2. Distributed Algorithms

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2019/2020

2.A. Time and Global States

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2019/2020

Contents

- **Time**
- Global states

Time

- Time in unambiguous in centralized systems
  - System clock keeps time, all entities use this
- No global time on distributed systems
  - Each node has a (crystal-based) system clock
    - Less accurate than atomic clocks
    - Results in **clock skew** (two clocks, two times) and **clock drift** (two clocks, two count rates)
      - Drifts 1 second every 11 days w.r.t. a perfect clock
  - Problem: An event that occurred after another may be assigned an earlier time
    - Use physical and logical clocks to deal with this

Contents

- Time
  - **Physical clocks**
  - Logical clocks
- Global states

Physical clocks

- Physical clocks allow to synchronize nodes …
  i. with a master node (with a UTC receiver)
     - UTC: Universal Coordinated Time is an international standard based on atomic time
       – Broadcasted through short-wave radio and satellite
  ii. with one another
- … within a given bound
  – **Perfect** clock synchronization is not feasible
    - Synchronization limited by network jitter and clock drift
    - Typical accuracy of milliseconds

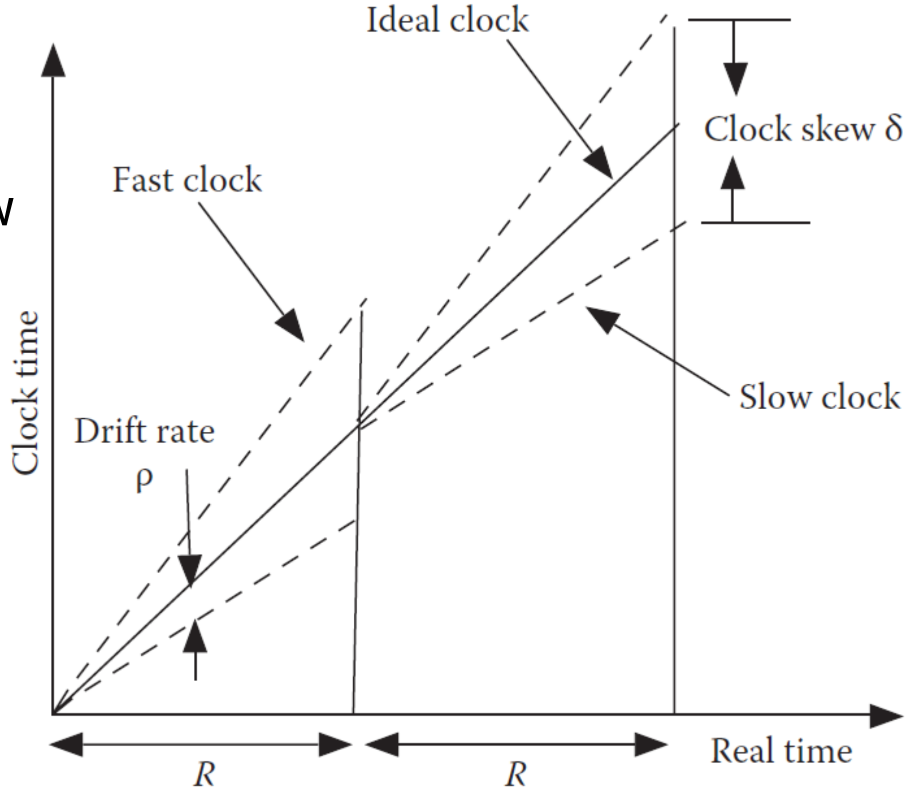
Physical clocks

- Synchronize at least every R < δ/2ρ to limit skew between two clocks to less than δ time units

R: resynchronization interval

ρ: maximum clock drift rate

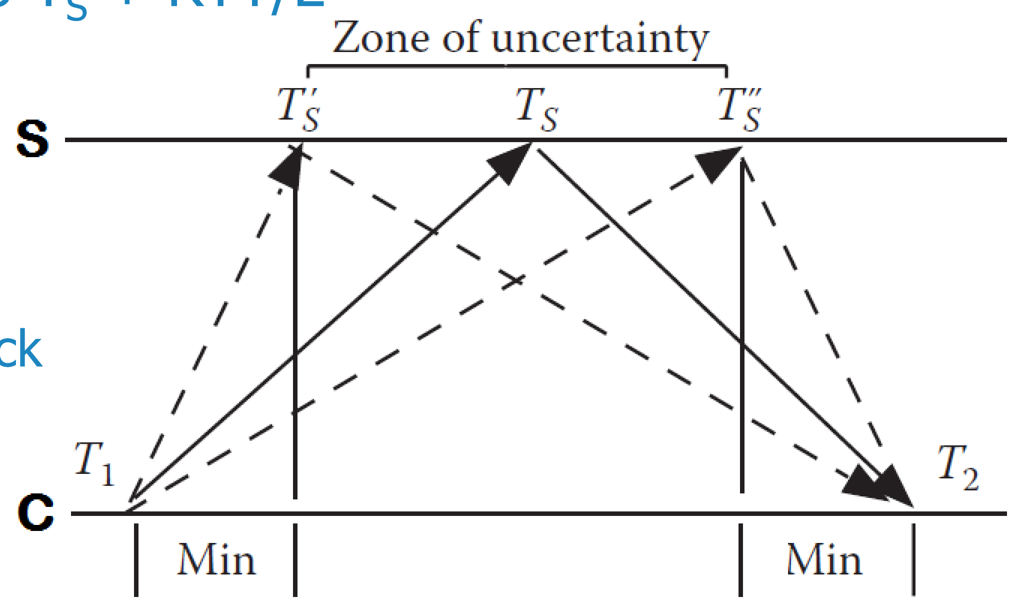
δ: maximum allowed clock skew



Cristian's algorithm

- Synchronize nodes with server with UTC receiver <u>within a given bound</u>: **External synchronization**
  - Intended for intranets with a UTC-sync server

1. Each client asks the time to the server at every R interval
2. Client sets the time to T_S + RTT/2
   - RTT: round-trip time
     - $T_2 - T_1$
   - Assumes symmetrical latency
   - Accuracy of client's clock is ±(RTT/2-Min)

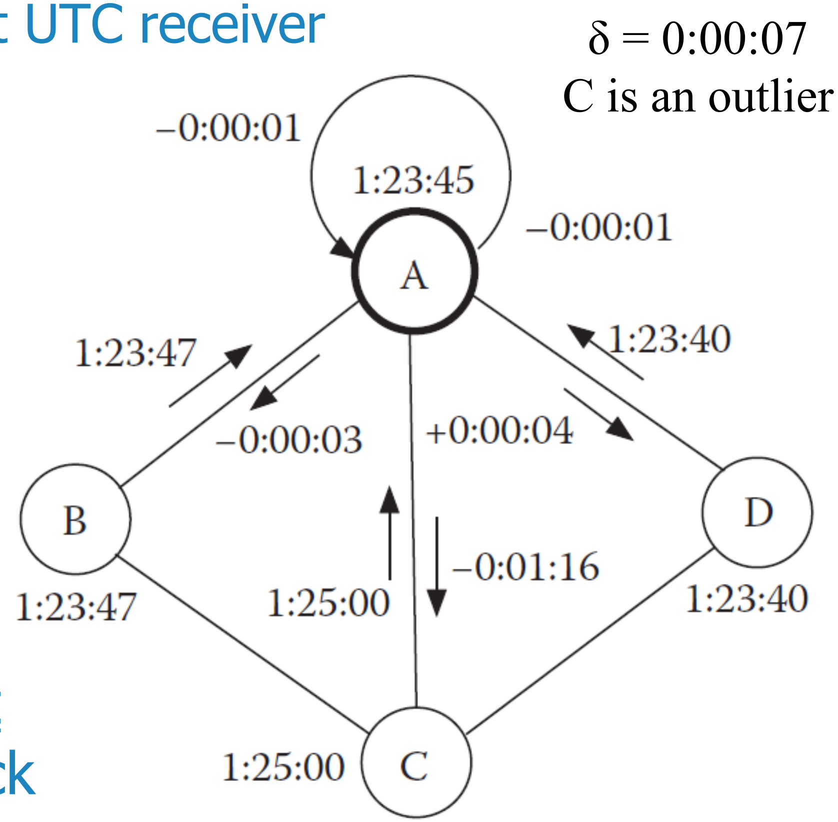


- NTP is based on the same concept

Berkeley algorithm

- Keep clocks synchronized with one another within a given bound: **Internal synchronization**
  - Intended for intranets without UTC receiver

1. Master polls the clocks of all the slaves at every R interval
   - Adapts them by considering round-trip times
2. Master calculates a fault-tolerant average
   - Clocks lying outside the given bound are discarded
3. Master sends the adjustment to be made to each local clock



Physical clocks

- Side effects of clock adjustments
  i. When setting the time forward
     - Some time instants are lost. This can affect potential events scheduled at these times
  ii. When setting the time back
     - Monotonicity property (time always moves forward) is violated. Future events can appear before than past ones
  - Workaround: Speed up or slow down local time until the adjustment has been achieved
- Clocks give a real time estimation but cannot be used deterministically to find out the **order** of any arbitrary pair of events

Contents

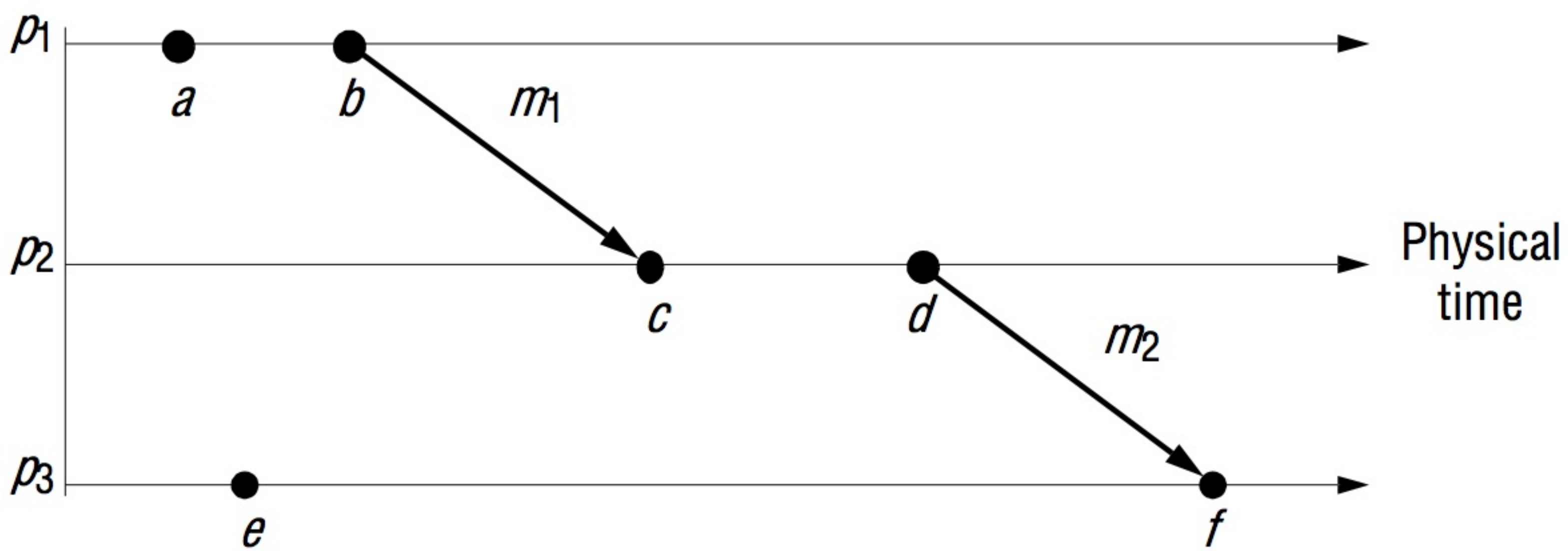
- **Time**
  - Physical clocks
  - **Logical clocks**
- Global states

Happened-before relation

- Processes need to know if event ‘a’ happened before or after event ‘b’
  - Agree on the **order** in which events occur rather than the **time** at which they occurred
- The happened-before relation
  - If **a** and **b** are two events in the same process, and **a** comes before **b**, then **a** $\rightarrow$ **b**
  - If **a** is the sending of a message, and **b** is the receipt of that message, then **a** $\rightarrow$ **b**
  - If **a** $\rightarrow$ **b** and **b** $\rightarrow$ **c**, then **a** $\rightarrow$ **c**

Happened-before relation

- Example:



- a→b, b→c, c→d, d→f, a→f but a||e (concurrent)
- The happened-before relation establishes a **partial** ordering

Logical clocks

- To capture the happened-before relation, we attach a timestamp **C(e)** to each event **e**, satisfying the following properties:
    1. If **a** and **b** are two events in the same process, and **a** → **b**, then we demand that **C(a) < C(b)**
    2. If **a** corresponds to sending a message **m**, and **b** to the receipt of **m**, then also **C(a) < C(b)**
- How to attach a timestamp to an event when there is no global clock?

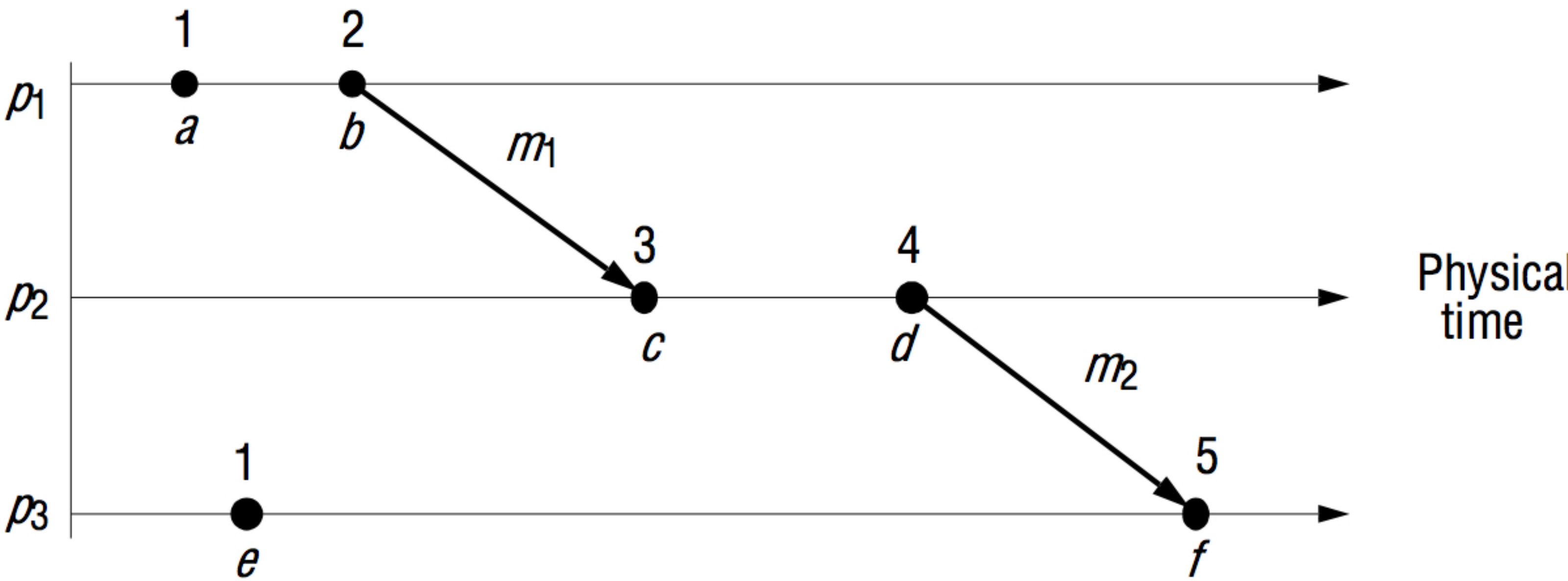
  ⇒ Use Lamport's logical clocks

Lamport's logical clocks

- Each process P_i maintains a local counter C_i
- C_i is used to attach a timestamp to each event at P_i
- C_i is adjusted according to the following rules:
  1. When an event happens at P_i, it increases C_i by 1
  2. When P_i sends message m to P_j, sets $ts(m) = C_i$
  3. When P_j receives m, sets $C_j = max(C_j, ts(m))$, and then increases by 1

Lamport's logical clocks

- Example



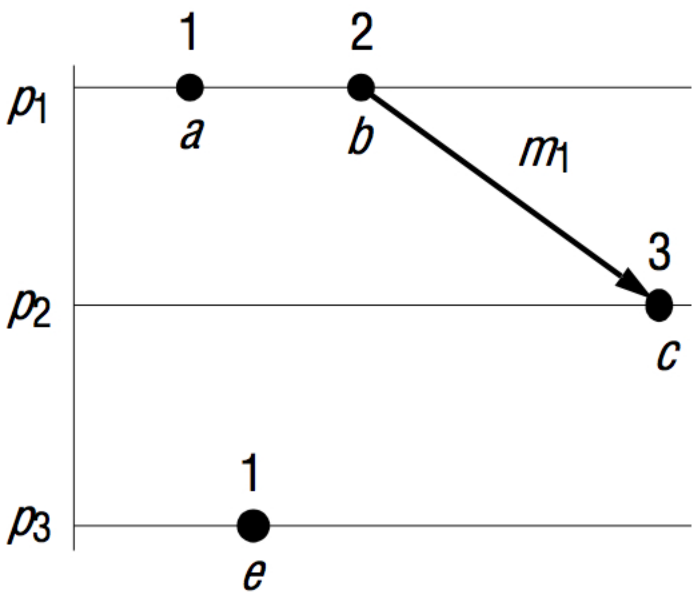
- Note that C(e) < C(b) but b||e

Lamport's logical clocks

- Lamport's clocks define a <u>partial</u> order that is consistent with the happened-before relation
  - i.e. it is consistent with causal order
- What if we need <u>totally-ordered</u> clocks?
  - Use Lamport's clocks, but in case two of them are equal, process IDs will be used to break the tie
    - $(C_i(a), i) < (C_j(b), j)$ iff
      - $C_i(a) < C_j(b)$ OR
      - $C_i(a) = C_j(b)$ AND $i < j$
  - Can be used for instance to order the entry of processes to a critical section

Logical clocks

- Lamport's clocks don't guarantee that if **C(a) < C(b)** then a <u>causally</u> preceded b (**a** $\rightarrow$ **b**)



- C(a) < C(c), and 'a' causally preceded 'c' (a $\rightarrow$ c)
- C(e) < C(c), but 'e' did not causally precede 'c' (c || e)

$\Rightarrow$ Use <u>vector clocks</u>

Vector clocks

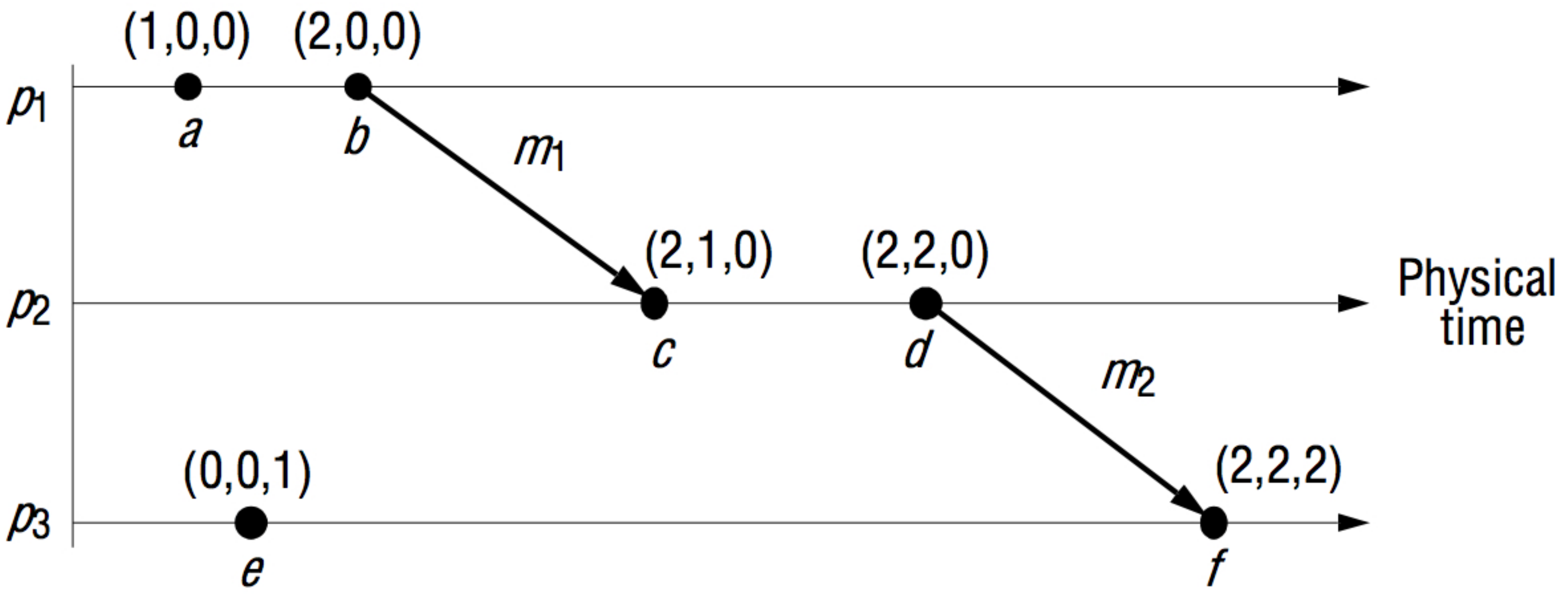
- Each process P_i has an array VC_i [1..n]
  - VC_i [j] denotes the number of events that process P_i knows have taken place at process P_j
    - i.e. VC_i [j] is the P_j logical clock at process P_i
- VC is adjusted as follows:
  1. When P_i sends a message m, it adds 1 to VC_i [i], and sends VC_i with m as <u>vector timestamp</u> ts(m)
  2. When P_j receives a message m from P_i, it updates each VC_j [k] to max{VC_j [k], ts(m)[k]} and then increments VC_j [j] by 1

