Distributed Systems

Fundamentals – Part Two

Professor Olivier Gruber

Université Joseph Fourier

Projet SARDES (INRIA et IMAG-LSR)

Message Fundamentals

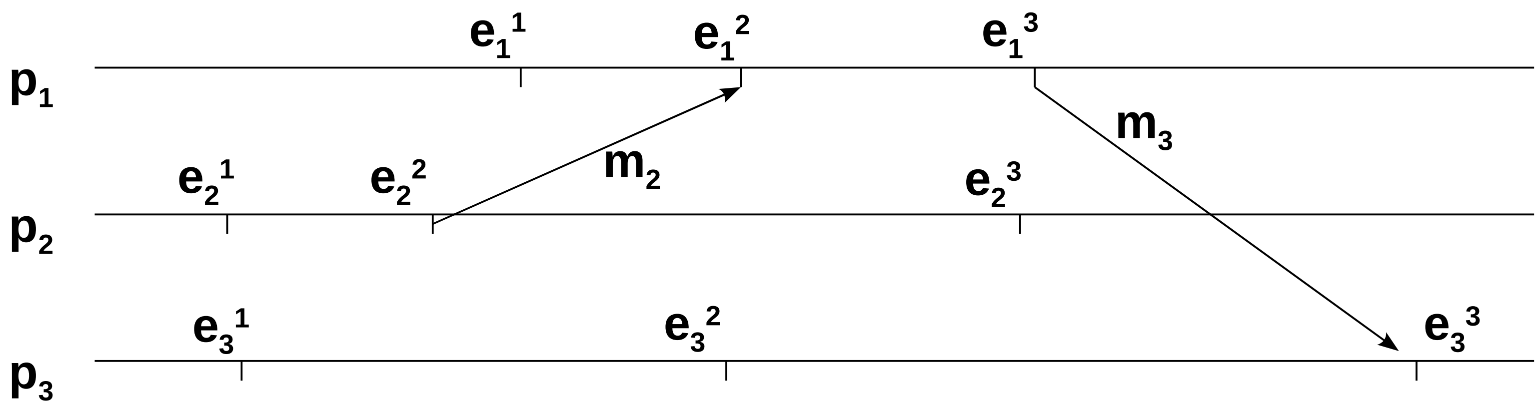
- Specific techniques
  - Logical clocks and totally-ordered multicast
  - Vector clocks and causally-ordered multicast
  - Matrix clocks and causal point-to-point messaging
  - Election in distributed systems
  - Replication
  - Consensus

Totally-Ordered Multicast

- Problem
  - How do we order multicast messages to a group of processes?
- Example – Bank Account Interest
  - You deposit 100€ to your account that contains 1000€
  - Banker applies your monthly interest 1%
  - Bank accounts are replicated in Paris and Berlin
    - Same execution order = 1110€
    - Different execution orders = 1111€
- Example – Deposit and Withdrawal
  - Same bank, you deposit 400€ and withdraw 1200€
    - Same execution order, accepted on all replicas
    - Different execution orders, one replica may reject the withdrawal

Execution Model

- Process model
  - Each process is a local sequence of events
    - p_i : $e_i^1, e_i^2, e_i^3, \ldots, e_i^k, \ldots$
  - An event is a local state change in the process
- Communication model
  - Process may exchange messages
  - Message delays are unknown, messages may be lost
  - Sending or receiving a message is a state change, thus an event

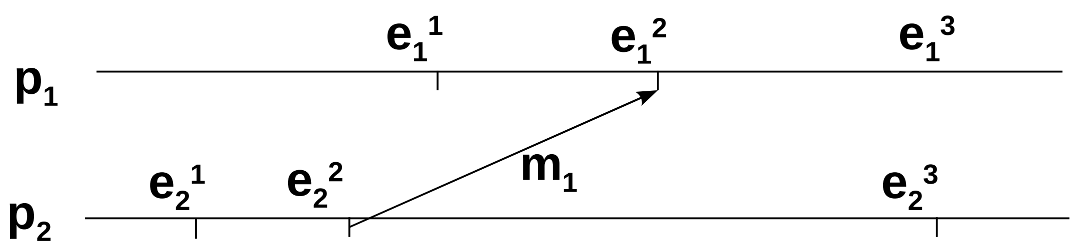


Causal Order

- Lamport (1978)
  - Causal order between two events is noted
    - $\mathbf{e} \rightarrow \mathbf{e'}$
  - It is defined as
    - e *happened-before* e'
  - In our execution model, we have $\mathbf{e} \rightarrow \mathbf{e'}$ if
    - e and e' happens in the same process and e happens before e'
    - e is the sending of a message *m* and e' is receiving that message
  - The causal relationship is transitive
    - If $\mathbf{e} \rightarrow \mathbf{e''}$ and $\mathbf{e''} \rightarrow \mathbf{e'}$ then $\mathbf{e} \rightarrow \mathbf{e'}$
  - Causal order is only a partial order
    - Not all events may be causally ordered

Causal Order

- Example
  - We have
    - $e_1^1 \rightarrow e_1^2 \rightarrow e_1^3$
    - $e_2^1 \rightarrow e_2^2 \rightarrow e_2^3$
    - $e_2^2 \rightarrow e_1^2$
  - Therefore we have
    - $e_2^2 \rightarrow e_1^3$
  - But we only have a ***partial order***
    - *We neither have* $e_1^1 \rightarrow e_2^1$ *or* $e_1^1 \rightarrow e_2^1$
    - *Noted as* $\boldsymbol{e_1^1 \parallel e_2^1}$

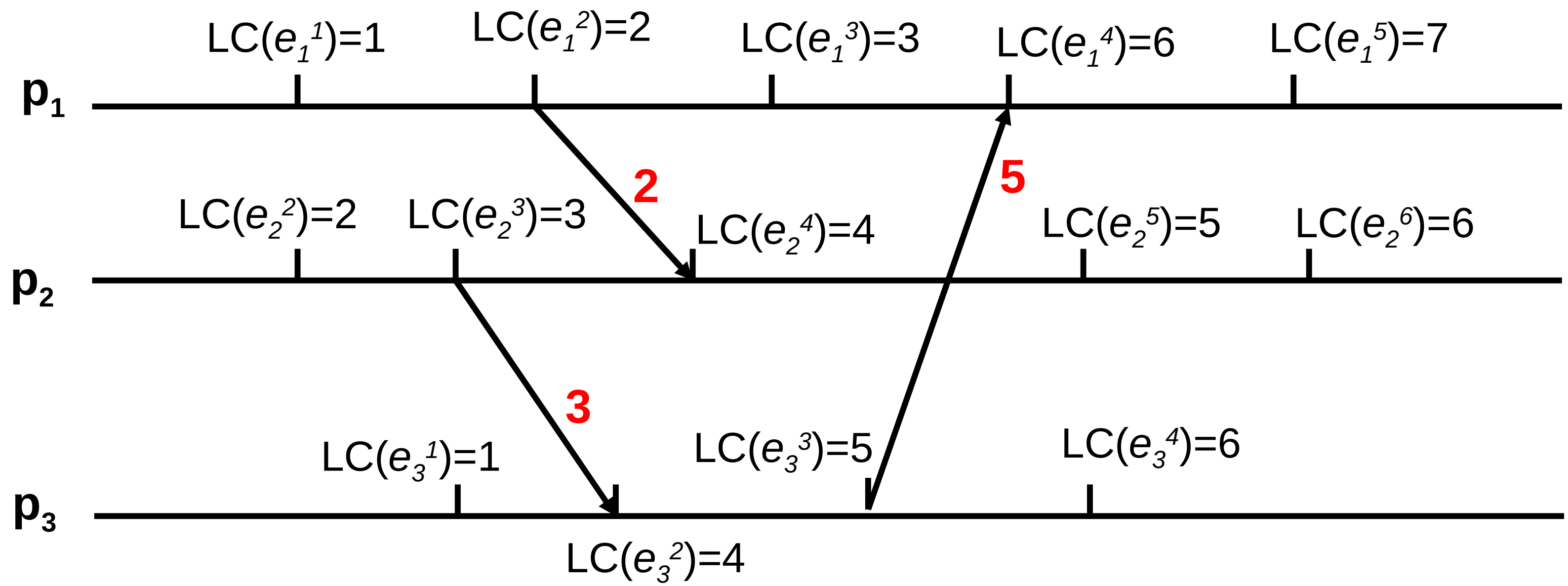


Logical Clocks

- Logical Clocks
  - Nothing to do with real time
  - Logical clock for an event e_i^k is noted **LC(e_i^k)**
  - Design
    - Logical clocks are maintained as local counters
    - For each new local event e_i^k: $LC(e_i^k) = LC(e_i^{k-1}) + 1$
- Regarding Messages
  - Sending a message M
    - This is a new local event e_i^k: $\mathbf{LC(e_i^k) = LC(e_i^{k-1}) + 1}$
    - **M is timestamped with LC(e_i^k)**
  - Receiving at P_j a message M(LC(e_i^k))
    - This is a new event e_j^r
    - $\mathbf{LC(e_j^r) = max(LC(e_j^{r-1}), LC(e_i^k)) + 1}$

Logical Clocks

Example



- By definition
  - $e_i^k \rightarrow e_j^r$ implies $LC(e_i^k) < LC(e_j^r)$
- Usage
  - $LC(e_i^k) < LC(e_j^r)$ implies $\neg(e_j^r \rightarrow e_i^k)$
  - That is $(e_i^k \rightarrow e_j^r)$ or $(e_i^k \parallel e_j^r)$

