Programação em Sistemas Distribuídos

MEI-MI-MSI

2018/19

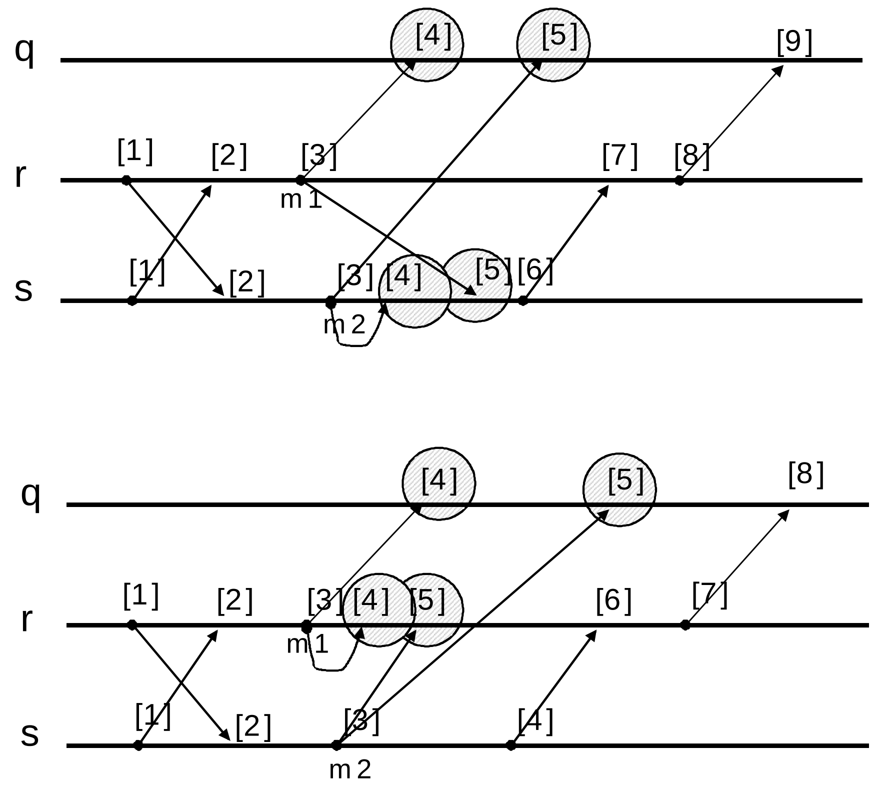
2. Distributed Systems Paradigms

Prof. António Casimiro

Ordering mechanisms and algorithms

Causal ordering with logical clocks

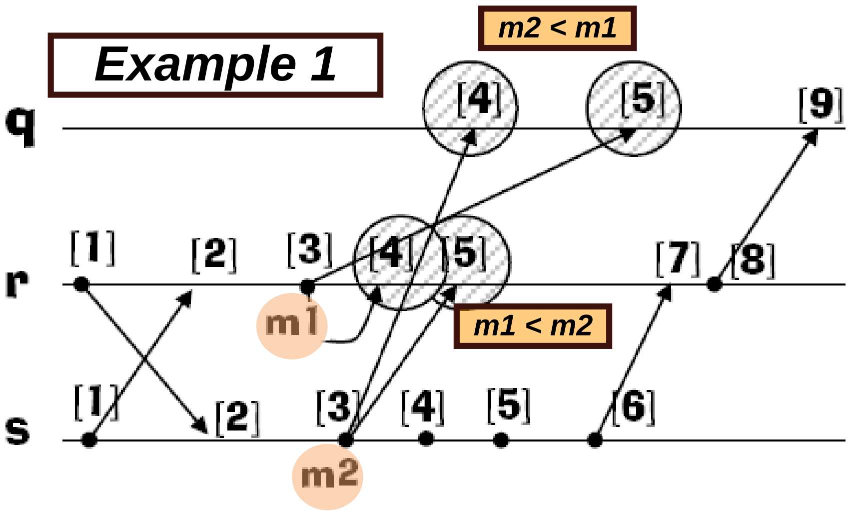
- **Objective**:
  - Order events by cause-effect or precedence (e1 → e2)
- **Implementation rules**:
  - **Initially**: LCi=0 forall i
  - **At each *e* local to *p***: incr. LCp
  - **At each send**: timestamp *m* with LC value: LC(m) = LCp
  - **At each *m* reception at *q*:** (a) update LCq with max [LCq,LC(m)] (b) increment LCq
- **Ordering rules**:
  - e1 → e2 ⇒ LC(e1) < LC(e2)

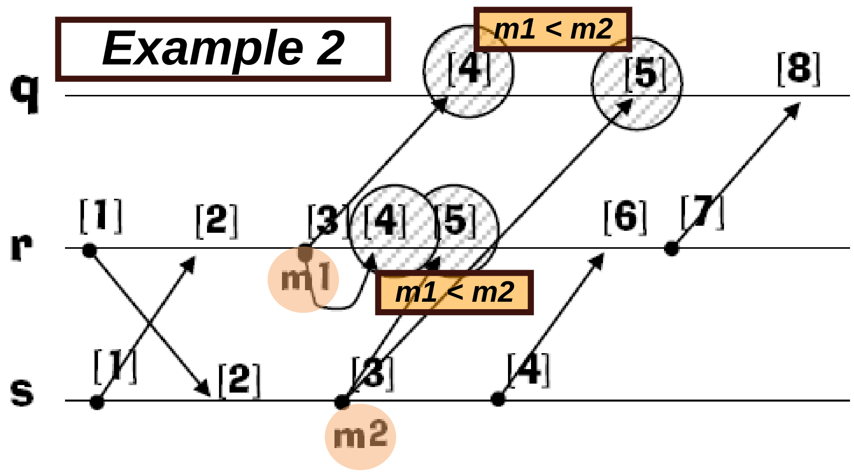


Logical clock limitations

Not (LC(e1) < LC(e2) $\Rightarrow$ e1$\rightarrow$ e2)