Vector clocks

- Compare vector clocks to detect causality ($VC(a) < VC(b) \Leftrightarrow a \rightarrow b$)
  - $VC(a) < VC(b)$ iff
    - $VC(a) \leq VC(b)$ & $VC(a) \neq VC(b)$
  - $VC(a) \leq VC(b)$ iff
    - $VC(a)[k] \leq VC(b)[k]$, $k = 1 \ldots N$
  - $VC(a) = VC(b)$ iff
    - $VC(a)[k] = VC(b)[k]$, $k = 1 \ldots N$

Vector clocks

- Example



- Neither VC(b) ≤ VC(e) nor VC(e) ≤ VC(b), so b||e

Contents

- Time

- **Global states**

Global states

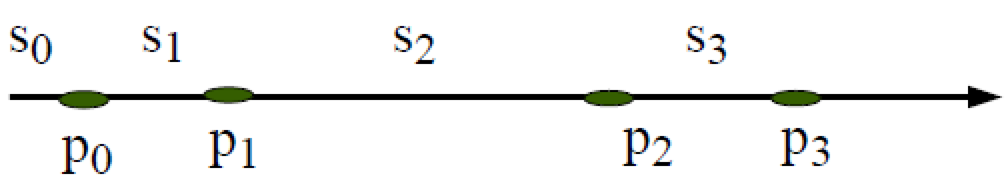
- Global state of the system is necessary for:
  - Failure recovery
    - In case of failure, recover by restoring the system to the last saved global state (i.e. checkpoint)
  - Detection of properties in the global state
    - Deadlock: are some processes mutually waiting to receive a message?
    - Termination: has a distributed program terminated?
      - Not as trivial as it seems since messages in transit can further activate a process
  - Debugging distributed software
    - The system is restored to a consistent global state and the execution resumes from there in a controlled manner

Global states

- Problem: how can we figure out the global state of a distributed system given that …
  1. Each process is independent
  2. There is not global clock and perfect clock synchronization is not feasible
     - If all processes had perfectly synchronized clocks, then they could agree on a time at which each process would record its state
  - We need to assemble a <u>meaningful</u> global state from local states recorded at different real times
- Let's start with some definitions

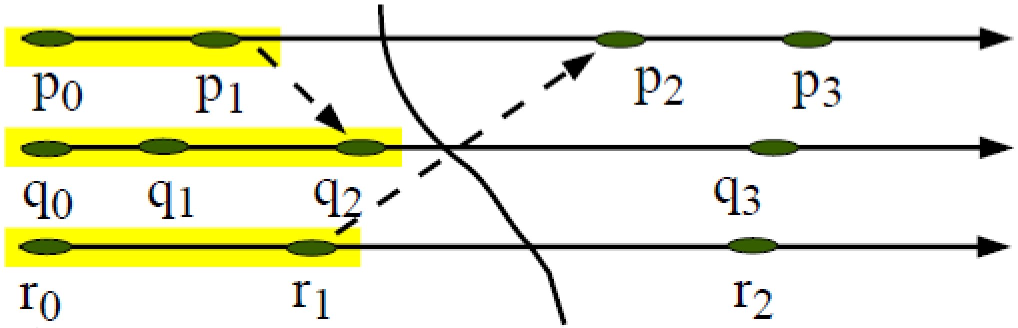
Global states

- The **history** of a process p, h(p), is the sequence of events that occur at the process
  - $h(p) = <p_0, p_1, \ldots>$
  - Each event either is an internal action of the process or is the sending or receipt of a message
- The **state** i of a process p, $s_i(p)$, is the finite prefix of the history before the i-th event
  - $s_i(p) = <p_0, p_1, \ldots, p_{i-1}>$



Global states

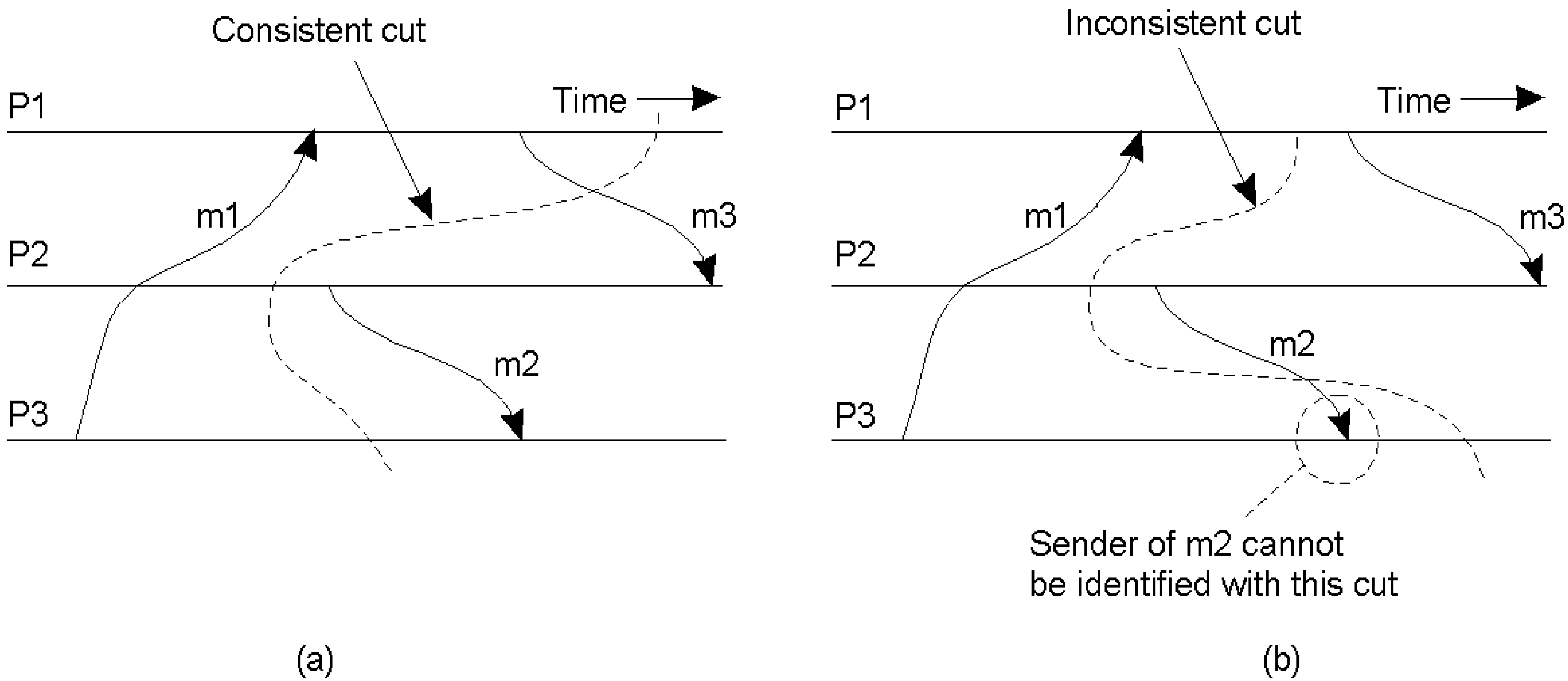
- The **global history** is the union of all the individual histories
- A **cut** is the global history up to a specific event in each process history
  - It is a union of prefixes of process histories (i.e., states), and corresponds to a **global state**
    - e.g., $S = (s_2(p), s_3(q), s_2(r))$



Global states

- A cut is **consistent** if for each event it contains, it also contains all the events that happened-before that event
  - i.e. if a process P has recorded the receipt of a message, there should also be a process Q that has recorded the sending of that message
- A consistent cut corresponds to a **consistent global state**
  - It is a possible state of the actual execution

Global states

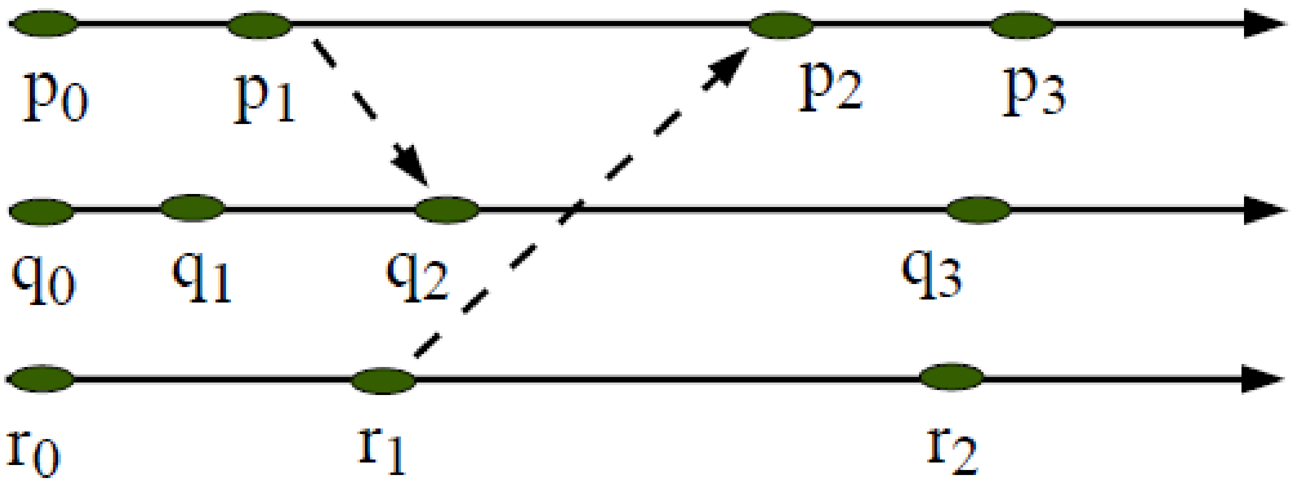


Linearization

- Execution goes through a series of transitions between global states ($S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$)
  - In each transition, precisely one event occurs at some single process in the system
- **Run**: total ordering of all events in a global history consistent with each local history
- A **linearization** or **consistent run** is a run consistent with the happened-before relation
  - It <u>only</u> passes through consistent global states
  - State S' is **reachable** from state S if there is a linearization passing through S and then S'

Linearization

- Ex: {p0, p1, q0, r0, q1, r1, p2, p3, q2, r2, q3}



$S_0=\{s_0(p), s_0(q), s_0(r)\}$
$S_1=\{s_1(p), s_0(q), s_0(r)\}$
$S_2=\{s_2(p), s_0(q), s_0(r)\}$
$S_3=\{s_2(p), s_1(q), s_0(r)\}$
$S_4=\{s_2(p), s_1(q), s_1(r)\}$
$S_5=\{s_2(p), s_2(q), s_1(r)\}$
$S_6=\{s_2(p), s_2(q), s_2(r)\}$
…

- If we collect all the events and we know the happened-before order, we can construct all possible linearizations
  - The actual execution took one of these paths

Contents

- Time

- **Global states**
  - **Distributed snapshot**
  - Global predicates

Distributed snapshot

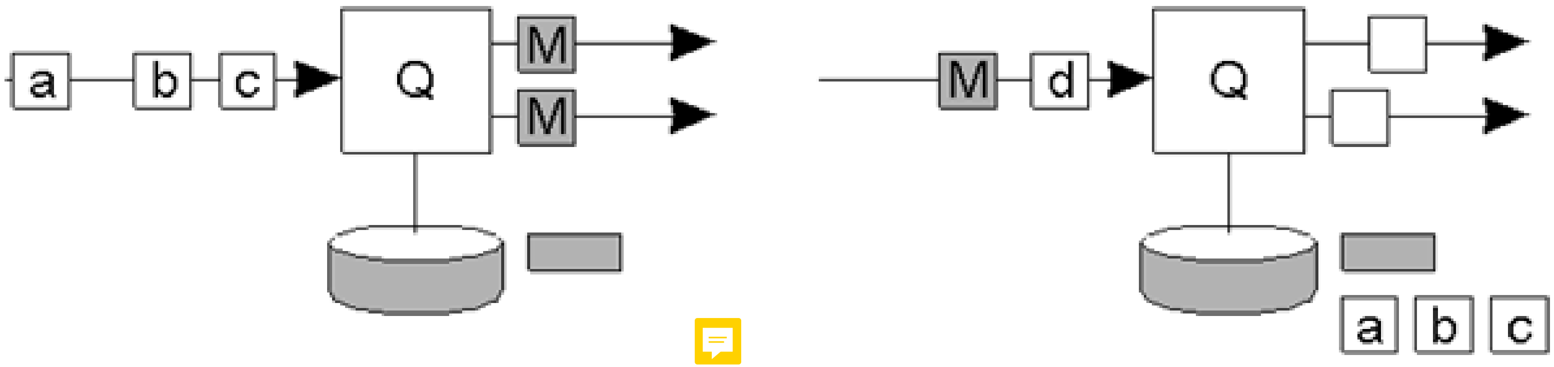
- How do we record a consistent global state?
  A. Freeze the entire system, record the states of the processes, and then resume the computation
     ↓ Interferes with the ongoing computation
     ↓ Algorithm should wait for all the computations to freeze and all the channels to be empty
  B. Chandy and Lamport's Snapshot Algorithm
     - Goal: Record a **consistent global state** while capturing messages that are in transit
     - Outcome: Fragments of a consistent global state are stored locally at processes
       - A further step is required to put these local states together (i.e. send them to a collector process)

Chandy-Lamport snapshot algorithm

- Assumptions
  - Processes and channels are reliable
  - Channels are unidirectional and FIFO
  - Topology is a strongly connected graph
    - There is a communication path between any two processes
  - <u>Processes may continue their execution while the snapshot takes place</u>
  - Any process may initiate a snapshot at any time
  - Separate snapshots may be initiated concurrently
    - They are distinguished by tagging the marker message

Chandy-Lamport snapshot algorithm

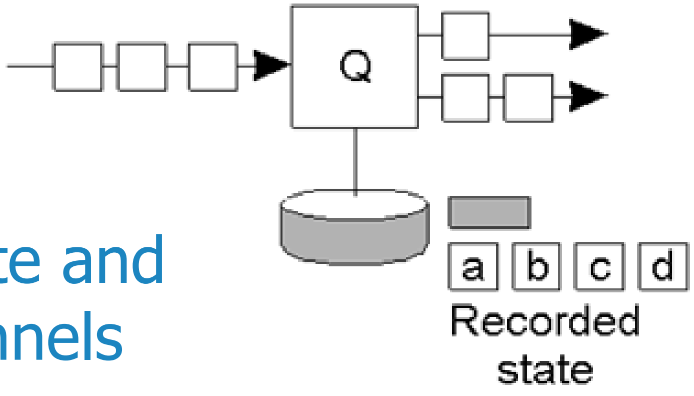
- Steps performed by the initiator process
  1. Record its own state
  2. Send special marker on every outgoing channel
  3. Start recording messages over all its incoming channels until a marker is received on each of those channels



Chandy-Lamport snapshot algorithm

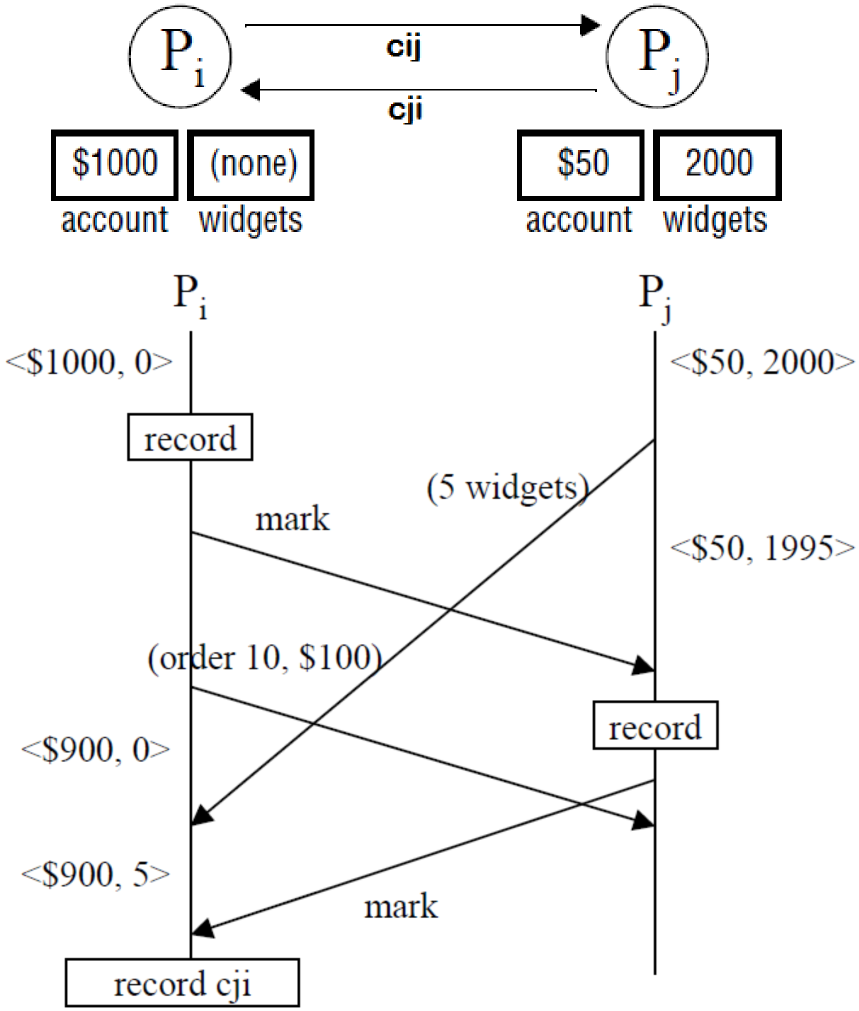
- On receiving a marker over channel c
  a) If process has not yet recorded its own state
     1. Record its own state
     2. Send special marker on every outgoing channel
     3. Record the state of c as the empty set
     4. Start recording messages over the other incoming channels until a marker is received on each of them

  b) Otherwise, record state of c as set of messages received over c since it saved its state



➢ Finish after recording process state and the state of ALL its incoming channels

Snapshot algorithm example



- Two processes trade in widgets at the rate of $10 per widget
- P_j is about to dispatch a previous order for 5 widgets to P_i
- P_i is about to order 10 more widgets
- P_i initiates the snapshot

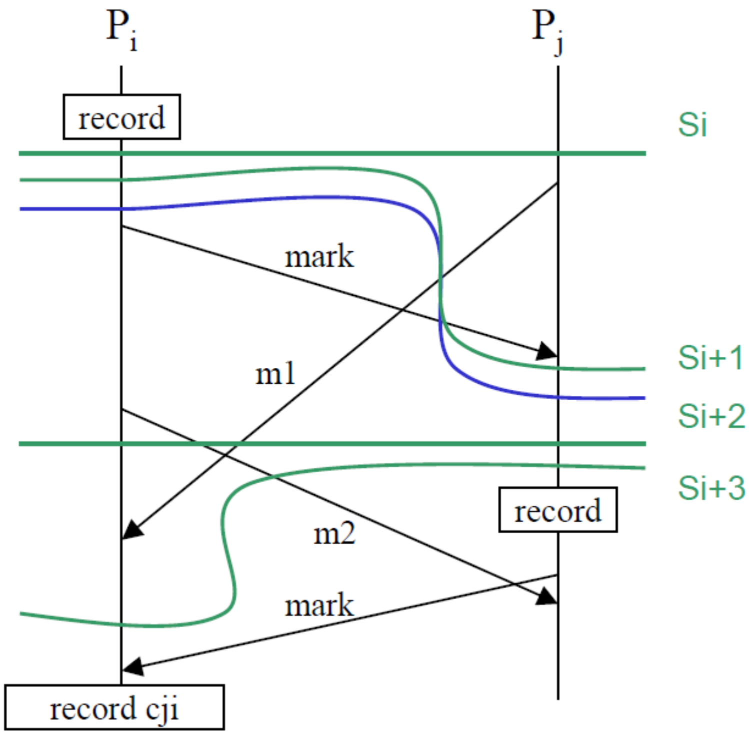
Observed state:

