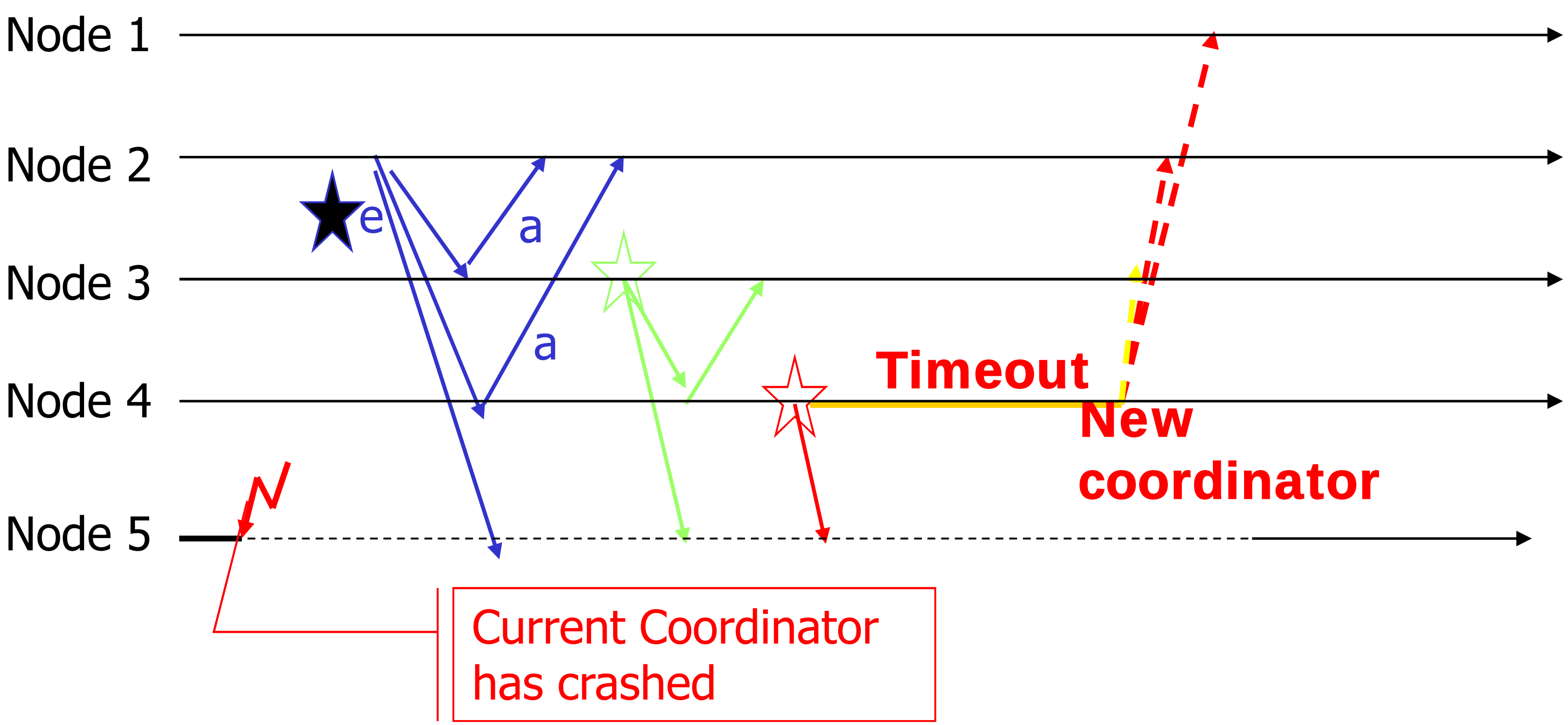
- *Election* ***e***, initiating the election
- *Answer* ***a***, confirming the reception of an ***e*** message
- *Coordinator* ***c***, telling all others, that it is the new coordinator

[1] Garcia-Molina, 1982

Steps of Bully Algorithm

1. Some node N_i sends *e*-messages to ***all other nodes N_j***, *$j > i$*.
2. If there is no answer within ***Δt***, N_i elects himself as coordinator sending this info via a *c*-message to all others N_j, $j < i$.
3. If N_i got an *a-message* within ***Δt*** (i.e. there is an active node with a higher number), it is awaiting another time-limit ***$\Delta t'$***. It restarts election, if there is no *c-message* within ***$\Delta t'$***
4. If N_j receives an e-message from N_i, it answers with an a-message to N_i and starts the algorithm for itself (step 1).
5. If a node N -after having crashed and being restarted- is active again, it starts step 1.
6. Highest numbered node declares itself to be the ***new coordinator***