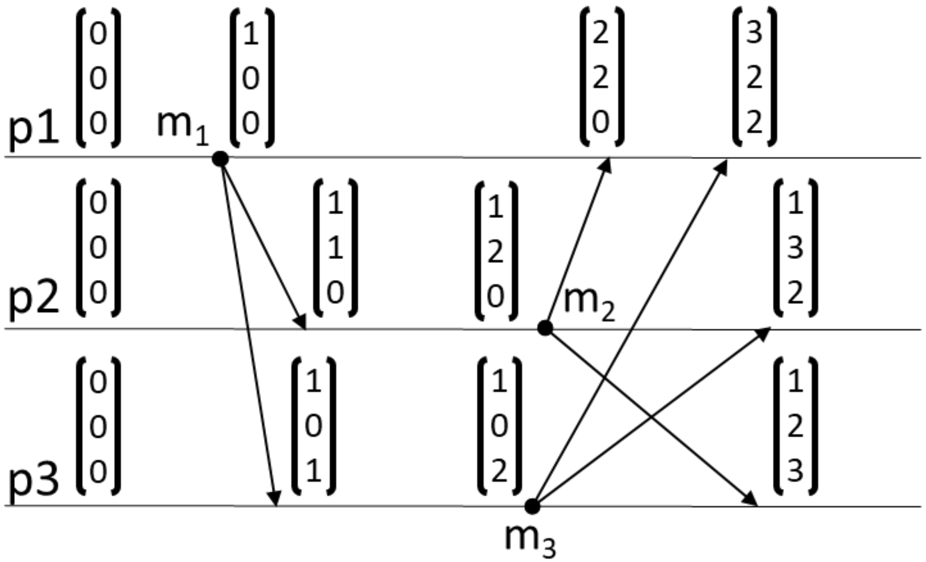
- Ordering rules:
  - e1$\rightarrow$ e2 $\Rightarrow$ LC(e1) < LC(e2)
  - not (LC(e1) < LC(e2) $\Rightarrow$ e1$\rightarrow$ e2)
    - **Clock values do not always mean precedence**
- Example 1:
  - m1 and m2 concurrent
  - Partial order: **OK**
- Example 2:
  - m1 and m2 concurrent
  - Total order: **NOK**
- **Conclusion**: timestamps will order more than necessary!
- This limitation is overcome with vector clocks





Causal ordering with vector clocks

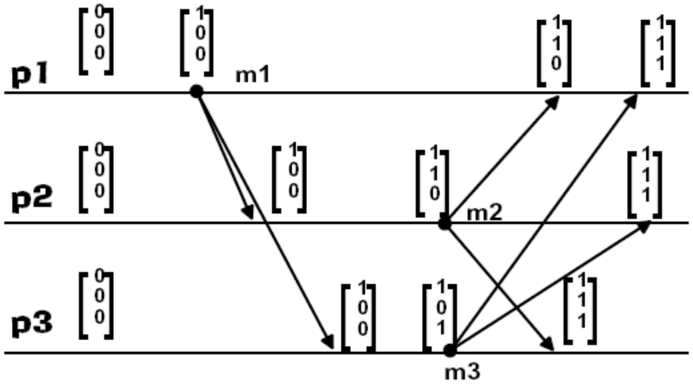
- **Objective**:
  - Order events by cause-effect or precedence (e1 $\rightarrow$ e2)
- **Implementation rules**:
  - Initially: VTi[k]=0 forall i,k
  - At each send m, incr. VTi[i]
  - Timestamp m with VTi: VT(m)
  - At each Rx, incr. VTi[i], and update VTi w/ max [VTi,VT(m)] per position
- **Ordering rules**:
  - m1 $\rightarrow$ m2 iff VT(m1) < VT(m2)
  - m1 || m2 iff VT(m1) || VT(m2)

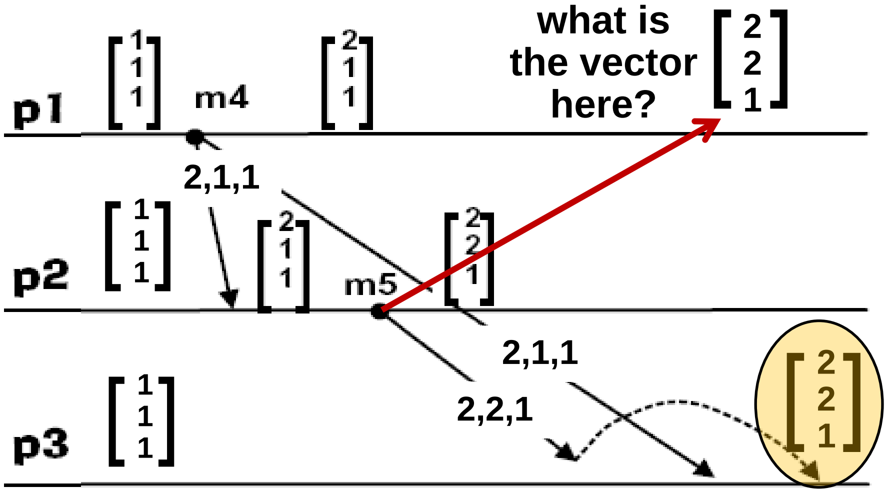


Causal ordering with vector clocks in action

ISIS CBCAST protocol

- **Implementation rules (Pi → Pj):**
  - **Initially**: VTi[k]=0 forall i,k
  - **At each send m**:
    (a) Increment VTi[i]
    (b) Timestamp m with Vti: VT(m)
  - **At each Rx at Pj**, delay delivery until:
    (a) msg m-1 from Pi seen:
    VT(Pj)[i]=VT(m)[i]-1
    (b) all msgs preceding m delivered at Pi, are also delivered at Pj:
    VT(m)[k]≤VT(Pj)[k], forall k
  - Pi does not delay messages to itself
  - **Upon each delivery**: update VT(Pj) w/ max[VT(Pj)[k],VT(m)[k]] forall k
  - **Do not** increment VTi[i] as in normal impl.