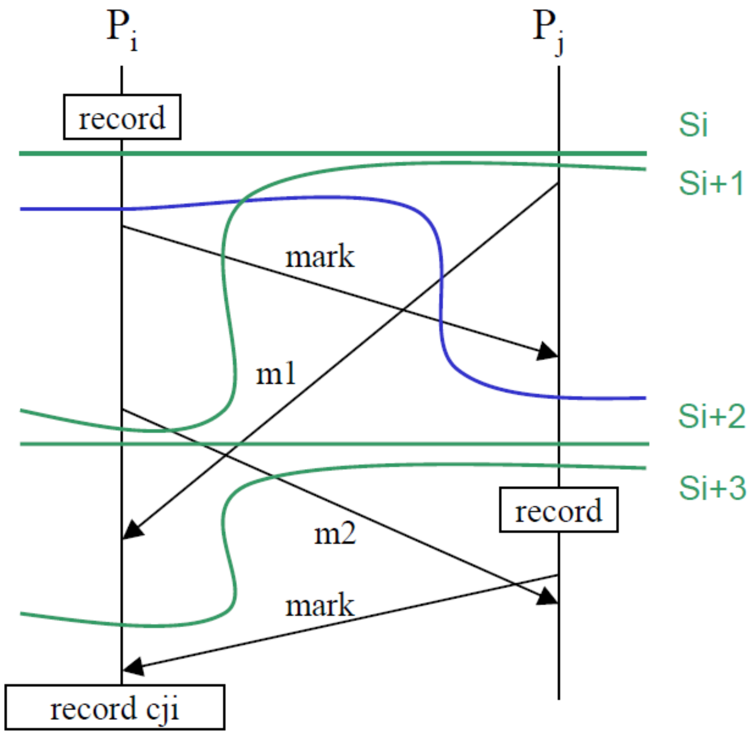
P_i: [$1000,0] ; c_{ji}: [(5 widgets)]

P_j: [$50,1995] ; c_{ij}: [ ]

Chandy-Lamport snapshot algorithm

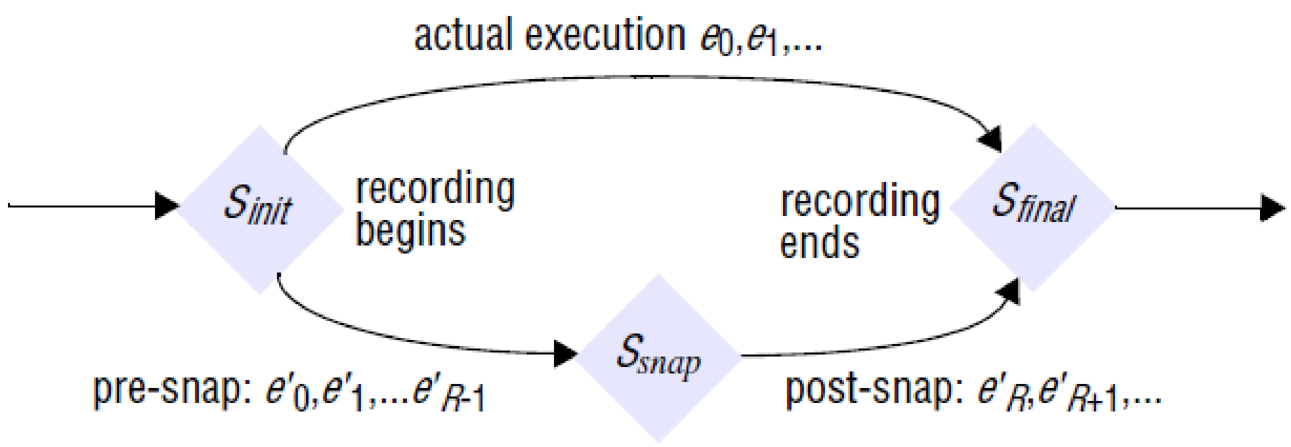
- A snapshot is a recording of a <u>consistent cut</u>
  - We have a recording of a global state that the execution might have passed through





Chandy-Lamport snapshot algorithm

- S_{final} is reachable from both the snapshot state S_{snap} and the actual execution
  - Stable properties of the actual execution can be evaluated in S_{snap}



Contents

- Time

- **Global states**
  - Distributed snapshot
  - **Global predicates**

Global state predicates

- A **global state predicate** is a property that is true or false for a global state
- Properties:

1. Safety
   - Safety for a (undesirable) predicate Φ means that Φ is false for all states reachable from S_0
2. Liveness
   - Liveness for a (desirable) predicate Φ means that Φ eventually evaluates to true for some state reachable from S_0 for any linearization from S_0

Global state predicates

3. Stability
   - **Stable**: if a predicate becomes true it remains true for all reachable states
     - e.g. deadlock, termination, object is garbage
     - <u>If a stable predicate is true in a snapshot then it is also true in the execution</u>
   - **Non-stable**: if a predicate can become true and then later become false
     - e.g. x=y
     - A non-stable predicate might be true in the snapshot but not necessarily during the actual execution
     - Can we make useful statements about whether a non-stable predicate occurred in an actual execution?

Non-stable predicates

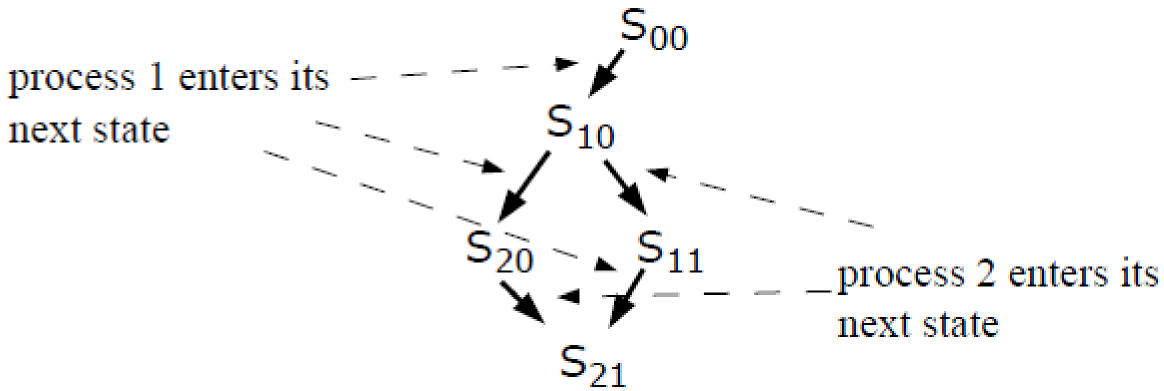
- We want to know if a non-stable predicate Φ **possibly** occurred or **definitely** occurred during the execution
  - Possibly Φ: there is a consistent global state S through which a linearization passes such that Φ(S) is true
  - Definitely Φ: for all the linearizations L, there is a consistent global state S through which L passes such that Φ(S) is true
- We must look at all possible execution states
  - How do we reconstruct all of them?

Non-stable predicates

- Process p_i records its state, time stamps it with process vector clock $V(s_i)$ and sends it to an external monitor process
  - Record initial state and when it changes
- Monitor collects states per process in a queue and groups them in a global state $S=\{s_0 \ldots s_n\}$
- A global state S is consistent if and only if $V(s_i)[i] \geq V(s_j)[i]$ for $i,j = 1,\ldots,N$
  - The 'happened-before' relationships are captured in the global state

Non-stable predicates

- Consistent global states form a **lattice** with reachability relation between states
  - The lattice is arranged in levels
    - S_{ij} is in level i+j



- The lattice shows all the linearizations corresponding to a global history

Non-stable predicates

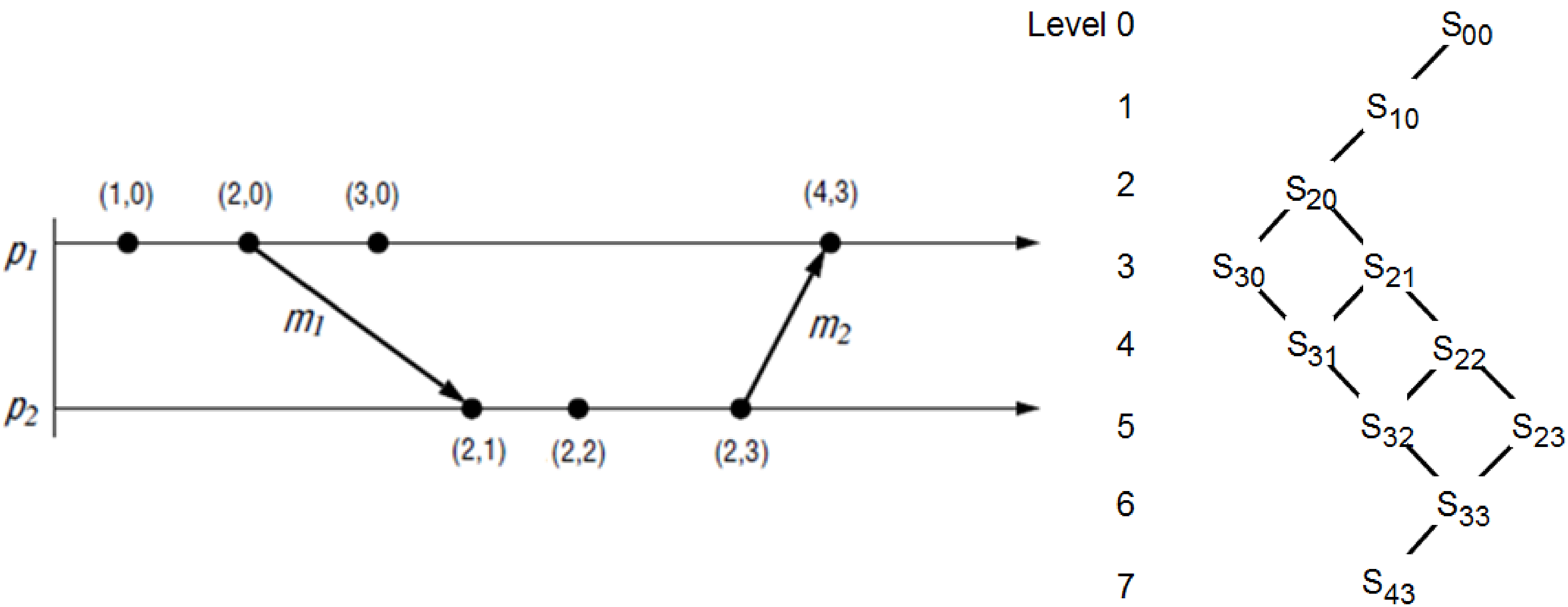
- How to evaluate predicate Φ using the lattice

1. Possibly Φ: Step through all the consistent global states of the lattice checking whether Φ is true for any of them

2. Definitely Φ: Step through all the paths from the initial to the final state of the lattice checking whether all the paths include a consistent global state for which Φ is true

Non-stable predicates

- Example:



Sij = global state after *i* events at process 1 and *j* events at process 2

Summary

- We can synchronize physical clocks, but only within a given bound
- Use logical clocks to find out events ordering
- A snapshot can record a consistent state that the execution possibly entered
  - Evaluate stable predicates
- Using vector clocks we can time stamp states and construct all possible linearizations
  - Evaluate non-stable predicates
- Further details:
  - [Tanenbaum]: chapters 6.1 and 6.2
  - [Coulouris]: chapter 14

2.B. Coordination and Agreement

Concurrence, Parallelism and Distributed Systems (CPDS)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2019/2020

Introduction

- Problem: A set of processes coordinate their actions or agree on one or more values in a distributed system
- This is needed for the following problems
  - Election of a new coordinator of a group of processes when the previous one has failed
  - Agree on which messages a group of processes receive and in which order
  - A set of processes agree on some value

Contents

- **Election algorithms**
- Multicast communication
- Consensus

Election algorithms

- Many distributed algorithms need one process to act as coordinator
  - e.g. centralized mutual exclusion, physical clock synchronization, primary-based replicated groups
  - No matter which process is, just need to pick one
    - Without loss of generality, we require that the elected process be chosen as the one with the largest identifier
- Election algorithms: technique to pick a **unique** coordinator (a.k.a. leader election)
  - Ensure that an election concludes with all the processes agreeing on who the new coordinator is

Election algorithms

- Desired properties for election algorithms:
  1. **Safety:** a participant is either non-decided or decided with the non-crashed process with the largest ID
  2. **Liveness:** all processes eventually participate & either decide on a elected coordinator or crash
- One could suggest to use a mutual exclusion algorithms to elect a leader, but:
  – Starvation is irrelevant in leader election
  – Exit from the critical section would be unnecessary
  – Leader needs to inform the rest about its identity

Election algorithms

- Any process P can initiate an election (several elections can run concurrently) but it can only initiate one at a time
  - When P notes that coordinator is not responding or P has just recovered from failure, it can initiate an election
- Algorithms assume that process identifiers are <u>unique</u> and <u>totally ordered</u>
- The goal is finding <u>the process with largest ID that is up</u> and make this the new coordinator

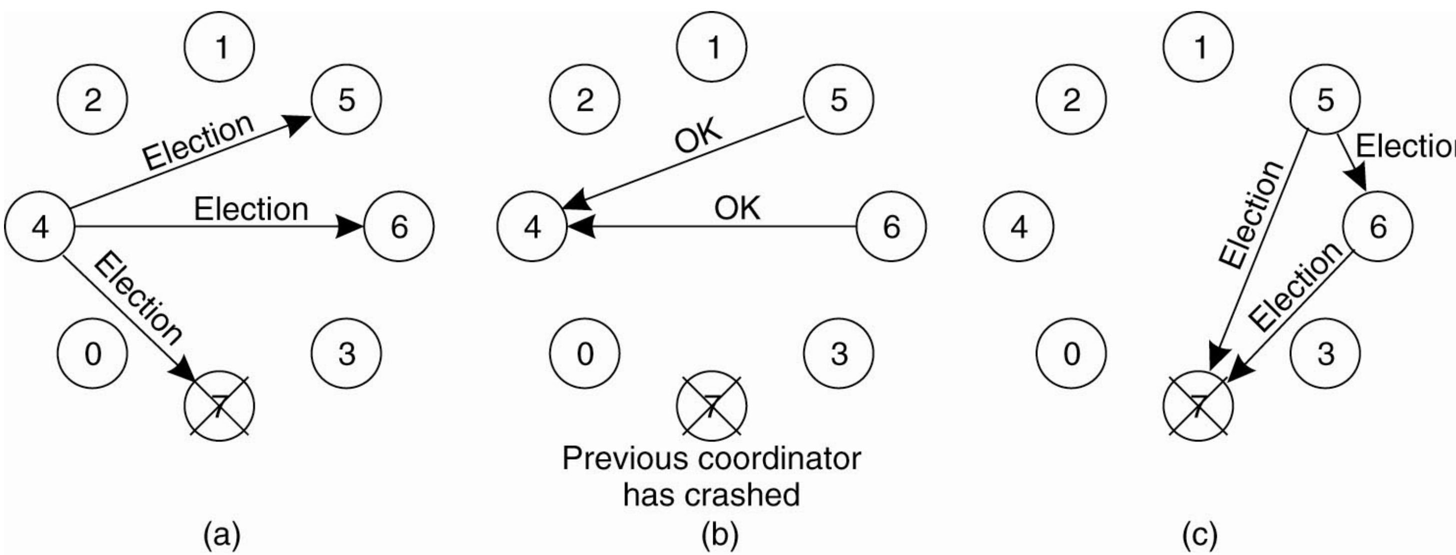
Bully algorithm

- Assumptions:
  - The system is <u>synchronous</u>
    - It can use **timeouts** to detect process failures and processes not responding to requests
  - Topology is a strongly connected graph
    - There is a communication path between any two processes
  - Message delivery between processes is reliable
  - Every process knows the ID of every other process, but not which ones are now up and which ones are down

Bully algorithm

a) P sends an **Election** message to all the processes with higher IDs and awaits **OK** messages

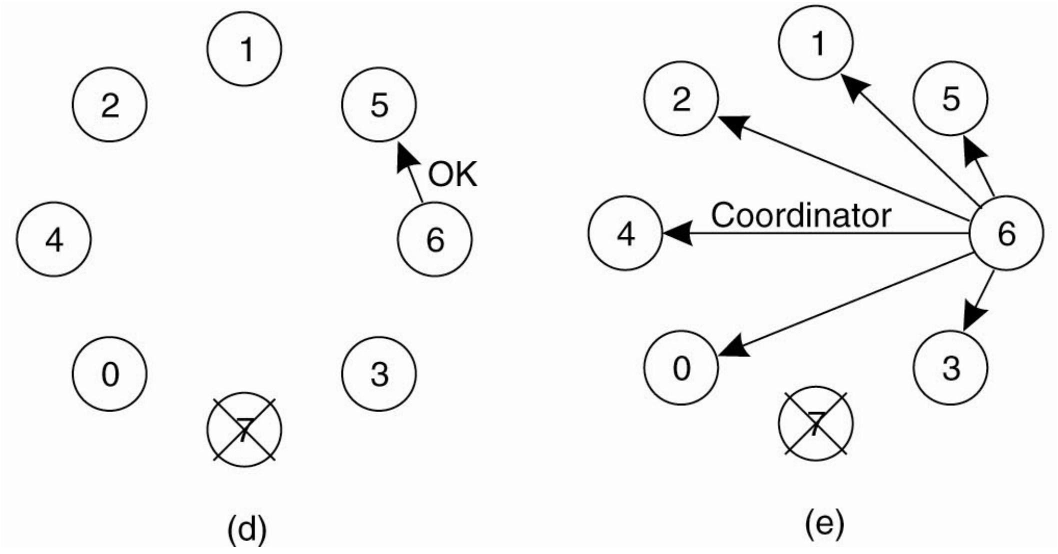
b) If a process receives an **Election** message, it returns an **OK** and starts another election, unless it has begun one already (c)



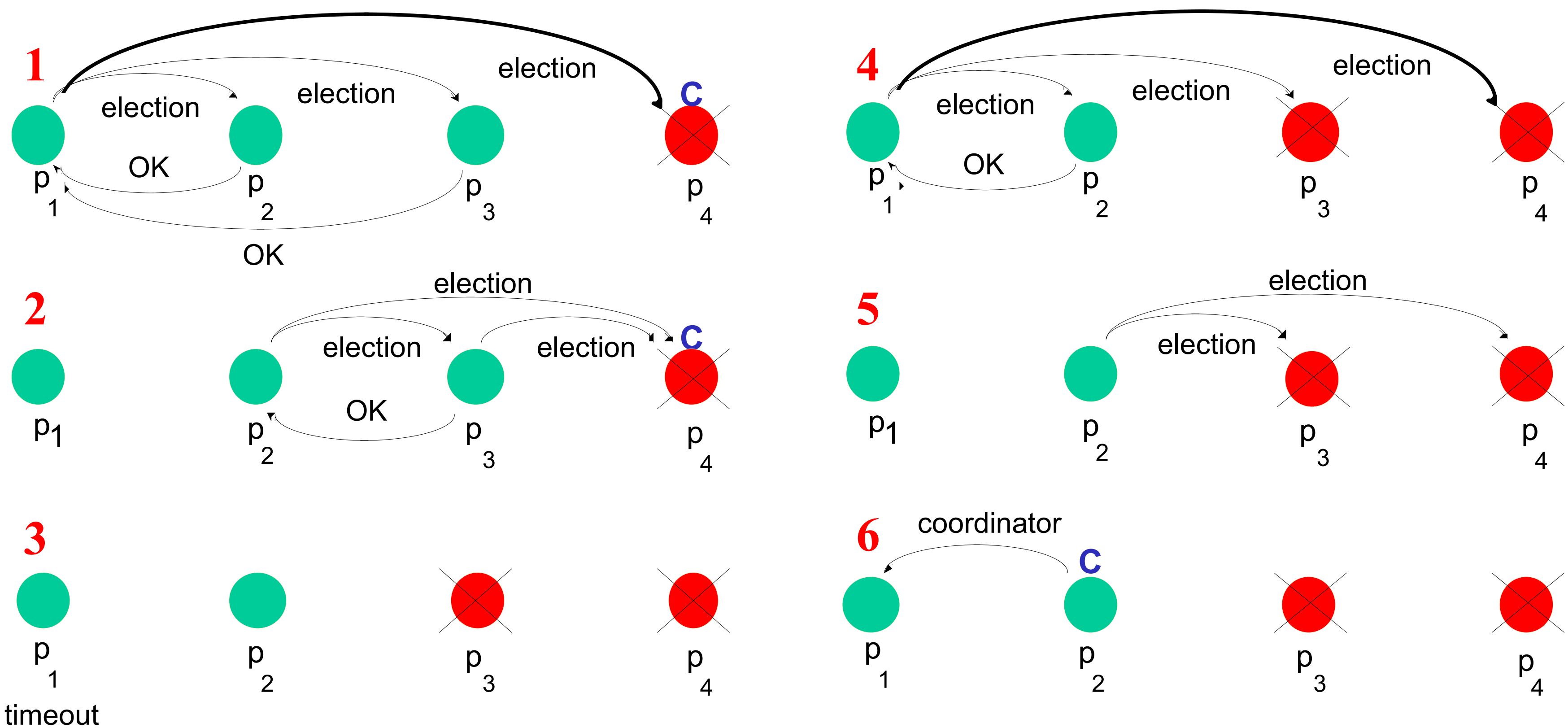
Bully algorithm

d) If P receives an **OK**, it drops out the election and awaits a **Coordinator** message (also (b))
   - P reinitiates the election if this message is not received

e) If P does not receive any **OK** before the timeout, it wins and sends a **Coordinator** message to the rest