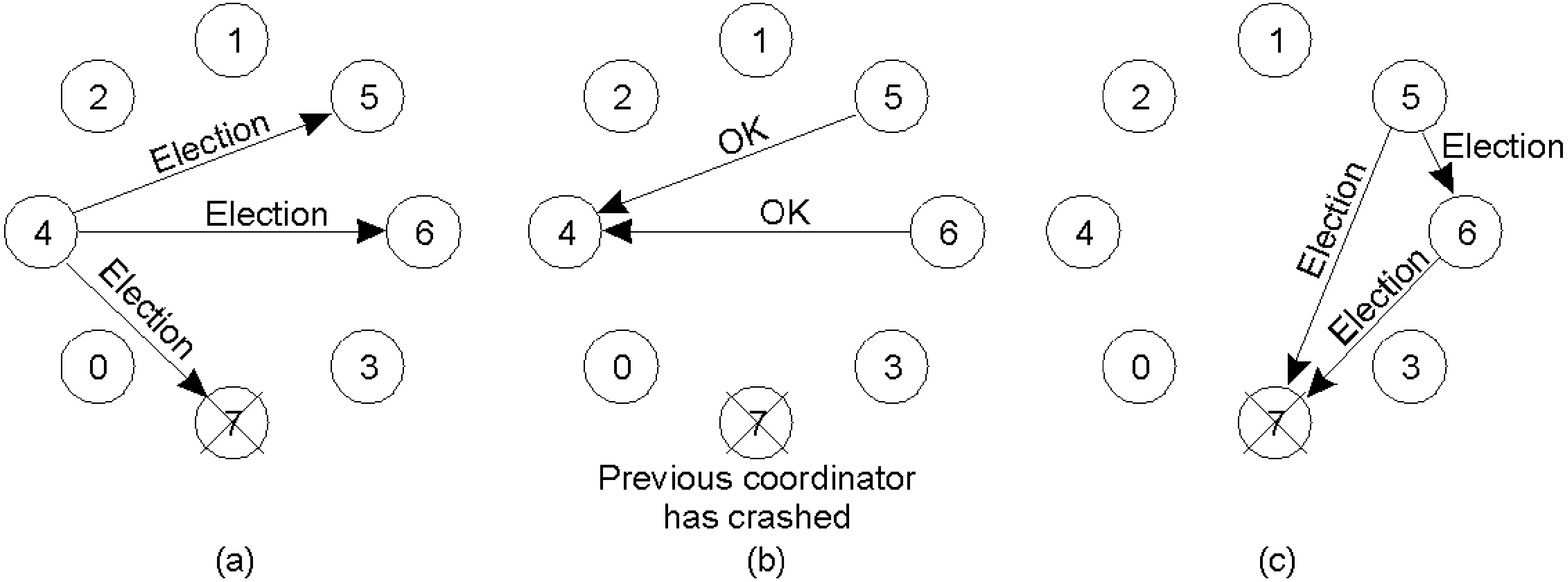
Example Bully Algorithm



Node 2 detects the false behavior of the coordinator

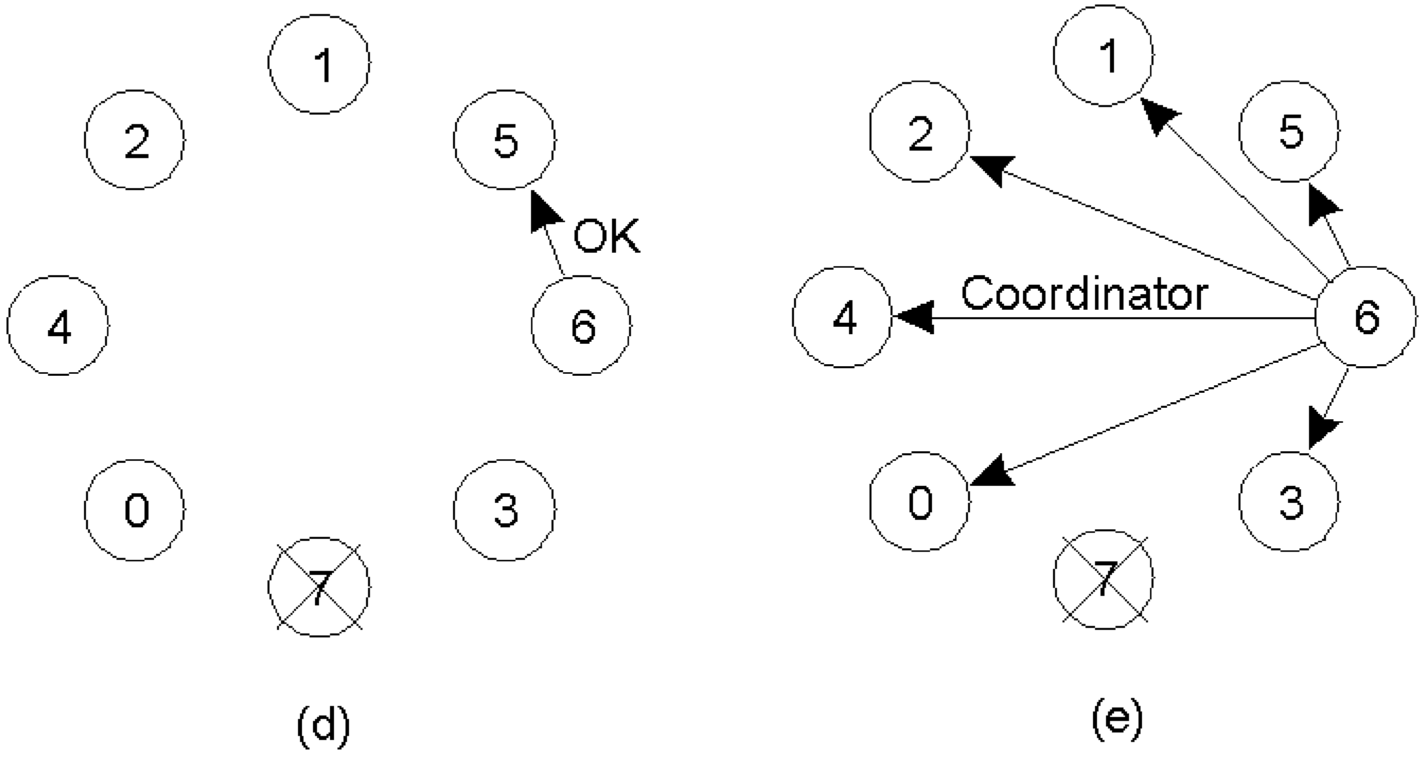
Nodes 3 and 4 have to start the algorithm due to their higher number telling node 2 to stop with its election algorithm

Bully Algorithm (1)



(a) Process 4 starts an election
(b) Process 5 and 6 respond, telling 4 to stop
(c) Now 5 and 6 each start an election

Bully Algorithm (2)



(d) Process 6 tells 5 to stop

(e) Process 6 wins and tells everyone

Analysis of Bully

- Properties
  - E_0 is satisfied, only one new coordinator
  - E_1 satisfied, since all identifiers are compared
  - E_2 follows from reliable communication property
- Performance
  - Best case: process p with second highest PID detects crash of old coordinator
    - Elects itself coordinator and send N-2 election messages
    - Requires $O(N^2)$ messages in worst case when lowest process detects coordinator crash
      - N-1 processes with higher Ids start the election

Comparison of 2 Election Algorithms

Algorithm	Number of Messages	Time
Bully	$O(n^2)$	$O(n)$
Ring	$2(n-1)$	$2(n-1)$

In M. Weber: “Verteilte Systeme” there is another election algorithm (from Mattern) based on a tree-topology