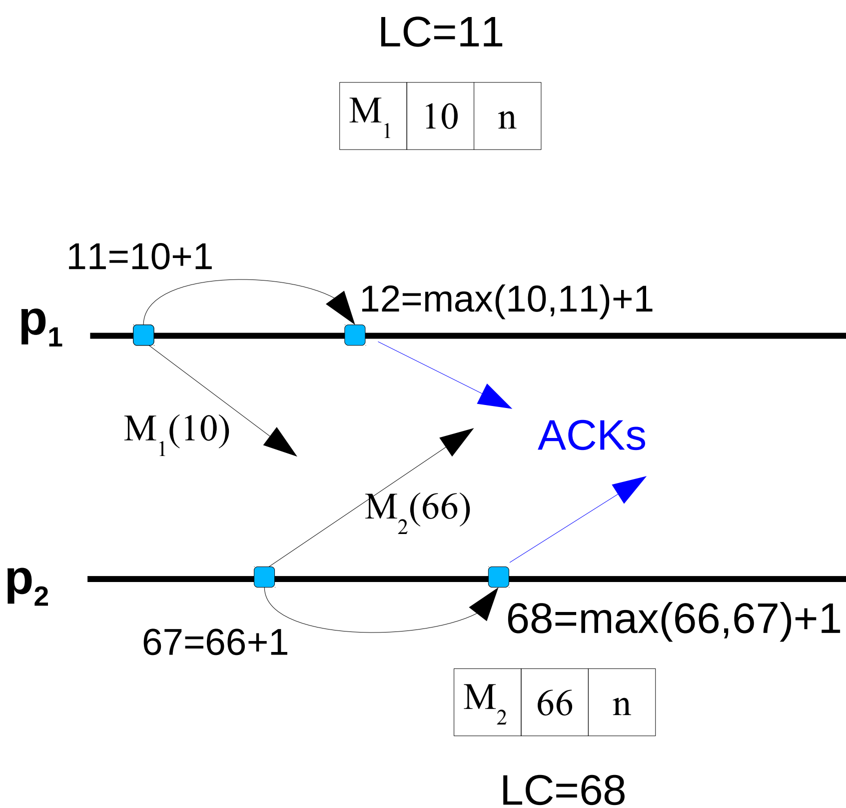
Look at $LC(e_3^1) < LC(e_2^3)$

It is a case where $(e_3^1 \parallel e_2^3)$

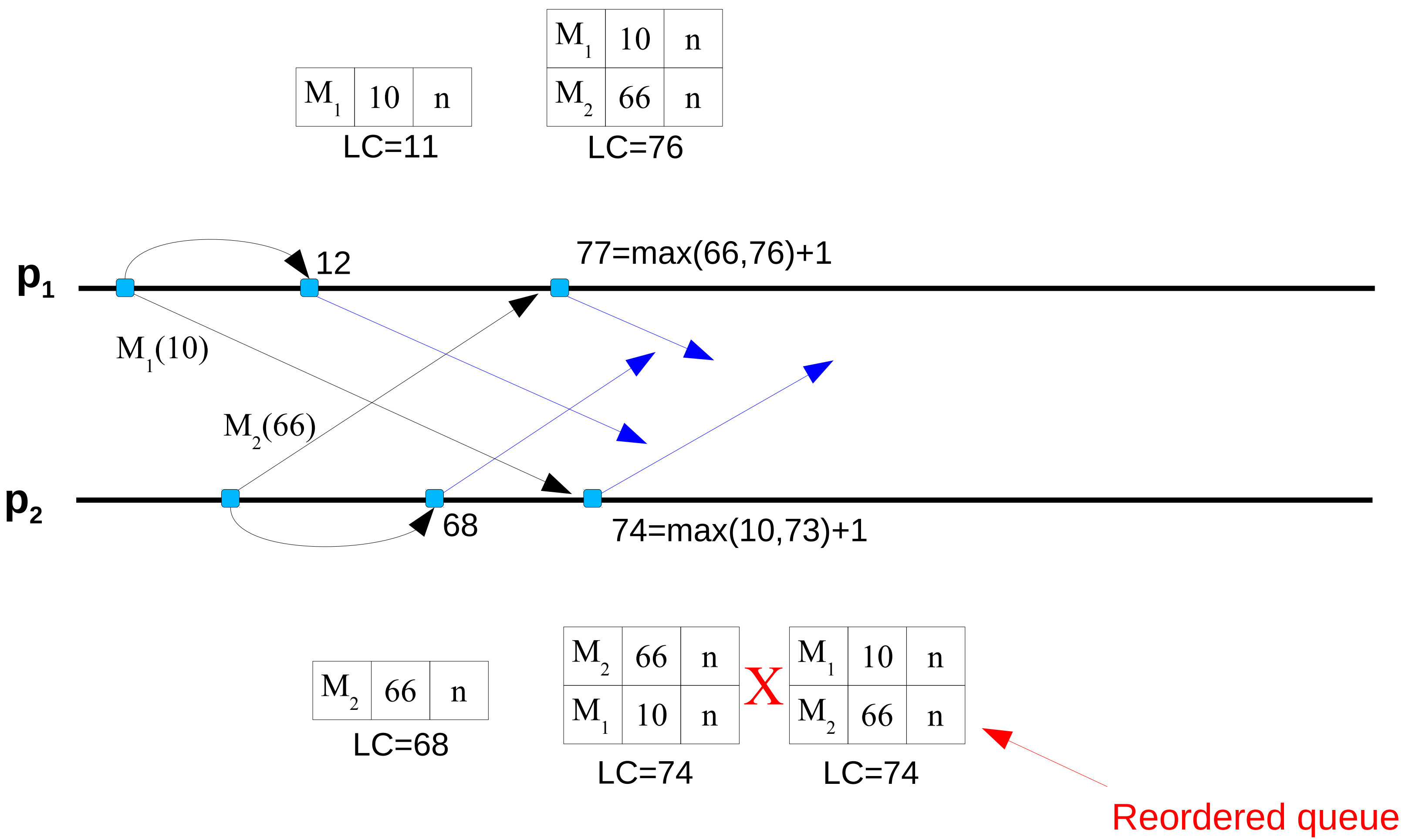
Totally Ordered Multicast

- Totally Ordered Multicast
  - Using Lamport's logical clocks
- Design
  - Between a group of N processes
    - They **must know each others** (concept of a group)
    - Each message from one process is **multicasted to the entire group**
    - We assume FIFO and loss-less communication channels
  - Each process:
    - Each message carries its normal timestamp (Lamport)
      - Build an ordered queue of messages based on the message timestamp
      - Acknowledge each message to the group (multicasted ack message)
    - Delivers a message only when
      - The message has been acknowledged by all other processes in the group
      - The message is at the top of the ordered queue

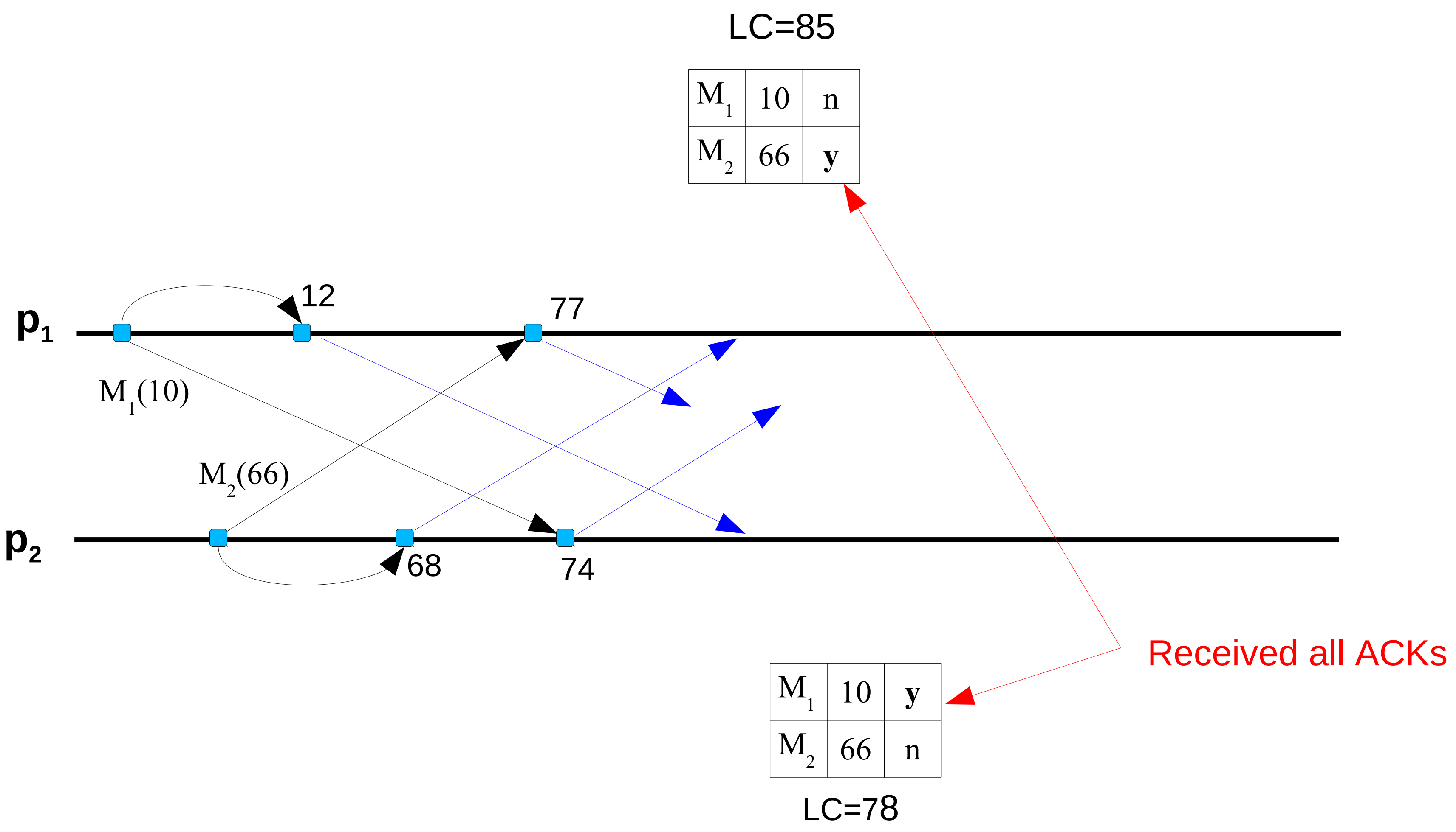
Totally Ordered Multicast



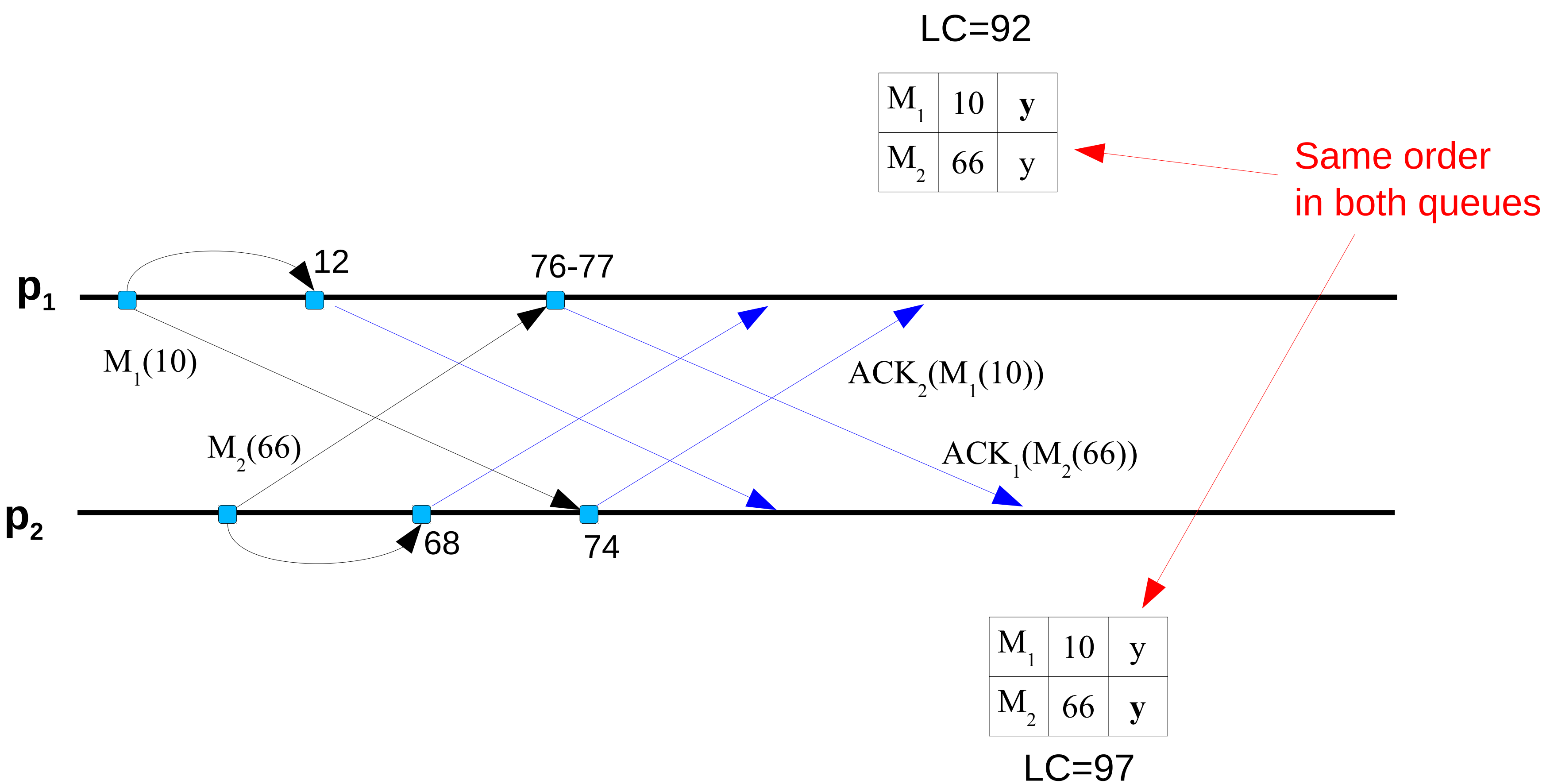
Totally Ordered Multicast



Totally Ordered Multicast



Totally Ordered Multicast



Totally Ordered Multicast

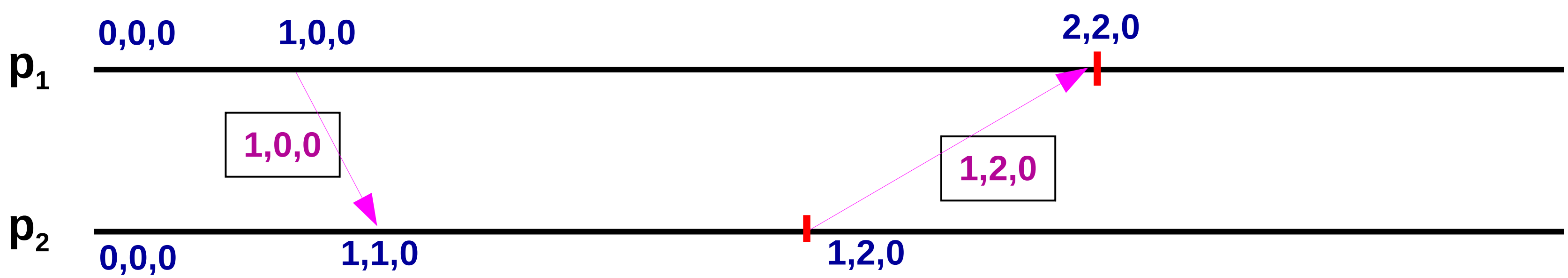
- Special Corner Case
  - Two multicast could have the same logical clock at two processes
  - Extends logical clocks with process identifiers,as decimals
    - When we had:
      - $LC(e_{32}^{k}) = 56$ and $LC(e_{24}^{k}) = 56$
    - We now have
      - $LC(e_{32}^{k}) = 56.32$ and $LC(e_{24}^{k}) = 56.24$
  - Use this extension any time you need a total order on logical clocks

