4. Coordination and Agreement

Sistemes Distribuïts en Xarxa (SDX)
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC)
2021/2022 Q2

Introduction

- Problem: A set of processes coordinate their actions or agree on one or more values in a distributed system. This includes:
  - Achieve mutual exclusion among processes to coordinate their accesses to shared resources
  - 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
    - Consensus not covered in this course

Contents

- **Mutual exclusion**
- Election algorithms
- Multicast communication

Mutual exclusion

- Problem: A set of processes in a distributed system want exclusive access to some shared resource (i.e. critical section problem)
  - Solutions based solely on **message passing**

1. Permission-based solutions
   A. Centralized algorithm
   B. Voting algorithms (Lin's and Maekawa's)
   C. Ricart & Agrawala's algorithm
2. Token-based solutions
   A. Token-ring algorithm

Mutual exclusion

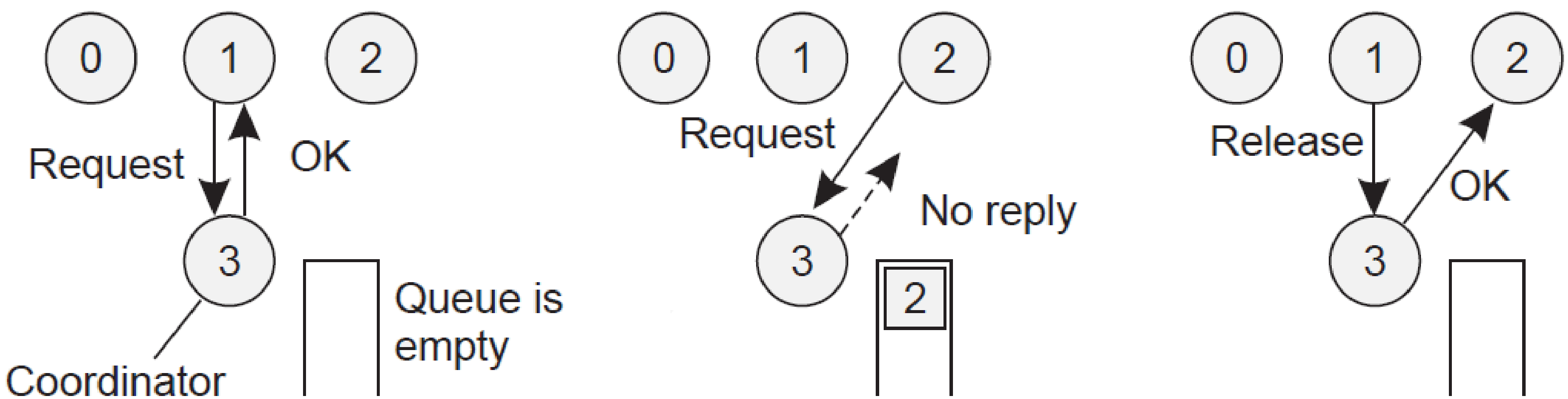
- Desired properties for mutual exclusion solutions:
  1. **Safety:** At most one process may execute in the critical section at a time
  2. **Liveness:** Requests to enter and exit the critical section eventually succeed
     - Free from both deadlock and starvation
  3. **Happened-before ordering:** If a request to enter the critical section happened-before another, then access to the critical section is granted in that order

Mutual exclusion

- Performance metrics for mutual exclusion solutions:
  1. **Bandwidth:** number of messages sent in each entry and exit operation
  2. **Client delay,** measured in message latencies, incurred by a process at each entry operation
     - When no other process is in the CS, or waiting
  3. **Synchronization delay,** in message latencies, between a process exiting the critical section and the next process entering it
     - When there is only one process waiting

Centralized algorithm

a) To enter the critical section (CS), a process sends a **Request** to coordinator and waits for permission
b) If no other process is in the CS, coordinator grants access (**OK**); otherwise, it queues the request
c) Send a **Release** to the coordinator to exit the CS
d) Coordinator removes the oldest process in the queue (if any) and grants access to it (**OK** message)



Centralized algorithm

- Problems:
  1. What if coordinator crashes?
     - A process cannot distinguish between ‘lock in use’ and ‘crashed coordinator’: we have a single point of failure
       – Workaround: deny permission explicitly
  2. Coordinator can be a performance bottleneck

Bandwidth	Client delay	Synch. delay	Safety (*)	Liveness (*)	Happened-before ordering
3 ({request + OK} + release)	round-trip (request + OK)	round-trip (release + OK)	Yes	Yes	No

(*) Assuming that processes do not fail and message delivery is reliable

Lin’s voting algorithm

- Decentralized algorithm with N coordinators
  - Using **voting**, gain quorum from the coordinators
    - To enter the critical section, a process needs to get a majority vote from $M > N/2$ coordinators

a) To enter the critical section, a process sends **Request** messages to all coordinators
  - Including its current Lamport’s clock and its process ID

b) Coordinator grants vote if it has not voted yet
  - Otherwise, it queues request (ordering by logical time)
  - In any case, it sends a **Response** message to requester including its current vote

Lin’s voting algorithm

c) Requester analyzes the votes in the responses:

1. If the requester obtains a majority vote (M > N/2 coordinators), it can enter the critical section
2. If someone else gets majority, the requester does nothing (it is already waiting in coordinators’ queues)
3. If nobody gets majority, the requester …

- <u>OPTION A</u>: releases its votes (to avoid deadlocks), backs off and retries after a random period
  - This might cause starvation (liveness violated)
- <u>OPTION B</u>: sends a **Yield** message (release+request) to all the coordinators that voted for him
  - Better performance and ensures liveness

Lin’s voting algorithm

d) On receiving a **Yield** message, a coordinator …
- Removes its vote
- Queues the request (<u>keeping the same time</u>)
- Gets the head of queue and votes for him
- Sends a **Response** message to the voted process
- If vote has changed, it notifies also the former process

e) To exit the critical section, requester sends a **Release** message to <u>all</u> the coordinators
- Each coordinator that voted for him …
  - Removes its vote
  - Gets the head of queue (if any) and votes for him
  - Sends **Response** message to the new voted process
- The rest of coordinators simply dequeue the process

Lin's voting algorithm

- Problems:
  1. Failure/recovery of coordinators
     - If a coordinator crashes, forgets previous vote. If there are f crashes and $N - M + f \geq M$ (i.e., $f \geq 2M - N$), the safety of algorithm will be violated
  2. Low efficiency if many Yield rounds are needed

Bandwidth (k Yield rounds)	**Client delay**	**Synch. Delay**	**Safety (*)**	**Liveness (*)**	**Happened -before ordering**
3N + f(k) ({N request + N vote + $\sum_{i=1}^{k}\binom{C_i\ yield\ +}{[C_i, 2C_i]\ vote}$} + N release)	round-trip (N request + N vote)	round-trip (N release + M vote)	Yes	Yes (using Yield)	Not guaranteed

Ricart & Agrawala's algorithm

- Multicast with logical clocks
  a) Process sends **Request** to <u>all</u> other processes
     - Message contains the requester's current <u>Lamport's logical time</u> and its process ID
       - Process ID used to break ties in logical time
     - Each process must know all processes in the group

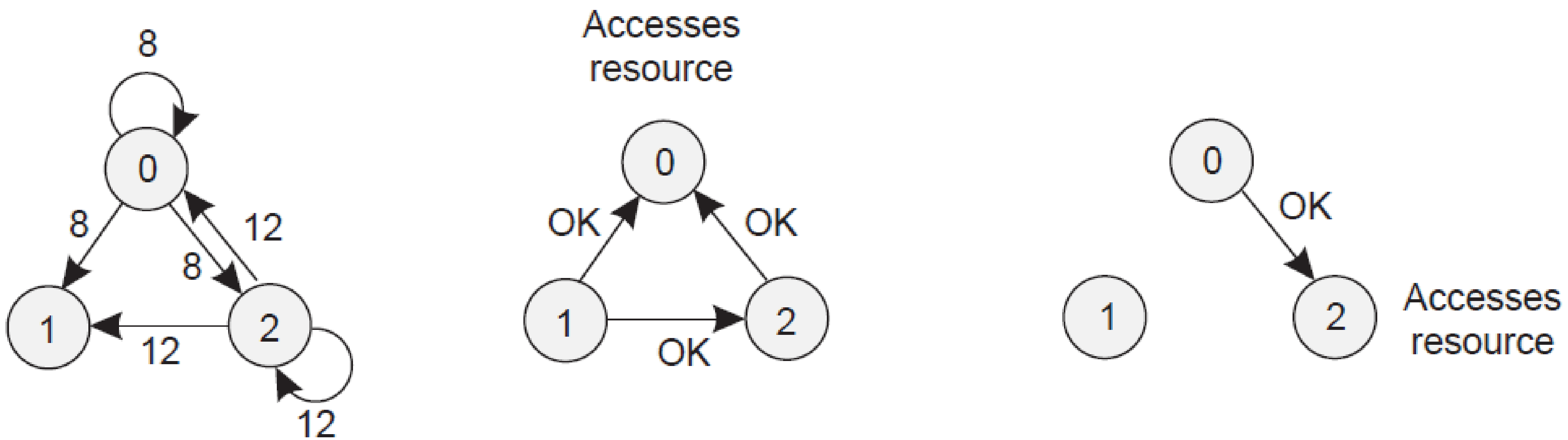
  b) When the requester receives **OK** messages from <u>all</u> processes, it can access the critical section

  c) When a process receives a **Request** message:
     1. If it is not accessing the critical section and does not want to access: send back an **OK** message
     2. If it is accessing now: no reply and queue the request

Ricart & Agrawala's algorithm

3. If it wants to access as well, but has not yet done so…
   - If the incoming message has <u>a lower logical time:</u> send back an **OK** message
   - Otherwise: no reply and queue the request

d) To exit the critical section, process sends an **OK** message to each process in its queue



Ricart & Agrawala's algorithm

- Problem:
  1. What if any process fails?
     - Requester cannot distinguish between 'denial of permission' and 'crashed process':
       - Workaround: deny permission explicitly
     - Variant: Get permission from a **majority** of the other processes

Bandwidth	**Client delay**	**Synch. delay**	**Safety (*)**	**Liveness (*)**	**Happened -before ordering**
2(N-1) (N-1 request + N-1 OK)	round-trip (N-1 request + N-1 OK)	1 (OK)	Yes	Yes	Yes

SEMINAR PREPARATION – Muty

[Ricart81] Ricart, G., Agrawala, A.K., *An Optimal Algorithm for Mutual Exclusion in Computer Networks*, Communications of the ACM, Vol. 24, No. 1, pp. 9-17, January 1981