Bully algorithm



It can violate the safety property if a crashed process with the highest ID restarts with an ongoing election (e.g. p_3 restarts during step 6)

Chang and Roberts' ring algorithm

- Processes are organized by ID in a logical unidirectional ring
  - Each process only knows its successor in the ring
- Assumes that system is asynchronous
- Multiple elections can be in progress
  - Redundant election messages are killed off

1. P sends an **Election** message (with its ID) to its successor, and becomes a participant
2. On receiving an **Election**, Q compares the ID in the message with its own ID

Chang and Roberts' ring algorithm

a) If the arrived ID is greater, Q forwards the message to its successor

b) If the arrived ID is smaller …
   - If Q is not a participant yet, Q replaces the ID in the message with its own ID and forwards it
   - If Q is already a participant, message is not forwarded

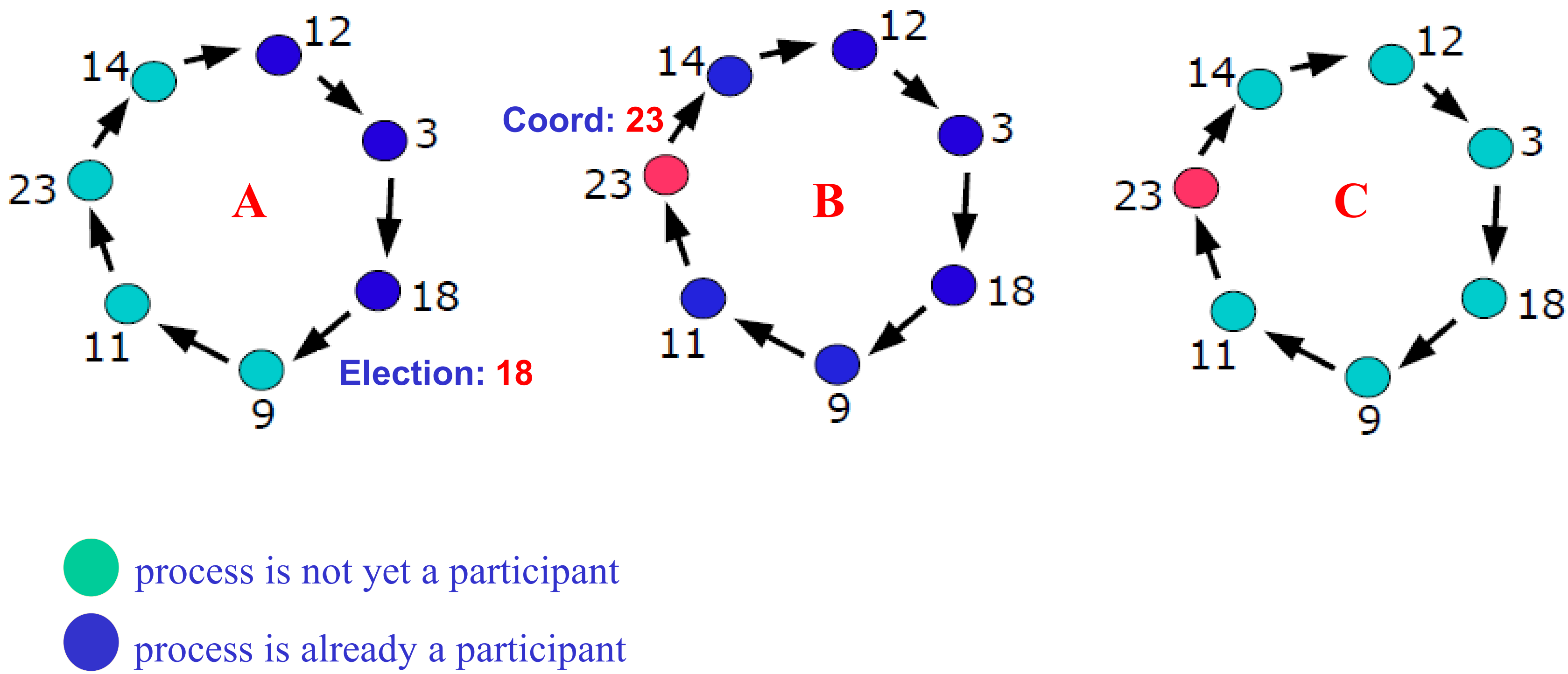
c) If the arrived ID is Q's ID, Q wins and sends a **Coordinator** message to its successor

– Q becomes a participant on forwarding an Election

3. When Q gets a **Coordinator** message, it forwards it to successor (unless Q is the new coordinator) and becomes a non-participant

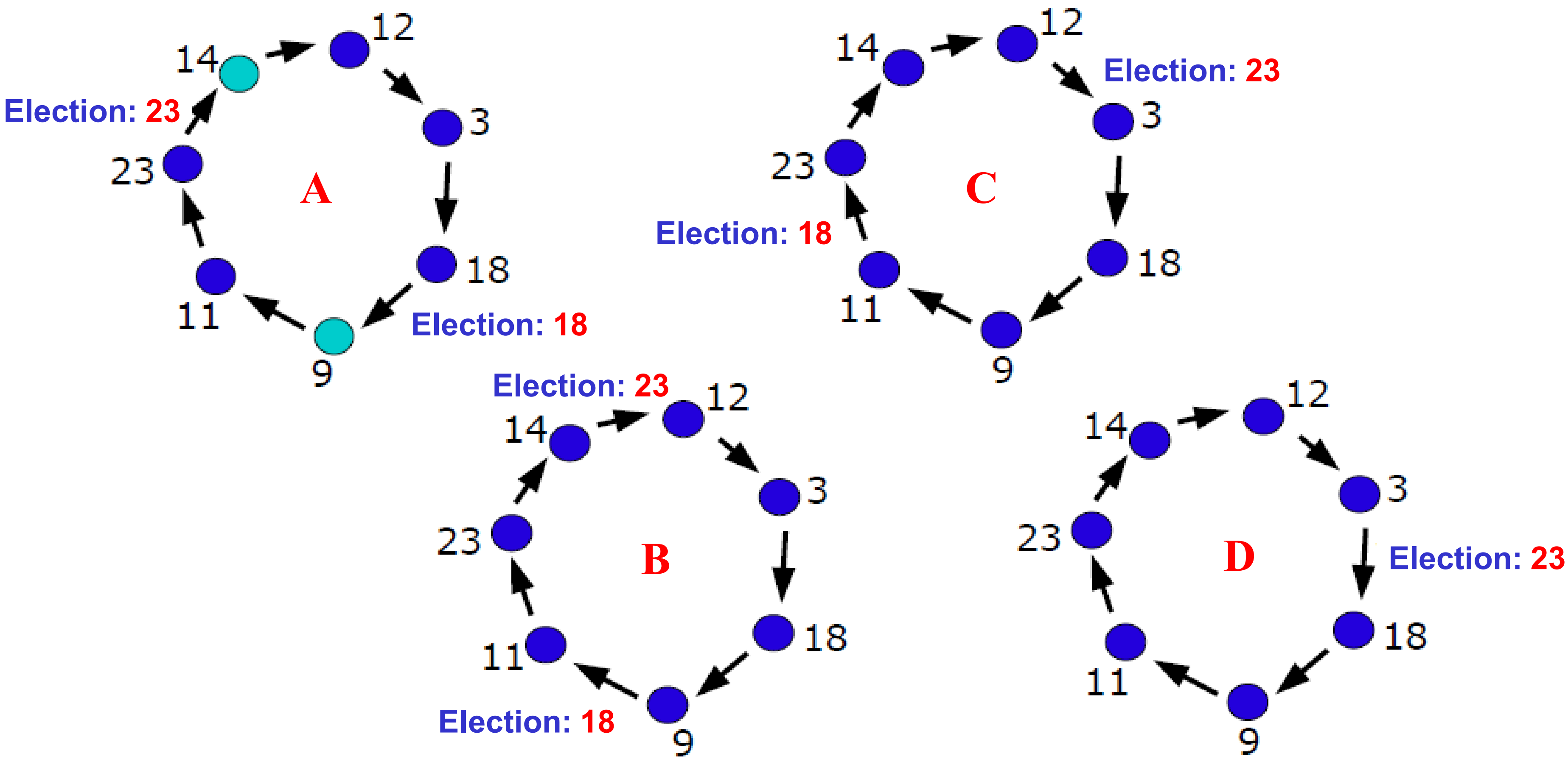
Chang and Roberts' ring algorithm

- Example: one election in progress



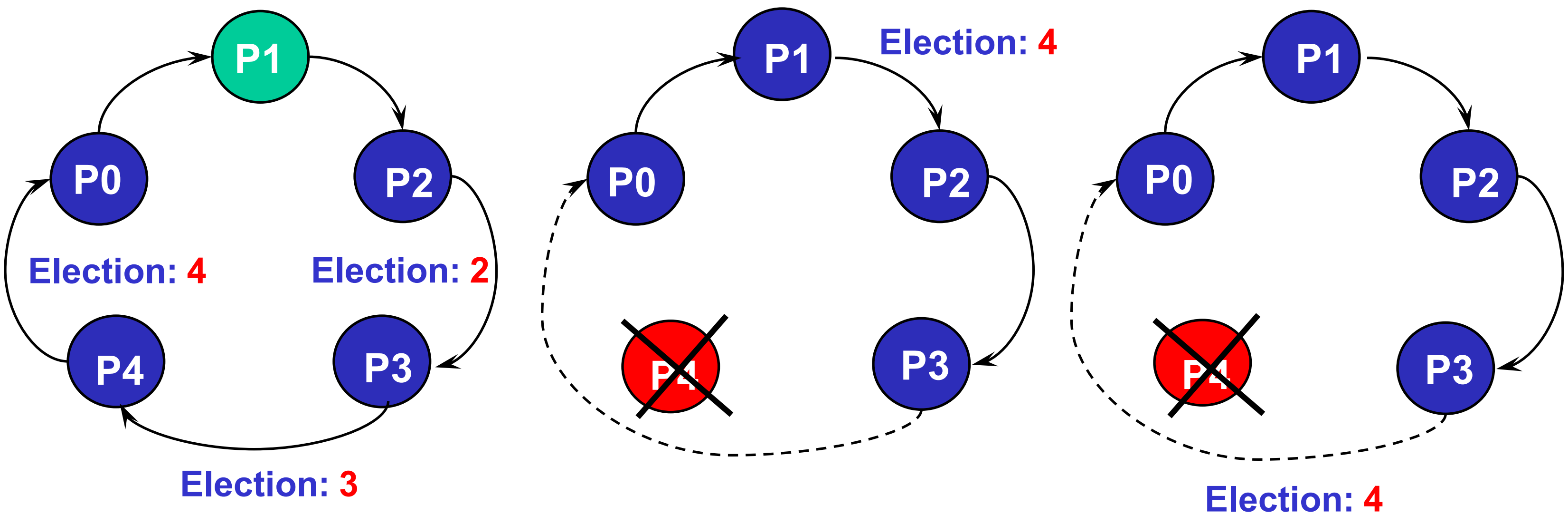
Chang and Roberts' ring algorithm

- Example: two elections in progress



Chang and Roberts' ring algorithm

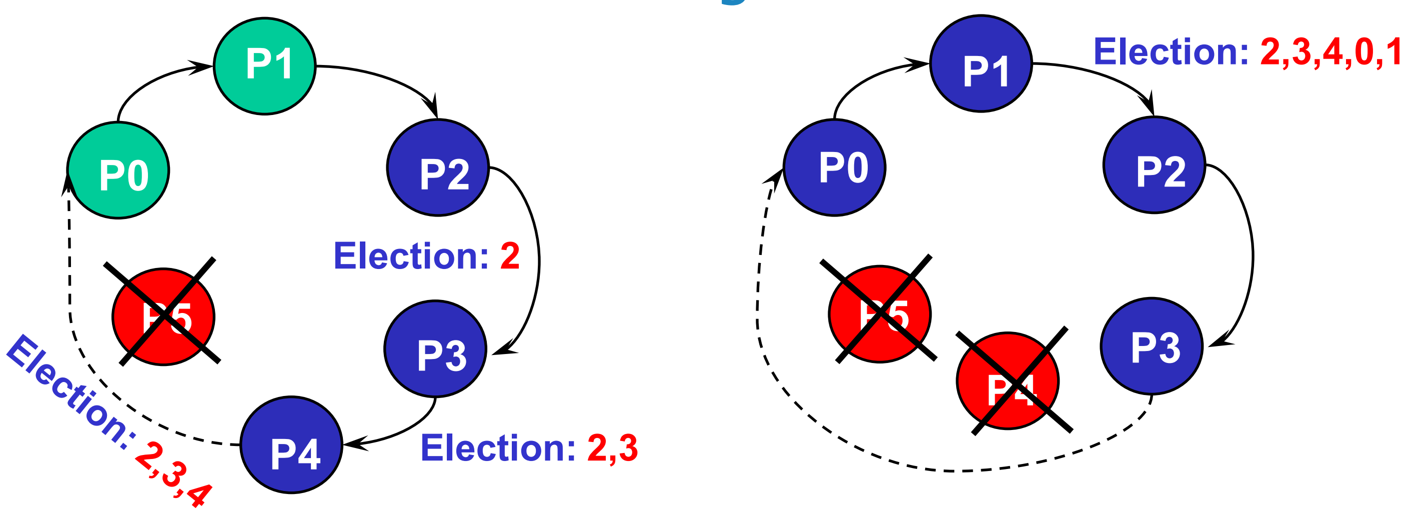
- Liveness violated when process failure occurs during the election
  - Ex: Which node will recognize 'Election: 4'?



Enhanced ring algorithm

a) P sends an **Election** message (with its process ID) to its closest alive successor
   - Sequentially poll successors until one responds
     - Each process must know all nodes in the ring

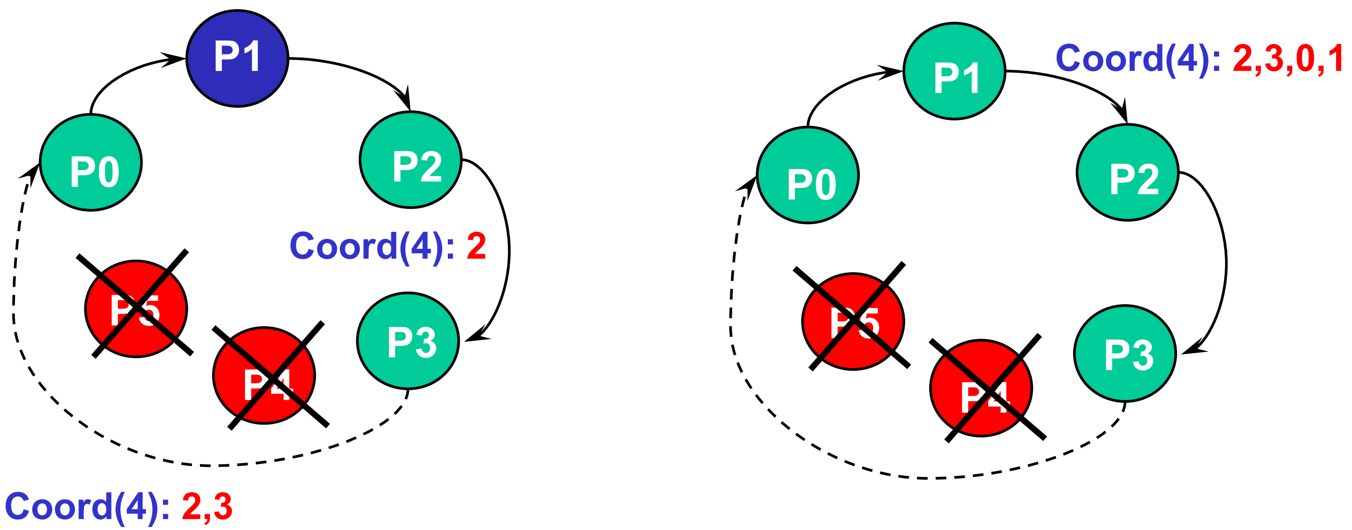
b) At each step along the way, each process adds its ID to the list in the message



Enhanced ring algorithm

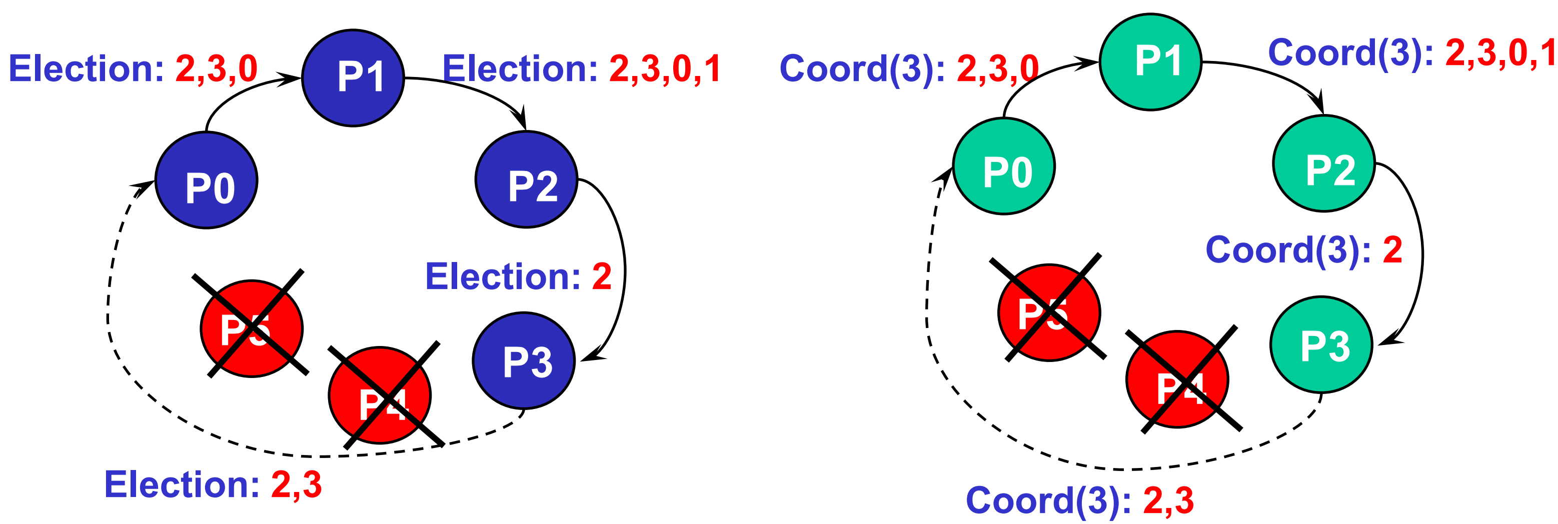
c) When the message gets back to the initiator (i.e. the first process that detects its ID in the message), it elects as coordinator the process with the highest ID and sends a **Coordinator** message with this ID

d) Again, each process adds its ID to the message



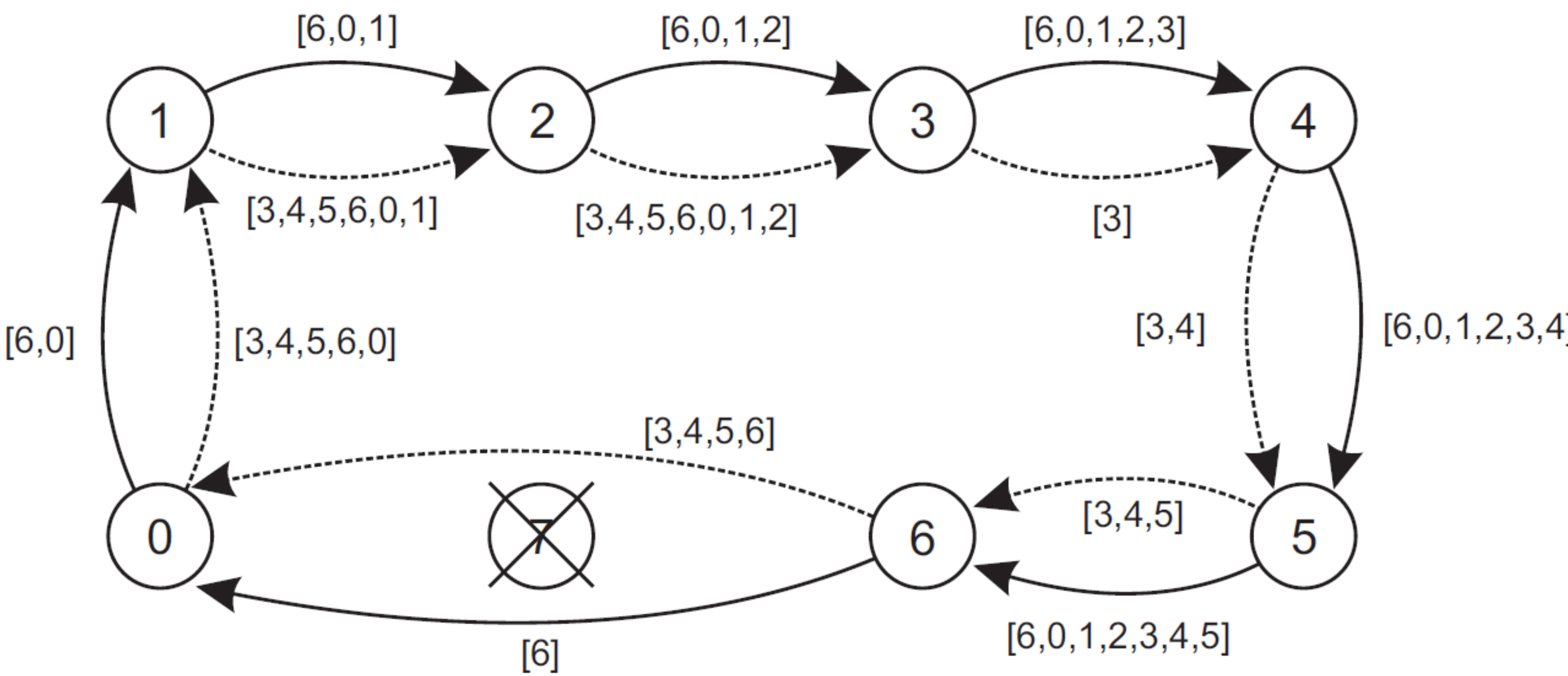
Enhanced ring algorithm

e) Once **Coordinator** message gets back to initiator
   - If elected process is in the ID list, election is over
     - Everyone knows who the coordinator is and who the members of the new ring are
   - Otherwise, the election is re-initiated