Totally-Ordered vs Causally-Ordered Multicast

- The newsgroup example
  - We have a group, messages are multicasted
- Totally-ordered multicast
  - Everyone in the group sees all messages in the same order
- Causally-ordered multicast
  - Everyone sees the question first and answers next
  - Answers may not be seen in the same order by everyone
  - Questions asked in parallel can be seen in different orders too

Vector Clocks

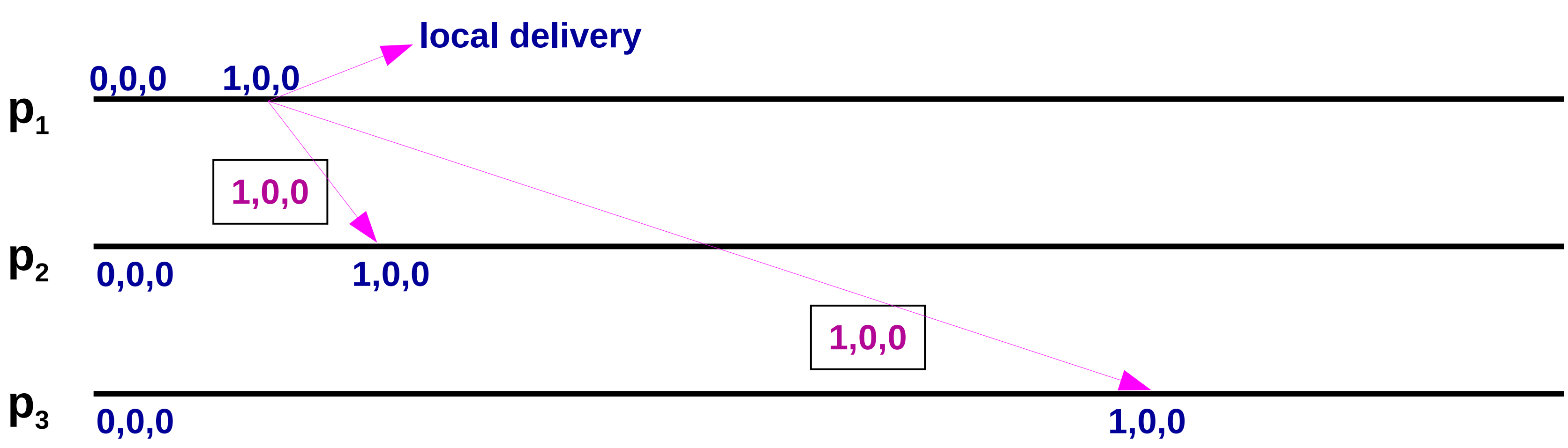
- Vector Clock (Fidge and Mattern, 1988)
  - A vector of logical clocks
    - One entry per known process P_i
    - VC[i] = max value of known LC(P_i)
  - Each event carries a vector clock
    - It gives the history at various processes that the event depends on
  - Each process P_i maintains a vector clock VC_i
    - Maintains the logical clocks that the current state of P_i depends on



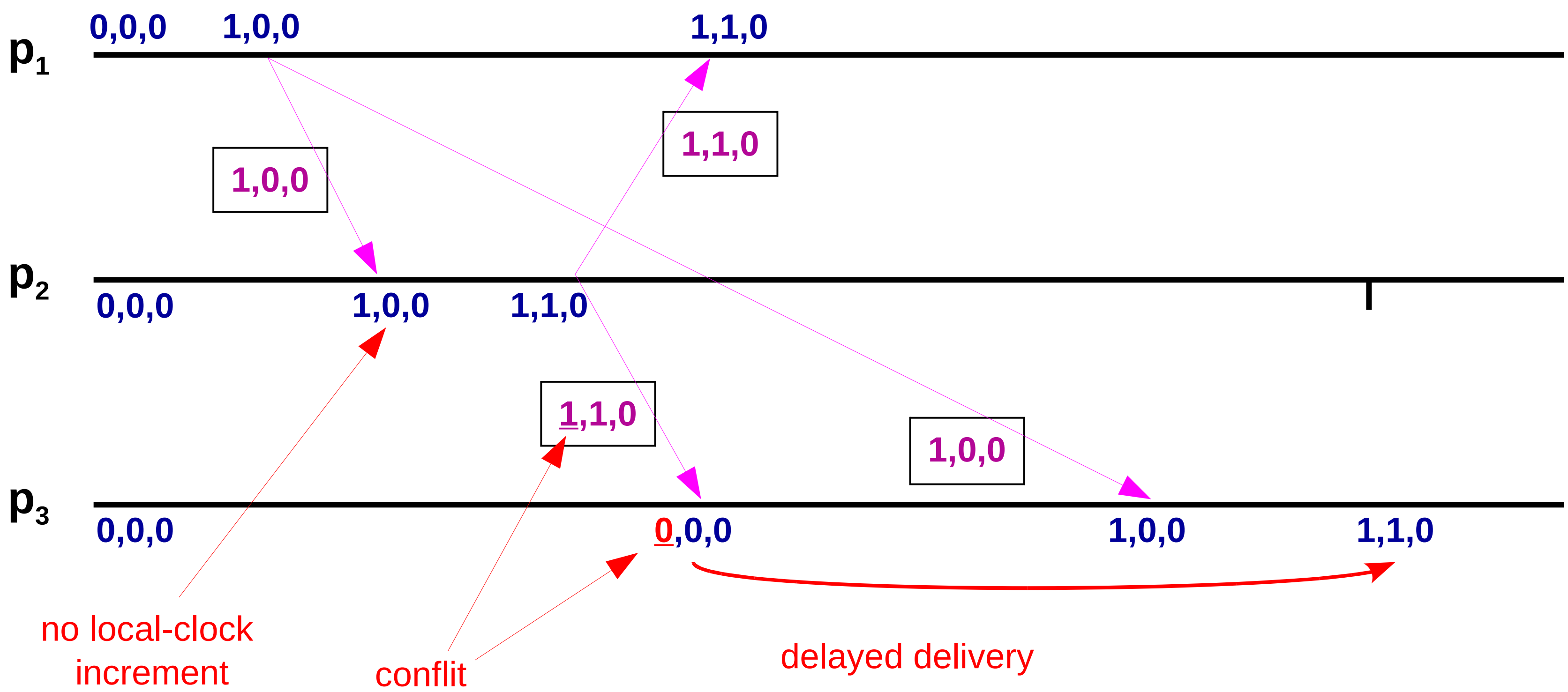
*Local state is now causally dependent on states (**1,0,0**)*

Causally-Ordered Multicast

- Causally Ordered Multicast
  - **Sending messages**
    - **Increment local logical clock only regarding multicasting (no other events)**
    - Timestamp messages with its VC_i
  - **Receiving messages** with a vector clock VC
    - $VC_i[k] = \max(VC_i[i], VC[k])$ for all $k \neq i$
    - **No increment of local logical clock**



Causally-Ordered Multicast



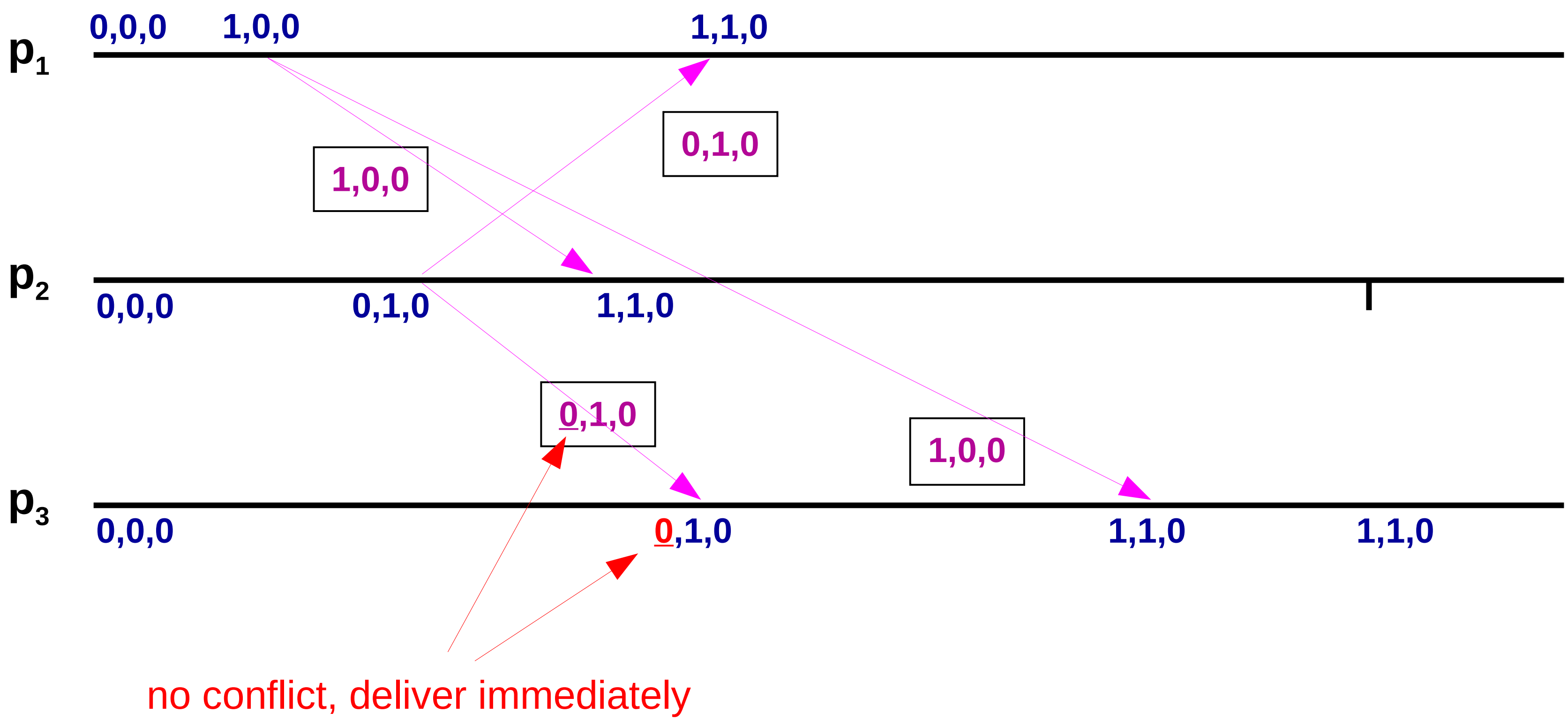
For a message M, received by P_r from P_s, with vector clock VC_m