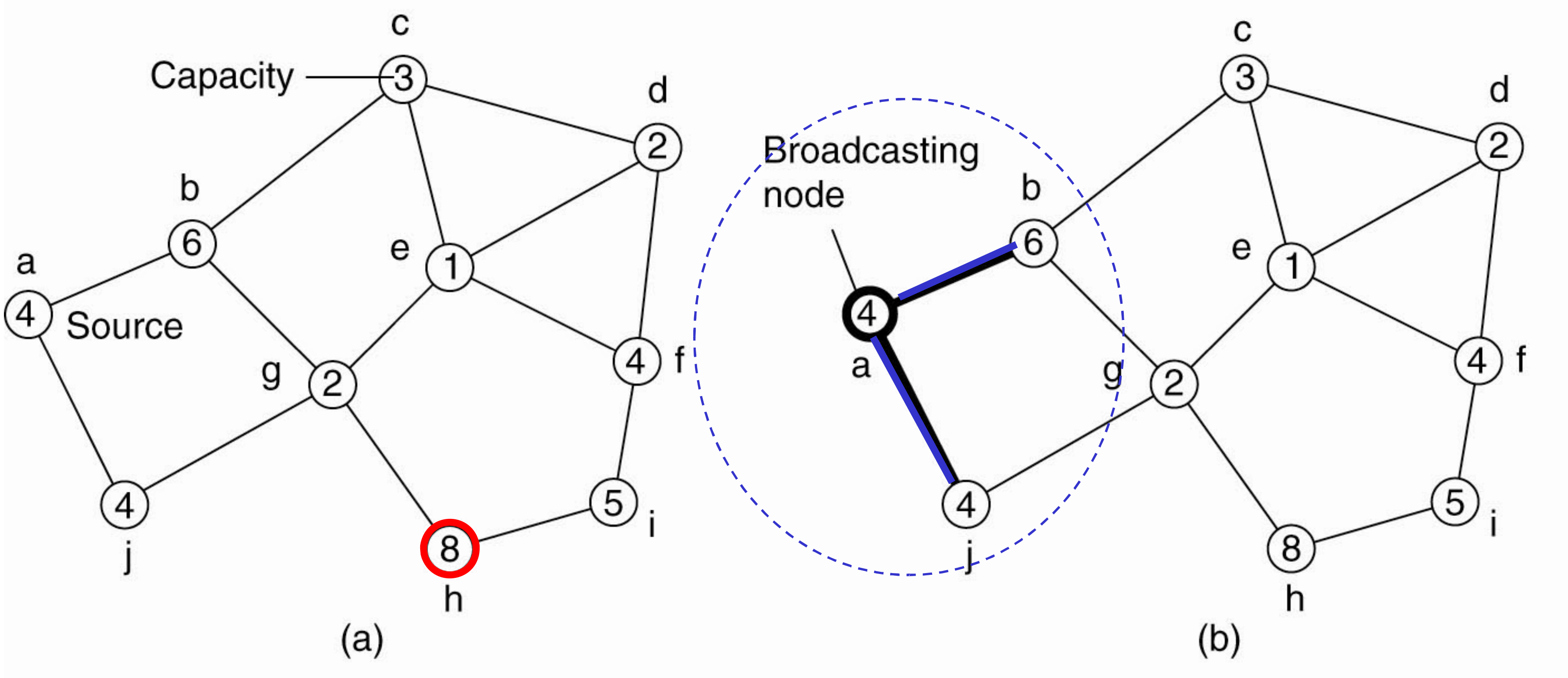
Election In Wireless Environments

Wireless Ad Hoc Nets with non moving nodes

Vasudevan et al.: “Design and Analysis of a Leader Election Algorithm for Mobile Ad Hoc Networks”, Proc. 12. International Conference on Network Protocols, 2004

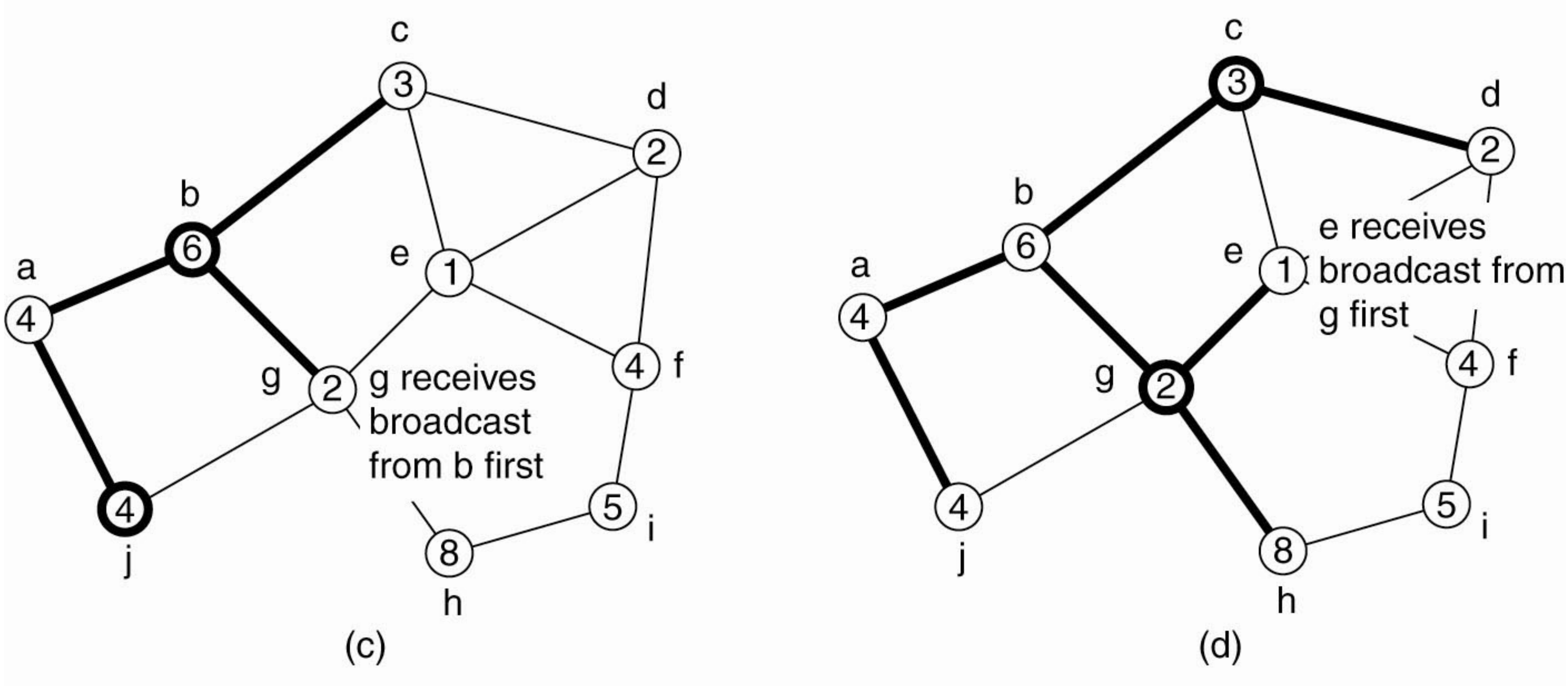
http://www-net.cs.umass.edu/~svasu/pubs.html