Enhanced ring algorithm

- ↑ Algorithm can support failures during the election
- ↓ Redundant election messages are not killed off



Comparison of election algorithms

(*) Assuming N processes, no failures, and a sole election in progress

- Bully algorithm
  - Worst case: initiator has the lowest ID: $\Theta(N^2)$ messages
    - Triggers N-1 elections: $\sum_{i=1}^{N-1}\big((N-i)\ Election + (N-i)\ OK\big) + (N-1)\ Coordinator$
  - Best case: initiator has the highest ID: N–1 Coordinator messages, which can be sent in parallel
- Chang and Roberts' ring algorithm
  - Worst case: initiator succeeds the node with the highest ID: 3N–1 (2N–1 Election + N Coordinator) sequential messages
  - Best case: initiator has the highest ID: 2N (N Election + N Coordinator) sequential messages
- Enhanced ring algorithm
  - 2N (N Election + N Coordinator) sequential messages always

Election algorithms

- Some of the presented algorithms can tolerate failures to some extent, but none of them can deal with network partitions
  - Multiple nodes (one for each network segment) may decide they are the leader
- Production systems typically use generic consensus algorithms for leader election
  - They tolerate failures and network partitions
  - They provide a single framework for all the agreement problems

Election algorithms

- e.g. Paxos (see ‘Consensus’ lesson)
  - Used by Google Chubby distributed lock service
    - Part of Google software stack (GFS, BigTable, …)
- e.g. ZooKeeper Atomic Broadcast (ZAB)
  - Part of Apache ZooKeeper coordination service
    - Used by Apache (Hadoop MapReduce & HBase, Solr, Kafka), Yahoo!, Rackspace
- e.g. Raft
  - Part of HydraBase by Facebook
  - Used by Kubernetes and Docker Swarm
  - Used by Consul by HashiCorp

Contents

- Election algorithms
- **Multicast communication**
- Consensus

Multicast communication

- Important service in distributed systems to:
  - Disseminate data reliably to large number of users
  - Implement collaborative applications where a common user view must be preserved
  - Implement consistency models of replicated data
  - Implement fault-tolerant (replicated) services
  - Monitor process groups and manage membership
  - e.g. JGroups (used for session replication and clustering in JBoss and JOnAS J2EE servers)
  - e.g. Isis (used by NY and Swiss Stock Exchange, French Air Traffic Control System, US Navy AEGIS)

Multicast communication

- Multicast: send a message to a process group
- Reliable multicast: deliver messages to all processes in a group or to none at all
  - Distinguish when the operating system **receives** a message and when is **delivered** to the application
- Ordered multicast: deliver messages while fulfilling ordering requirements
- Atomic multicast: deliver messages in the same order to all processes and any process can fail

Multicast communication

- Desired properties for reliable multicast:
  1. **Integrity:** A correct process delivers every message at most once
  2. **Validity:** If a correct process multicasts message m, then it will eventually deliver m
  3. **Agreement:** If a correct process delivers message m, then all other correct processes in the group will eventually deliver m

  ⇒ Sounds simple, but what happens …
    - … if a message is lost?
    - … if the sender crashes half-way sending the multicast?
    - … if a process joins the group during communication?

Contents

- Election algorithms
- **Multicast communication**
  - **Basic reliable multicast**
  - Scalable reliable multicast
  - Ordered multicast
  - Atomic multicast
- Consensus

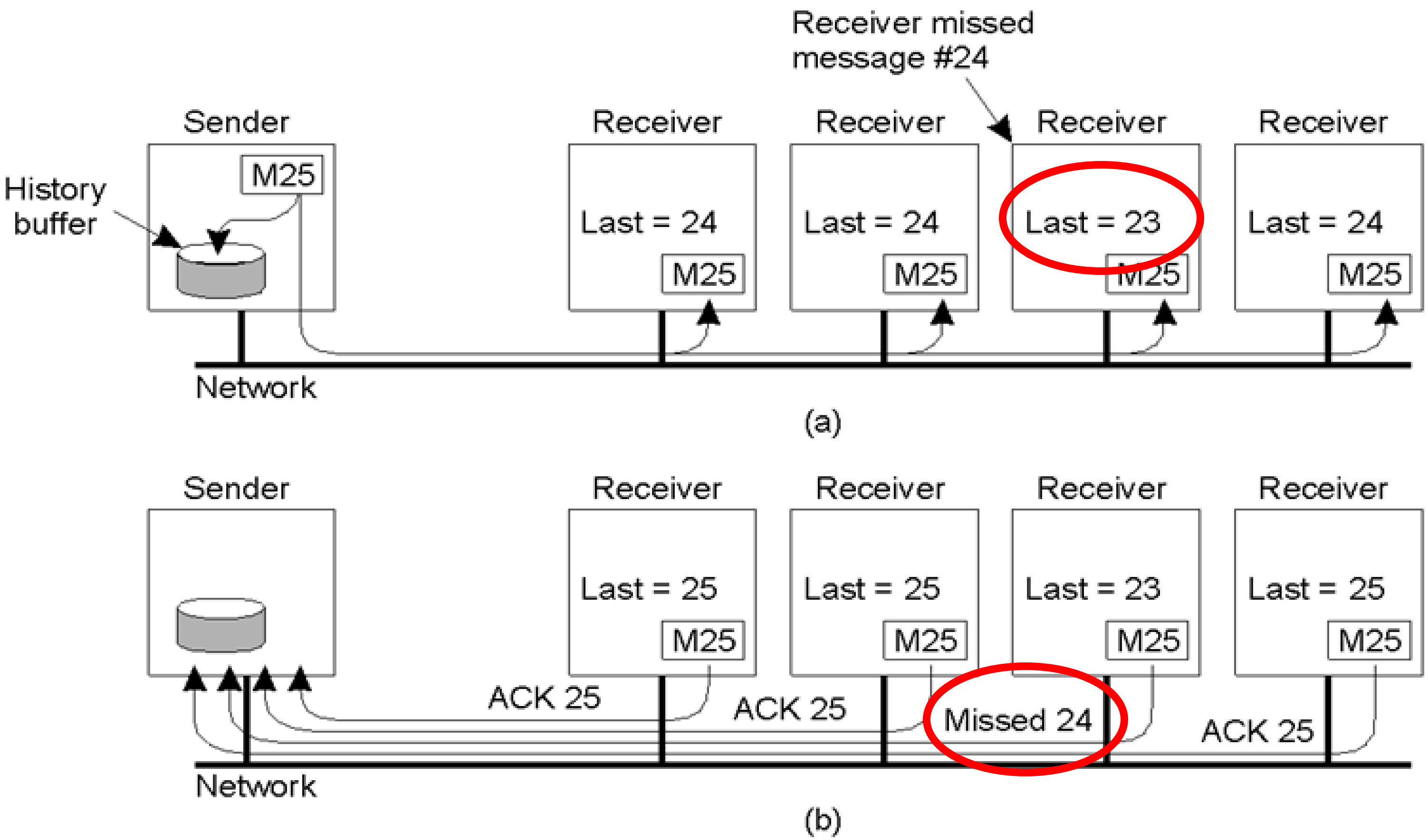
Basic reliable multicasting

- Simple solution assuming that processes do not fail and do not join/leave the group
  - Sender P assigns a sequence number S_P to each outgoing message
    - Makes easy to spot when a message is missing
  - P stores a copy of each outgoing message in a history buffer
    - P removes a message from the history buffer when everyone has acknowledged receipt
  - Each process Q records the number of the last message it has delivered coming from any other process P ($L_Q(P)$)

Basic reliable multicasting

- When process Q receives a message from P:
  - If $S_P = L_Q(P) + 1$: Q delivers the message, increases $L_Q(P)$ and acknowledges the receipt to P
  - If $S_P > L_Q(P) + 1$: Q keeps the message in a *hold-back queue* and requests the retransmission of missing messages
    - Queued message will be delivered (and acknowledged) when its sequence number is the next expected number
    - Retransmissions are also multicast messages
  - If $S_P <= L_Q(P)$: Q has already delivered the message before and thus it discards it

Basic reliable multicasting



↓ Poor scalability: too many ACKs (feedback implosion)

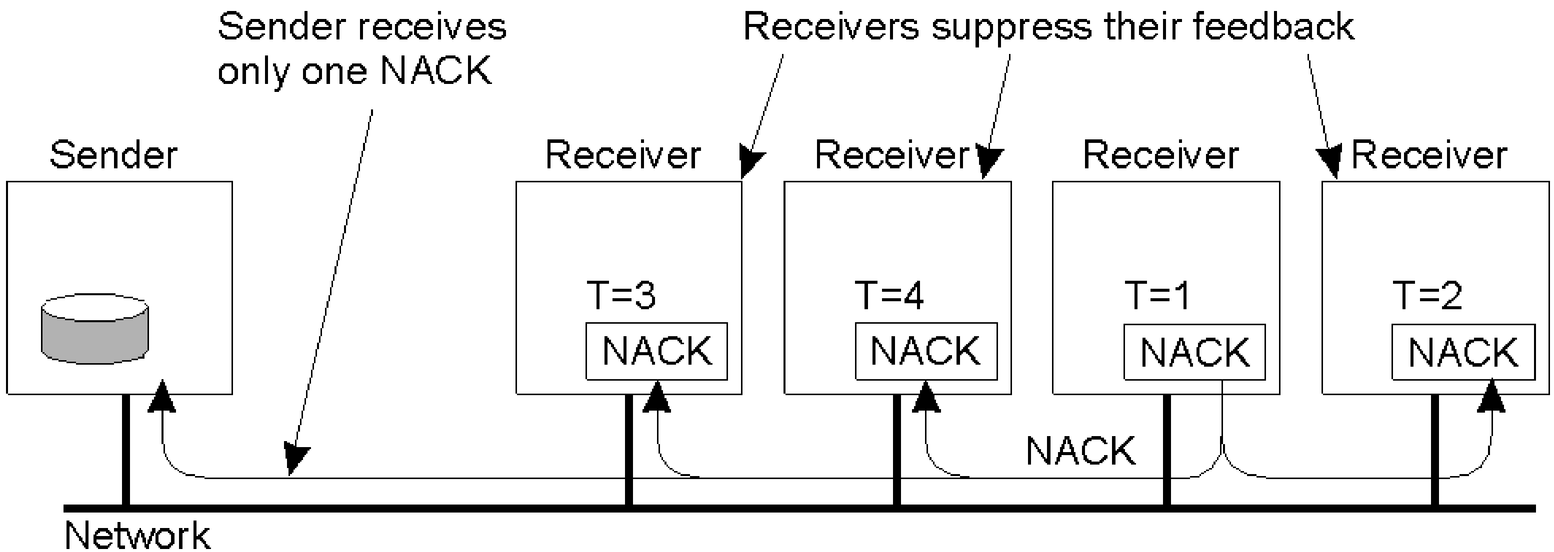
Contents

- Election algorithms
- **Multicast communication**
  - Basic reliable multicast
  - **Scalable reliable multicast**
  - Ordered multicast
  - Atomic multicast
- Consensus

Scalable reliable multicasting

- Main idea: use sequence numbers but <u>reduce</u> the number of feedback messages to sender
- Only missing messages are reported (NACK)
  - NACKs are multicast to all group members
  - Successful delivery is never acknowledged
- Each process waits a random delay prior to send a NACK
  - If a process is about to NACK, this is suppressed as a result of the first multicast NACK
  - In this way, only one NACK is delivered to the sender

Scalable reliable multicasting



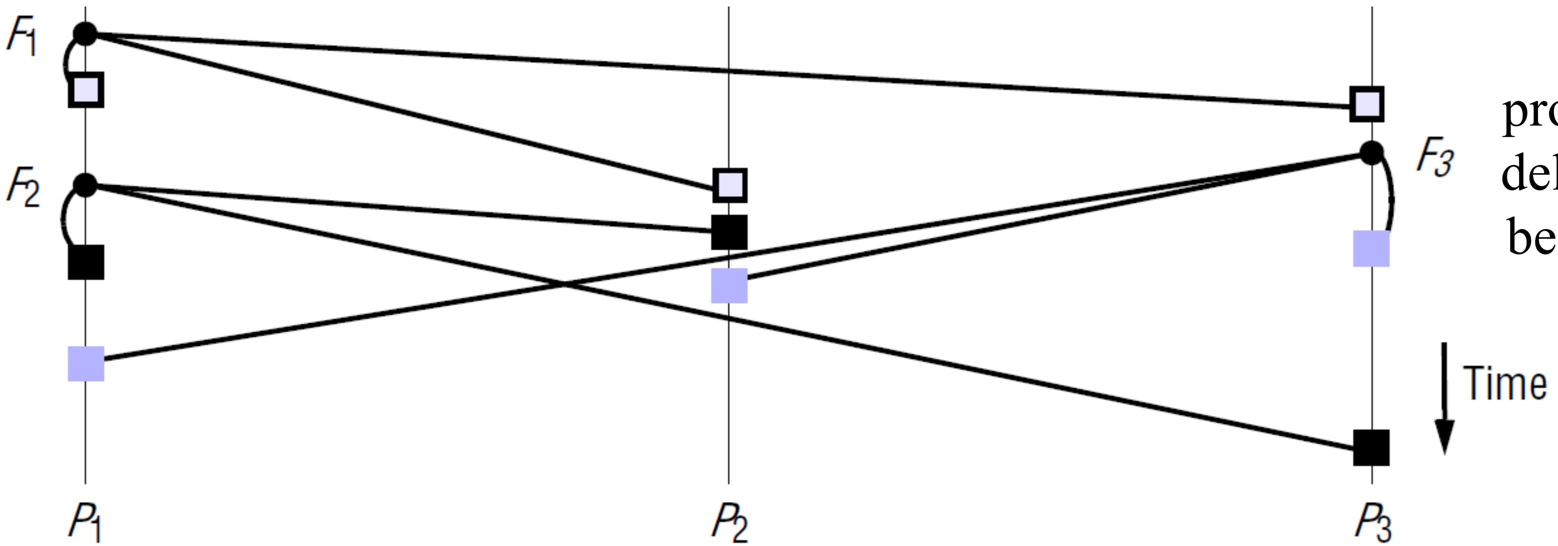
- ↑ Better scalability
- ↓ Setting timers to ensure only one NACK is hard
- ↓ Sender should keep messages in the history buffer forever to guarantee all retransmissions
    - In practice, messages are deleted after some time

Contents

- Election algorithms
- **Multicast communication**
  - Basic reliable multicast
  - Scalable reliable multicast
  - **Ordered multicast**
  - Atomic multicast
- Consensus

Ordered multicast

- Due to the latency, messages might arrive in different order at different nodes
- Common ordering requirements:
  A. <u>FIFO ordering</u>: messages from the same process delivered in the sent order by all processes

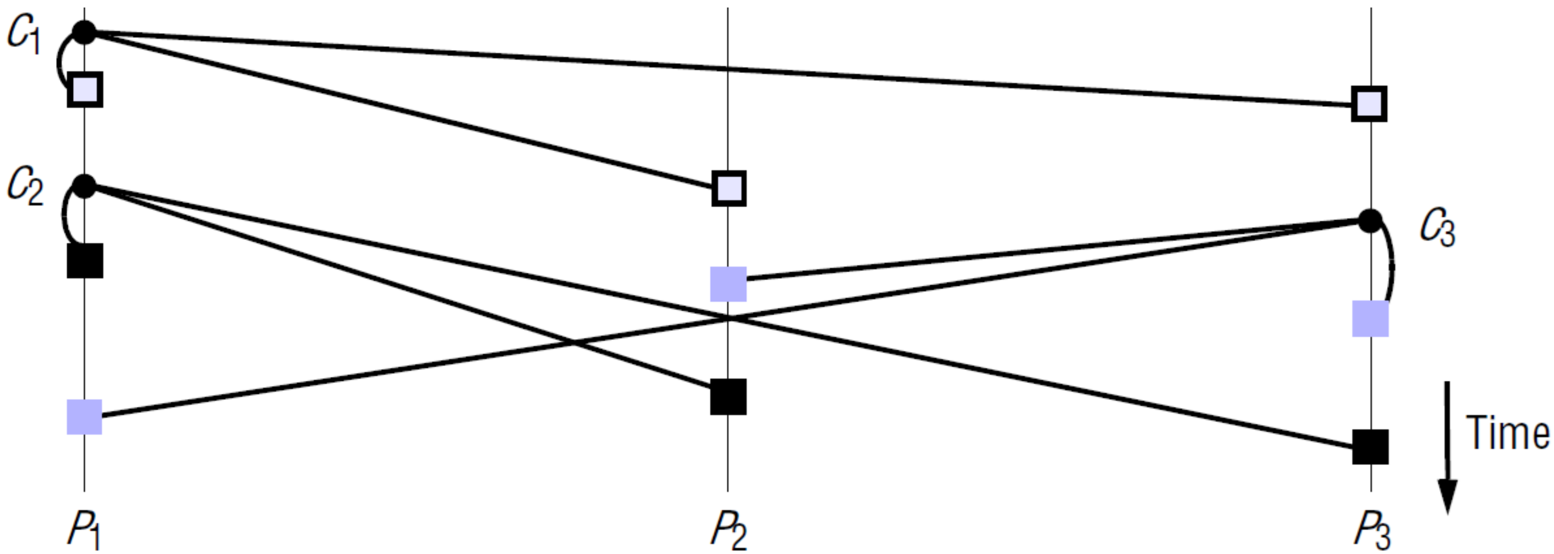


All processes deliver F_1 before F_2

Ordered multicast

B. <u>Causal ordering</u>: happened-before-related messages delivered in that order by all processes

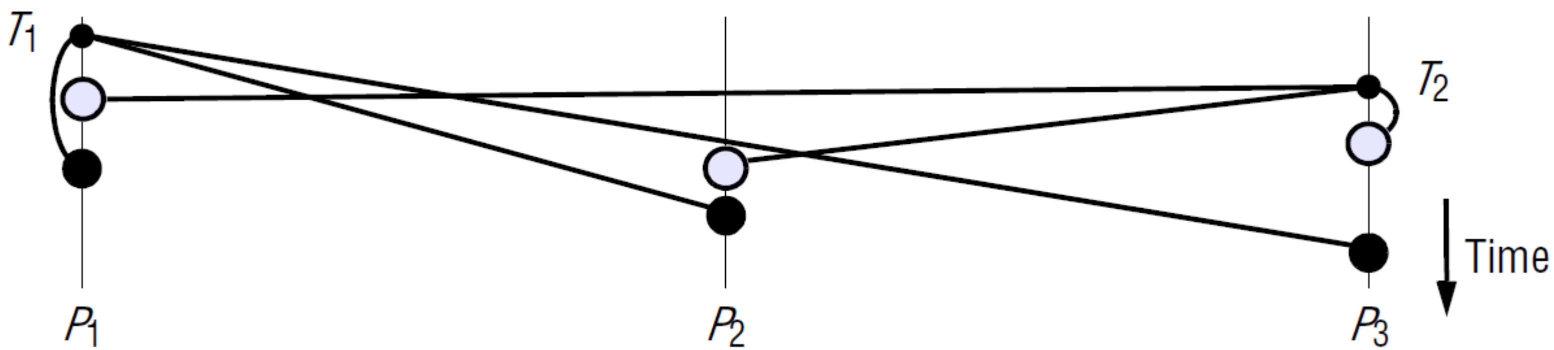
- Causal ordering implies FIFO ordering



C_1 and C_2 are FIFO related ; C_1 and C_3 are causally related
All processes deliver C_1 before C_2 and C_1 before C_3