Delay delivery until

$$VC_m[s] = VC_r[s]+1$$

$$VC_m[k] \leq VC_r[k] \text{ for all } k \neq s$$

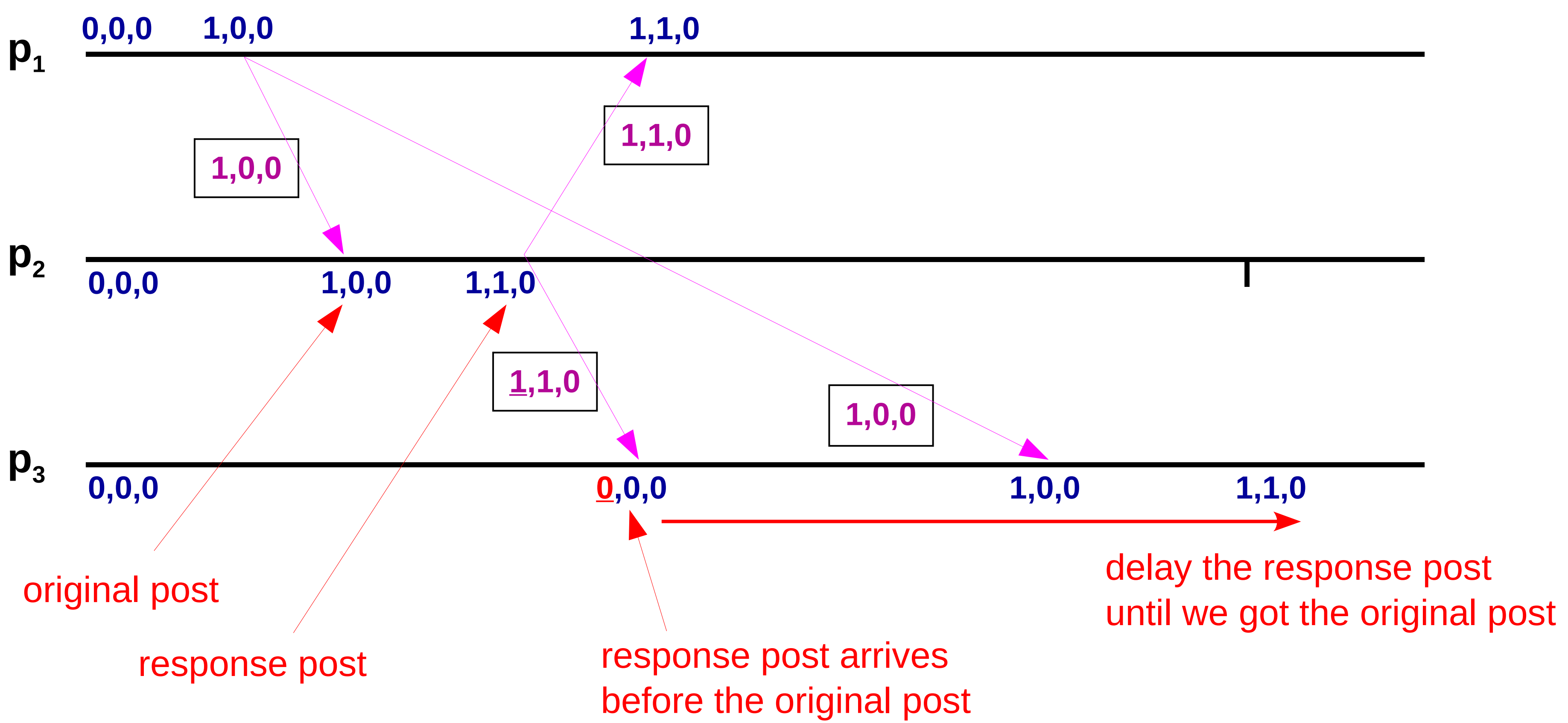
Causally-Ordered Multicast



Notice that
we avoided all the acknowledgment messages
of the totally-ordered multicast

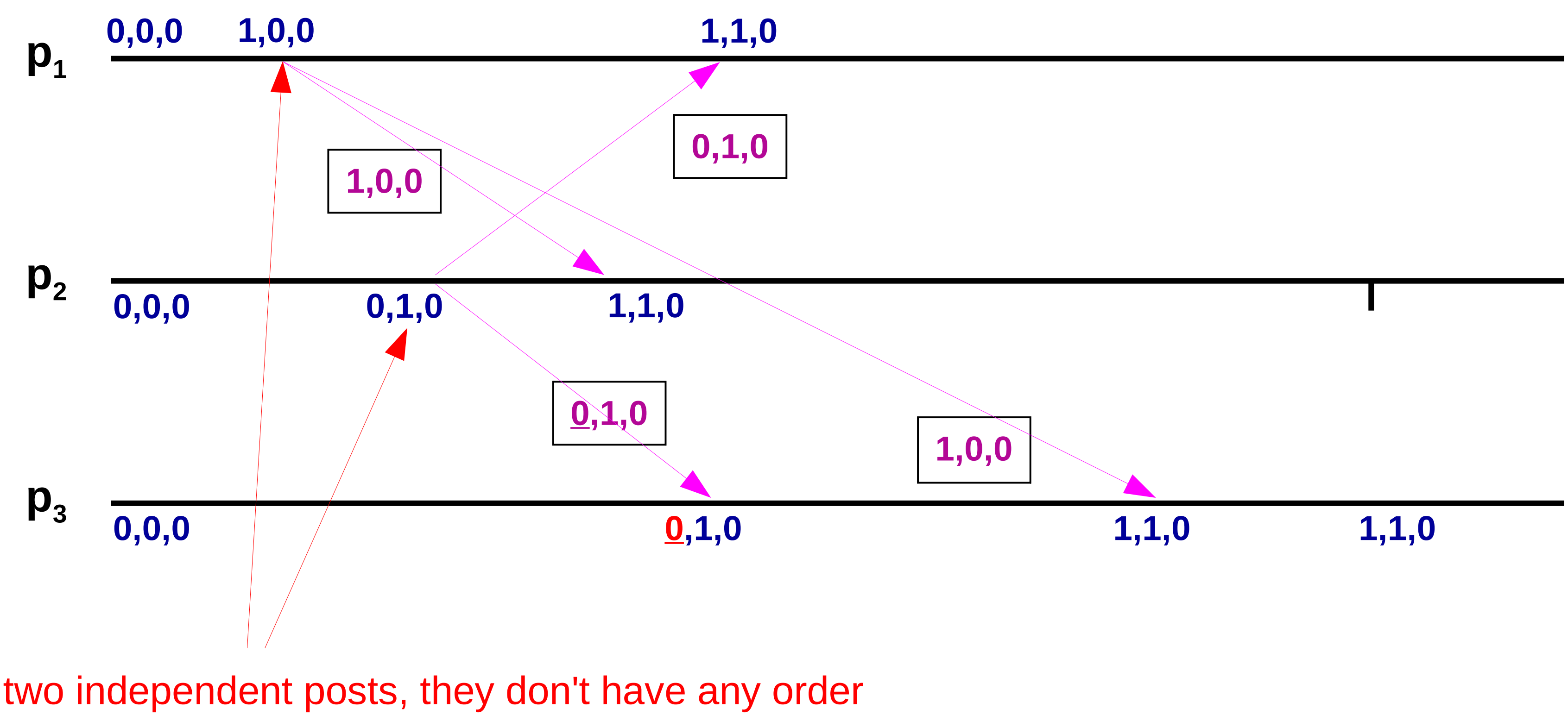
Causally-Ordered Multicast

- Example: newgroups
  - We want to avoid response posts to appear before the original posts



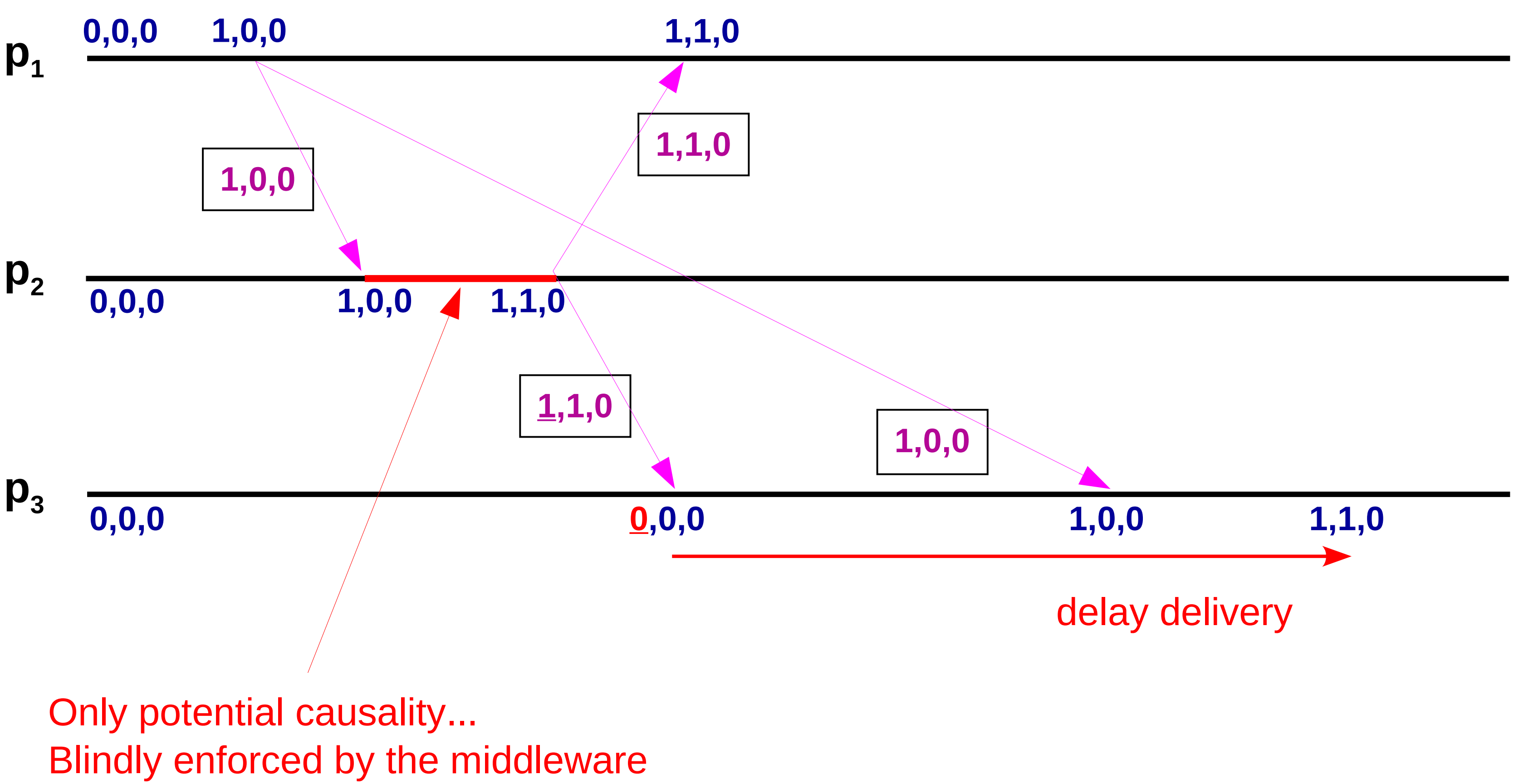
Causally-Ordered Multicast

- Example: newsgroup
  - But we don't need to order original posts...

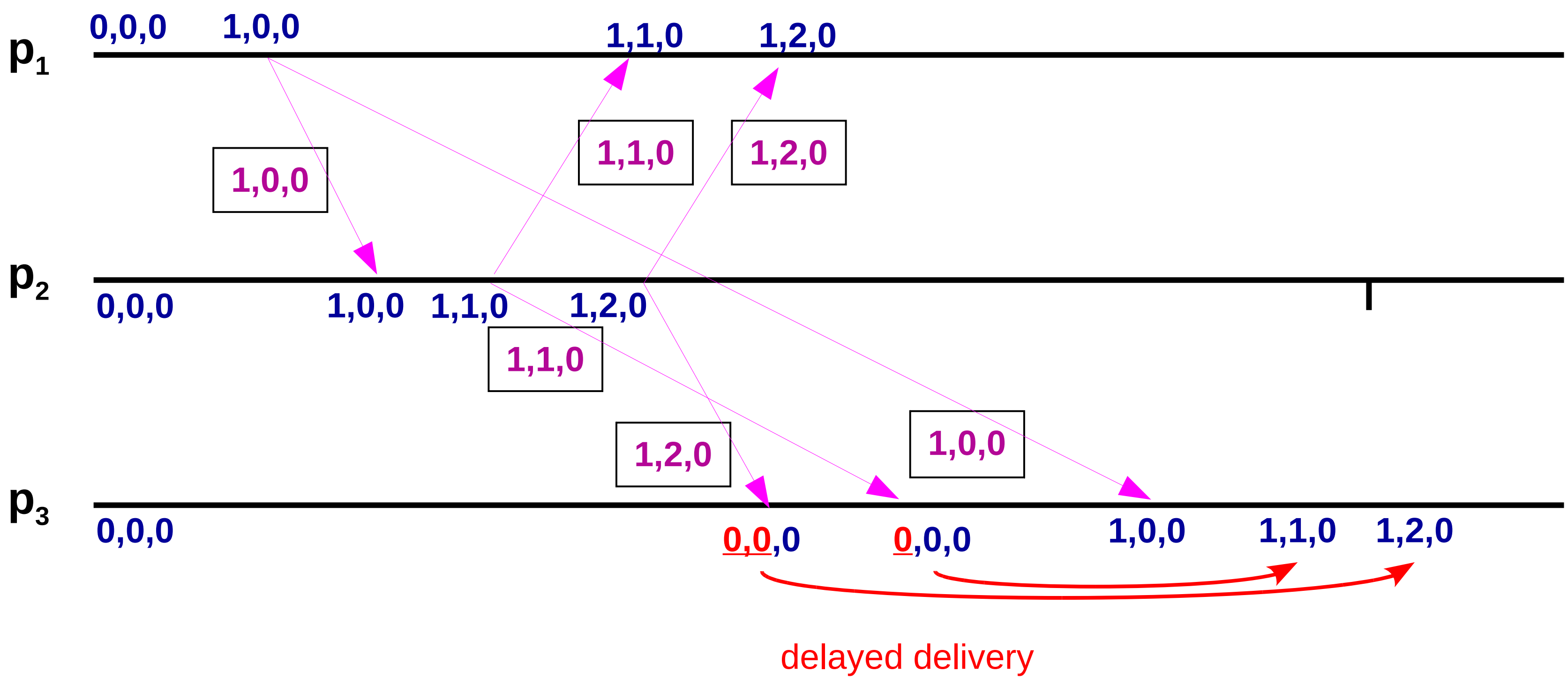


Causally-Ordered Multicast

- Example - newsgroups
  - Notice that we don't know for a fact if the message is a response or original post
  - Middleware is blind to application-level semantics