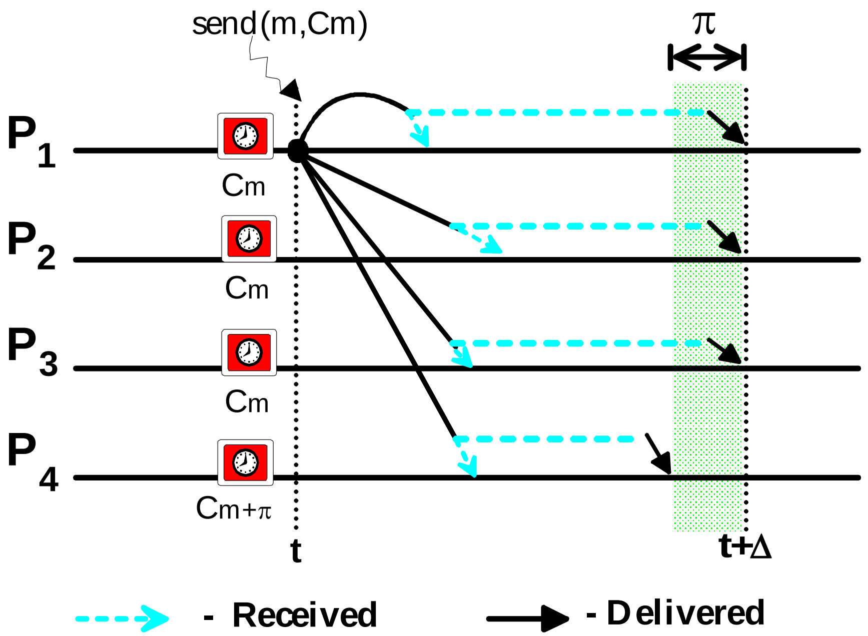




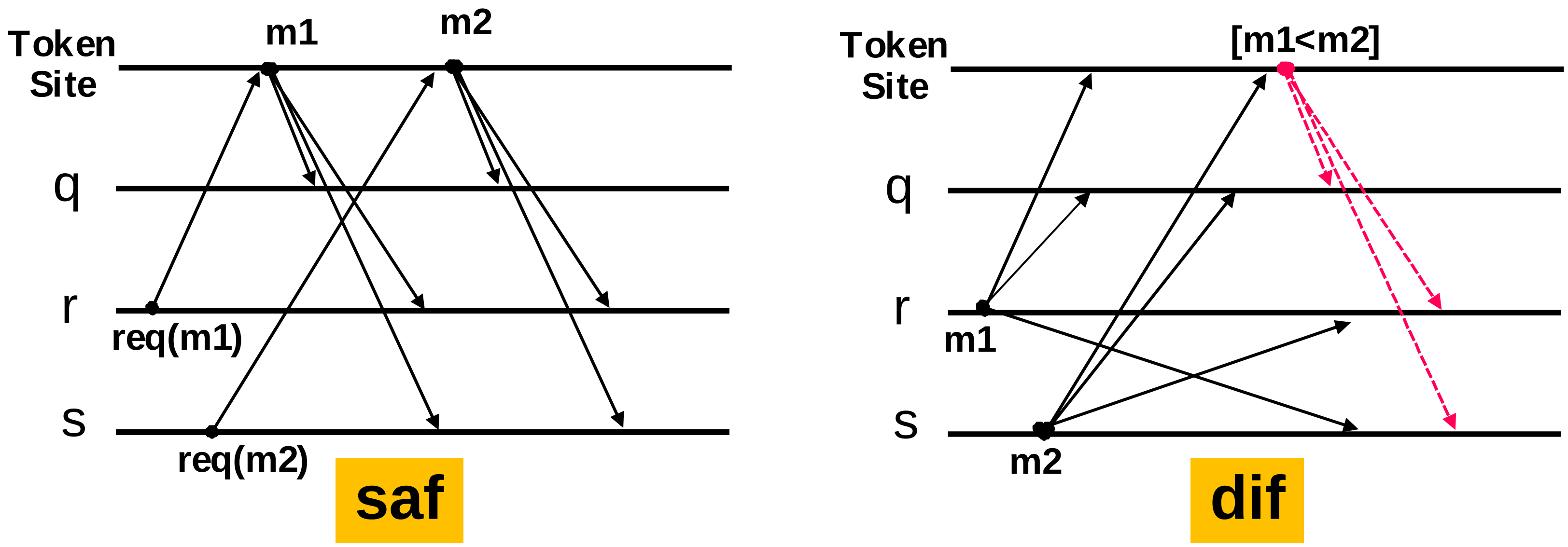
- **Example:**
  - When p3 receives m5, do VT3=[1,1,1]
  - Since VT(m) =[2,2,1], msg from p1 is missing at p3
  - Wait until rx m4, which sets VT3=[2,1,1], and deliver m5, setting VT3=[2,2,1]

Total ordering with physical clocks

- Clock-driven ($\Rightarrow$ stable)
- Reliable diffusion through space redundancy
- Total causal ordering through physical clock timestamps
- Delays delivery until received everywhere
- Delivery of message with timestamp C_m is done at time ($C_m + \Delta$)
- i.e. after C_{m-1} and before C_{m+1} everywhere