Ordered multicast

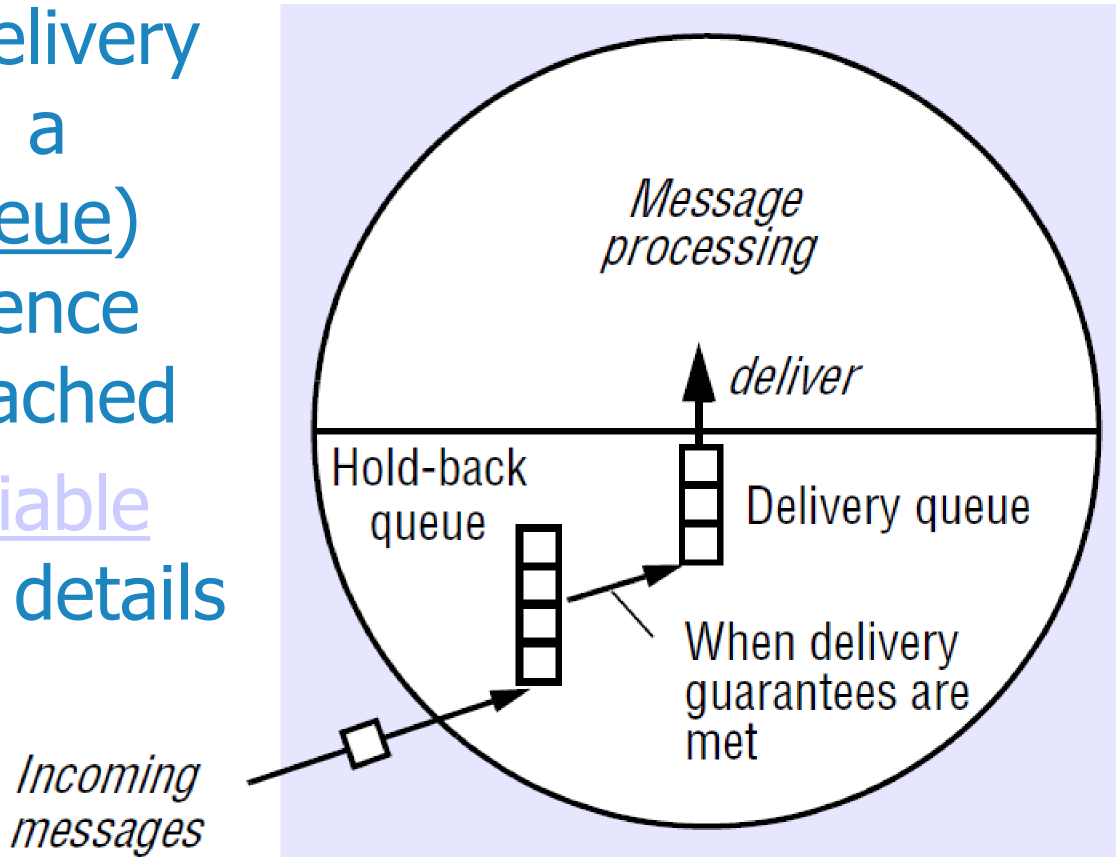
C. Total ordering: all messages delivered in the same order by all processes
   - Hybrid approaches such as FIFO+total ordering and causal+total ordering are also possible



All processes deliver T_2 before T_1

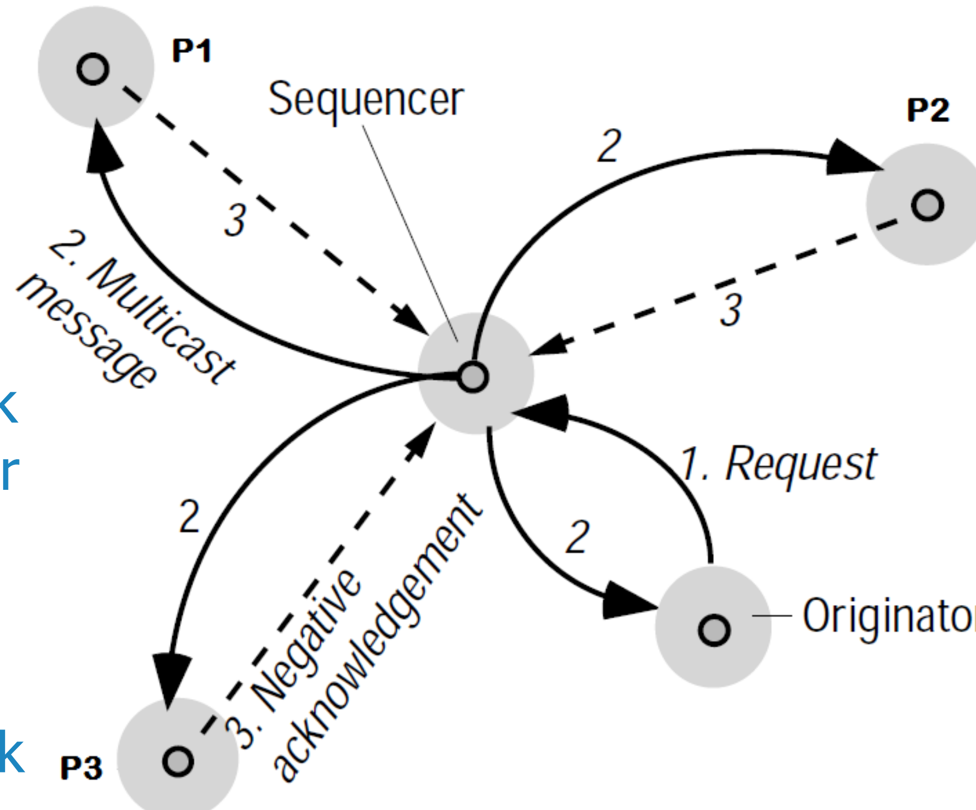
Implementing FIFO ordering

- Using **sequence numbers per sender**
- A message delivery is delayed (in a <u>hold-back queue</u>) until its sequence number is reached
- See <u>'basic reliable multicast'</u> for details



Implementing total ordering

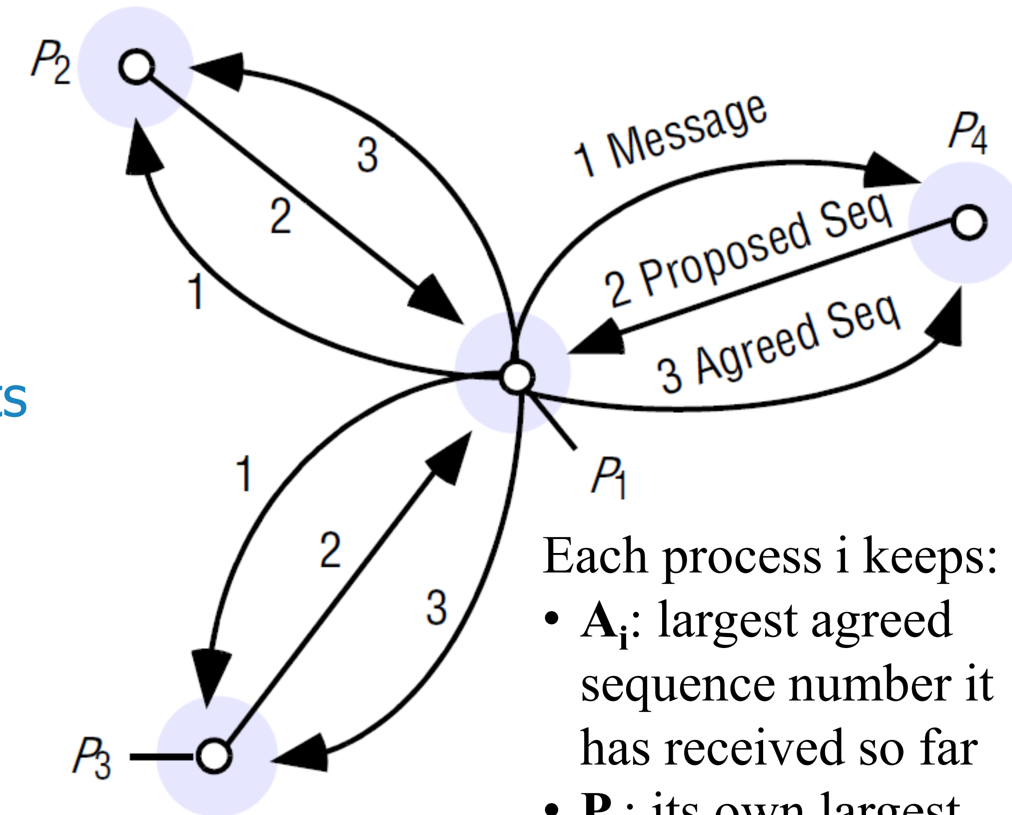
- Using **sequence numbers per group**
  a) Send messages to a **sequencer**, which multicasts them with numbering
    - A message delivery is delayed (in a hold-back queue) until its number is reached
    - Sequencer is a single point of failure and a performance bottleneck



Implementing total ordering

b) Processes jointly agree on sequence numbers

1. The sender multicasts message m
2. Each receiver j replies with a proposed sequence number for message m (including its process ID) that is $P_j = Max(A_j, P_j) + 1$ and places m in an ordered hold-back queue according to P_j



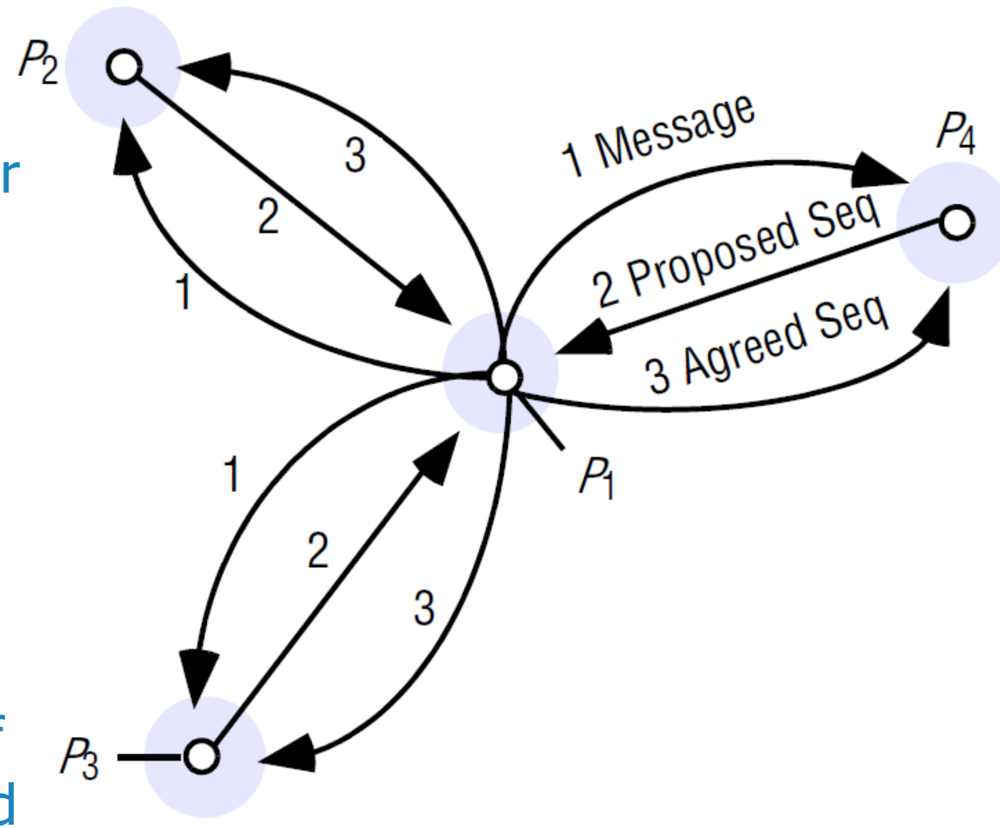
Each process i keeps:

- $\mathbf{A_i}$: largest agreed sequence number it has received so far
- $\mathbf{P_i}$: its own largest proposed number

Implementing total ordering

b) Processes jointly agree on sequence numbers

3. The sender selects the largest of all proposals, N, as the agreed number for m and multicasts it
4. Each receiver j updates A_j=Max(A_j,N), tags message m with N, and reorders the hold-back queue if needed
5. A message is delivered when it is at the front of the hold-back queue and its number is agreed



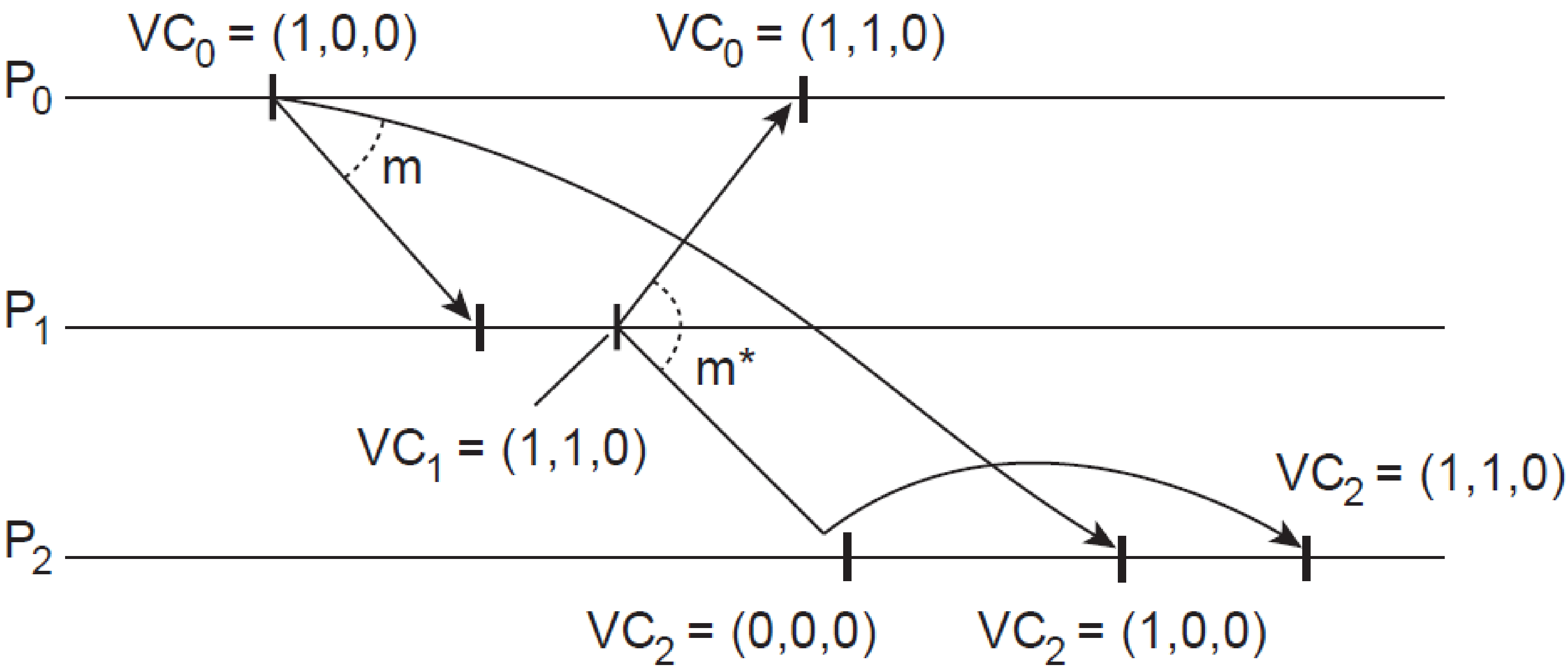
Implementing causal ordering

- Using **vector clocks**
  - A message is delivered only if all causally preceding messages have already been delivered

1. P_i increases $VC_i[i]$ only when sending a message
2. If P_j receives message m from P_i, it postpones its delivery until the following conditions are met:
   a. $ts(m)[i] = VC_j[i]+1$
      – m is the next expected message from P_i
   b. $ts(m)[k] \leq VC_j[k] \; \forall k \neq i$
      – P_j has seen all the messages seen by P_i before m
3. P_j increases $VC_j[i]$ after delivering m

Implementing causal ordering

- Causal ordering example



Contents

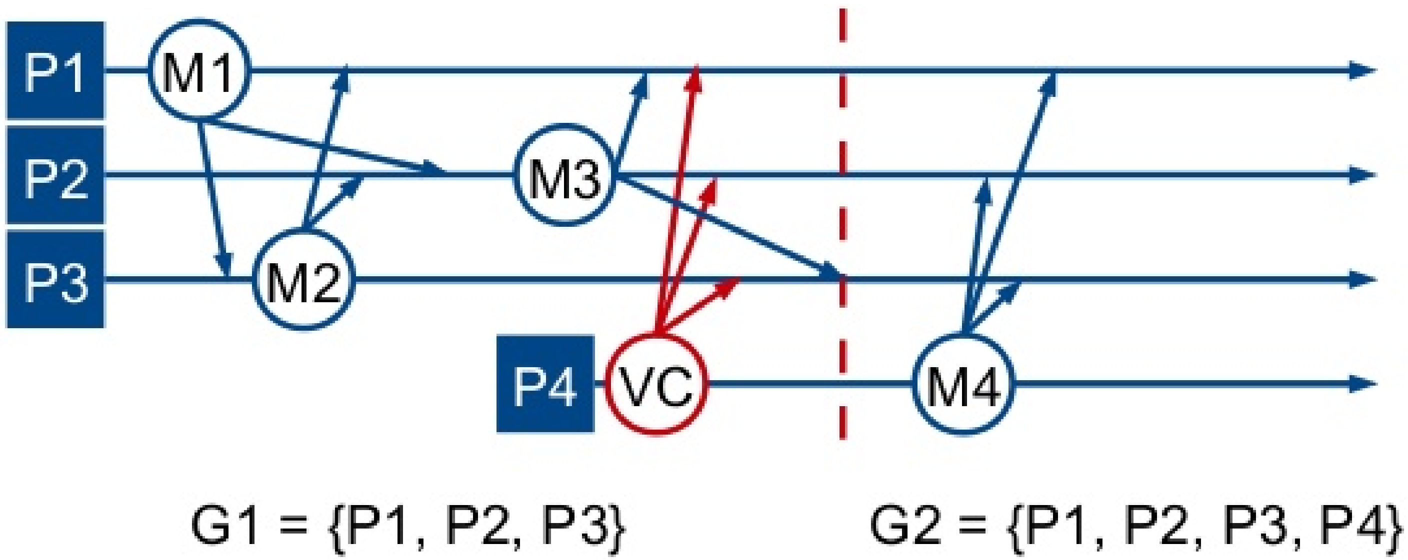
- Election algorithms
- **Multicast communication**
  - Basic reliable multicast
  - Scalable reliable multicast
  - Ordered multicast
  - **Atomic multicast**
- Consensus

Atomic multicast

- Solution for reliable multicasting in open groups (with **faulty** processes)
- Guarantee that a message is delivered to either all processes or none at all
- A message is delivered only to the current <u>non-faulty members</u> of the group
  - Processes have to agree on the current group membership
- a.k.a. virtual synchrony or view-synchronous multicast