Maekawa's voting algorithm

- Get permission to access the CS by obtaining votes from a **subset** of the rest of processes
  - Each process p_i gets access to CS by means of a unique voting set S_i
  - All the voting sets have the same size K
  - Each process p_i belongs to M sets:
    - Its own voting set: $p_i \in S_i$
    - M-1 of the other voting sets
  - Intersection of any two voting sets is non-empty
    - Processes in the intersection of two sets ensure the safety property by voting for only one candidate

S_0	=	$\{0, 1, 2\}$
S_1	=	$\{1, 3, 5\}$
S_2	=	$\{2, 4, 5\}$
S_3	=	$\{0, 3, 4\}$
S_4	=	$\{1, 4, 6\}$
S_5	=	$\{0, 5, 6\}$
S_6	=	$\{2, 3, 6\}$

Ex: K=M=3

Maekawa's voting algorithm

a) p_i sends **Request** messages to all K members of S_i
   - p_i needs **OK** from all of them to access the critical section

b) When a process in S_i receives a **Request** message:
   - If it is not accessing the critical section and it has not already replied ('voted') since it last received a Release message, sends an **OK** message immediately
   - Otherwise, it queues the request (in the order of its arrival) but does not yet reply

c) To exit the critical section, p_i sends **Release** messages to all K processes in S_i

d) When a process in S_i receives a **Release** message:
   - It removes the head of its queue (if any) and sends an **OK** message (a 'vote') to it

Maekawa's voting algorithm

- Problems:
  1. Liveness not guaranteed (system may deadlock)

  – Example:
    - S0 ={0, 1}, S1 ={1, 2}, S2 ={0, 2}
    - p0, p1, p2 send request to S0, S1, S2, respectively
    - From S0, p0 sends OK to p0, but p1 sends OK to p1
    - From S1, p1 sends OK to p1, but p2 sends OK to p2
    - From S2, p2 sends OK to p2, but p0 sends OK to p0

    $\Rightarrow$ p0 waits for p1, p1 waits for p2, and p2 waits for p0

Maekawa's voting algorithm

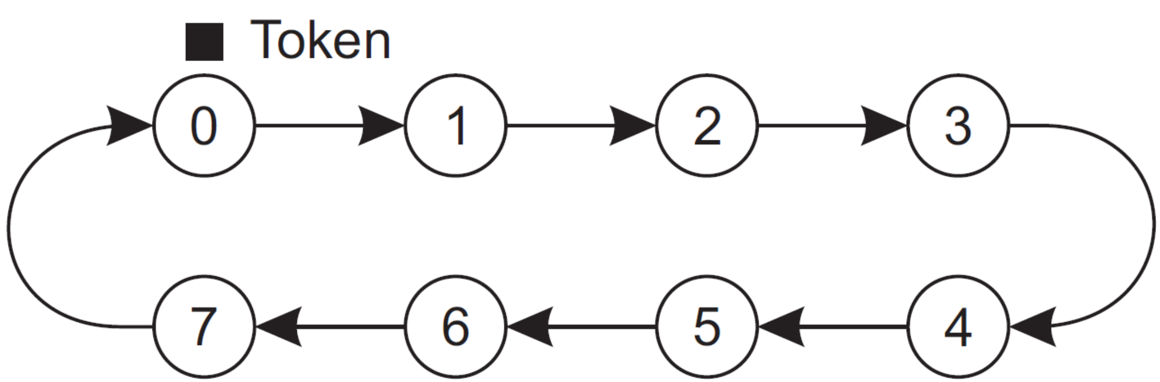
- Problems:
  2. Cannot tolerate failures within the required voting set, but can tolerate failures in other sets

Bandwidth	Client delay	Synch. delay	Safety (*)	Liveness (*)	Happened-before ordering
3K ({K request + K OK} + K release)	round-trip (K request + K OK)	round-trip (release + OK)	Yes	No	No

- Maekawa showed that K=M≈$\sqrt{N}$ is the optimal solution that minimizes K while providing safety

Token ring algorithm

- Organize processes in a logical unidirectional ring (a process only knows its successor)
- A **token** message circulates around the ring
- Only the process holding the token can enter the critical section
- To exit the critical section or if the process does not want to enter, send the token to successor



Token ring algorithm

- Problems:
  1. What if token is lost?
     - If the token is ever lost, it must be regenerated
     - Detecting token loss is difficult, requires coordination
  2. What if a process crashes?
     - Send the token to the next member down the line
       - Each process must know all nodes in the ring

Bandwidth	Client delay	Synch. delay	Safety (*)	Liveness (*)	Happened-before ordering
1 (all enter) to ∞ (none enter)	0 (token just received) to N–1 (token just departed)	1 (successor will access) to N–1 (predecessor will access)	Yes	Yes	No

Mutual exclusion

- None of these algorithms can tolerate process failures and network partitions well, and for this reason, they are not generally used in production systems
- How do production systems typically achieve mutual exclusion?
  a) Some systems offer only eventual consistency (they favor availability over consistency), and do not need mutual exclusion
    - e.g. Amazon Dynamo

Mutual exclusion

b) Many systems use generic consensus algorithms, which can tolerate failures and network partitions
- e.g. Paxos
  - Used by Google Chubby distributed lock service
    - » Part of Google software stack (GFS, BigTable, …)
- e.g. ZooKeeper Atomic Broadcast (ZAB)
  - Used by Apache ZooKeeper coordination service
    - » Used by Apache (Hadoop MapReduce & HBase, Solr, Kafka), Yahoo!, Rackspace
- e.g. Raft
  - Used by HydraBase (i.e. evolution of Apache HBase)
    - » Used by Facebook
  - Used by Consul by HashiCorp

Contents

- Mutual exclusion
- **Election algorithms**
- Multicast communication

Election algorithms

- Many distributed algorithms need one process to act as coordinator
  - e.g. centralized mutual exclusion, physical clock synchronization, primary-based replicated groups
- Use <u>leader election</u> algorithms
  - They ensure that an election concludes with all the processes <u>agreeing</u> on a <u>**unique**</u> coordinator
  - No matter which process is, just need to pick one
    - Without loss of generality, we require that the elected process is the one with the <u>largest identifier that is up</u>
      - Identifiers must be <u>unique</u> and <u>totally ordered</u>

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
- 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

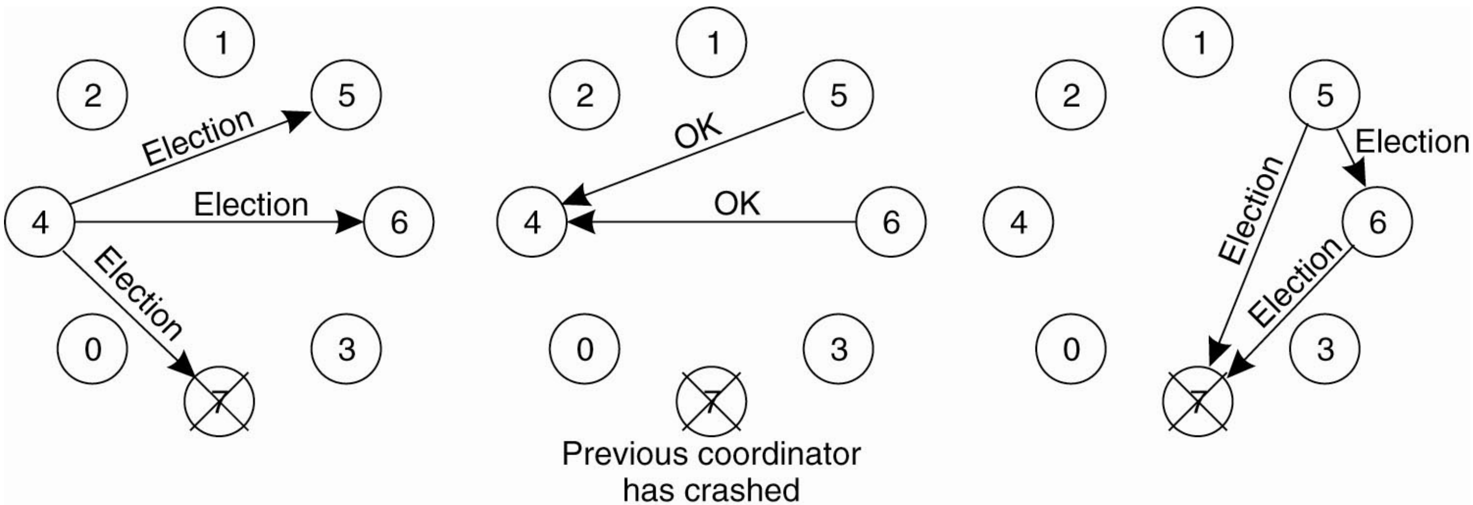
Bully algorithm

- Assumptions:
  - The system is synchronous
    - 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)

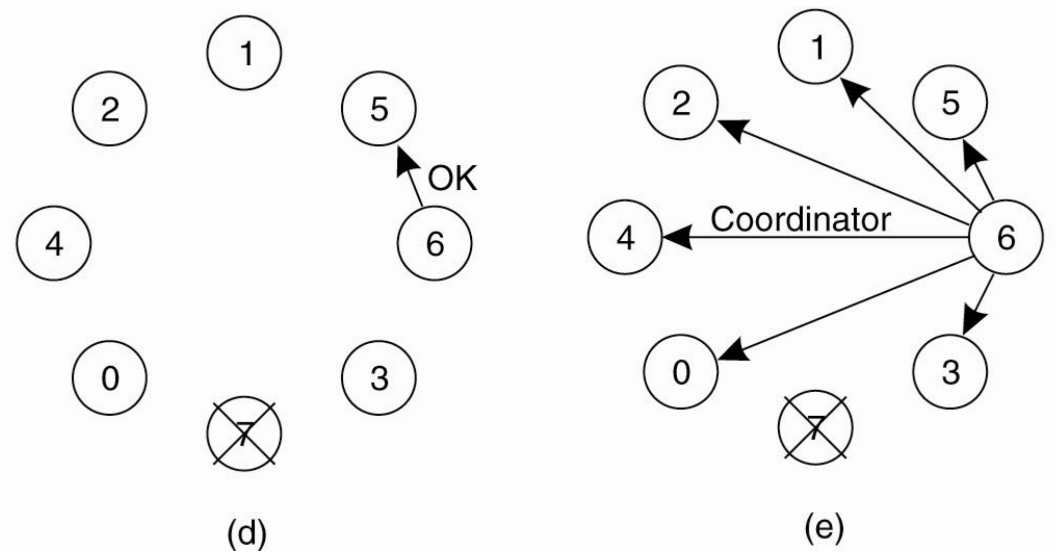


(a) (b) (c)