Total ordering with sequencer

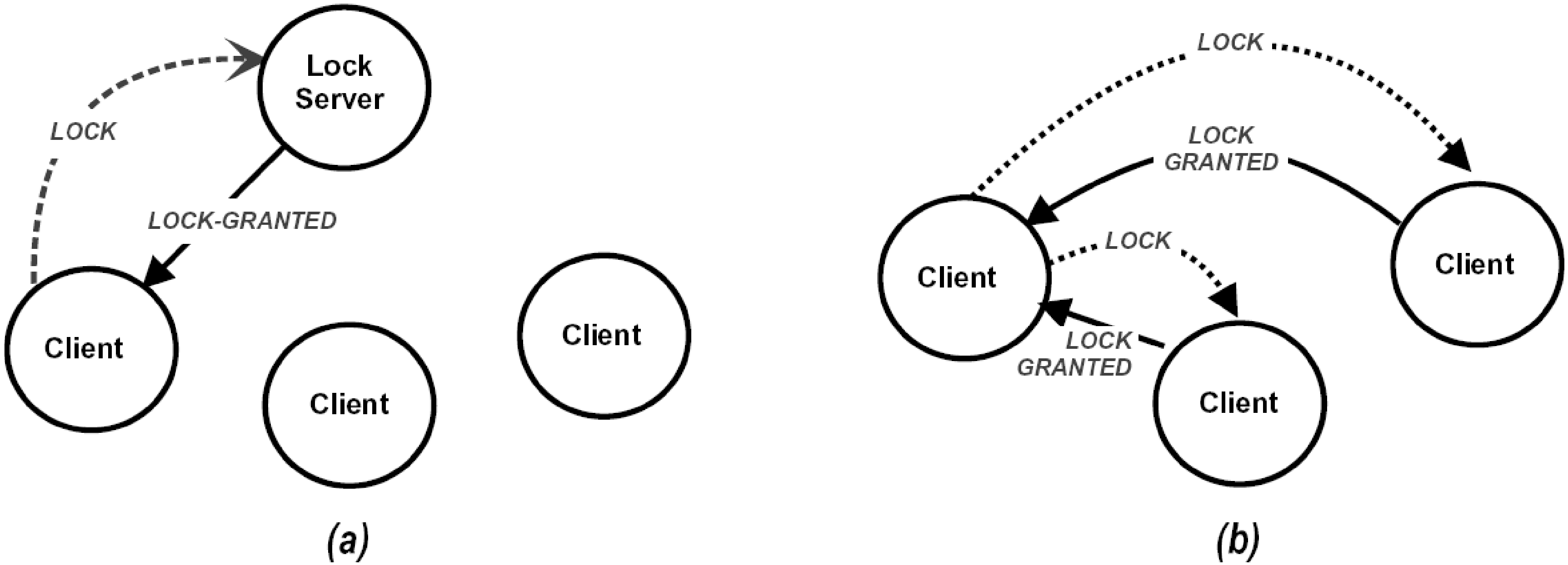


- Reliability by positive ack (saf) or by negative ack (dif)
- Token-based: sequencer decides ordering and propagates to all
- Vulnerable to token-site failure; sol.: rotating token-site or coordinator
- Ordering techniques (total non-causal order):
  - **Store-and-forward (saf)** – senders send to sequencer, who retransmits in order it chooses
  - **Diffusion (dif)** – senders send to all, sequencer only retransmits ordering sequence

Coordination and Consistency

Distributed Mutual exclusion

Control: (a) Centralized; (b) Decentralized



Distributed Mutual exclusion

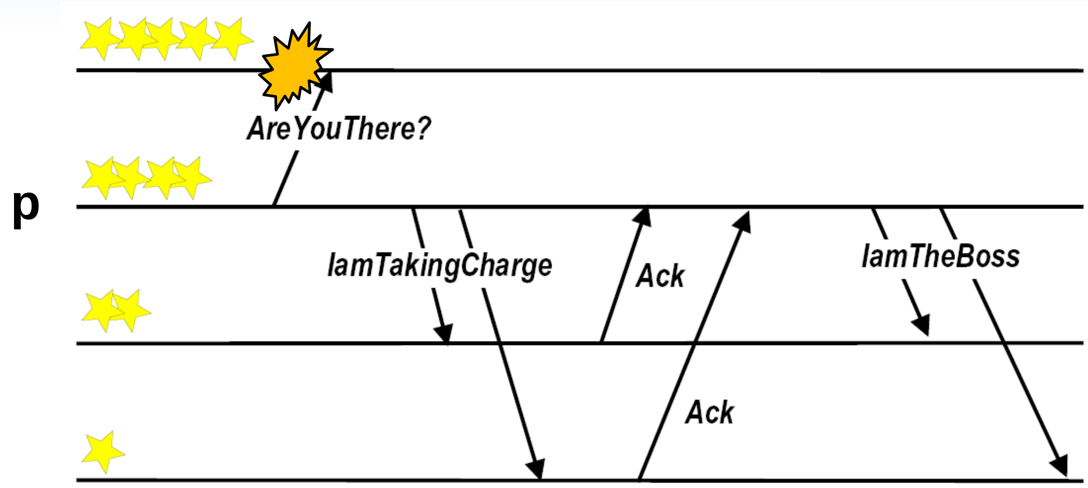
Lamport algorithm *[Lamport:78]*

Ciências ULisboa

- Decentralized, based on logical clocks enhanced with UniqueProcessID (UID) to achieve causal total order (see Ordering):
  - **LOCK** requests are disseminated to all clients and inserted in a waiting queue, organized in the order of request timestamps + UID
  - Receiving process stores request in the waiting queue and returns a timestamped **REPLY** message
  - As soon as a REPLY message has been received from every other process, the request is marked as **stable** (it was received by all processes)
  - A process enters the critical section when:
    (a) its own request is at the head of the waiting queue
    (b) the request is stable
  - When leaving the critical section, the request is removed from the head of the queue and a **RELEASE** message is broadcast
  - When the RELEASE message is received, the respective lock request is removed from the queue

Leader election

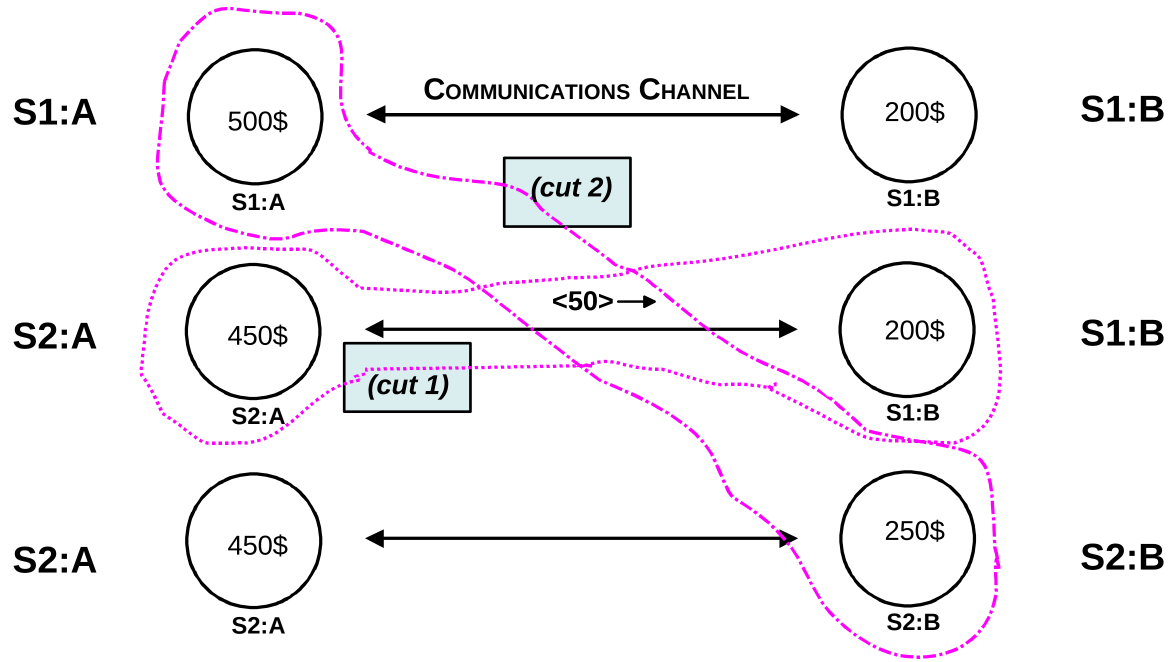
Bully algorithm *[Garcia-Molina:82]*



- Always elects the highest ranked active candidate (process with the lowest id)
- Instead of sending a message to every process, process p trying to become leader just sends **AreYouThere**? to higher ranks, if someone replies, p silently gives up
- If nobody replies, p tells all lower ranks of his intention, sending **IAmTakingCharge**, and waits for **Ack** from each
- When all **Acks** come (or a timeout occurs, since some of the processes with lower rank may have crashed), p assumes the leadership by sending **IAmTheBoss** to all processes.