Elections in Wireless Environ. (1)



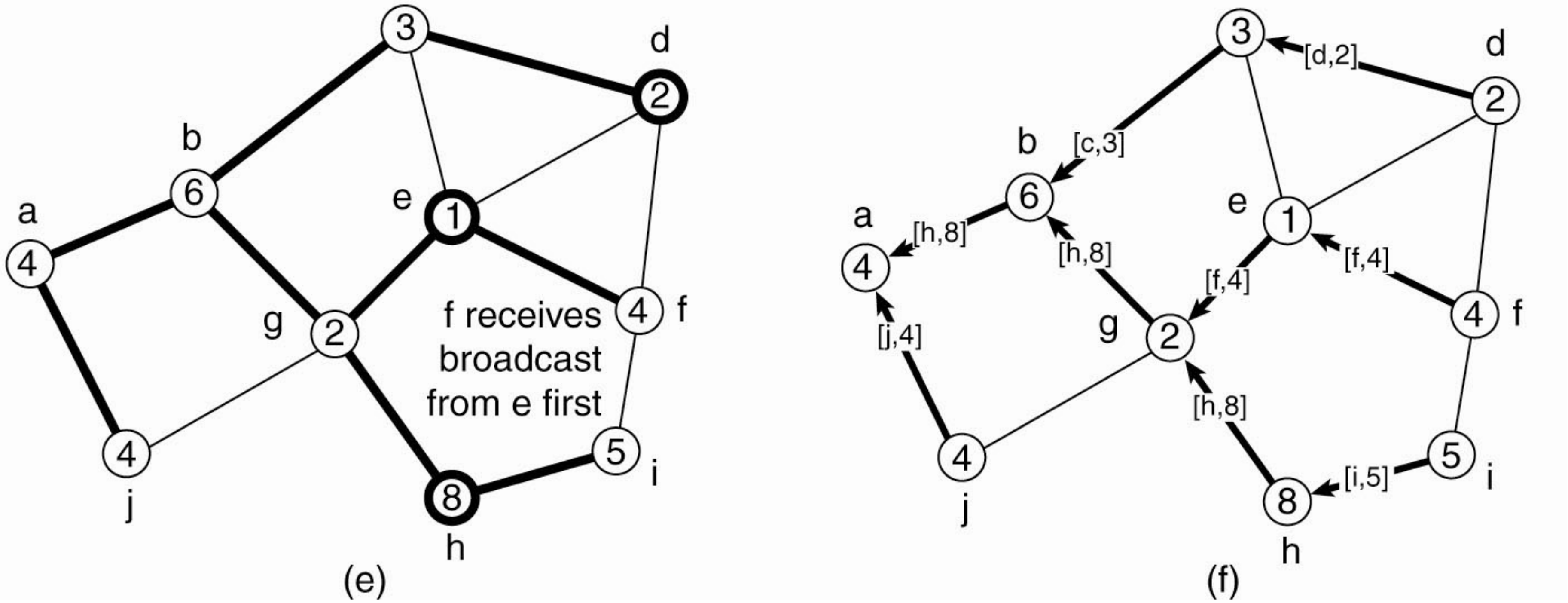
- Election algorithm in a wireless network, with node a as the source. (a) Initial network. (b)–(e) The build-tree phase

Elections in Wireless Environ. (2)



(c) (d)

Elections in Wireless Environ. (3)



Elections in Large-Scale DS

Study of your own

Elections in Large-Scale Systems (1)

- Requirements for superpeer selection:

1. Normal nodes should have low-latency access to superpeers.
2. Superpeers should be evenly distributed across the overlay network.
3. There should be a predefined portion of superpeers relative to the total number of nodes in the overlay network.
4. Each superpeer should not need to serve more than a fixed number of normal nodes.

Superpeer Election

In a DHT system:

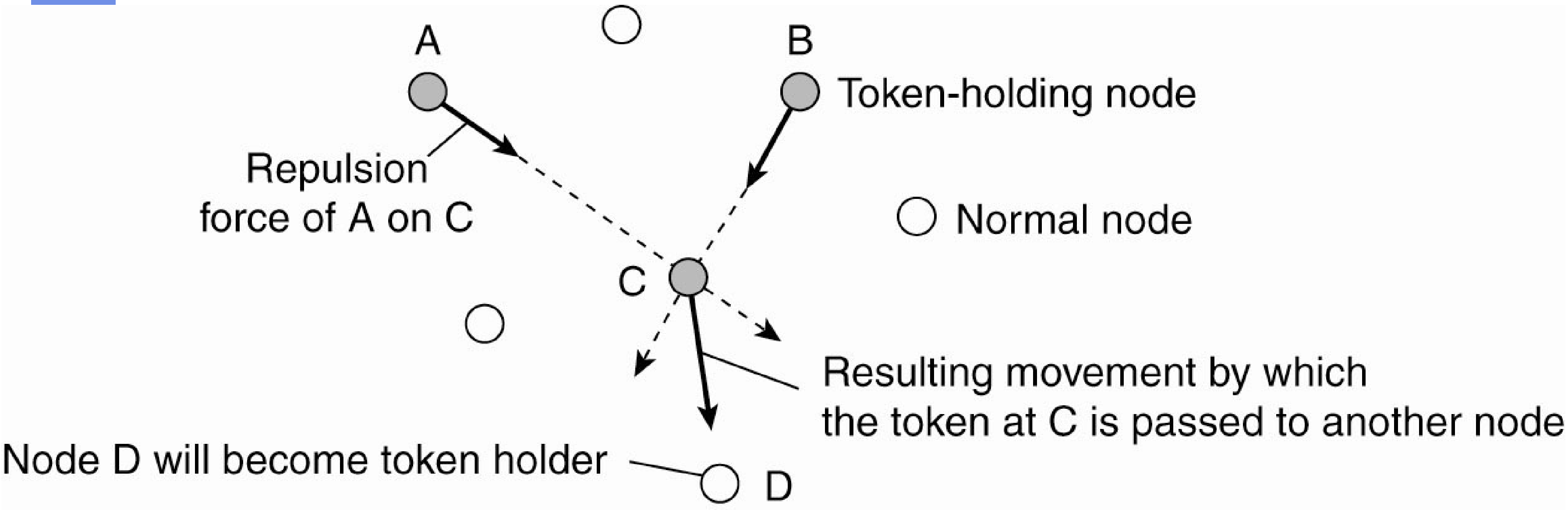
Reserve a fixed part of the ID space for superpeers