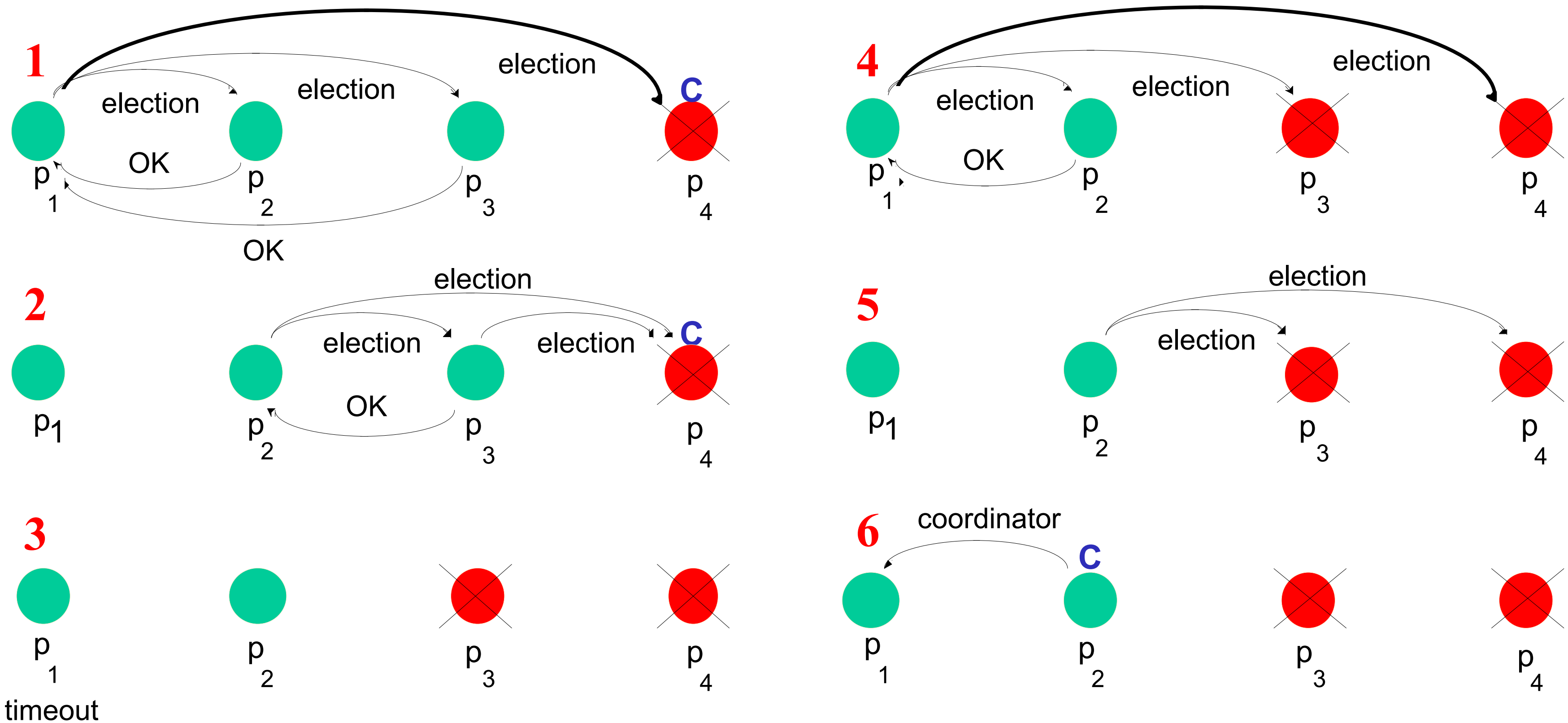
Bully algorithm

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

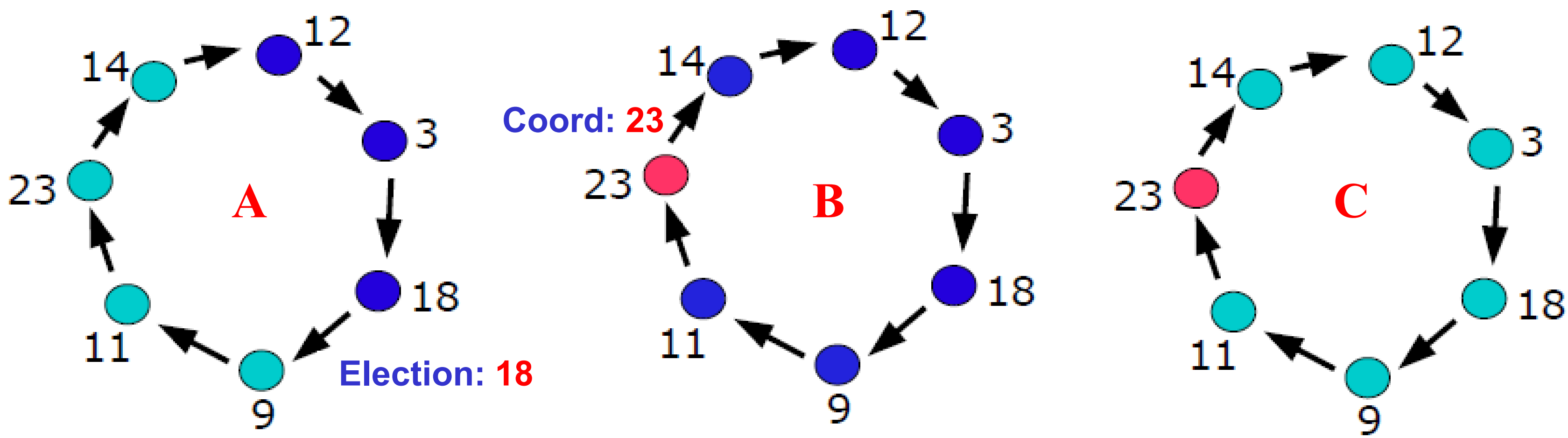
a) If the arrived ID is greater, Q forwards the message to its successor
b) If the arrived ID is smaller …
   - If Q is not a participant yet, Q replaces the ID in the message with its own ID and forwards it
   - If Q is already a participant, message is not forwarded
c) If the arrived ID is Q's ID,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