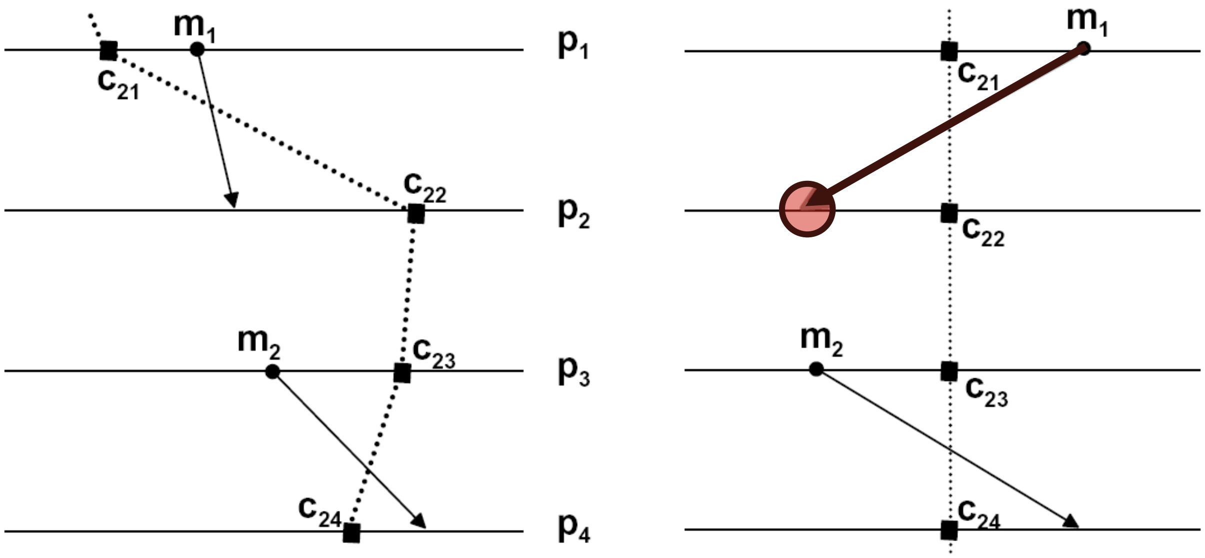
Consistent Global States

Issues with Ad-hoc State Snapshots



- total=700$; money transfer A-> B, 50$; during transfer, ad-hoc snapshot is done, from external node sending messages to A and B.
- cut 1: <S2:A=450$, S1:B=200$>=650$! (msg sent, not received!)
- cut 2: <S1:A=500$, S2:B=250$>=750$! (msg received, not sent!)
- A correct snapshot protocol will flush the channels to ensure consistency

Inconsistent Cuts



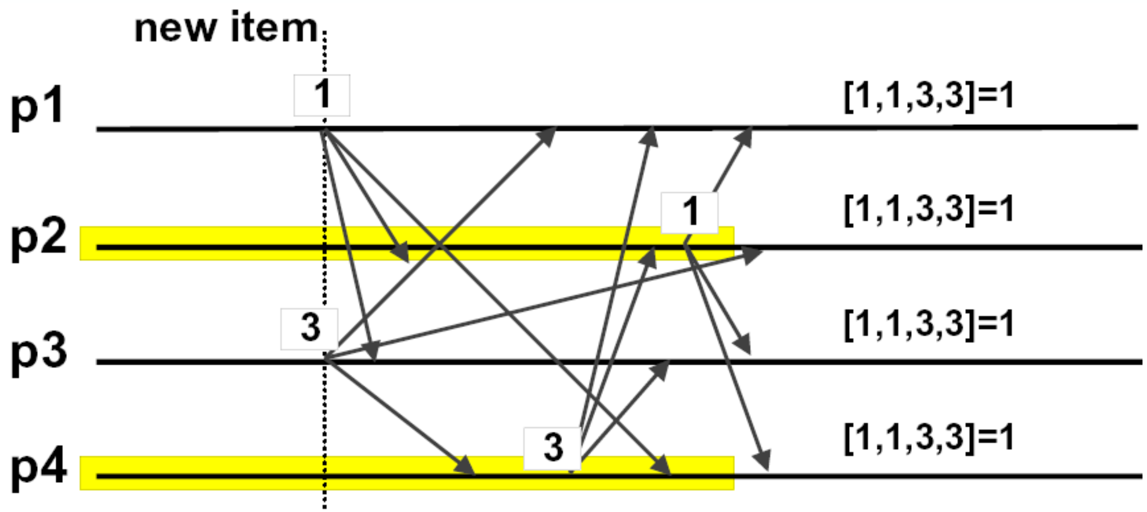
- What you get if you don't use the right algorithm:
  - Is there anything strange with message m1?
  - **And now?**
  - A correct snapshot protocol will discard messages "not sent" to ensure consistency

Consensus

- **Validity**
  - If a process decides *v*, then *v* was proposed by some process
  - No process decides more than once
- **Agreement**
  - No two correct processes decide differently
- **Termination**
  - Every correct process eventually decides

- **Consensus is equivalent to atomic broadcast**
  - That is, one can implement one with the other
  - Does not mean that all such implementations are efficient!