Causally Ordered Multicast



For a message M
Received by P_r from P_s with vector clock VC
Delay delivery until

$VC[s] = VC_r[s]+1$

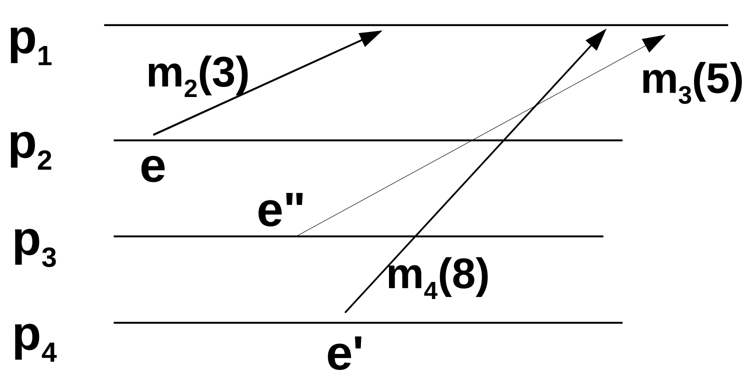
$VC[k] \leq VC_r[k]$ for all $k \neq s$

Point-to-Point Causality

- The Challenge of Point-to-Point Causality
  - When should we deliver **$m_4(8)$** ?
  - Do we have to wait for **$m_3(5)$**?
  - How do we detect missing or delayed events?
  - Undistinguishable situation from P_1 perspective

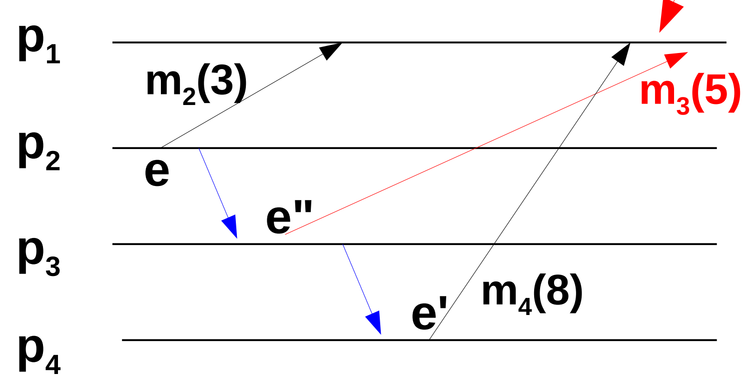
$$\boxed{\begin{aligned} &\textbf{send (m)} \rightarrow \textbf{send (m')} \\ &\Rightarrow \textbf{deliver (m)} \rightarrow \textbf{deliver (m')} \end{aligned}}$$

$(e_2 \parallel e''_3)$ and $(e''_3 \parallel e'_4)$



$(e_2 \rightarrow e''_3)$ and $(e''_3 \rightarrow e'_4)$

violates causality

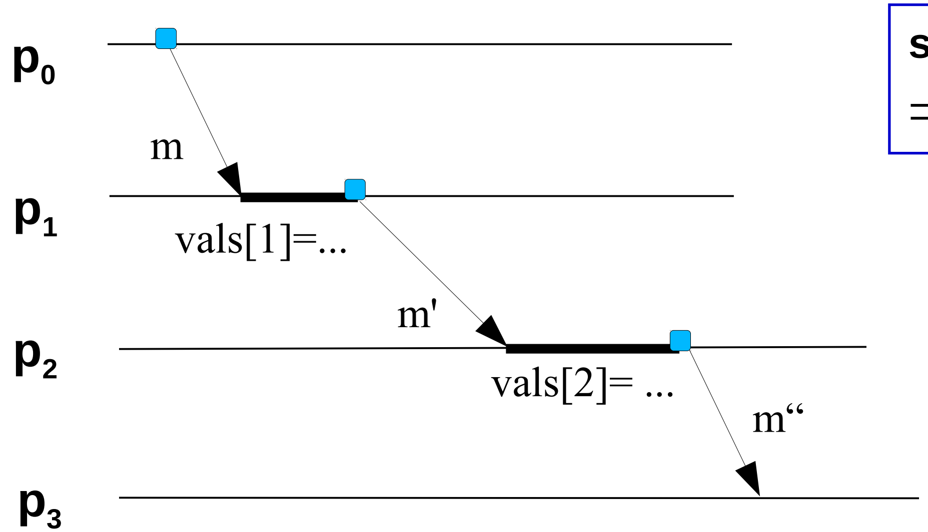


Example

- A simple loop:

```
int vals[]={0,1,2,3}
for (int i=1; i<vals.length;i++)
      vals[i] = vals[i] + vals[i-1];
```

Distributed values: vals[i] on processus Pi
Distributed the computation



send (m) → send (m') → send (m'')
⇒ deliver (m) → deliver (m') → deliver (m'')

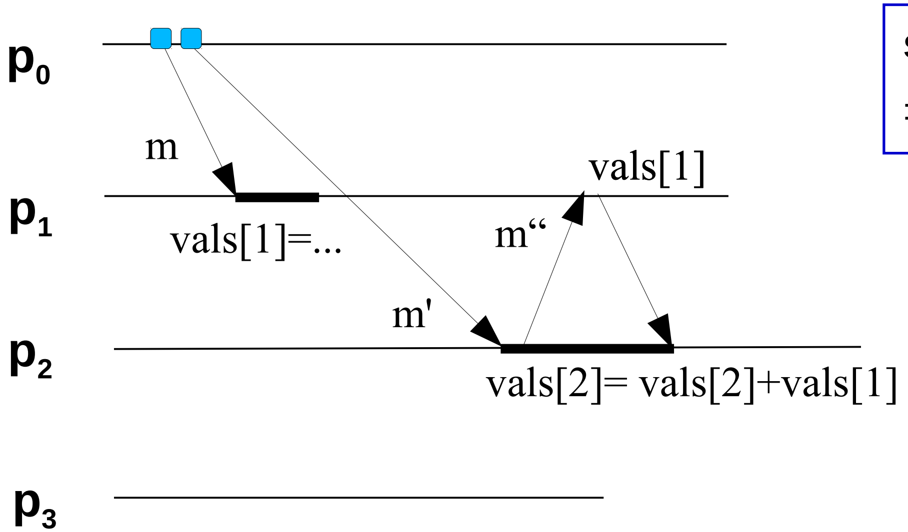
The simple and correct design...

Example

- A simple loop:

```
int vals[]={0,1,2,3}
for (int i=1; i<vals.length;i++)
      vals[i] = vals[i] + vals[i-1];
```

Distributed values: vals[i] on processus Pi
Distributed the computation



send (m) $\rightarrow$ send (m') $\rightarrow$ send (m'')
$\Rightarrow$ deliver (m) $\rightarrow$ deliver (m') $\rightarrow$ deliver (m'')

You must have point-to-point causality to be correct...

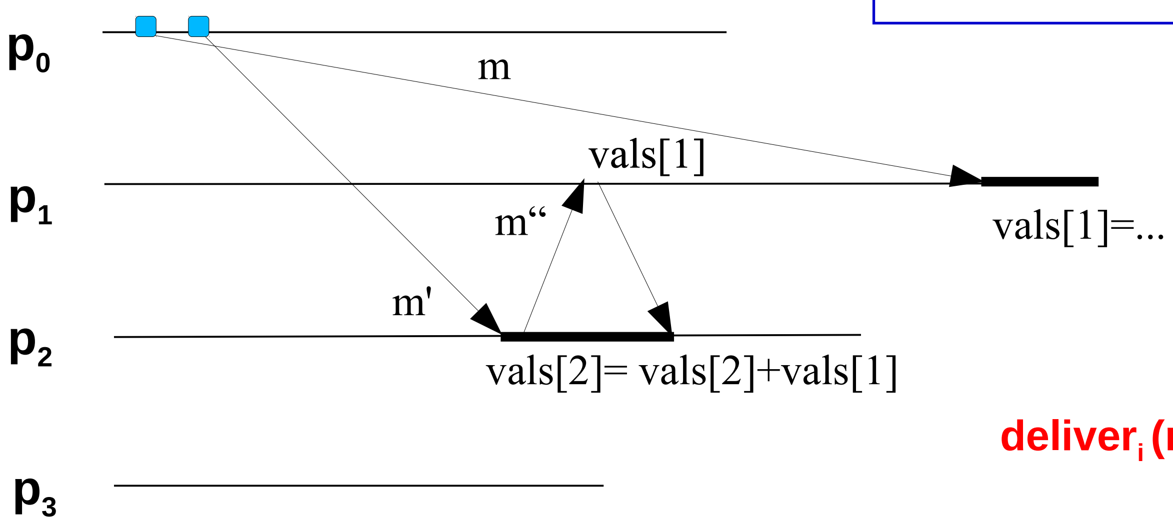
Example

- A simple loop:

```
int vals[]={0,1,2,3}
for (int i=1; i<vals.length;i++)
      vals[i] = vals[i] + vals[i-1];
```

Distributed values: vals[i] on processus Pi

send (m) → send (m') → send (m'')



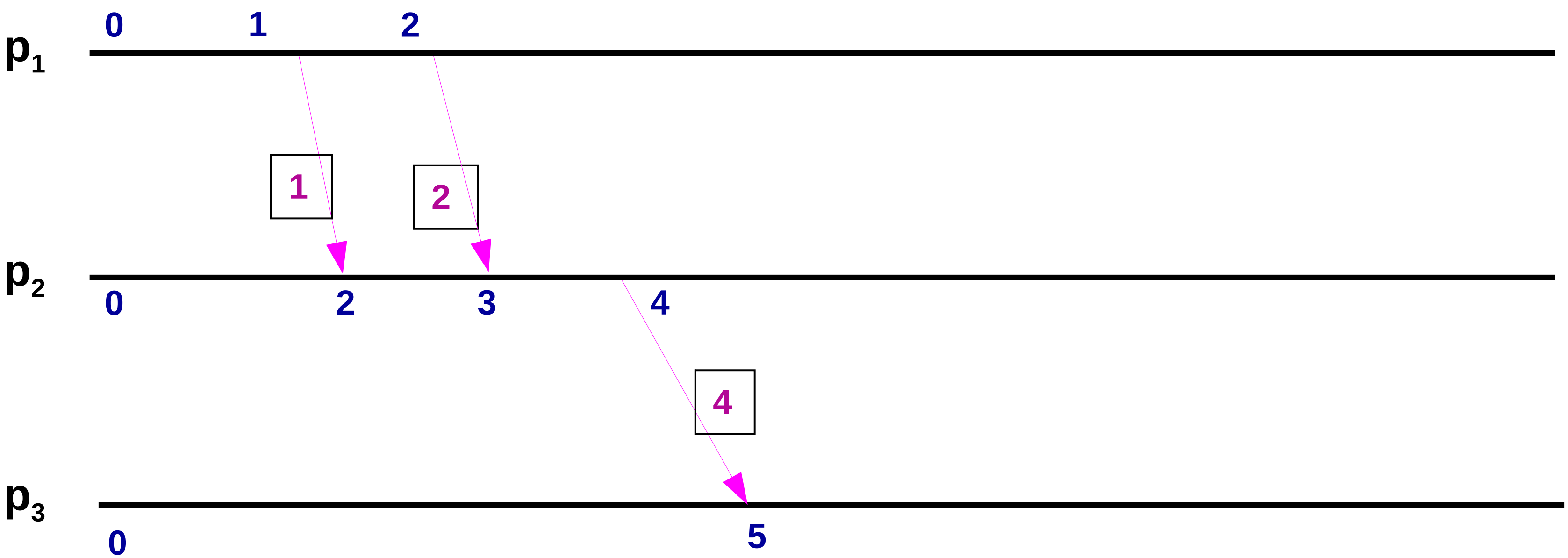
If not...

deliver$_i$ (m') → deliver$_i$ (m'') → deliver$_i$ (m)

⇒ incorrect execution!