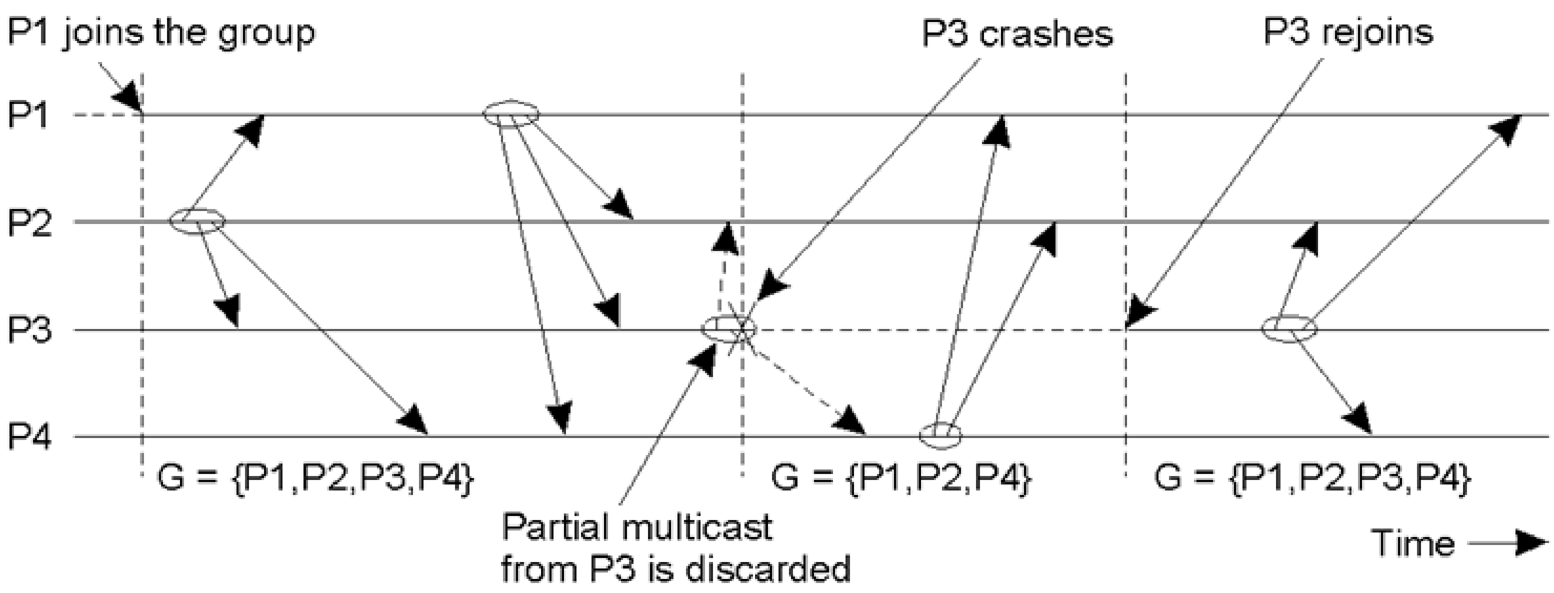
Atomic multicast

- A membership service keeps all members updated on who the current members of the group are
- Send **view messages** of group membership which must be delivered to members in <u>total order</u>
- View changes when processes join/leave the group

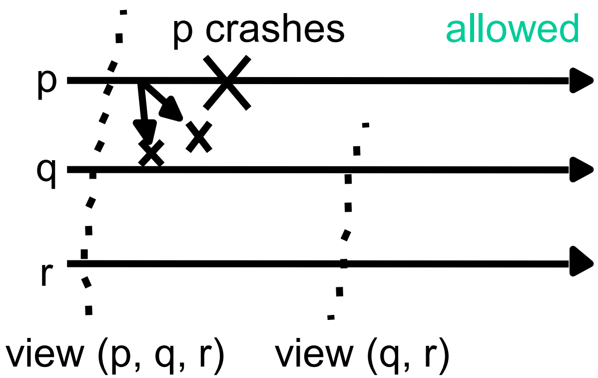


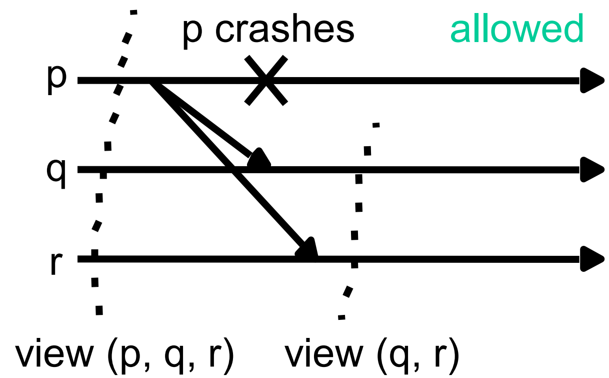
Atomic multicast

- Each message is associated with a group view
  - The one the sender had when transmitting
- Multicasts cannot pass across view changes
  - All multicasts that are in transit while a view change occurs must be completed before the new view comes into effect



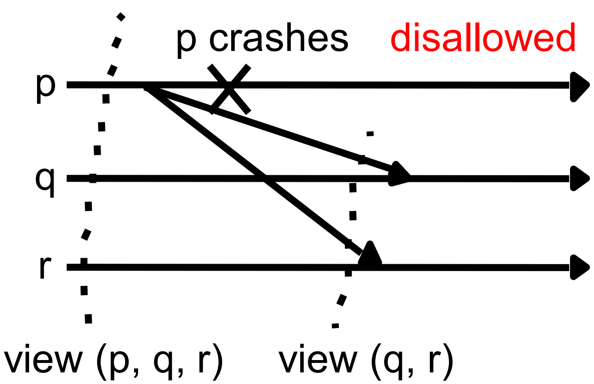
Atomic multicast

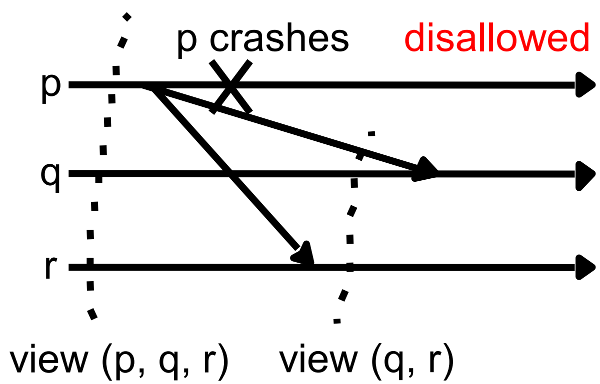


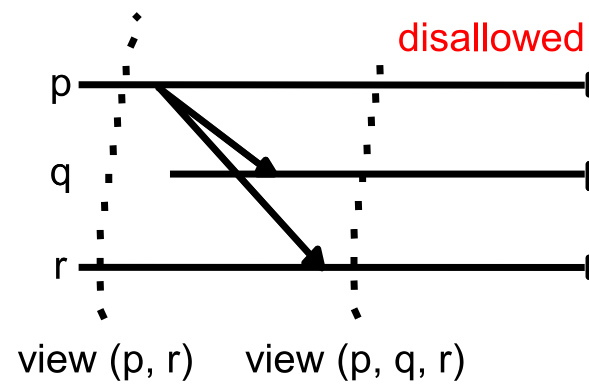


p crashes before m reaches any other process: none of them delivers m

m has reached at least 1 process when p crashes: both q and r deliver first m and then the view







not allowed for any process to deliver first the view (q, r) and then m

q must not deliver delayed messages from the previous view before it joined

Contents

- Election algorithms
- Multicast communication
- **Consensus**

Consensus

- We have seen algorithms that are tailored to individual types of agreement
  - e.g. electing a coordinator, synchronizing a set of clocks, deciding to commit or abort a transaction or which process can enter a critical section, …
- Let's consider a general form of agreement
  - Some processes must agree on a value (in a finite number of steps) after one or more of them have proposed what that value should be
  - Consider the problem <u>in the presence of failures</u>

Consensus

- Desired properties for consensus solutions:
  1. **Termination:** Every non-faulty process must eventually decide
  2. **Agreement:** The final decision of every non-faulty process must be identical
  3. **Validity:** If all the non-faulty processes proposed the same value, then the final decision for any non-faulty process has to be that value

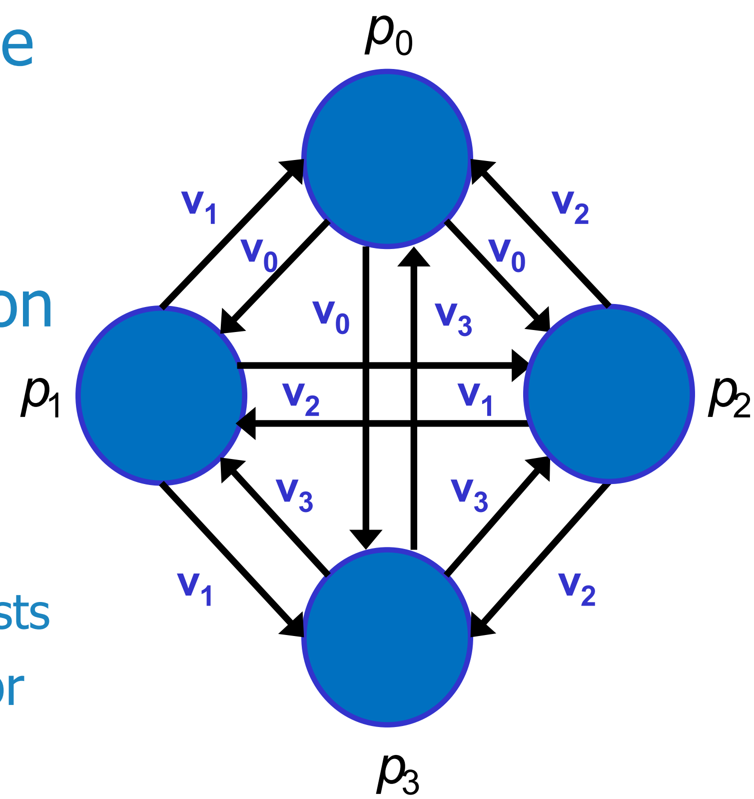
Consensus

1. With correct processes and reliable communication agreement is straightforward

a) Each process proposes a value

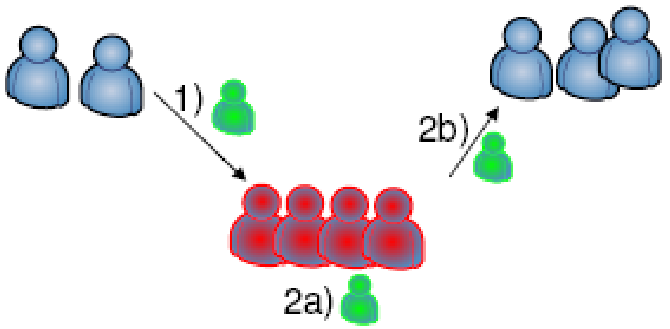
b) Processes exchange values through multicast

c) Each process applies a function on the collected values and sets a decision variable
   - Typically, use a majority function
     - Special value ⊥ if no majority exists
   - If values are ordered, minimum or maximum functions may be used



Consensus

2. With <u>unreliable</u> communication agreement between even two processes **cannot be guaranteed**: 'Two-army problem'
   - The two blue armies (2000 + 3000 soldiers) must agree to attack the red army (4000 soldiers) at the same time



   - A messenger is used to communicate, but can be captured by red army (unreliable communication)
   - Agreement <u>cannot be guaranteed</u> because ACKs can be lost as easily as the original message

Consensus

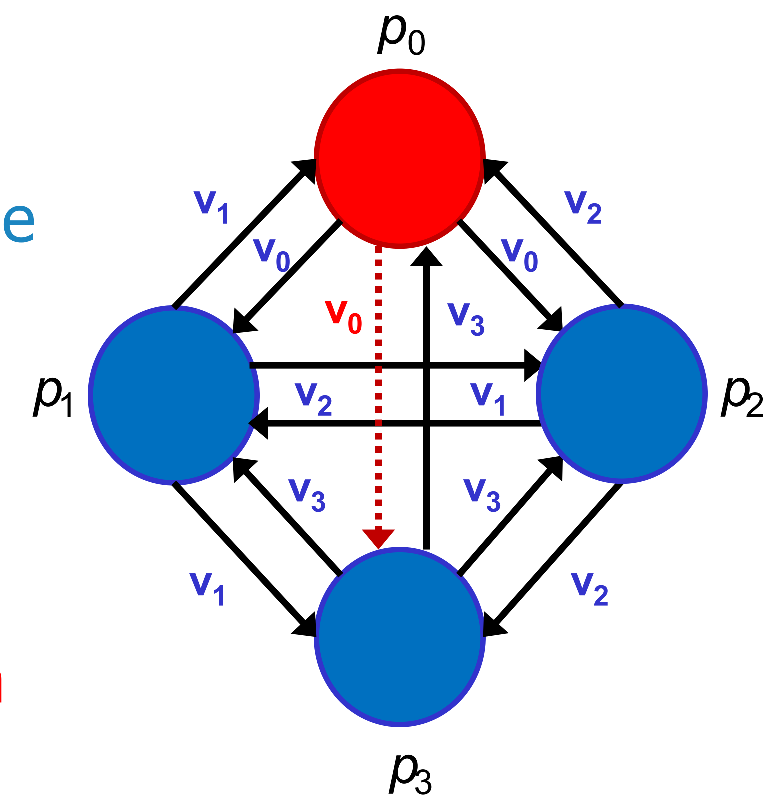
3. <u>Crash-faulty</u> processes and <u>reliable</u> communication in a <u>synchronous</u> system

   – Up to f of the N processes exhibit crash failures

   – Basic algorithm with a single round does not work

     - p1 decides $f(v_0,v_1,v_2,v_3)$
     - p2 decides $f(v_0,v_1,v_2,v_3)$
     - p3 decides $f(v_1,v_2,v_3)$

   ⇒ Dolev & Strong's algorithm



Dolev & Strong's algorithm

- Algorithm proceeds in **f+1** rounds

a) Each process proposes a value

b) From round 1 to round f+1, each process …

   i. Multicasts NEW values (not sent in previous rounds). Initially, sends its proposed value

   ii. Collects values from other processes and records any new values

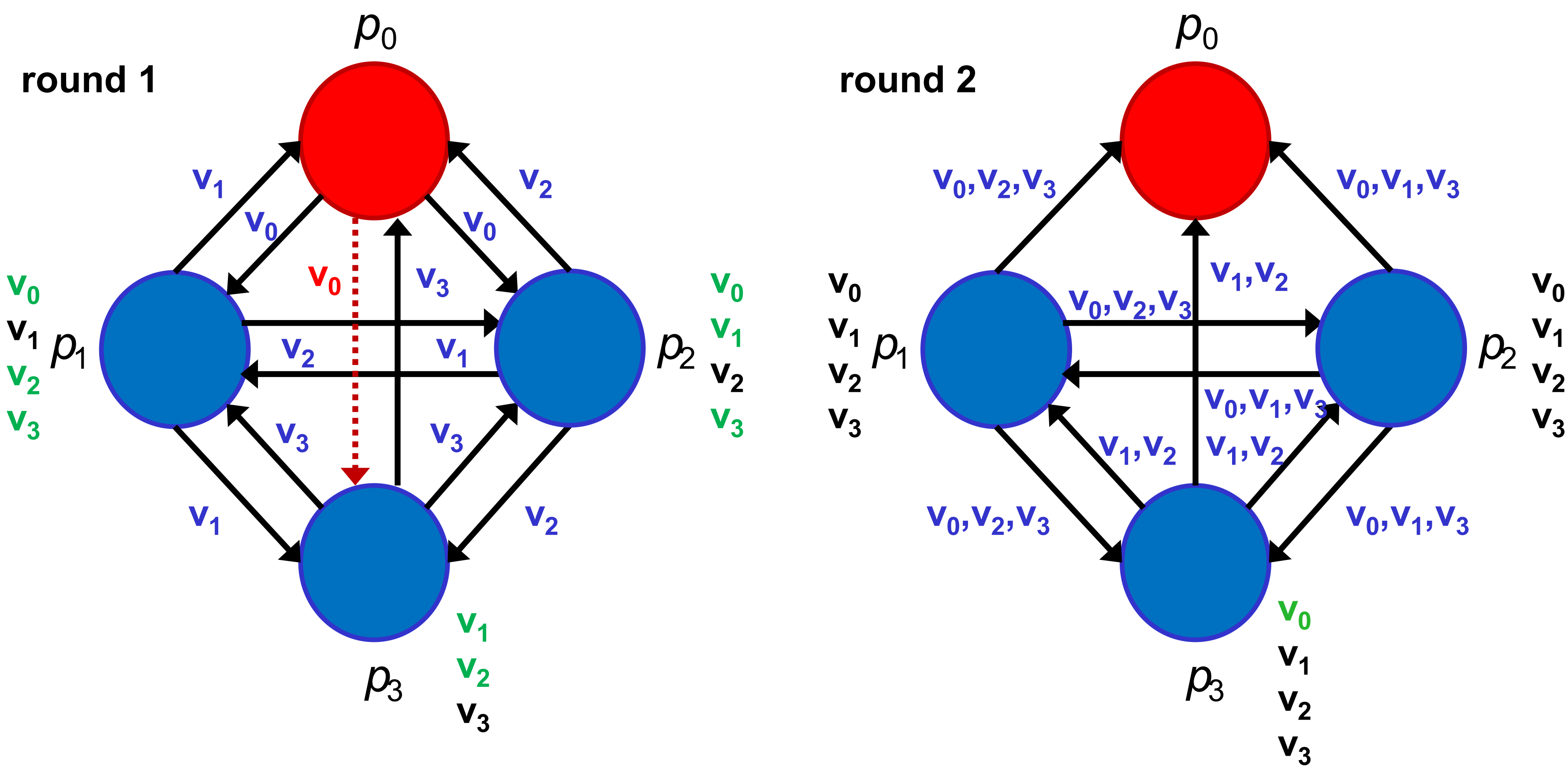
   – Round terminates when values from all processes are collected or by timeout (based on the maximum message latency)

c) Each process decides against collected values

Dolev & Strong's algorithm

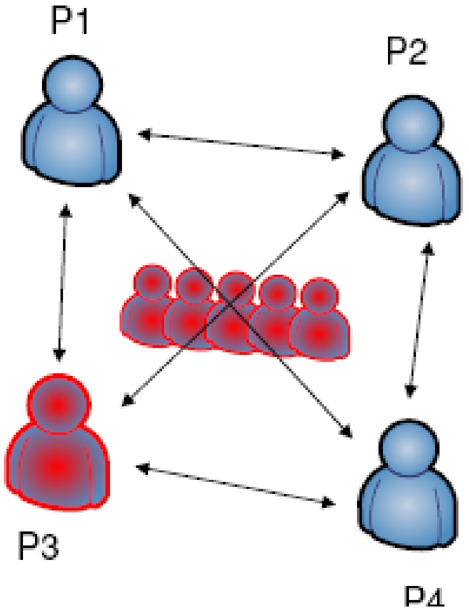
- Example (f=1)

v_i: new values
v_i: known values



Consensus

4. <u>Byzantine-faulty</u> processes and <u>reliable</u> communication in a <u>synchronous</u> system
   - 'Byzantine generals problem'
     - A group of Byzantine generals camped around an enemy city must agree on a common plan
       - e.g. attack or retreat
     - Direct communication (reliable)
     - Some of the generals may be traitors (i.e. faulty process)
     - Traitors may work together maliciously to confuse loyal generals and avoid their agreement (byzantine failures)



Byzantine generals problem

- Problem formalization for n generals:
  1. Let v(i) be information provided by ith general
  2. Generals communicate v(i) values to one another
  3. Decision is the majority among values v(1)...v(n)
     - Loyal generals will agree if they **use the same inputs**
       - A loyal general cannot use the values he received directly from the others since traitors can send conflicting values to different generals, BUT …
       - If a general is loyal then the value he sent must be used as his value by every other loyal general

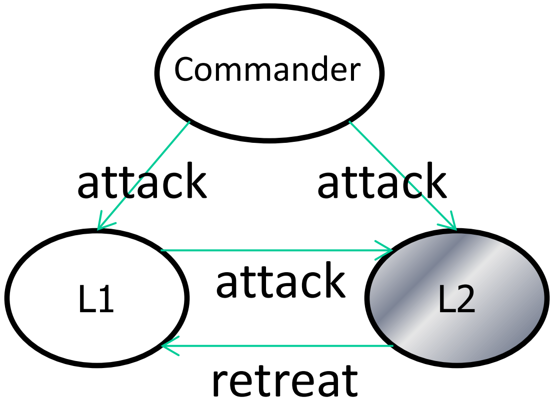
  $\Rightarrow$ We can restrict to the problem of one general sending his value to the others

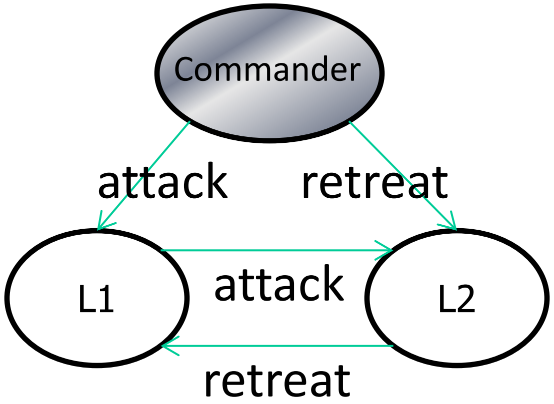
Byzantine generals problem

- A commanding general (commander) must send an order to his n-1 lieutenants
  - Generalization to original problem?
    - Each general sends his value by using this solution, with other generals acting as lieutenants
- Interactive consistency requirements:
  - IC1: All loyal lieutenants obey the same order (agreement)
  - IC2: If the commander is loyal, then every loyal lieutenant obeys the order he sends (integrity)