Distributed consensus
Intuition



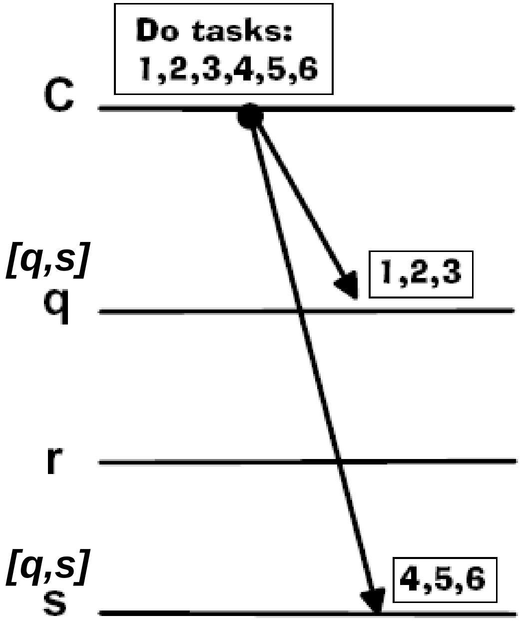
- Set of processes must agree on one action, in a decentralized way: decide who keeps each new item that arrives, all send their votes to all
- New item arrives, p2 and p4 are busy, so only p1 and p3 offer to pick it
- p2 receives the proposal from p1 in the first place, thus it supports p1, while for the same reason p4 supports p3
- When all votes are collected, all have same vector to decide from (1,1,3,3)
- Any agreed deterministic function will do, ex:
  - "winner is the one with more votes, in case of tie, the smaller ID wins"
- The item is assigned to p1

Membership

- Group membership
  - Set of processes belonging to the group at a given point in time
- Membership service:
  - Keeps track of membership and provides info to group members
- Group view:
  - Subset of members mutually reachable at a given point
- Group membership is often dynamic:
  - In response to user demand or changes in the runtime environment (load, failures, etc)
  - It may grow, by letting new processes **join** the group
  - It may shrink, by letting members **leave** the group
  - View changes when processes **fail** or when they recover

Agreement on Membership
Decentralized applications

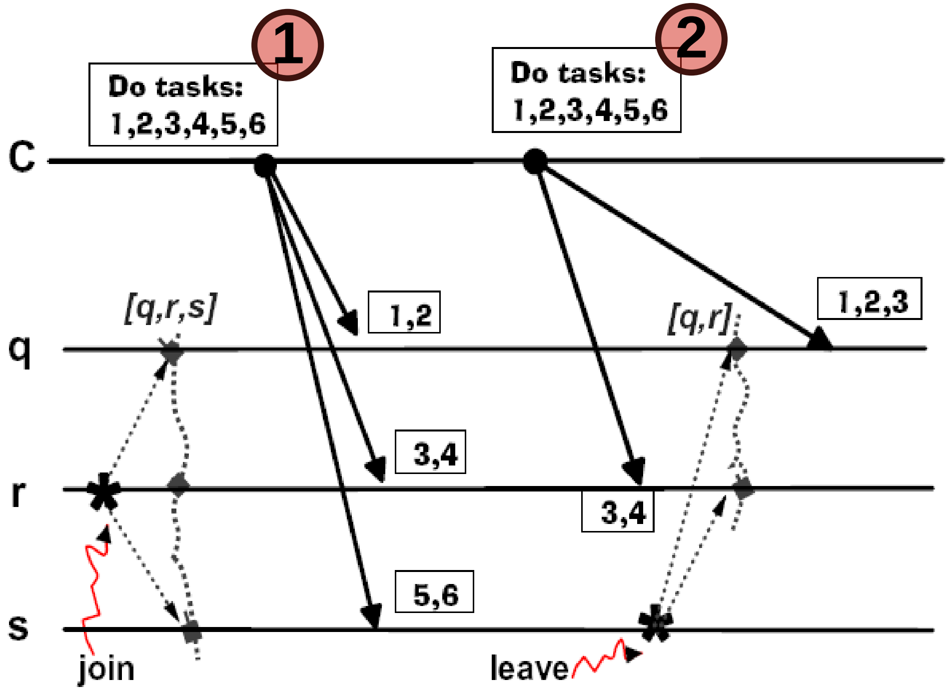
- Coherent notion of membership useful for a number of applications
- E.g. decentralized dispatcher
  - Group of workers (set of parallel processors) is currently [q,s]
  - They divide a task requested by client by the current number of elements
  - Dispatch is local, split is dynamic
- Processors may come and go:
  - How to do it?



Agreement on Membership

Issues with ad-hoc view change

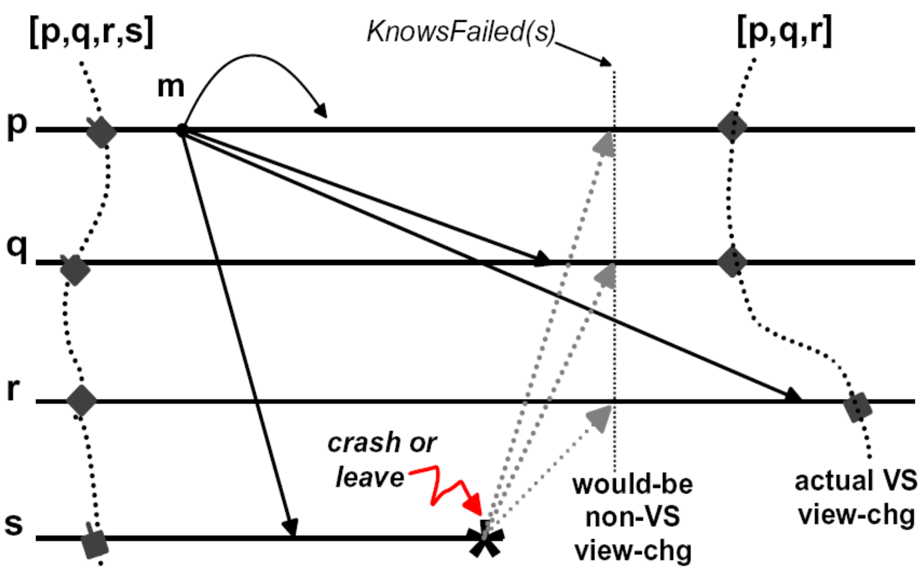
- r joins group, view is {q,r,s}, (1) is performed, then s leaves, view is {q,r}, (2) is performed
- Change notification not consistent:
  - r gets request 2 in view {q,r,s}, so picks <3,4>
  - q gets request 2 in view {q,r}, so picks <1,2,3>
- What went wrong:
  - <3> is performed twice
  - <5,6> are not performed



View Synchrony

View-synchronous view change

- Solution to previous problem:
  - Membership changes notified **consistently with message flow**!
  - If a message m is delivered to a process p in view V_i, then for all q in V_i, m is also delivered to q in view V_i.
- How to ensure all processes deliver same messages in same view?
  - Flush messages until a consistent cut