Logical Clocks – Not Enough

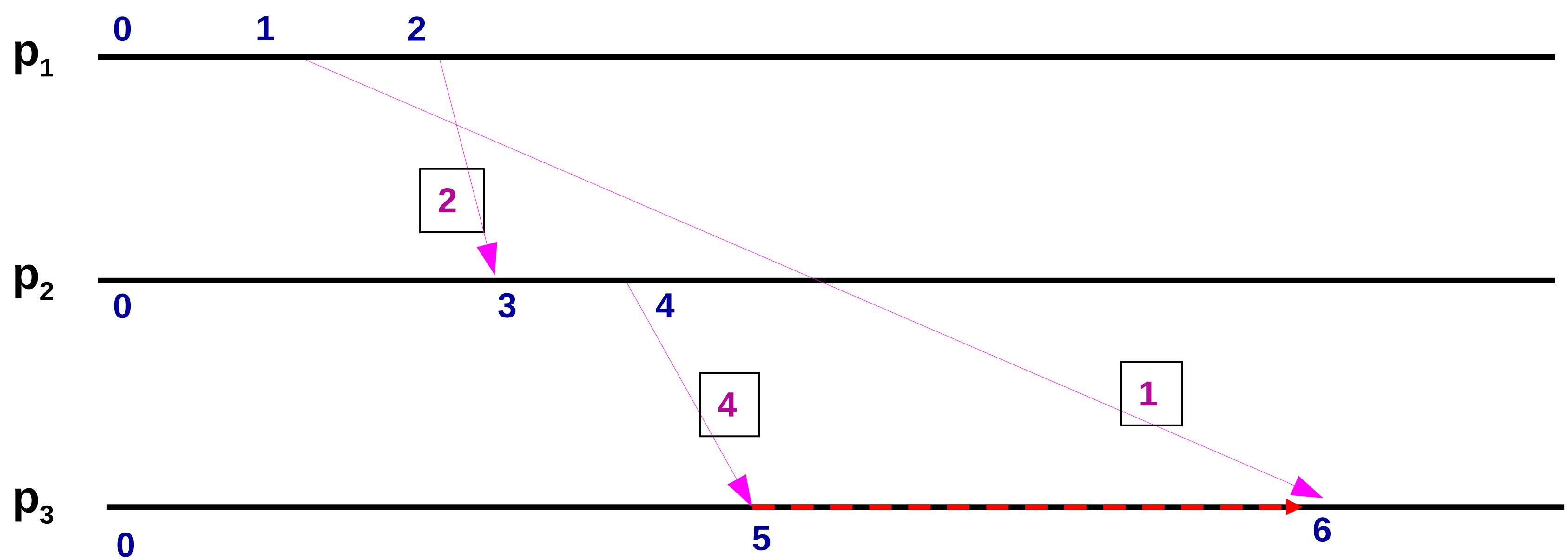
send (m) $\rightarrow$ send (m')
$\Rightarrow$ deliver$_i$ (m) $\rightarrow$ deliver$_i$ (m')



Does not violate causality

Logical Clocks – Not Enough

send (m) $\rightarrow$ send (m')
$\Rightarrow$ $deliver_i$ (m) $\rightarrow$ $deliver_i$ (m')



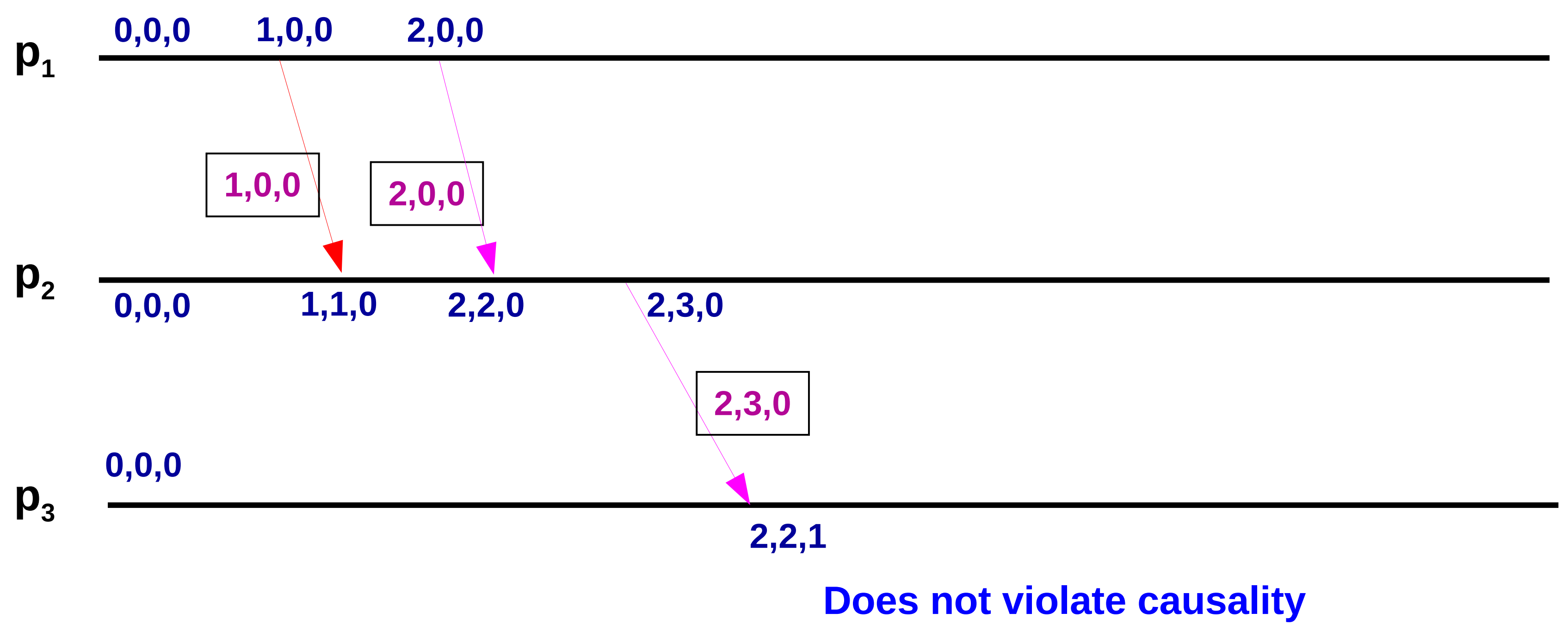
violates causality

the logical clocks do not carry any knowledge of late messages

NOT DISTINGUISHABLE FROM P3

Vector Clocks – Not Enough

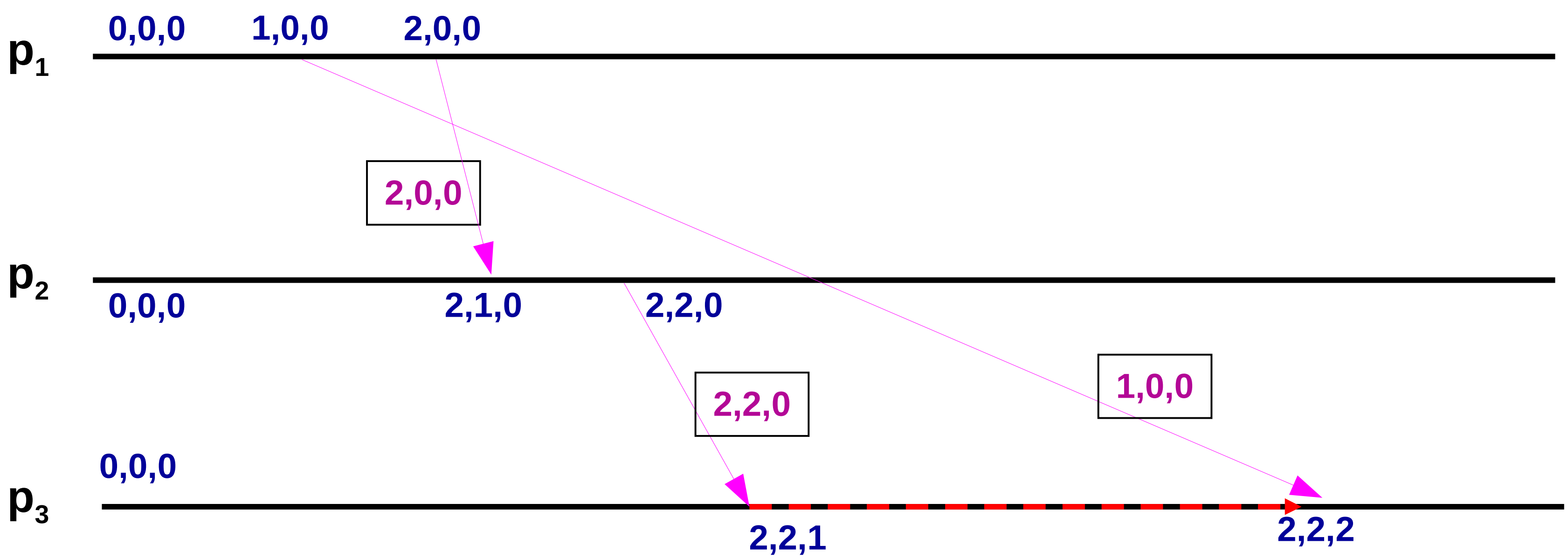
- Causal execution if P_1 sent the first message to another process than P_3
- Not distinguishable from P_3 perspective



Vector Clocks – Not Enough

send (m) $\rightarrow$ send (m')

$\Rightarrow$ deliver$_i$ (m) $\rightarrow$ deliver$_i$ (m')



violates causality

the vector clocks do not carry any knowledge of late messages

NOT DISTINGUISHABLE FROM P3

Matrix Clocks

- Towards a more complete history
  - Logical Clocks
    - LC_i = what P_i knows is just a number, used in a global order
  - Vector Clocks
    - $VC_i[j]$ = what P_i knows about P_j
  - Matrix Clocks
    - $MC_i[j, k]$ = what P_i knows about what P_j knows about P_k

Matrix Clocks

- Within a group of n process
  - Each process P_i maintains a matrix clock $MC_i[n,n]$
  - Each event e_i^k is timestamped with the matrix MC_i
  - Each message is timestamped with the matrix MC_i