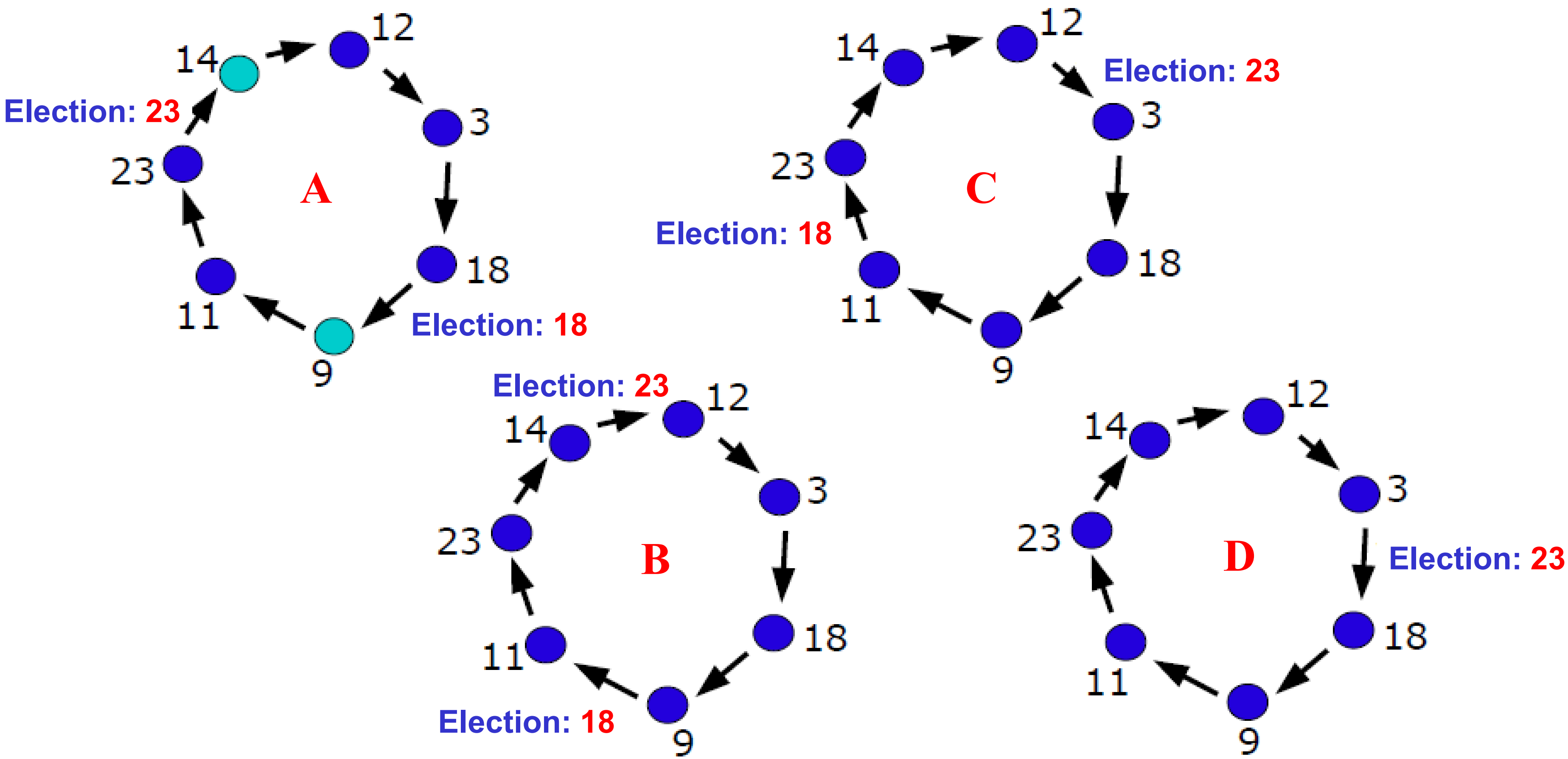


process is not yet a participant

process is already a participant

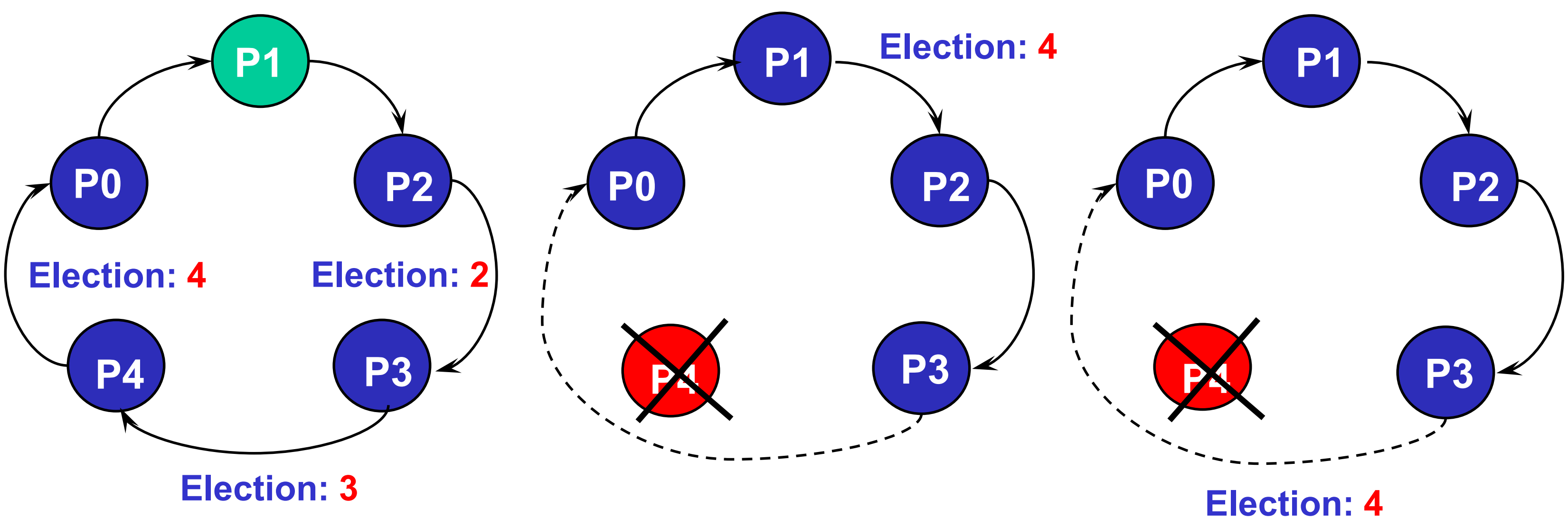
Chang and Roberts' ring algorithm

- Example: two elections in progress



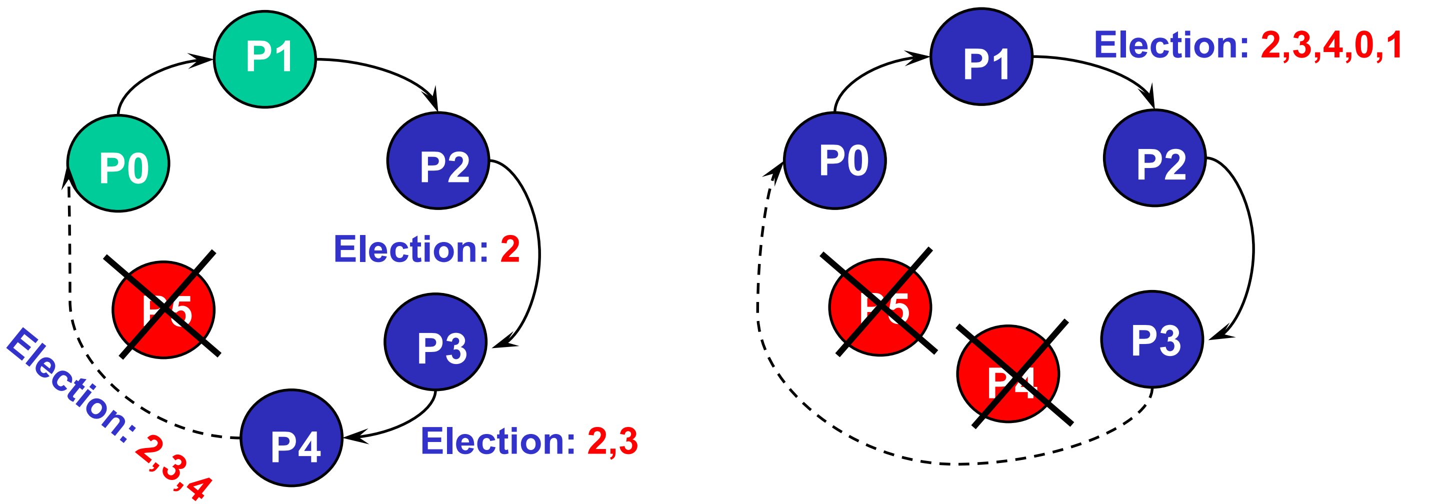
Chang and Roberts' ring algorithm

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



Enhanced ring algorithm

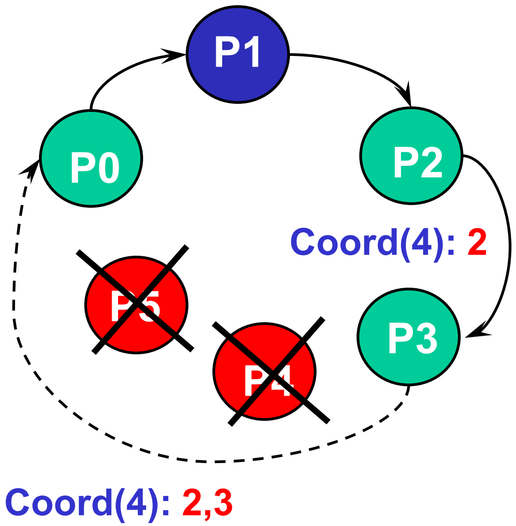
a) P sends an **Election** message (with its process ID) to its closest alive successor
   - Sequentially poll successors until one responds
     - Each process must know all nodes in the ring
b) At each step along the way, each process adds its ID to the list in the message

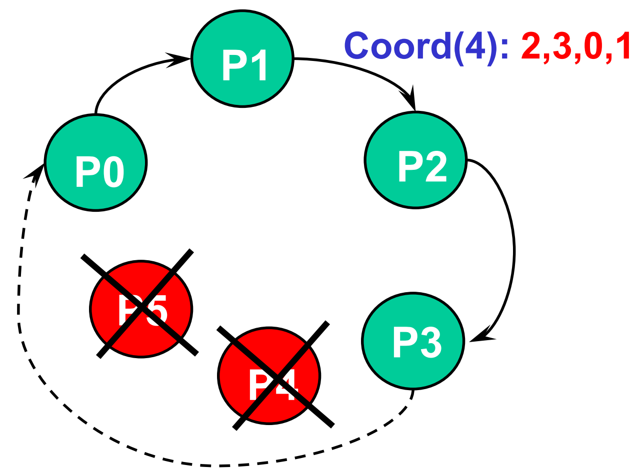


Enhanced ring algorithm

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

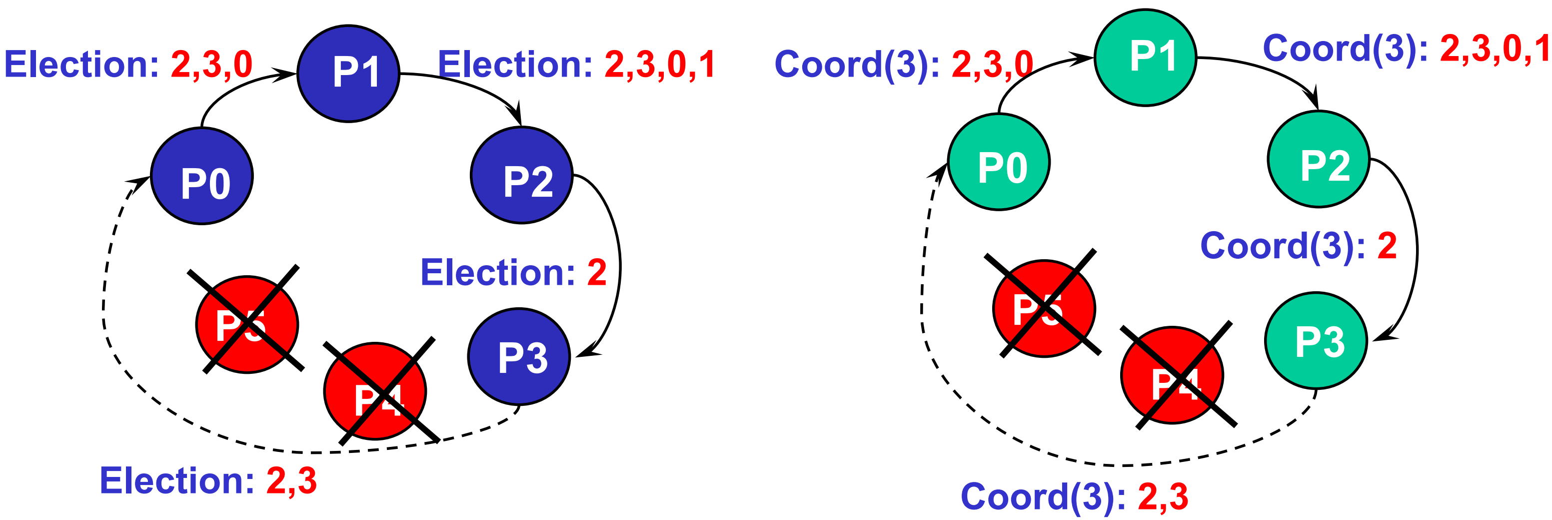
d) Again, each process adds its ID to the message





Enhanced ring algorithm

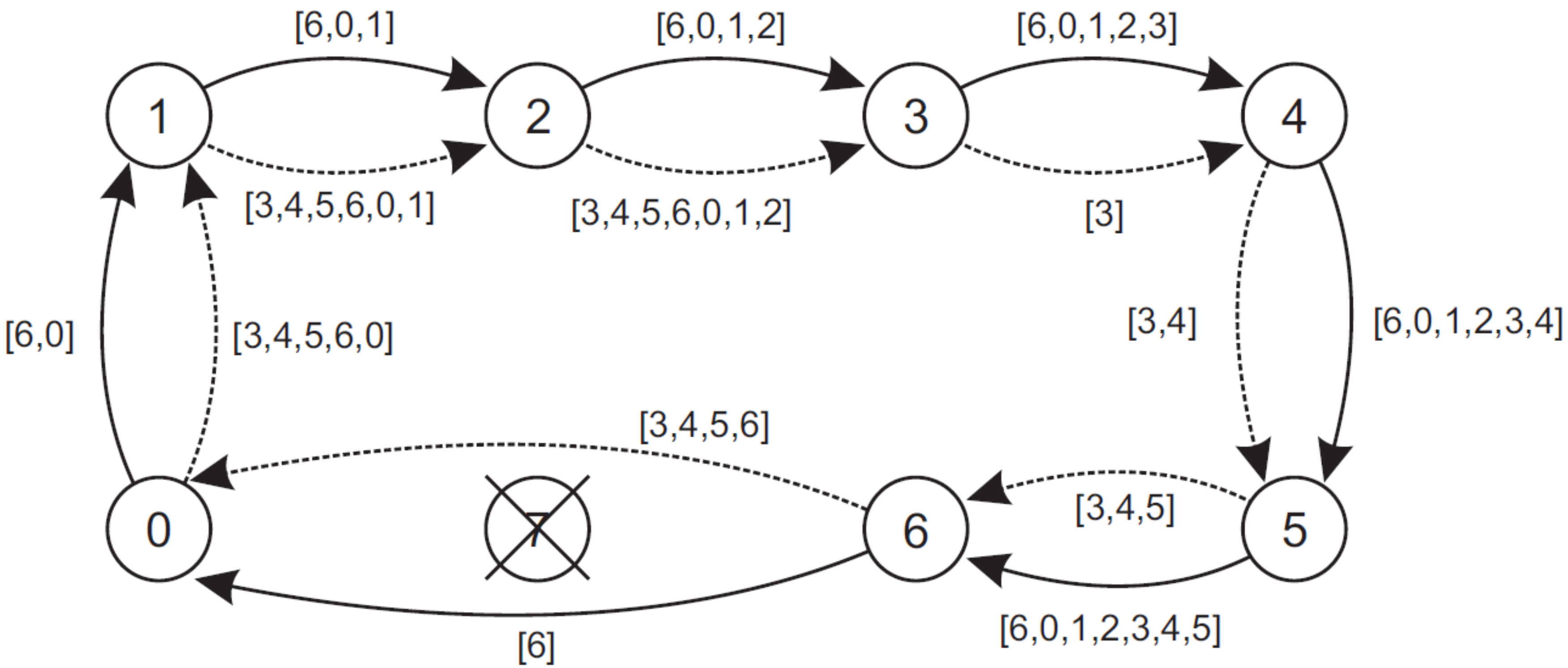
e) Once **Coordinator** message gets back to initiator
  - If elected process is in the ID list, election is over
    - Everyone knows who the coordinator is and who the members of the new ring are
  - Otherwise, the election is re-initiated