Example:

- If s superpeers are needed for the DS that uses ***m-bit identifiers***, simply reserve **$k = \log_2 S$** leftmost bits for superpeers
- With n nodes we'll have on average

  $2^{k-m} *n$ superpeers
- Routing to superpeer: send message for key ***p*** to node responsible for ***p AND 11…1100…000***

Elections in Large-Scale Systems (3)



- Moving tokens in a two-dimensional space using repulsion forces

Deadlock Detection

Outline

- Deadlocks
  - Deadlock Conditions
  - Centralized Detections
  - Path Pushing
  - Distributed Detection

How to deal with deadlocks

- Transactions
  - Transactions in Local systems
  - Characteristic of Transactions
  - Serializability
  - Two Phase locking Protocol
  - Distributed Transactions

How to support complicated distributed applications

Methods against Deadlocks in DS

- Prevention (in some transaction oriented systems)
- Avoidance (too complicated and time consuming)
- *Ignoring* (still popular)
- Detecting (sometimes, if really needed) combined with repairing

Deadlocks in Distributed Systems

In a DS a distinction is made between:

- Resource deadlock: tasks are stuck waiting for resources held be each other
- Communication dl: tasks are stuck waiting for message to arrive

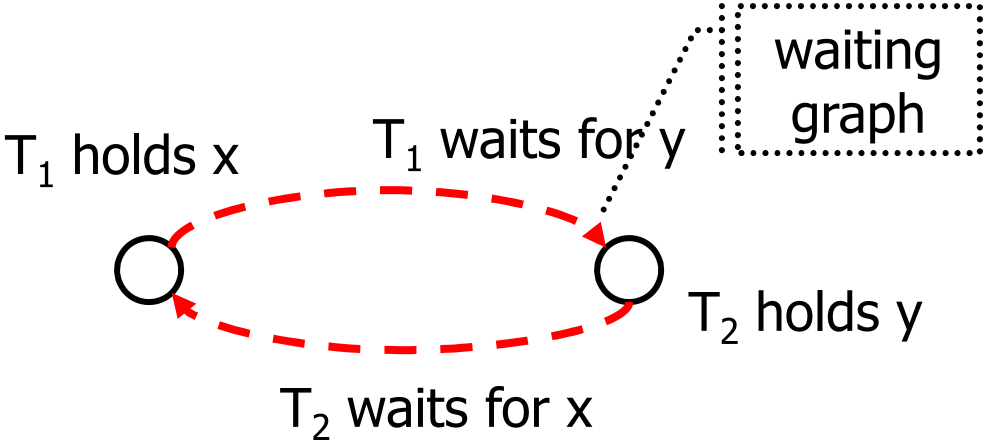
- However, message buffers ~ resources

Distributed Deadlocks

- Using “*locks*” within transactions may lead to deadlocks:

T_1	T_2
…	…
	….
lock(x)	
…	lock(y)
	…
lock(y)	
	lock(x)

time



T_1 holds x

T_1 waits for y

waiting graph

T_2 holds y

T_2 waits for x

A *deadlock* has occurred
if *global waiting graph* contains a *cycle*.

Deadlock Prevention

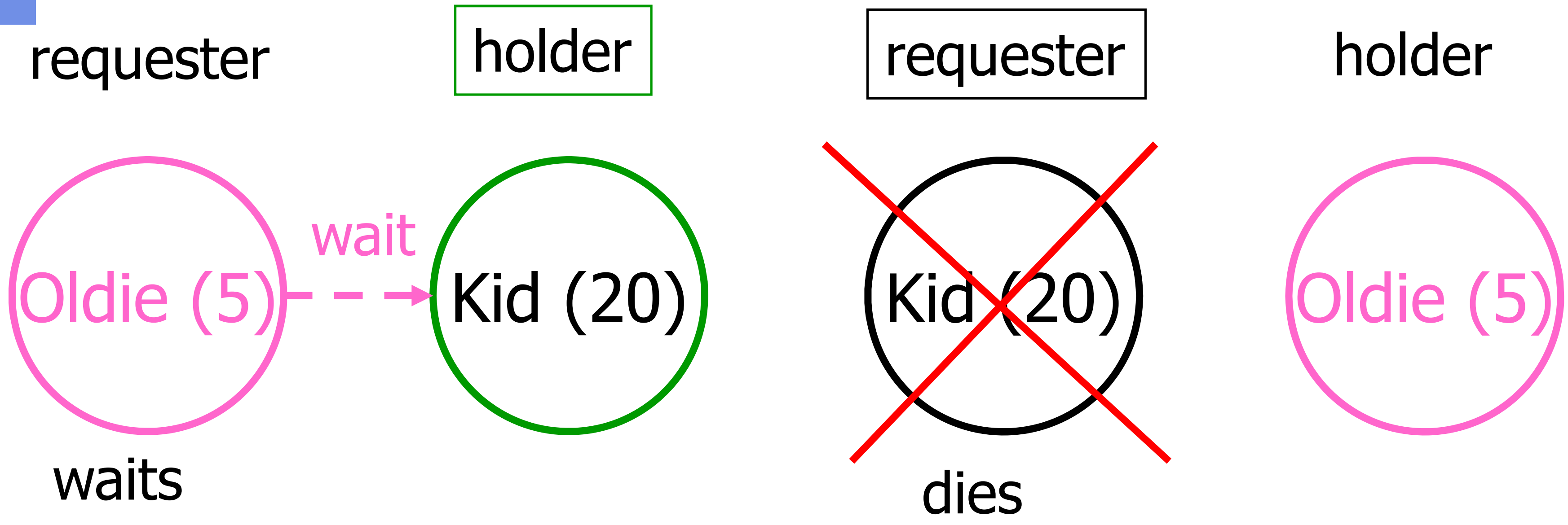
1. Task may hold only *1* resource at the same time (=> no cycles possible)
2. Pre-allocation of resources (⇒ resource inefficiency)
3. Release old resources if requesting a new one
4. Acquire in order (It's quite a cumbersome task to number all resource types in a DS)
5. "*Senior rule*": each application gets a "timestamp" (according to Lamport's time).