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{array}$$

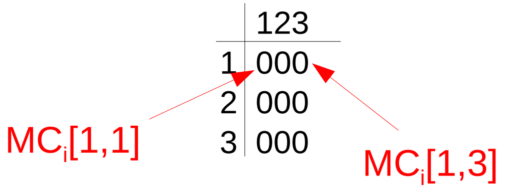
$MC_i[2,3]$

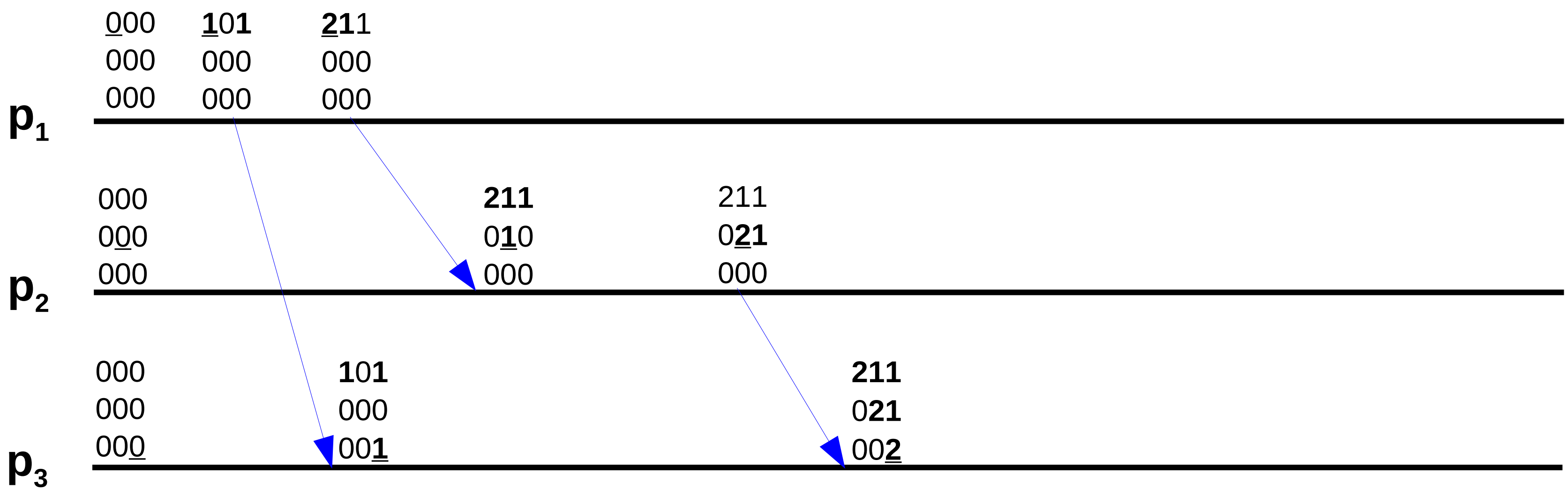
$MC_i[3,1]$

- Matrix definition
  - $MC_i[j,k]$ = number of messages sent by P_j to P_k that P_i causally knows about
    – A column k represents what a process P_k has received from other processes P_j that P_i knows about
  - $MC_i[i,i]$ = local events (local logical clock)

Matrix Clocks

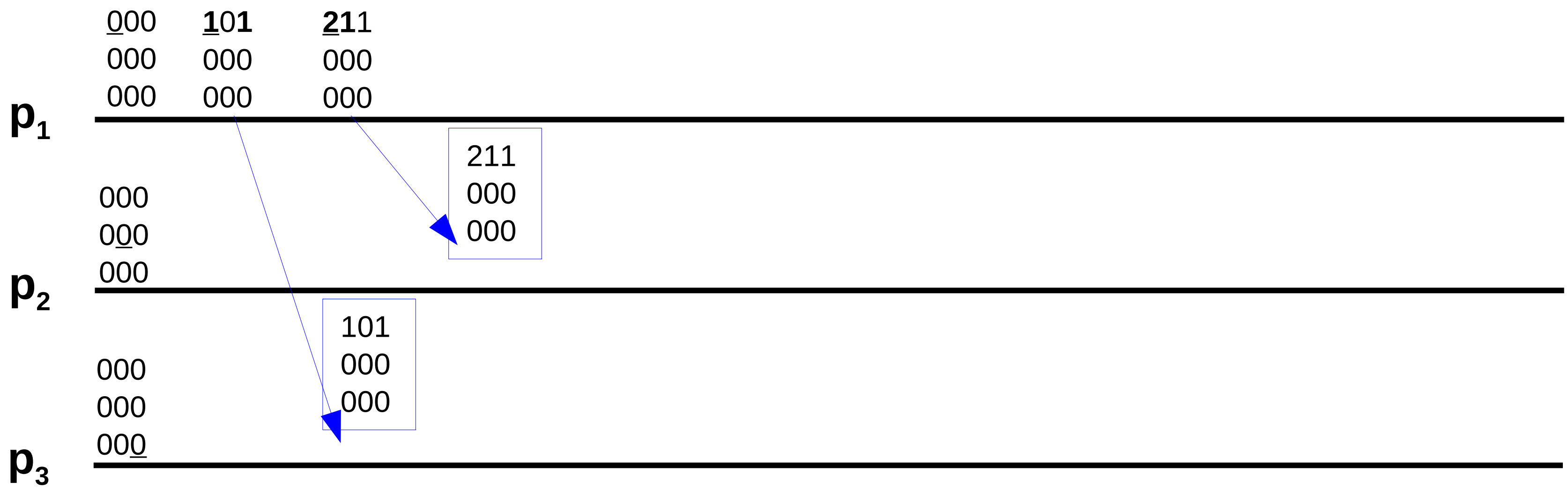
- Matrix definition
  - $MC_i[j,k]$ = number of messages sent by P_j to P_k that P_i causally knows about
  - $MC_i[i,i]$ = local events (local logical clock)





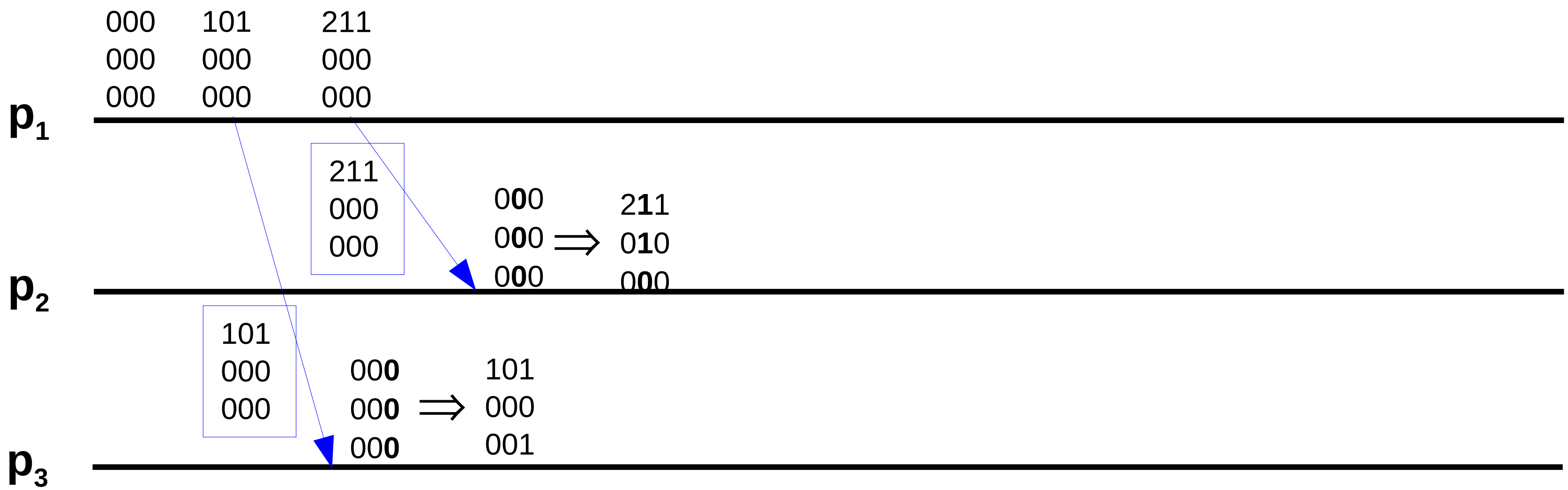
Matrix Clocks – Rules

- Local Event:
  - $MC_i[i,i] = MC_i[i,i] + 1$
- Sending a message from P_i towards P_k
  - $MC_i[i,k] = MC_i[i,k] + 1$
  - $MC_i[i,i] = MC_i[i,i] + 1$



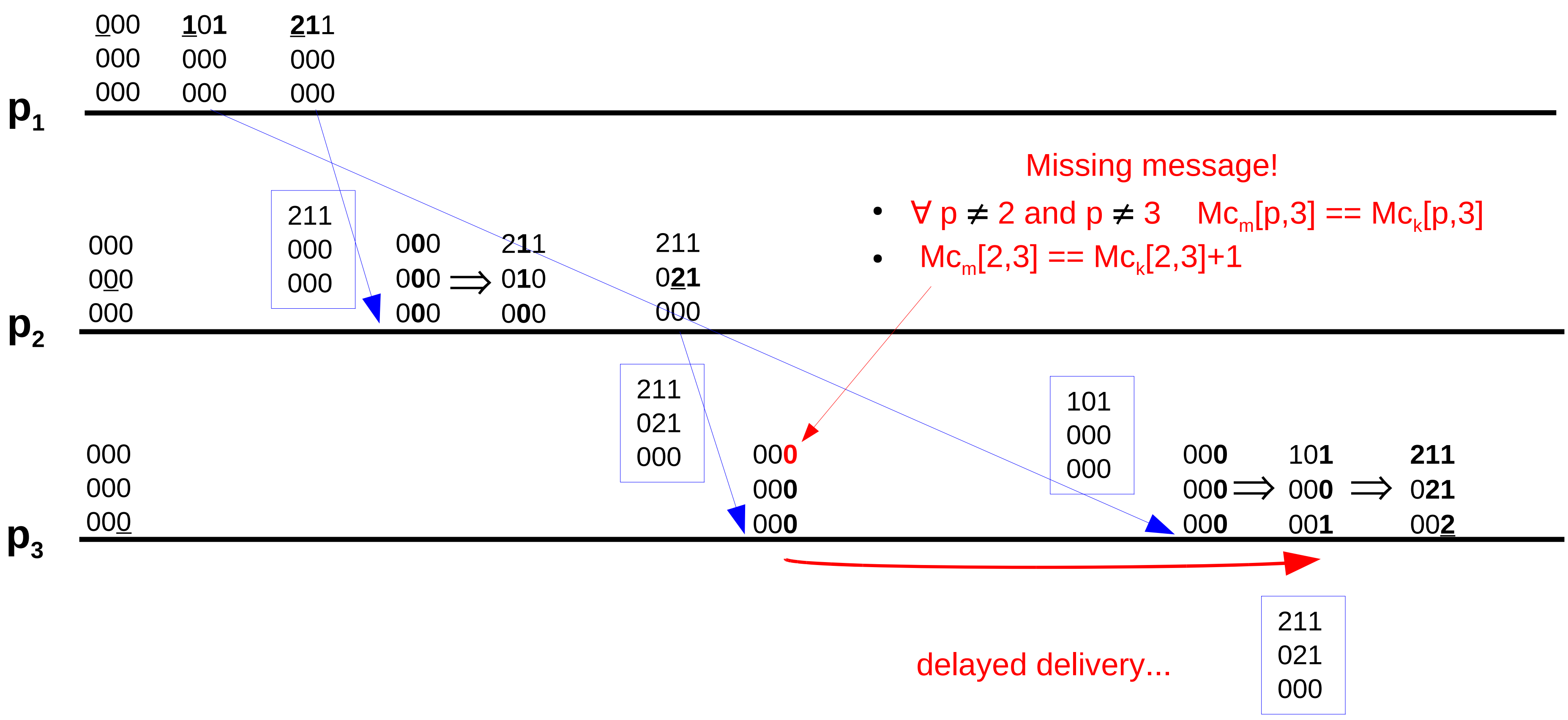
Matrix Clocks – Rules

- Delivery condition at P_k of a message from P_i timestamped with MC_m
  - $\forall$ p ≠ i and p ≠ k $Mc_m[p,k] == Mc_k[p,k]$
  - $Mc_m[i,k] == Mc_k[i,k]+1$ (FIFO order on channel from P_i to P_k)
- Receiving a message timestamped with MC_m from P_i at P_k
  - $MC_k[p,q] = \max(MC_k[p,q],MC_m[p,q])$ with p ≠ k (P_k knows best what it received)
  - $MC_k[k,k] = MC_k[k,k] + 1$ (increment local clock)



Matrix Clock

- Delivery condition at P_k of a message from P_i timestamped with MC_m
  - $\forall$ p ≠ i and p ≠ k $Mc_m[p,k] == Mc_k[p,k]$
  - $Mc_m[i,k] == Mc_k[i,k]+1$ (FIFO order on channel from P_i to P_k)
  -