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
- As with mutual exclusion, production systems typically use generic consensus algorithms for leader election (refer to slide 24)
  - They tolerate failures and network partitions
  - They provide a single framework for all the agreement problems

Contents

- Mutual exclusion
- Election algorithms
- **Multicast communication**

Multicast communication

- Important service in distributed systems to:
  1. Disseminate data reliably to many users
  2. Implement collaborative applications where a common user view must be preserved
  3. Implement consistency models of replicated data
  4. Implement fault-tolerant (replicated) services
  5. 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

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

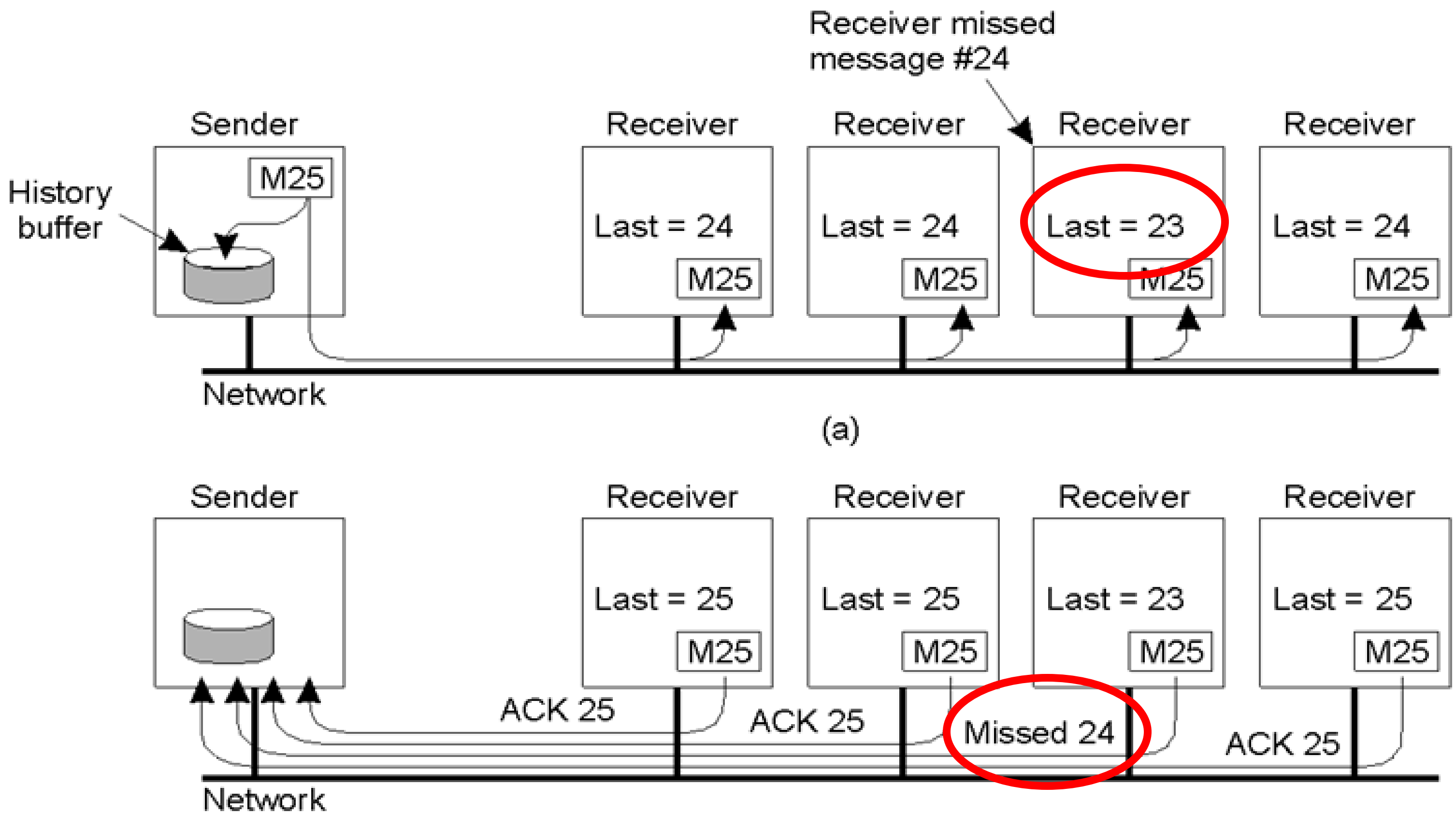
Basic reliable multicasting

- 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



(b)

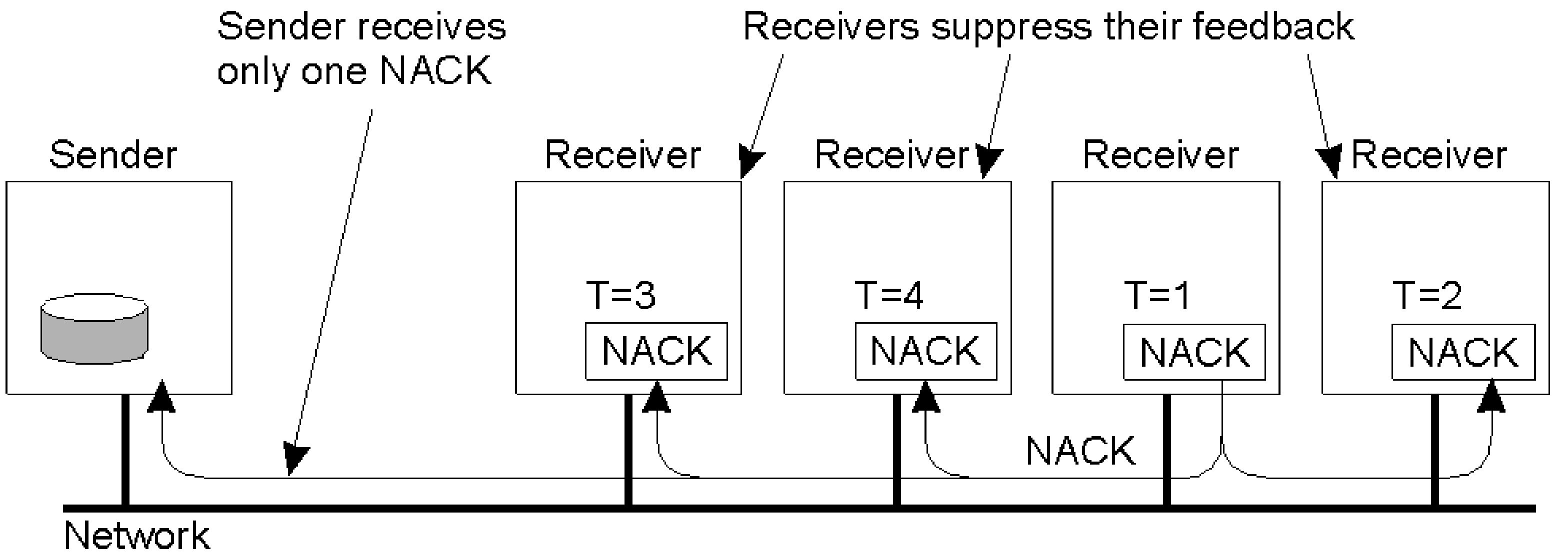
↓ Poor scalability: too many ACKs (feedback implosion)

Contents

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



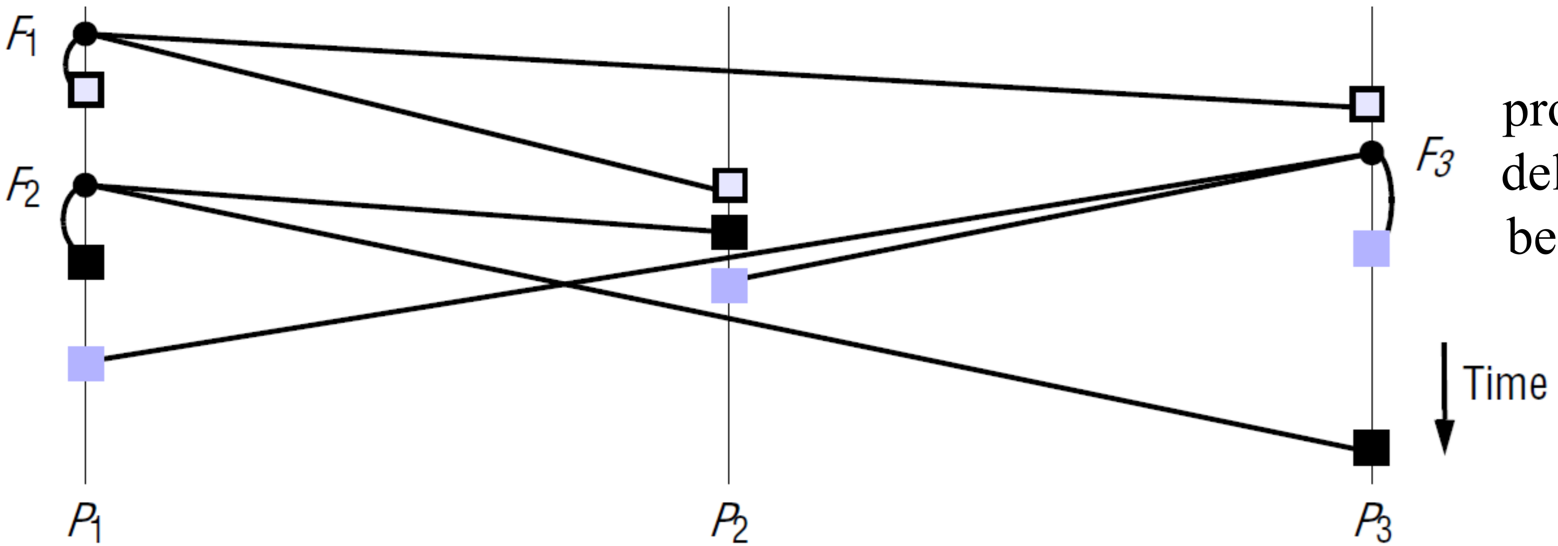
- ↑ Better scalability
- ↓ Setting timers to ensure only one NACK is hard
- ↓ Sender should keep messages in the history buffer forever to guarantee all retransmissions
  - In practice, messages are deleted after some time