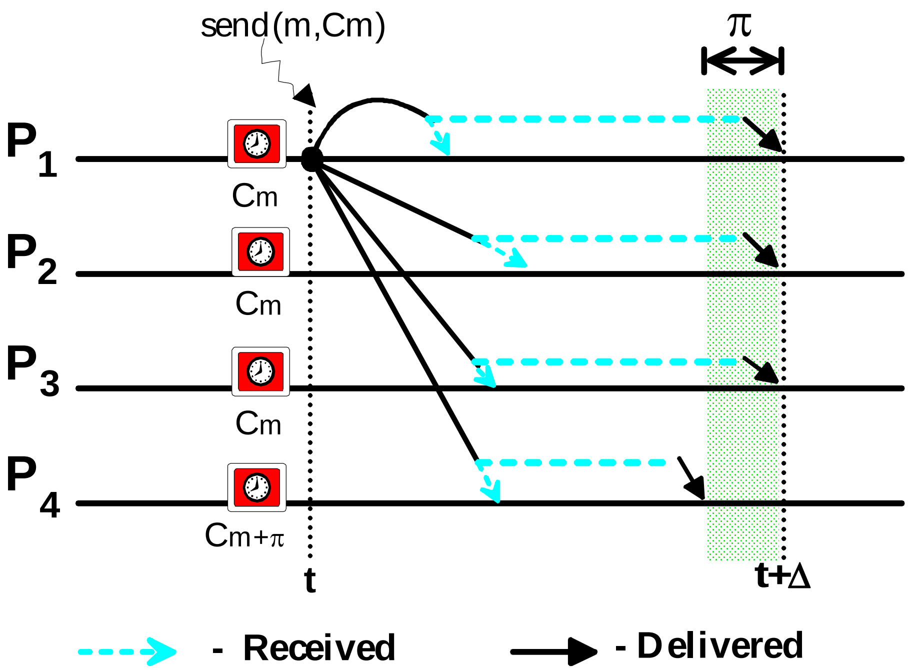


Atomic Broadcast
Properties

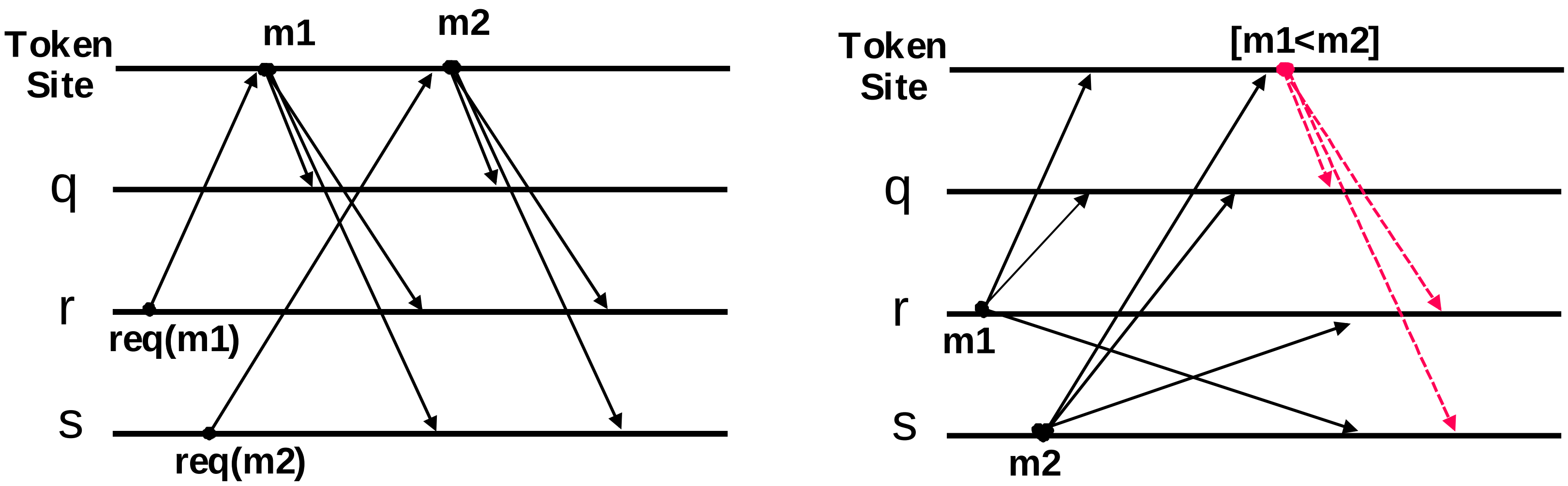
- **Validity**
  - If a correct processor broadcasts a message M, then some correct processor eventually delivers M
- **Agreement**
  - If a correct processor delivers a message M, then all correct processors eventually deliver M
- **Integrity**
  - For any message M, every correct process p delivers M at most once
  - If process p delivers M and sender(M) is correct, then M was previously broadcast by sender(M)
- **Total order**
  - If two correct processors deliver two messages M1 and M2 then both processors deliver the two messages in the same order

Atomic Broadcast
Symmetric approach - intuition

- Reliability through space redundancy or tx-w-resp
- Total causal ordering through physical clock timestamps
- Delivers by message timestamp order
- Disambiguates e.g. by UID or MAC address, etc.
- I.e. msg(Cm) after Cm-1 and before Cm+1 everywhere
- Can be done with logical timestamps



Atomic Broadcast
Asymmetric approach - intuition



- Reliability by tx-w-resp w/ store-and-forward or diffusion w/ negative ack
- Token-based: sequencer decides ordering and propagates to all
- Total non-causal order