⇒ ***Oldies* (seniors) are preferred**

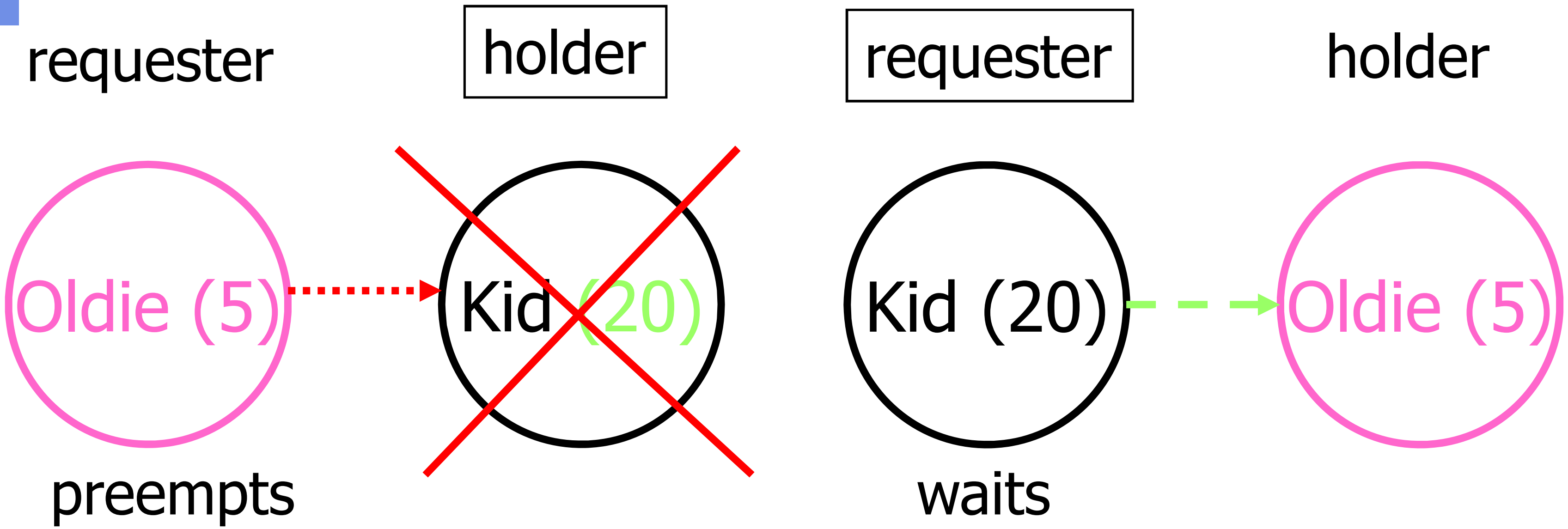
Wait-Die Deadlock Prevention

- Each transaction gets a *time stamp* when it starts
- If „old" transaction (with lower time stamp) requests resource -held by a younger one- then oldie has to wait and it is *queued according to its time stamp*
- If a younger transaction requests a resource -held by an oldie- the young transaction is *aborted* and later on *restarted*

„Wait-Die" Prevention



„Wound-Wait“ Prevention



requester

holder

requester

holder

Oldie (5)

Kid (20)

Kid (20)

Oldie (5)

preempts

waits

Deadlock Avoidance

Avoidance* in DS almost never used because:

1. Every node must keep track of *global state* of DS ⇒ substantial ***storage & communication overhead***
2. Checking for a global state *safe* must be *mutual exclusive*, otherwise two concurrent checks may violate the state safe
3. Checking for a *global safe state* requires substantial processing and communication

*Deadlock avoidance rarely used even in local systems

Deadlock Detection in DS

Increased problem:

If there is a deadlock within a DS
resources from different nodes may be involved

Several approaches:

1. Centralized Control
2. Hierarchical control
3. Distributed Control

In any case:

Deadlock *must be detected* within a *finite amount of time*

Deadlock Detection in DS

Correctness in a waiting-graph depends on:

- progress
- safety

Deadlock Detection in DS

General remarks:

- Message delay and out of date data may cause false cycles to be detected (***phantom deadlocks***)
- After a "possible" deadlock has been detected, one has to **double check** if it is a real one
- Having detected a deadlock, delete and restart task, if it‘s transaction oriented.

Centralized Deadlock Detection

- Local and global deadlock detector (LDD and GDD) (if a LDD detects a local deadlock it resolves it locally!).
- The GDD gets status information from the LDD
  - on waiting-graph updates
  - periodically
  - on each request
- If a GDD detects a deadlock involving resources at two or more nodes, it has to resolve this deadlock globally!)

Centralized Deadlock Detection

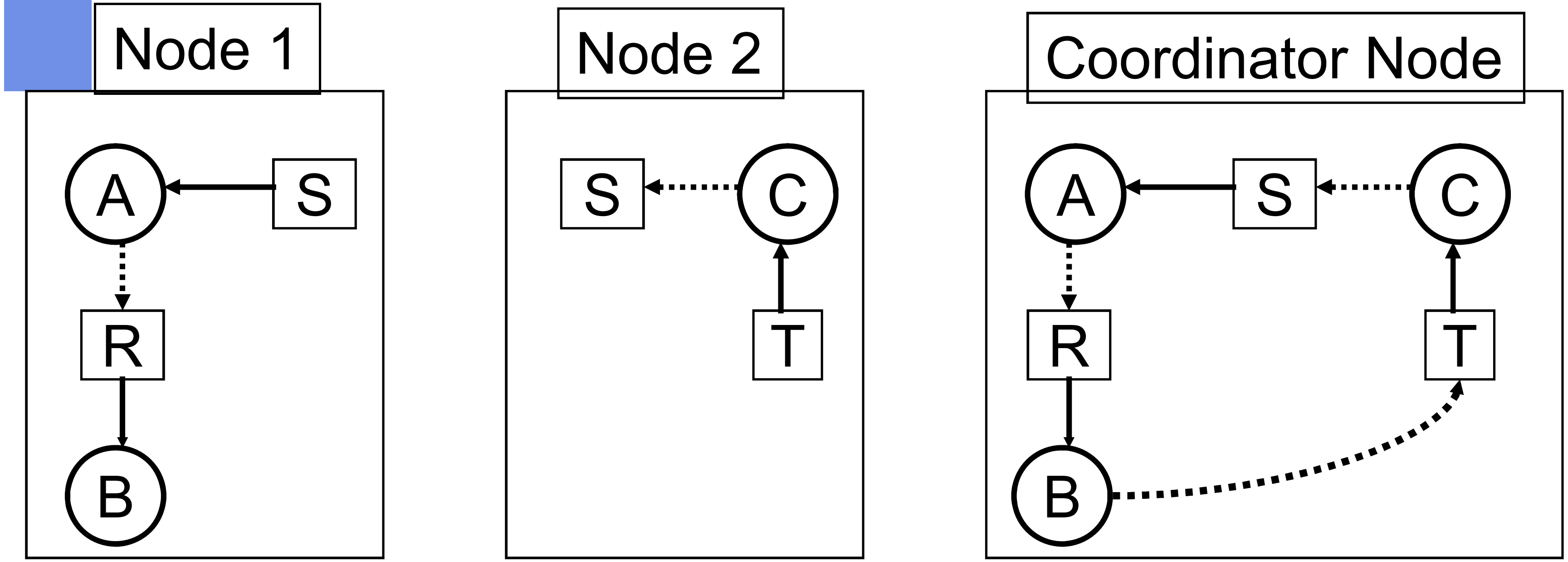
Major drawbacks:

- The node hosting the GDD = *point of single failure*
- "*Phantom deadlocks*" may arise because the global waiting graph is not up to date

Centralized Deadlock Detection

- Each node preserves its local waiting graph (respectively its resource usage graph)
- Central coordinator preserve a global waiting graph (union of the local ones)
- If coordinator detects a cycle it kills one task to break the deadlock
- Problem: Does the global waiting graph correspond to the current global state?

Phantom Deadlocks



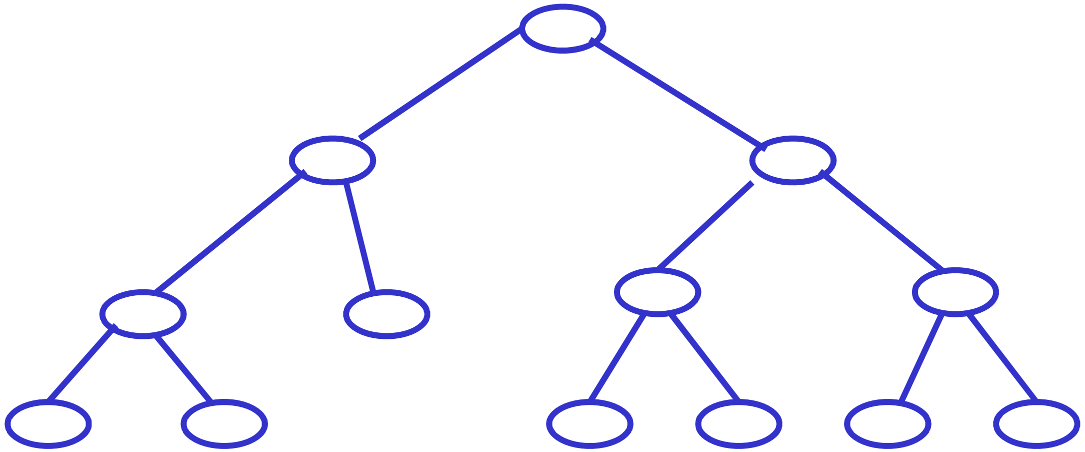
Question: B having released R, requests T, what may happen?

How to solve? Using “Lamport time stamps“ per message

Hierarchical Deadlock Detection

- hierarchy of deadlock detectors (controllers)
- waiting graphs (union of waiting graphs of children)
- deadlocks resolved at lowest level possible

Hierarchical Deadlock Detection



- Each node in tree (except of a leaf node) keeps track of the resource allocation information of itself and of all “kids”
- A deadlock that involves a set of resources will be detected by the node that is the common ancestor of all nodes whose resources are among the objects in conflict.

Simple Distributed Deadlock Detection[1]

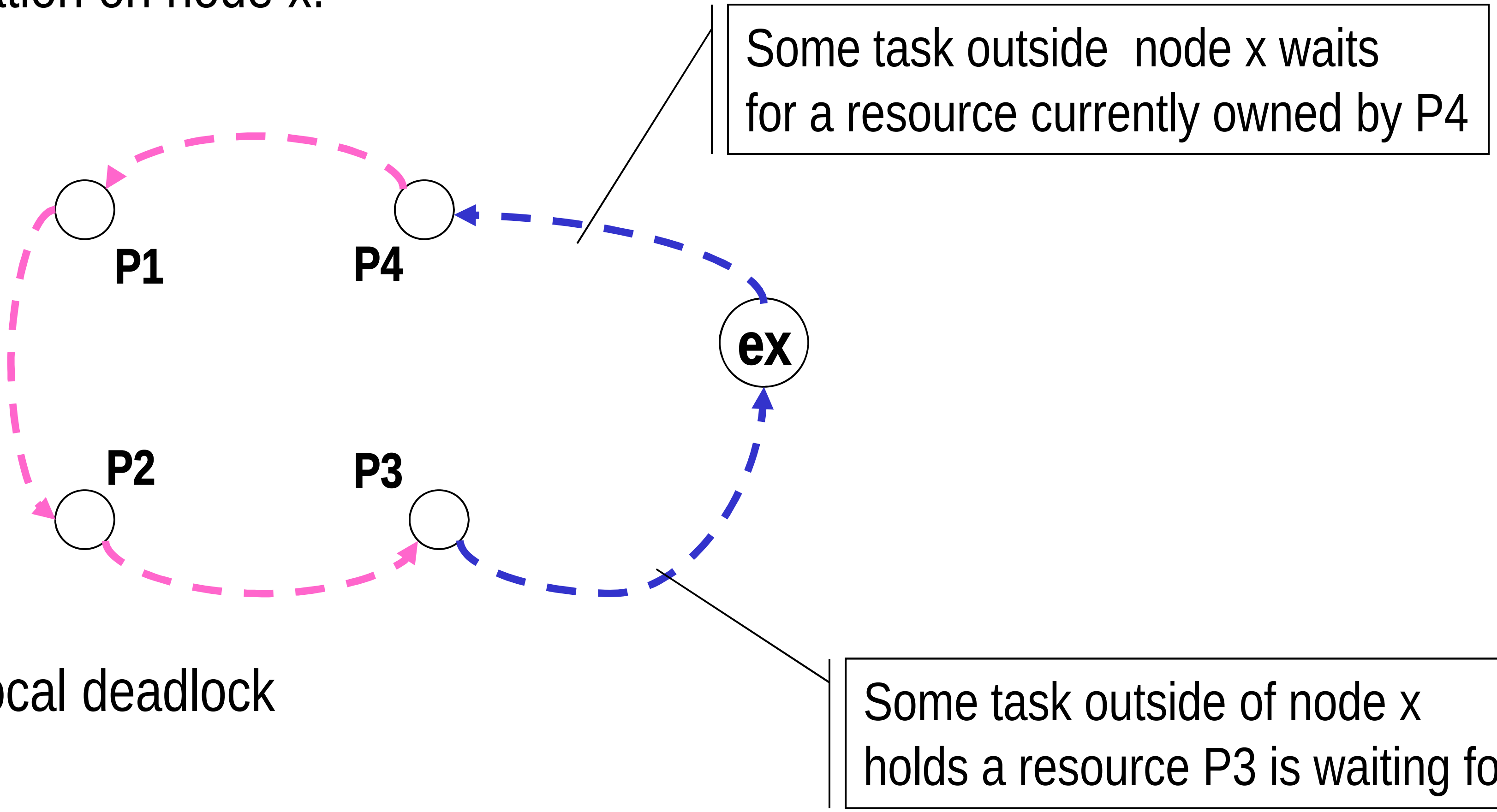
- no global waiting-graph
- deadlock detection cycle:
  - wait for information from other nodes
  - combine with local waiting-information
  - break cycles, if detected
  - share information on potential global cycles