Contents

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

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

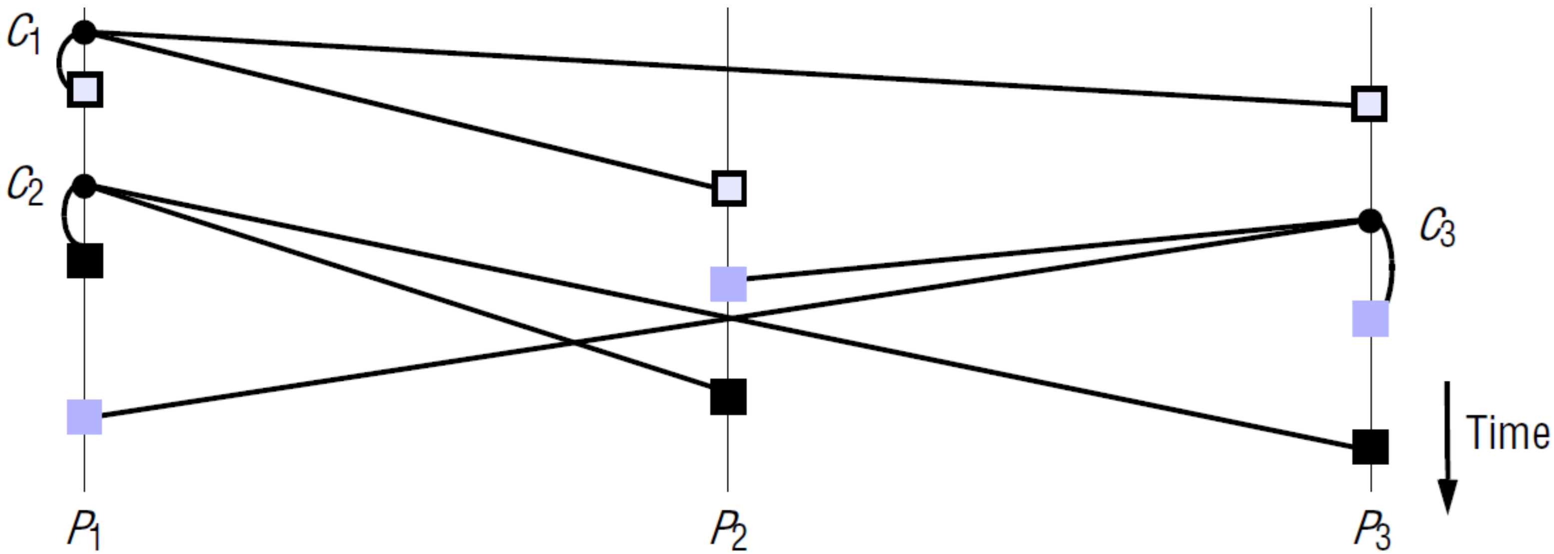


All processes deliver F_1 before F_2

Ordered multicast

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

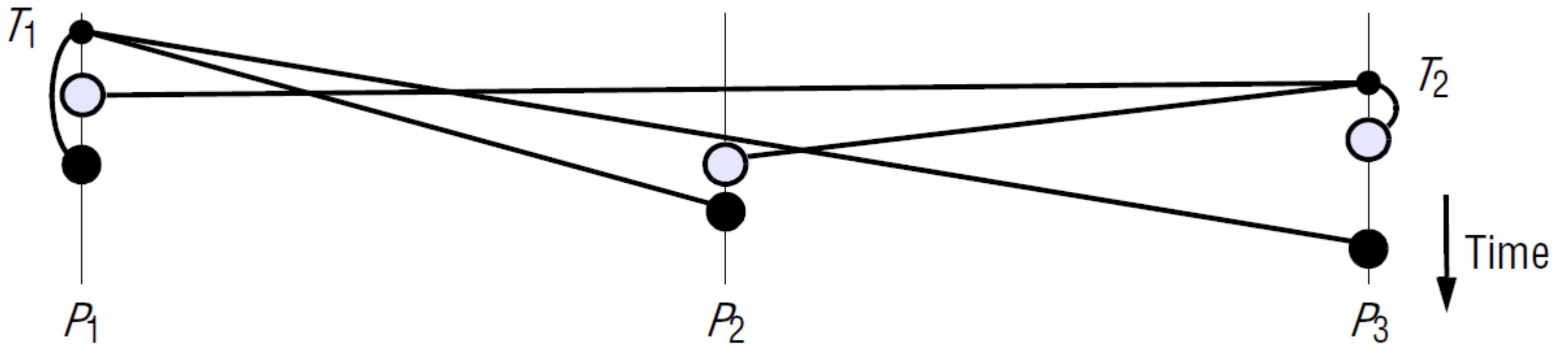
- Causal ordering implies FIFO ordering



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

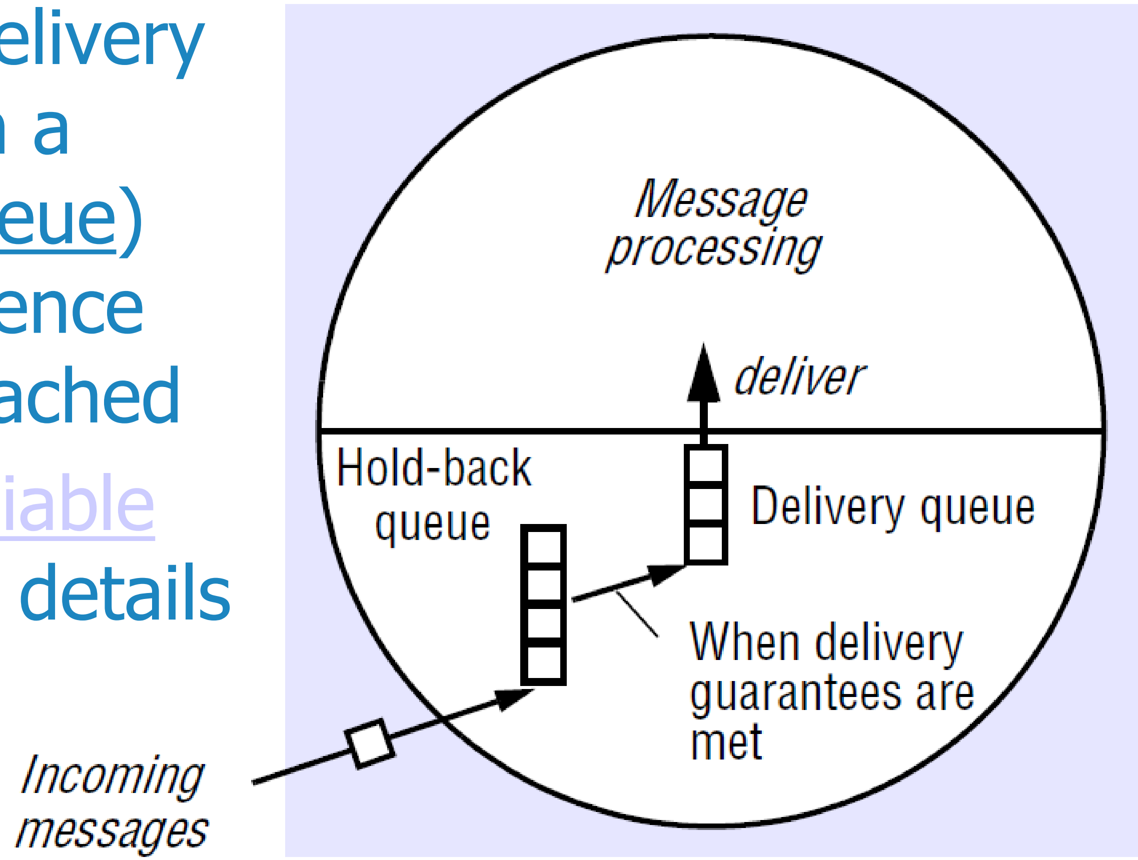
C. <u>Total ordering</u>: all messages delivered in the same order by all processes
   - Hybrid approaches such as FIFO+total ordering and causal+total ordering are also possible