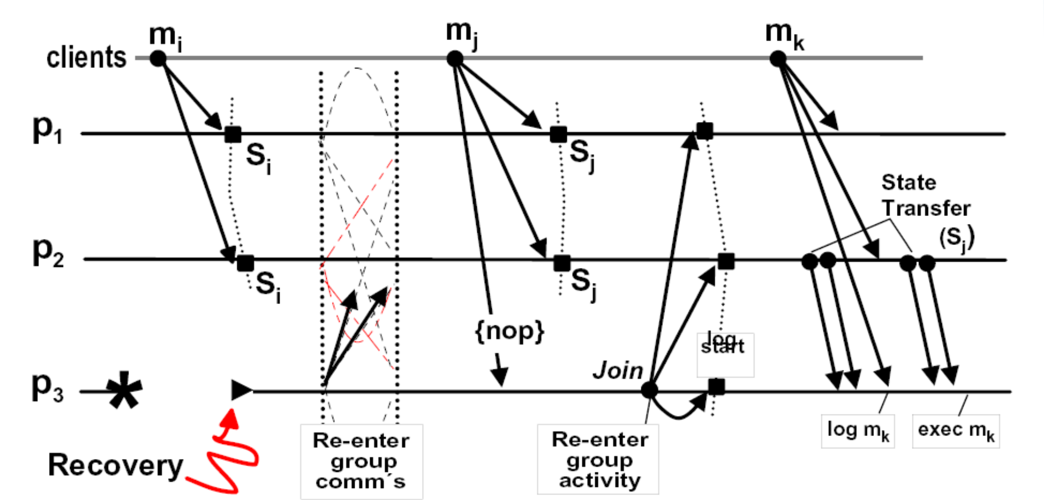
Replicated computations

- Distributed applications may run replicated pieces of code which should behave in the same way (e.g. fault tolerance, performance)
- Atomic broadcast guarantees, in a decentralized way, that replicas receive the same sequence of inputs:
  - Same requests, in the same order

Replica determinism

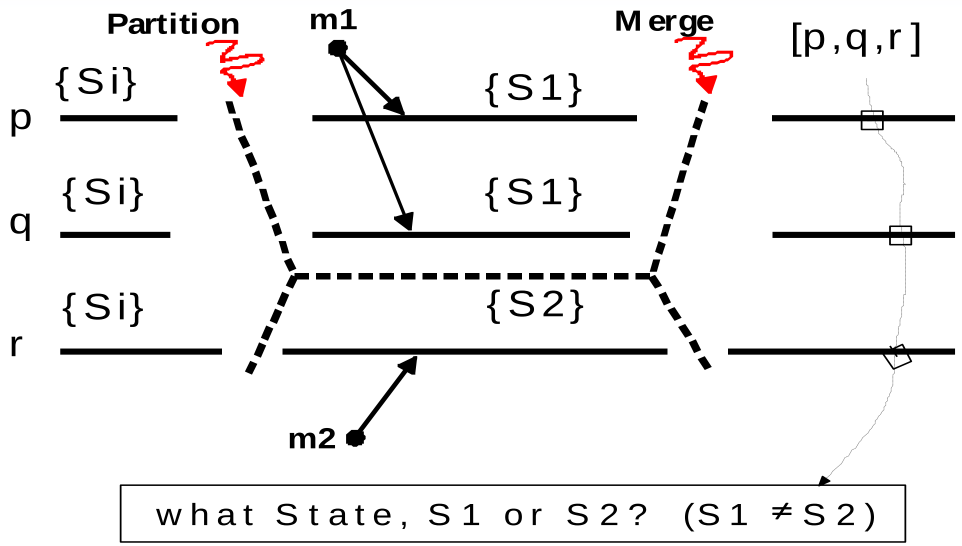
- Replica determinism:
  – Two replicas, departing from the **same initial state** and subject to a **same sequence of inputs** reach the **same final state** and produce the **same sequence of outputs**
- Atomic broadcast:
  – Guarantees "**same sequence of inputs**" objective
  – The rest lies with the replica itself
- Issues:
  – Deterministic coding
  – Replica failure and recovery
  – State divergence with partitioning

Replica failure and recovery



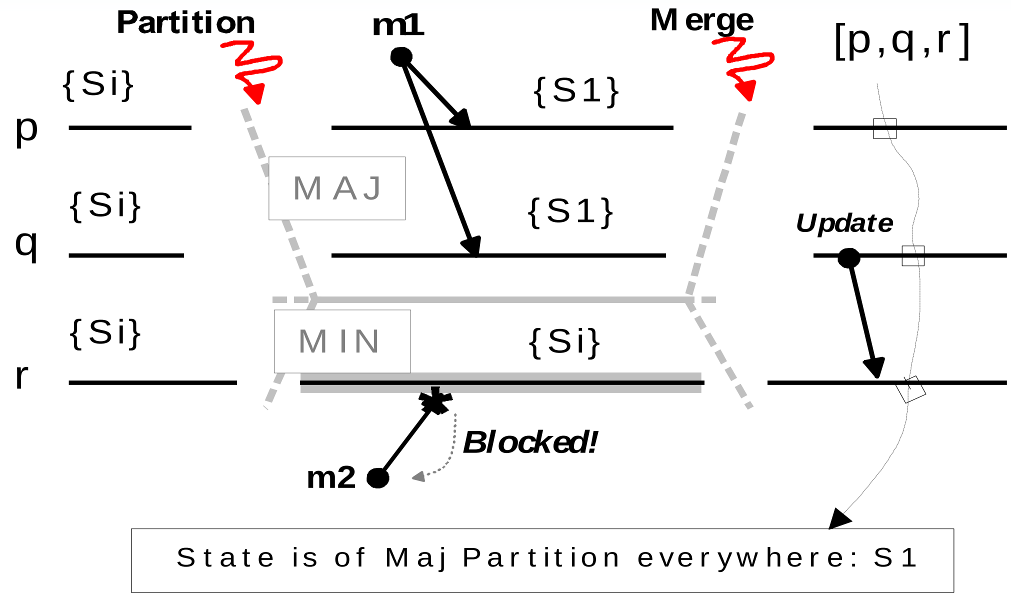
- Recovering replica (p3) starts by resuming communication with the replica set, e.g. if the set was using some form of group communication
- It starts receiving all messages, but still discards them
- Next, sends a request to join the replica group activity, delivered in total order to all replicas, including the joining replica, marking a cut Sj in the global system state request, which triggers a state-transfer operation
- p2 checkpoints its state at this point (Sj), and sends it to p3
- p3 starts logging any messages that arrive after the cut Sj
- New requests (mk) can continue to be processed by all replicas except p3

State Divergence with Partitioning



- Partitioning occurs, p,q execute cmd m1 and assume state S1
- r executes cmd m2 and assumes state S2
- What is system state after merger?
- E.g., if m1 and m2 produced conflicting results, it is impossible to find a coherent common state without special-purpose *reconciliation* (application dependent)

Avoiding State Divergence
Primary partition



- As before, but now only primary partition continues executing
- PP has majority of replicas, i.e. <p,q>
- <r> stays blocked in state Si
- <p,q> continues, processing m1, and goes to state S1
- After merger, <r> requests state update to set <p,q>
- Since Si (of <r>) is a prefix of S1 in PP history, there is no divergence