Remark: The non-local portion of the global waiting-graph is an abstract node “ex”

[1] Obermark, 1982

Simple Distributed Deadlock Detection

Situation on node x:



Some task outside node x waits for a resource currently owned by P4

Some task outside of node x holds a resource P3 is waiting for.

No local deadlock

Distributed Deadlock Detection[1]

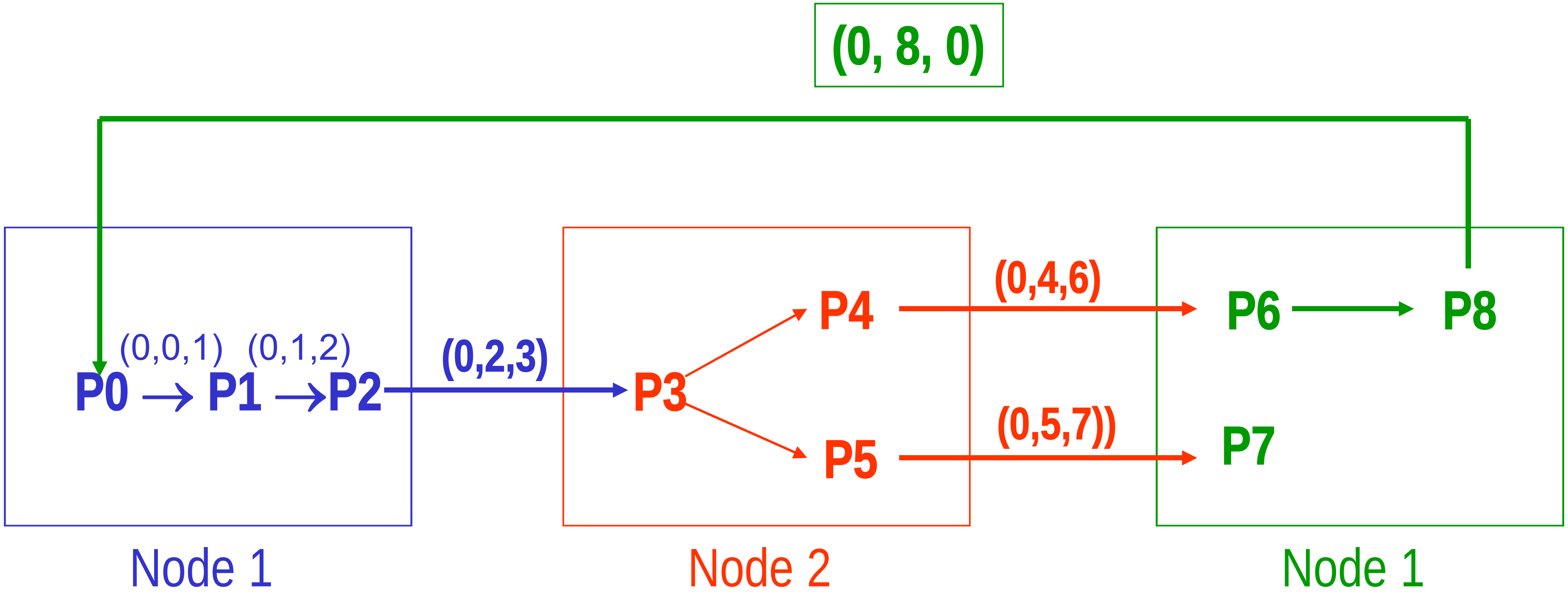
- A probe message <i, j, k> is sent whenever a task blocks
- This probe message is sent along the edges of the waiting-graph if the recipient is waiting for a resource
- If this probe message is sent to the initiating task, then there is a deadlock

[1] Chandy/Misra/Haas 1983)

Distributed Deadlock Detection

- If P has to wait for resource R it sends a message to current resource-owner O
- This message contains:
    - PID of waiting task P
    - PID of sending task S
    - PID of receiving task E
- Receiving process E checks, if E is also waiting. If so, it modifies the message:
    - First component of message still holds
    - 2. Component is changed to: PID(E)
    - 3. Component is changed to PID of that process, process E is waiting for.
- If message ever reaches waiting process P $\Rightarrow \exists$ deadlock

Example of DDD in DS



Distributed Deadlock Detection

Recommended Reading:

Knapp, E.:	Deadlock Detection in Distributed Databases, ACM Comp. Surveys, 1987
Sinha, P.:	Distributed Operating Systems: Concepts and Design, IEEE Computer Society, 1996
Galli, D.:	Distributed Operating Systems: Concepts and Practice, Prentice Hall, 2000

Deadlocks with Communication

1. Deadlocks may occur if each member of a specific group is waiting for a message of another member of the same group.

2. Deadlocks may occur due to unavailability of message buffers etc.

3. Study for yourself: Read Stallings: Chapter 14.4., p. 615 ff

Recommended Literature

http://link.springer-ny.com/link/service/series/0558/tocs/t2584.htm

A. Schiper, A.A. Shvartsman, H. Weatherspoon, B.Y. Zhao (Eds.): *Future Directions in Distributed Computing*
Research and Position Papers (currently online available)

Part I: Foundations of DS: What to expect from theory?

Part II. Exploring Next-Generation Communication Infrastructures and Applications

Part III. Challenges in Distributed Information and Data Management

Part IV. System Solutions: Challenges and Opportunities in Applications of Distributed Computing Technologies