All processes deliver T_2 before T_1

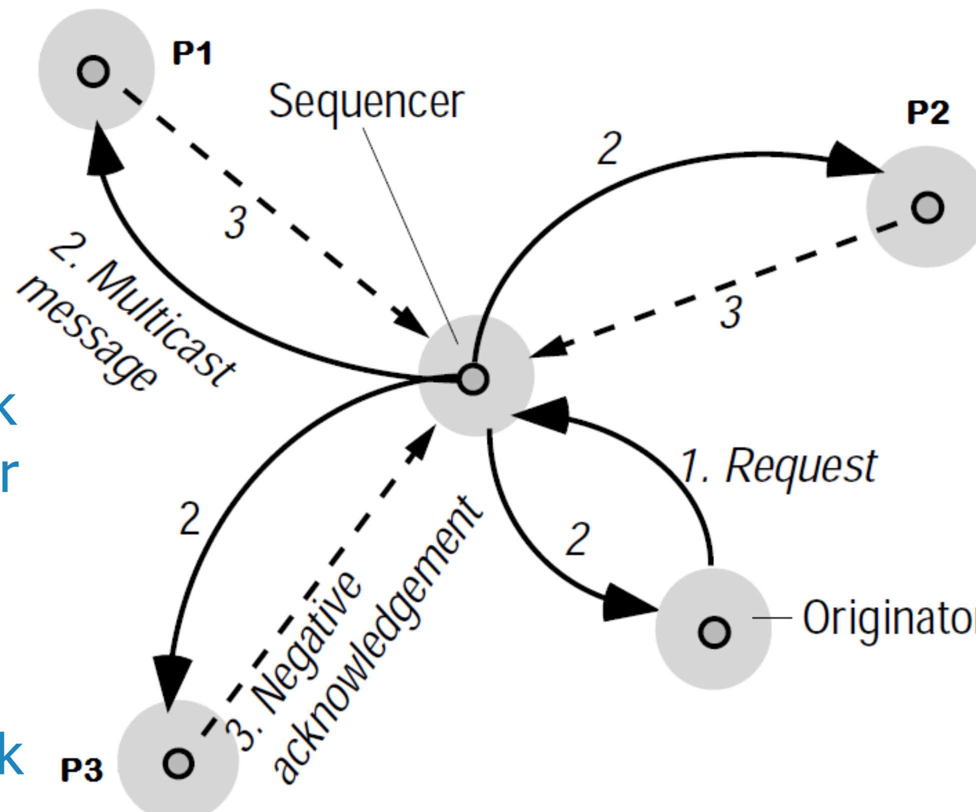
Implementing FIFO ordering

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



Implementing total ordering

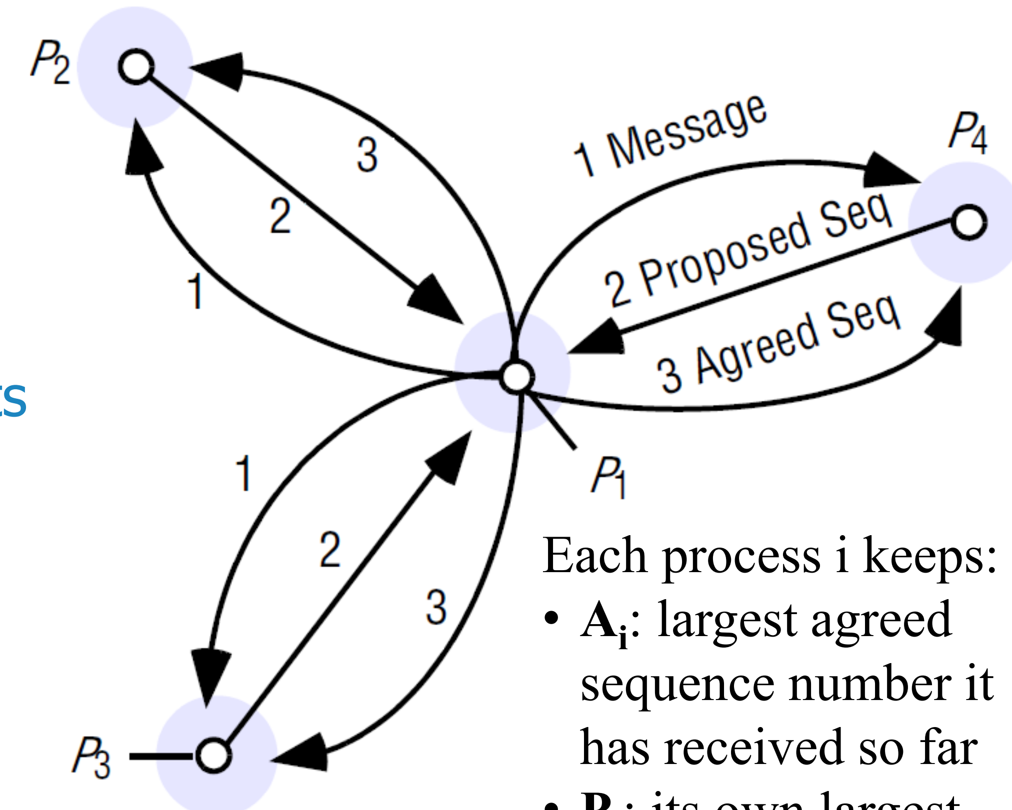
- Using **sequence numbers per group**
  a) Send messages to a **sequencer**, which multicasts them with numbering
    - A message delivery is delayed (in a hold-back queue) until its number is reached
    - Sequencer is a single point of failure and a performance bottleneck



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



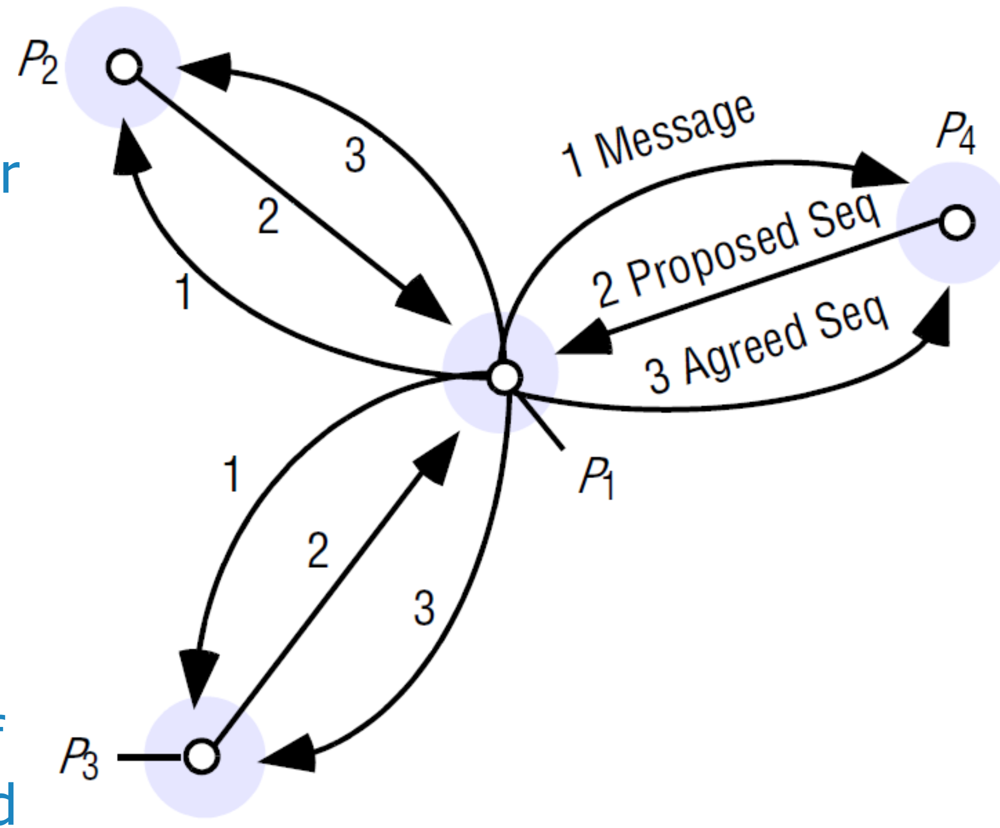
Each process i keeps:

- $\mathbf{A_i}$: 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



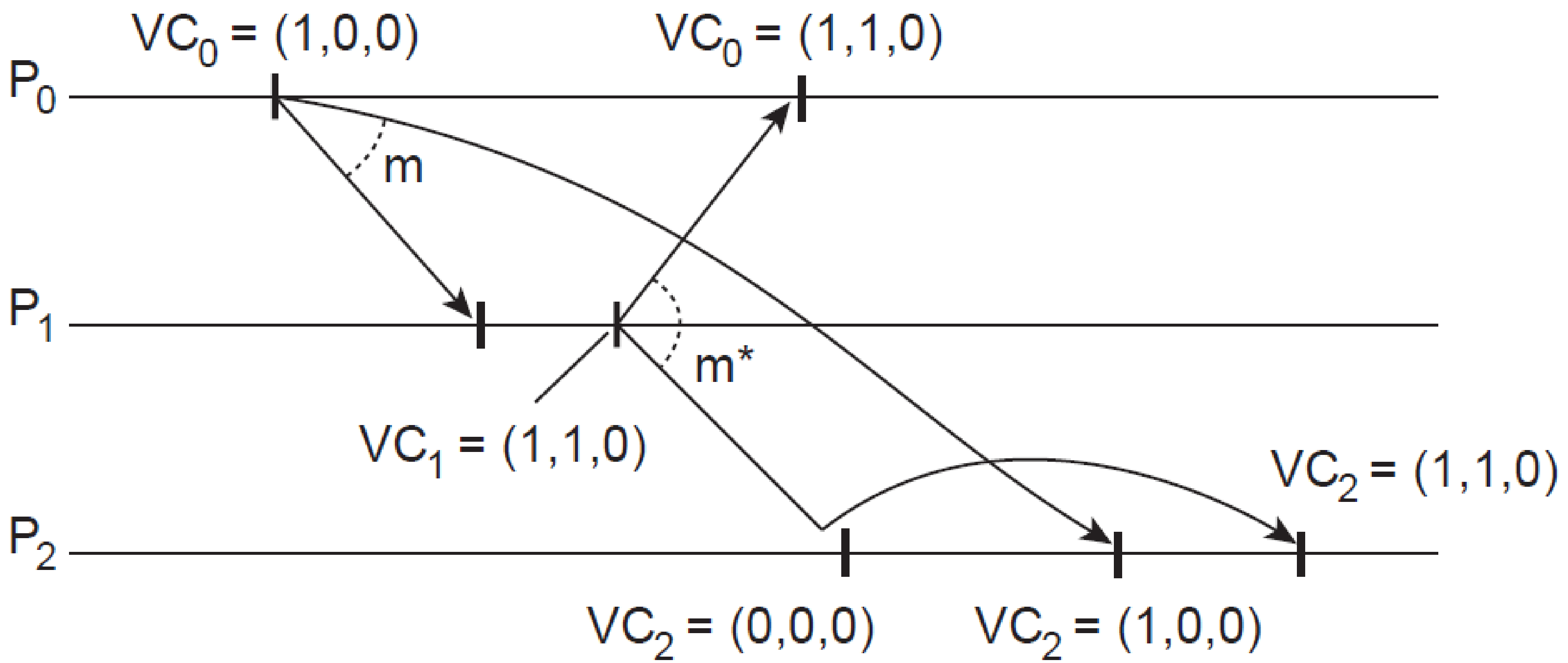
Implementing causal ordering

- 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. $VC(m)[i] = VC_j[i]+1$
      - m is the next expected message from P_i
   b. $VC(m)[k] \leq VC_j[k]\ \forall k \neq i$
      - 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



SEMINAR PREPARATION – Ordy

[Birman91] Birman, K.P., Schiper, A., Stephenson, P., *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems, Vol. 9, No. 3, pp. 272–314, August 1991

[Dasser92] Dasser, M., *TOMP: A Total Ordering Multicast Protocol*, ACM SIGOPS Operating Systems Review, Vol. 26, No. 1, pp. 32-40, January 1992