Byzantine generals problem

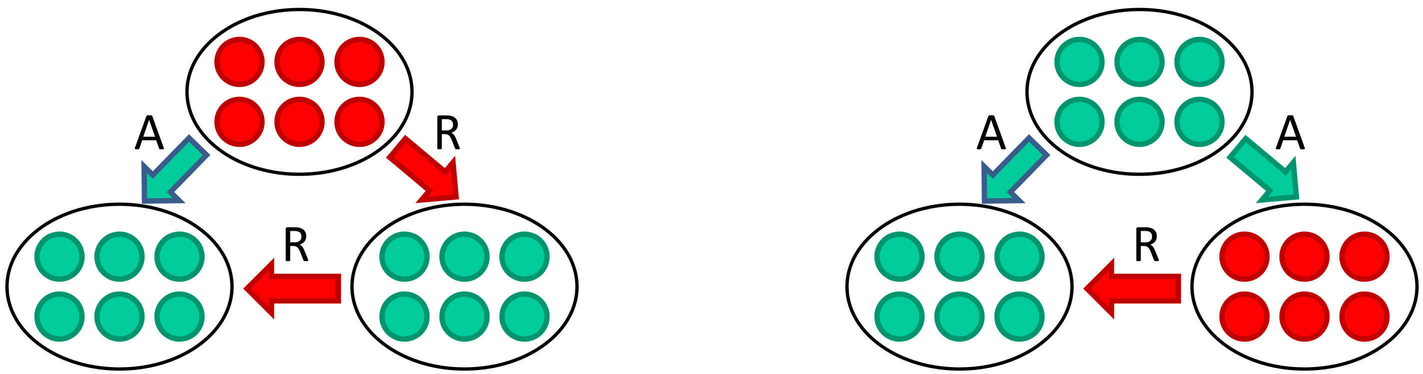
- Impossibility result: with three processes, no solution can work with even one traitor
  1. L1 has to attack to satisfy IC2 in Fig 1
  2. L1 cannot distinguish between both scenarios, so it will also attack in Fig 2
  3. By symmetry L2 will retreat in Fig 2, violating IC1





Byzantine generals problem

- Corollary: No solution with fewer than **3m+1** generals can cope with **m** traitors



⇒To reach a consensus in the presence of **m** traitors, **$n \geq 3m+1$** generals are needed

OM(m) algorithm

- OM(m): A solution with oral messages with at most m traitors and at least 3m+1 generals
  - Assumptions:
    - Messages may be lost, but this can be detected
    - Messages are not corrupted in transit
    - Receiver of a message knows the identity of the sender
  - OM(m) algorithm is recursive
- **OM(0)**

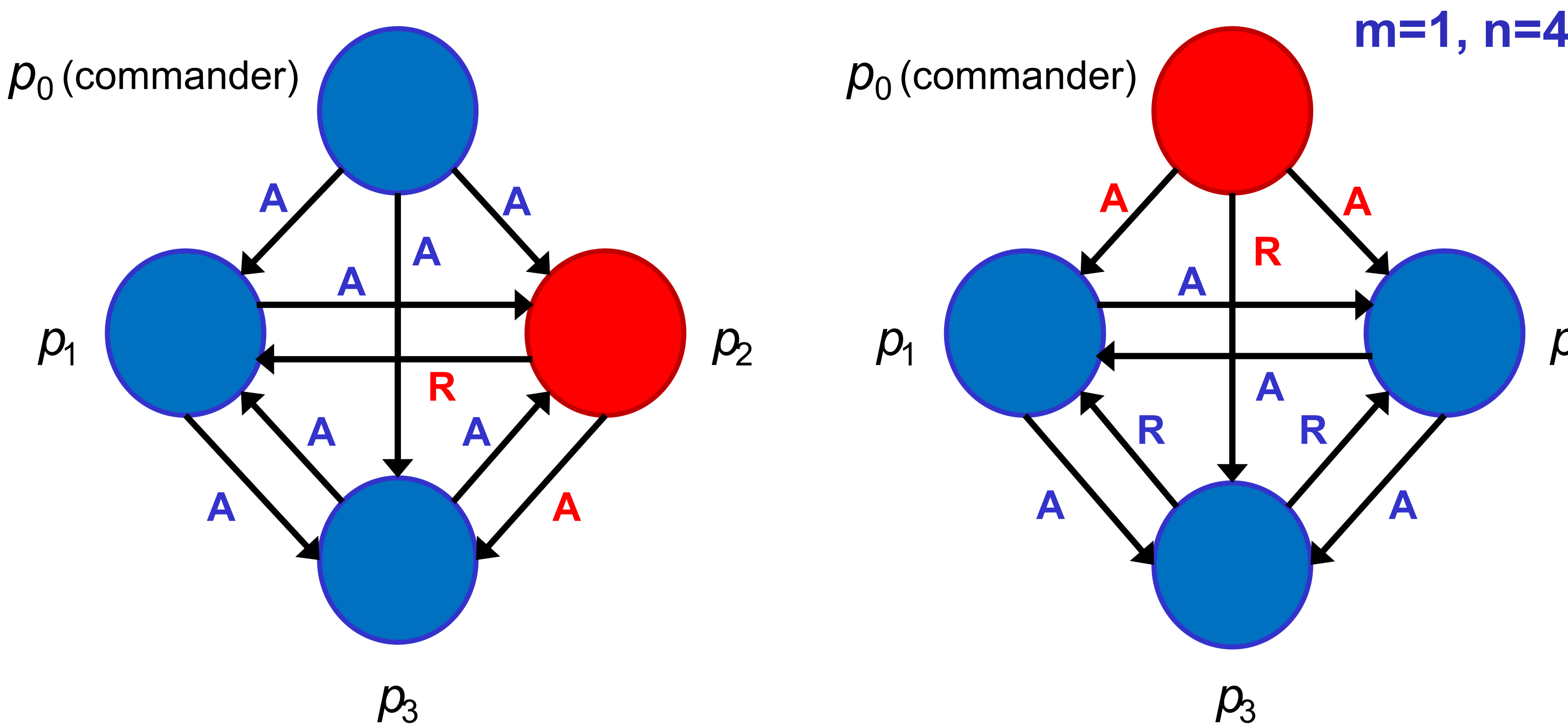
1. The commander sends his value to every lieutenant
2. Each lieutenant accepts the value he receives as the order from the commander

OM(m) algorithm

- **OM(m), m > 0**

1. The commander sends his value to every lieutenant
2. Each lieutenant i acts as a commander in *OM(m–1)* to broadcast the value he got from the commander to each of the remaining n–2 lieutenants
3. Each lieutenant i accepts the value $majority(v_1, v_2, \ldots, v_{n-1})$ as the order from the commander
   - v_i: value directly received from the commander in 'step 1'
   - v_j, $\forall j \in \{1, n-1\}$, $i \neq j$: values indirectly received from the other lieutenants in 'step 2'
   - If a value is not received, it is substituted by a default value

OM(m) algorithm



p1 gets {A,R,A} = A

p3 gets {A,A,A} = A

p1 gets {A,A,R} = A

p2 gets {A,A,R} = A

p3 gets {A,A,R} = A

OM(m) algorithm

- OM(m) requires **m+1** rounds
- Why? This example should require 3 rounds, but seems to work using only 2
  1. Commander sends value
  2. Lieutenants exchange values

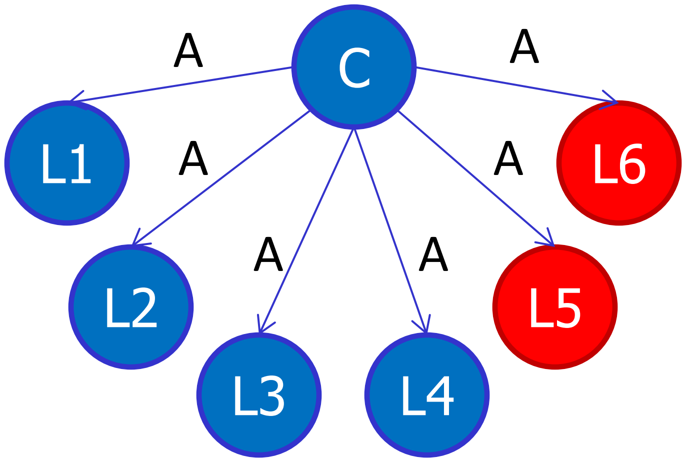
m=2, n=7

L1 gets {A,A,A,A,R,R}: A

L2 gets {A,A,A,A,R,R}: A

L3 gets {A,A,A,A,R,R}: A

L4 gets {A,A,A,A,R,R}: A

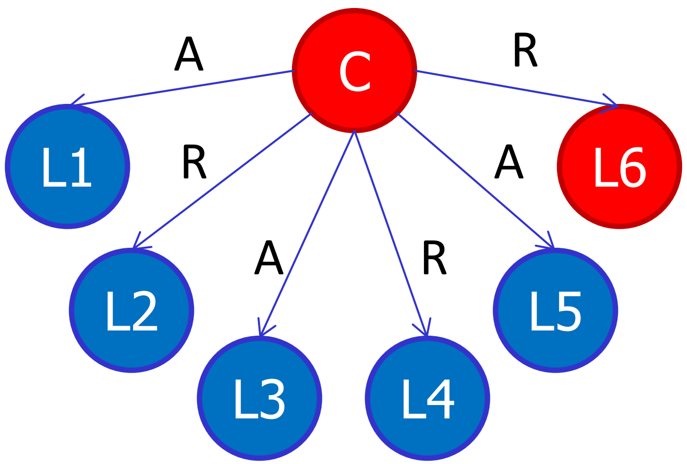


OM(m) algorithm

- Another example with 2 traitors:

L1 gets {A,R,A,R,A,A}: A
L2 gets {A,R,A,R,A,R}: ⊥
L3 gets {A,R,A,R,A,A}: A
L4 gets {A,R,A,R,A,R}: ⊥
L5 gets {A,R,A,R,A,A}: A

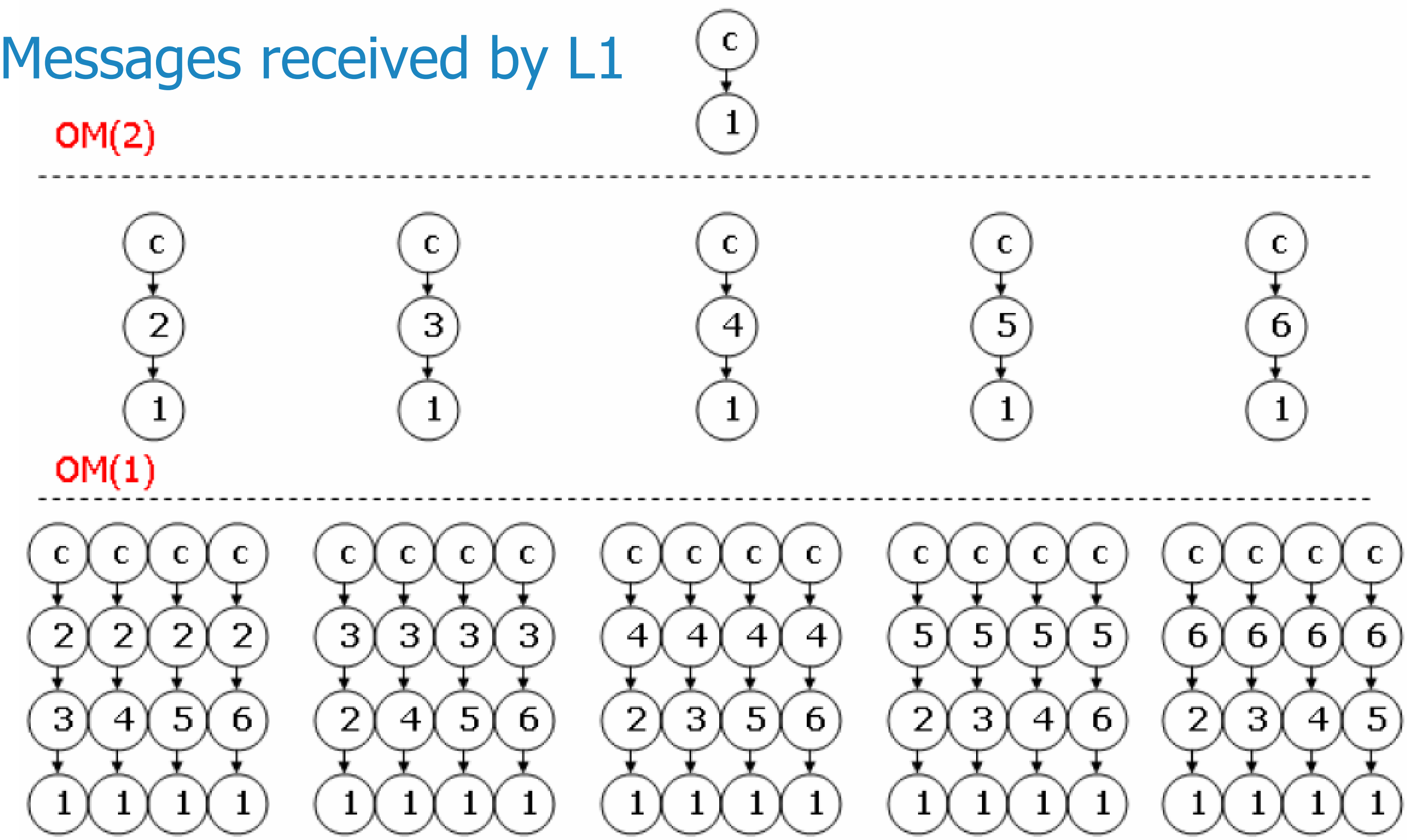
m=2, n=7



- All loyal lieutenants do NOT agree same value
  - We need to verify that lieutenants tell each other the same thing, thus another round is needed

OM(m) algorithm

Messages received by L1



OM(m) requires $O(n^{m+1})$ messages

OM(m) algorithm

Messages received by L1 and decision making

OM(2): {C, A}
OM(1): {L2, R}, {L3, A}, {L4, R}, {L5, A}, {L6, A}
OM(0): {L2, {L3, A}, {L4, R}, {L5, A}, {L6, R}}
{L3, {L2, R}, {L4, R}, {L5, A}, {L6, A}}
{L4, {L2, R}, {L3, A}, {L5, A}, {L6, R}}
{L5, {L2, R}, {L3, A}, {L4, R}, {L6, A}}
{L6, {L2, A}, {L3, R}, {L4, A}, {L5, R} }
OM(2): L1: { A R, A, R, A, A } = A

…

OM(2): L2: { A R, A, R, A, A } = A
OM(2): L3: { A R, A, R, A, A } = A
OM(2): L4: { A R, A, R, A, A } = A
OM(2): L5: { A R, A, R, A, A } = A

Consensus

5. Faulty processes and reliable communication in an asynchronous system
   - No algorithm can **guarantee** to reach consensus, even with only one faulty process [FLP85]
     - Cannot distinguish crash failure from arbitrary long delay
   - Consensus can be reached with some probability:

   a) Fault masking + basic algorithm (slide 99)
     - Assume that failed processes always recover and can reintegrate into the group (by recovering important data from persistent storage)
     - A round terminates when every expected message is received (if a process is not responding, wait longer)

Consensus

b) Failure detectors + synchronous algorithm (e.g. Chandra & Toueg, Mostefaoui & Raynal)

- Determine which processes have crashed
- A round terminates when every expected message is received, or <u>failure detector suspects its sender</u>
- Consensus with failure detectors requires that:
  1. Each faulty process is permanently suspected at least by some correct process
  2. At least a correct process is never suspected
- These items do not hold forever in asynchronous systems, but it is enough that they hold for a long enough period for correct processes to decide

Consensus

c) PAXOS algorithm

- Consensus protocol based on gaining votes from a quorum. Assumes that:

1. Processes may operate at an arbitrary speed and may crash (and subsequently recover)
2. Restarted processes will remember where in the protocol they were
   - They need access to stable, persistent storage
3. Messages can take arbitrary long time to be delivered, get lost or duplicated but not corrupted

Paxos: Roles

- Proposers: Propose values aiming to agree on a single one among the proposed values
- Acceptors: Decide whether to accept values
  - Could be a single acceptor (which chooses the first received proposal) but further progress is not possible if it fails $\Rightarrow$ Use multiple acceptors
- Learners: Check if any value has been chosen (a **majority** of acceptors have accepted it)

(*) A single process may play more than one role

Paxos: Design

- An acceptor must accept the first proposal that it receives
  - To allow a value to be chosen even if only one value is proposed by a single proposer
- An acceptor may accept several proposals
  - To ensure that majorities can be formed
- Each proposal is assigned with a sequence number (totally ordered)
  - A proposal consists of <u>a number and a value</u>
  - Number includes the proposer's ID to break ties

Paxos: Design

- The proposer operates in rounds
  - If a proposal is not accepted in a given round, the proposer backs off and starts another round with a higher sequence number
- All accepted proposals have the same value
  - If a proposal with number N and value V is accepted, then every higher-numbered proposal issued by any proposer has value V
  - This defines a number of requirements both for proposers and acceptors

Paxos: Design

a) The proposer must:
   1. Learn the highest-numbered proposal with number less than N, if any, that has been accepted so far
   2. Extract promises from acceptors not accept any more proposals numbered less than N

b) The acceptor needs to keep track of:
   1. The highest-numbered promise it has made
   2. The highest-numbered proposal it has voted for

   – <u>It must recover this information from persistent storage when it recovers from crash</u>

Paxos: Algorithm

A. Phase 1

1. Proposer sends **prepare** request with new sequence number N to each acceptor

'Please, do not vote for any proposal with number less than N'

2. Acceptor replies, if N is greater than any prepare request to which it has already responded, with:
   - **Promise** not to vote never again for any proposal numbered less than N
   - Highest-numbered proposal (sequence number + value) it has voted, if any

'I promise not to vote for any proposal with number less than N, and proposal with number M and value V is the highest-numbered proposal I have voted for'

Paxos: Algorithm

B. Phase 2

1. If the proposer collects promises from a majority of acceptors
   - Cast a ballot with sequence number N and value V
     - V: value of the highest-numbered proposal among the responses, or its own value if none reported
   - Send **accept** messages to request votes for the proposal

   'Please, vote for proposal with number N and value V'

2. Acceptor **votes** for the proposal unless it has already promised a number greater than N, and acknowledges the acceptance to the learners

   'I vote for proposal with number N'

Paxos: Algorithm

C. Phase 3

- When the learners collect notifications from a majority of acceptors for a given proposal, this becomes the final decision value
  - ➢ We have reached Consensus!!!
- At this point, each learner may take action and inform the client about the outcome
  - e.g. if processes were agreeing about the next operation to execute, each learner will run the agreed operation

Paxos: Properties

- Ensures Safety
  - Validity: Only a proposed value may be chosen
  - Agreement: Only a single value is chosen
- Using majorities allows good fault tolerance
  - 2f+1 acceptors can tolerate f failures
  - Can tolerate network partitions, because no two majorities can exist simultaneously
- Message cost (best case)
  - 6f+4 messages: (2f+1) + (f+1) + (2f+1) + (f+1)
  - 4 messages delays

Paxos: Properties

- Cannot guarantee termination (Liveness: A proposed value is eventually chosen)
  - e.g. 2 proposers alternate sending out prepare messages with increasing numbers
    - Acceptor cannot vote(N) once it has promised(N+1)
  - Those conditions are very improbable in real world
  - How to enhance the liveness of the algorithm?
    a) Proposers pick some increasingly large random delay before starting a new round to allow eventually one of them to get done without interference
    b) Elect a distinguished proposer, whose messages will be privileged when conflict arises

Paxos: Production use

- Used extensively in production systems:
  - Google Chubby distributed lock service
    - Part of Google software stack (GFS, BigTable, ...)
  - Linearizable consistency support in Cassandra distributed database
  - Replica management in OpenReplica service
  - Replicated log in Apache Mesos cluster manager
  - Leader election in VMware NSX
  - Transactional journal in Amazon WS platform
  - Monitoring system in Ceph storage
  - Lease negotiation in XtreemFS file system

SEMINAR PREPARATION – Paxy

- **[Lamport01]** Lamport, L., *Paxos Made Simple*, ACM SIGACT News, Vol. 32, No. 4, Distributed Computing Column 5, pp. 51-58, December 2001

Summary

- Election algorithms are primarily used in cases where the coordinator crashes
  A. Bully algorithm
  B. Ring algorithms (Chang & Roberts'; Enhanced)
- Multicast allows sending a message to a specified group of nodes
  - Message is delivered to all nodes or to none at all
    - Even when there are faulty nodes in the group
  - Messages can have also ordering requirements
    - FIFO, causal, and total

Summary

- Agreement algorithms: All non-faulty processes reach consensus on some value
  A. Dolev & Strong's
  B. Byzantine generals (OM)
  C. Paxos
- Further details:
  - [Tanenbaum]: chapters 6.5, 8.2.3, and 8.4
  - [Coulouris]: chapters 15, 18.2, and 21.5.2