Contents

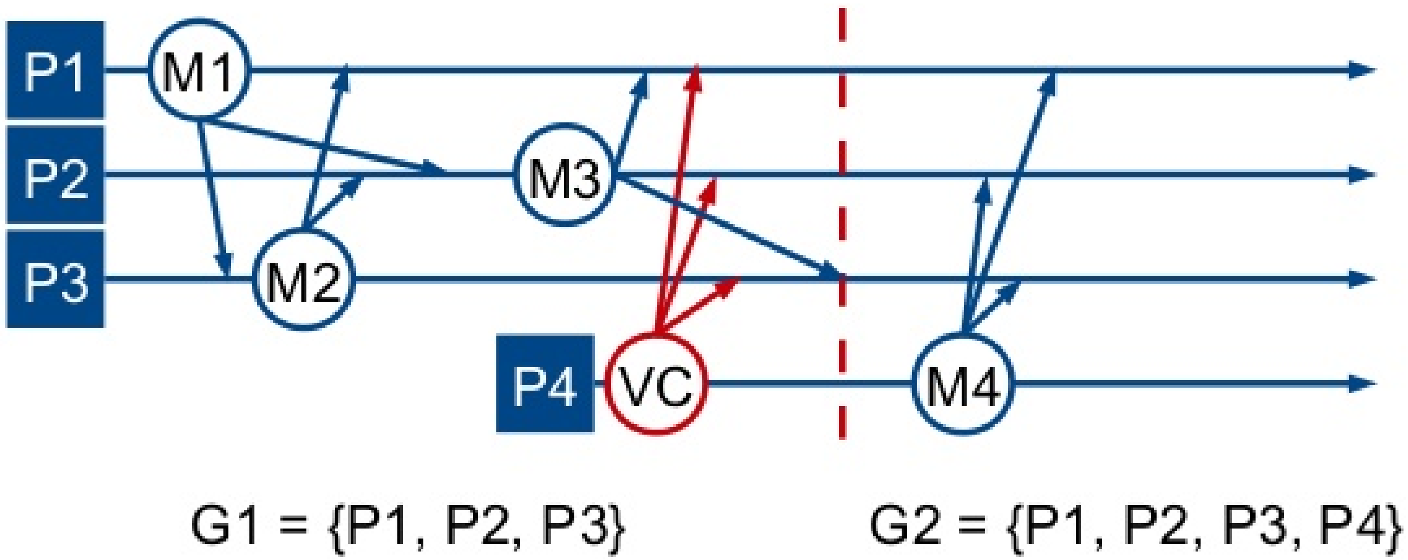
- Mutual exclusion
- Election algorithms
- **Multicast communication**
  - Basic reliable multicast
  - Scalable reliable multicast
  - Ordered multicast
  - **Atomic multicast**

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

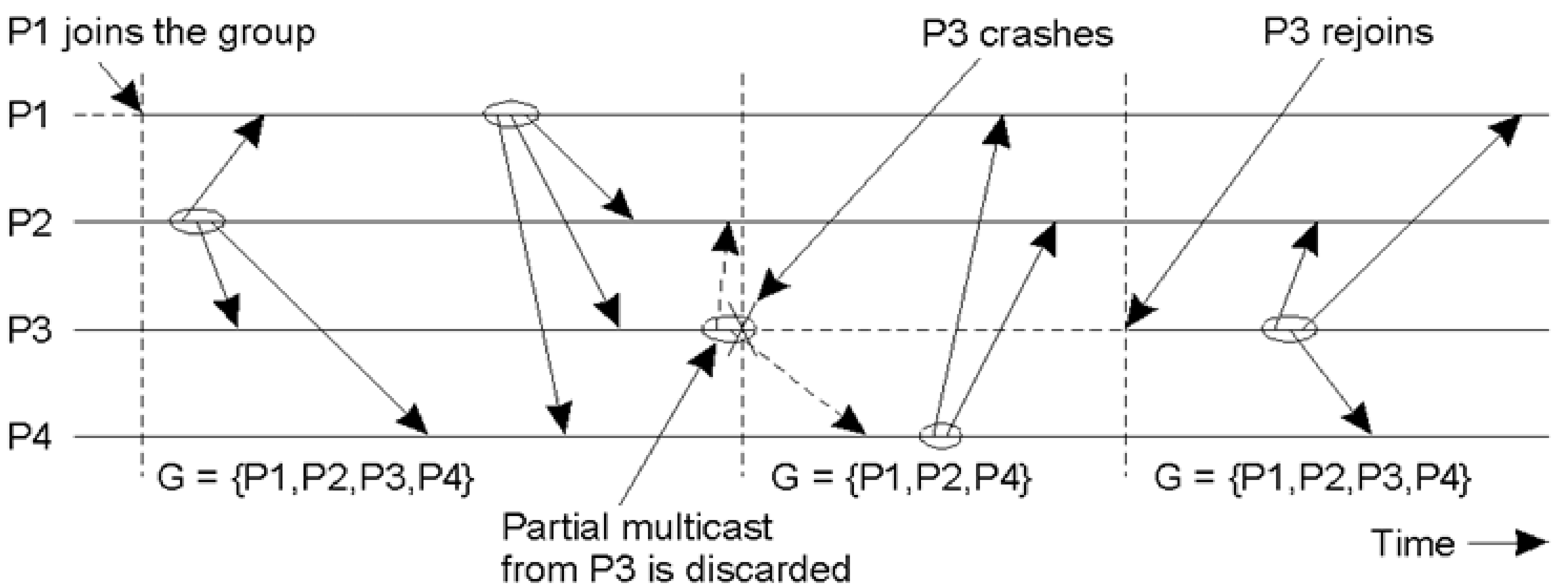
Atomic multicast

- 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

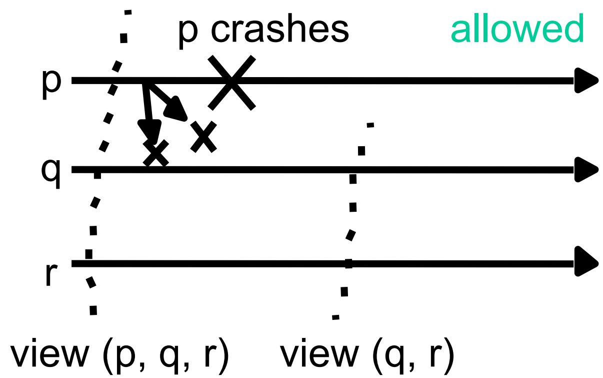


Atomic multicast

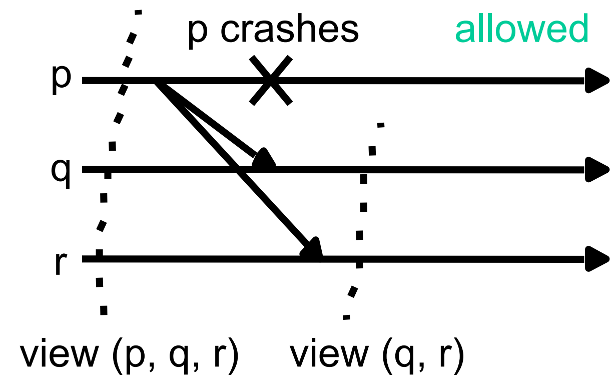
- Each message is associated with a group view
  - The one the sender had when transmitting
- Multicasts cannot pass across view changes
  - All multicasts that are in transit while a view change occurs must be completed before the new view comes into effect



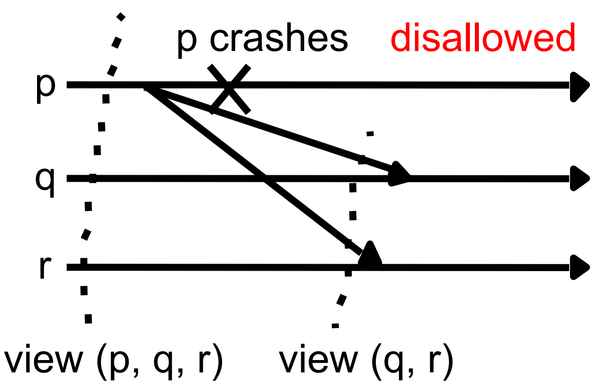
Atomic multicast

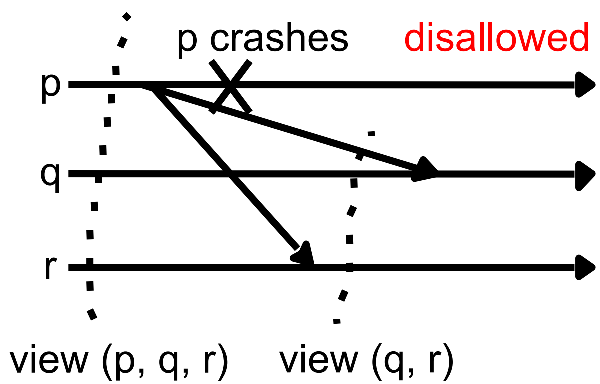


p crashes before m reaches any other process: none of them delivers m

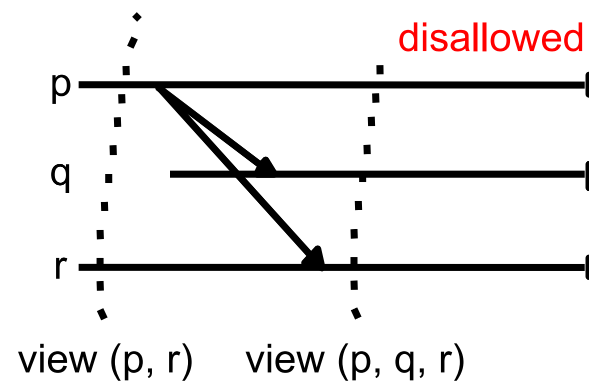


m has reached at least 1 process when p crashes: both q and r deliver first m and then the view





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

SEMINAR PREPARATION – Groupy

[Kaashoek89] Kaashoek, M.F., Tanenbaum, A.S., Hummel, S.F., Bal, H.E., *An Efficient Reliable Broadcast Protocol*, ACM SIGOPS Operating Systems Review, Vol. 23, No. 4, pp. 5-19, October 1989

[Schiper93] Schiper, A., Sandoz, A., *Uniform Reliable Multicast in a Virtually Synchronous Environment*, 13th International Conference on Distributed Computing Systems (ICDCS'93), Pittsburgh, USA, May 25-28, 1993, pp. 561-568

Summary

- Mutual exclusion algorithms ensure that at most one process at a time has access to a shared resource
  A. Centralized algorithm
  B. Voting algorithms (Lin’s and Maekawa’s)
  C. Ricart & Agrawala’s algorithm
  D. Token ring algorithm
- Election algorithms are primarily used in cases where the coordinator crashes
  A. Bully algorithm
  B. Ring algorithms (Chang & Roberts’; Enhanced)

Summary

- Multicast allows sending a message to a specified group of nodes
  - The message is delivered to all nodes in the group or to none at all
    - Even when there are faulty nodes in the group
  - Messages can have also ordering requirements
    - FIFO, causal, and total
- Further details:
  - [Tanenbaum]: chapters 6.3, 6.5, and 8.4
  - [Coulouris]: chapters 15 and 18.2