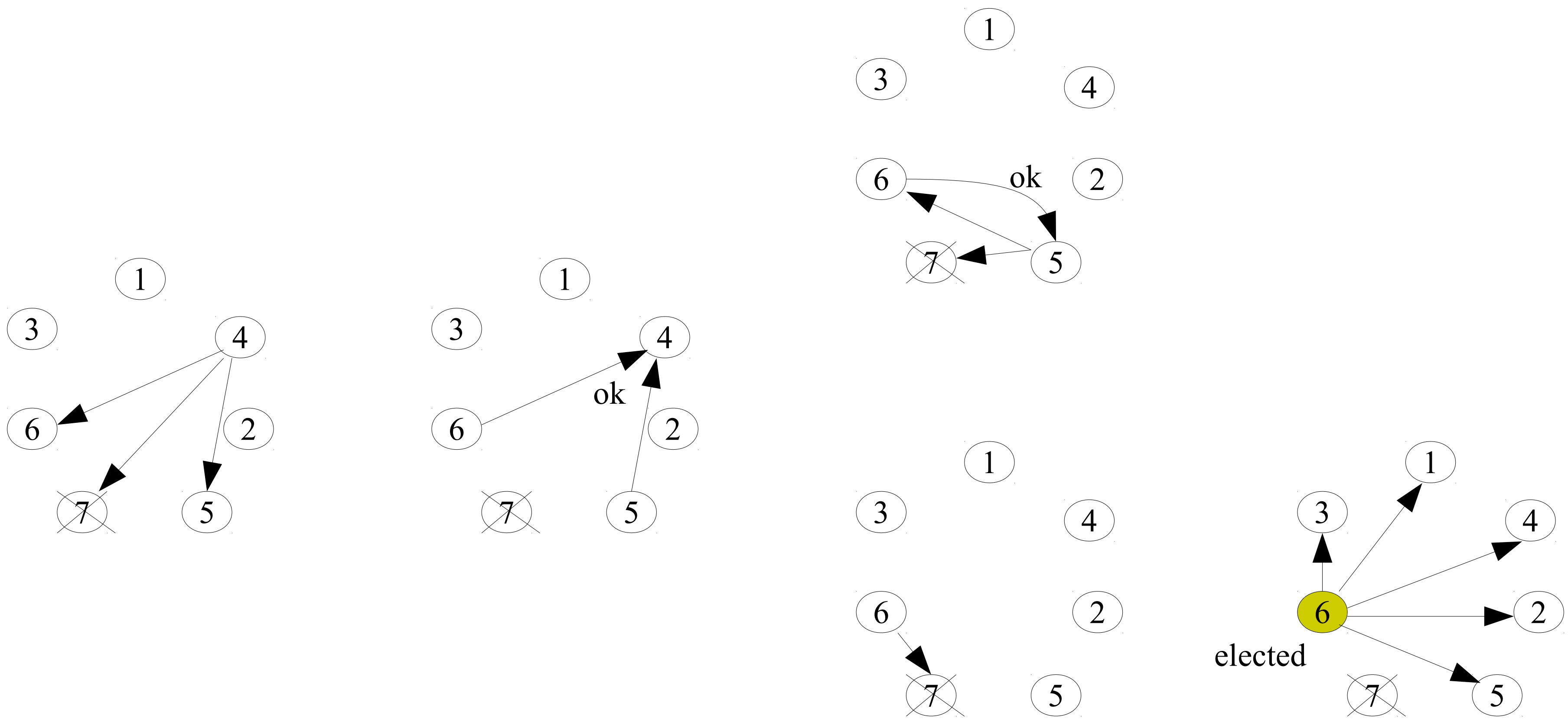
The Election Challenge

- Context
  - A distributed system with N processes
    - Processes know each others
      - The knowledge of the static group
    - A process does not know which process is running or down or failed
      - No knowledge of the dynamic group (currently correct processes)
    - Synchronous network (bounded delivery)
  - Elect cooperatively one process to perform a certain task
    - One process needs to be selected and only one
    - All processes need to agree on which process is elected
  - Necessary in many circumstances
    - Mutual exclusion coordinator (centralized algorithm)
    - Transaction commit (coordinator)
    - Data replication

Election Algorithms

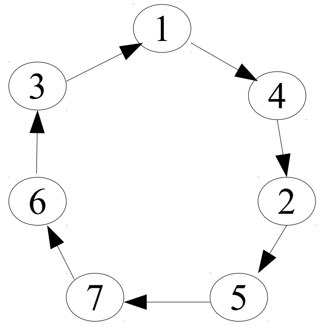
- Bully algorithm
  - Processes are all uniquely identified
  - There is a total order on process identifier
  - For example, machine IP and local creation time
- Simple design
  - Any process may initiate the election at any time
    - A process P sends an ELECTION message to all processes with higher identifiers
    - If no one responds, P wins the ELECTION
    - Notify all processes of the new elected coordinator (process P)
    - If one of the process responds, it takes over the election process
  - Upon receiving an ELECTION message
    - Returns an OK message to indicate that it is alive and takes over the election
    - If it is already holding an election process, just keep going
    - If it is not already holding an election process, apply the algorithm above

Bully Algorithm



Election Algorithms

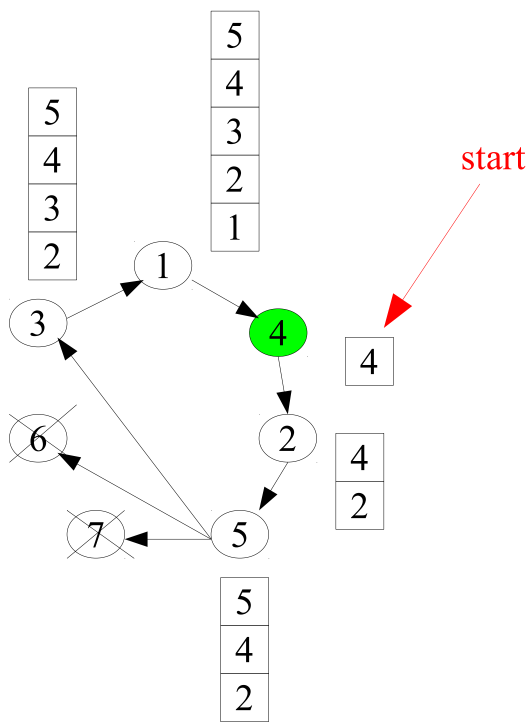
- A ring algorithm
  - N processes are organized as a ring overlay
  - Synchronous network, loss-less and FIFO

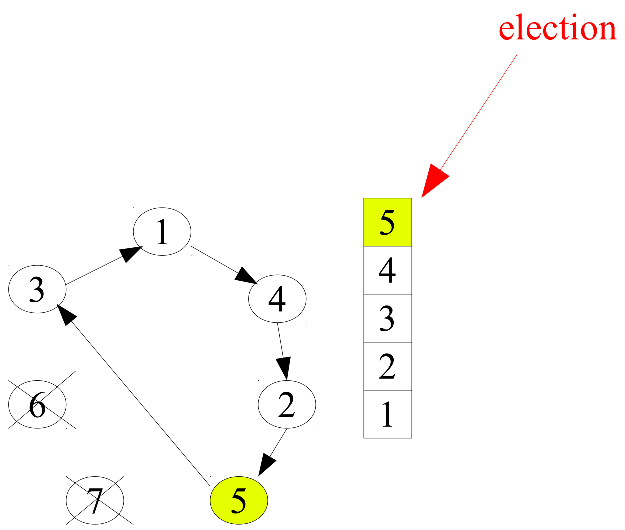


Election Algorithms

- A ring algorithm
  - Any process needing a coordinator
    - Creates an ELECTION message with its own identity
    - Sends a ELECTION message to the next node on the ring
      - Loops on the overlay until it finds one successor alive
      - If none are alive, it self-elects as a coordinator
  - Any process receiving an ELECTION message
    - Add its own identity to the message
    - Forwards the message to the next node on the ring
    - Loops on the overlay until it finds one successor alive
  - First loop is done
    - The ELECTION message comes back to the originator
      - Elects the process with the highest identifier as the coordinator
    - Circulate the COORDINATOR message notifying
      - Who the coordinator is
      - Who is in the overlay (removing failed processes)

Ring Algorithm



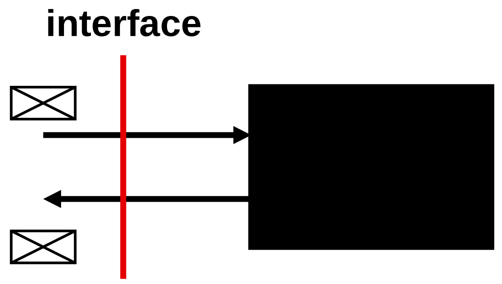


Discussing Failures

- Kinds of failures
  - Messages may be lost or delayed enormously
    - Impossible to detect the difference in practice
  - Processes may fail
    - Fail-stop
      - Works correctly or not at all
      - How do we differentiate between lost or delayed messages and failed process?
    - Partially fail (algorithm failure, boundary condition, etc.)
      - May accept message and make erroneous answers
- Impacts on previous algorithms
  - Totally-ordered multicast blocks
  - Causally-ordered multicast may partially block
  - Elections support fail-stop processes with a synchronous assumption
    - Synchronous assumption = known bound for message delivery

Definitions

- Failed System
  - *A system has failed when it does not behave according to its specification*
    - This is not a precise definition, it is system-dependent
    - This assumes that the specification is complete and correct
  - Black-box model
    - A distributed system is a collection of collaborating parts
      - Each part is considered a black-box from a failure model perspective
      - We will call each part a component
    - Failures are witnessed from outside
      - A component does not behave according to its specification
      - Example: it does not reply to messages



Fault Tolerance

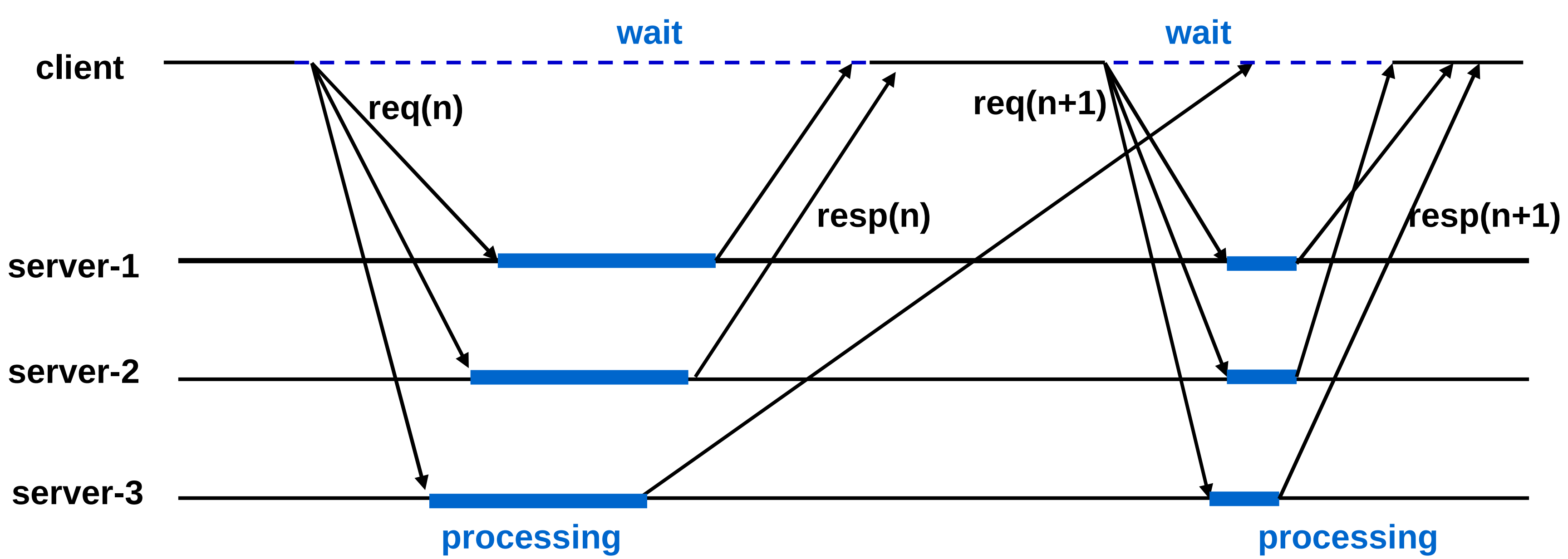
- Fault masking
  - Faults are transparently recovered
    - Enough redundancy and error checking
    - Done real low in the architecture, often in hardware or in drivers
  - Example:
    - Memory parity errors and checksum recovery
    - Redundant processing units and majority vote
    - RAID disks
- Fault recovery
  - Faults do happen and software components do fail
  - To ensure good performance and long-term operation
    - Failures must be detected
    - Failures must be recovered from
  - Classical approach
    - **Fail-stop, repair, and reinsert**

Replicated Servers

- Goal
  - High-availability servers, wanting to resist server failures
- Architecture
  - For clients
    - The model must be equivalent to a centralized server
  - Replicated servers
    - N servers resist up to **N-1 concurrent failures**
    - Failed servers are **repaired and re-inserted**
    - **Assume fail-stop servers**
  - Two models
    - Primary-based replication
    - Active replication

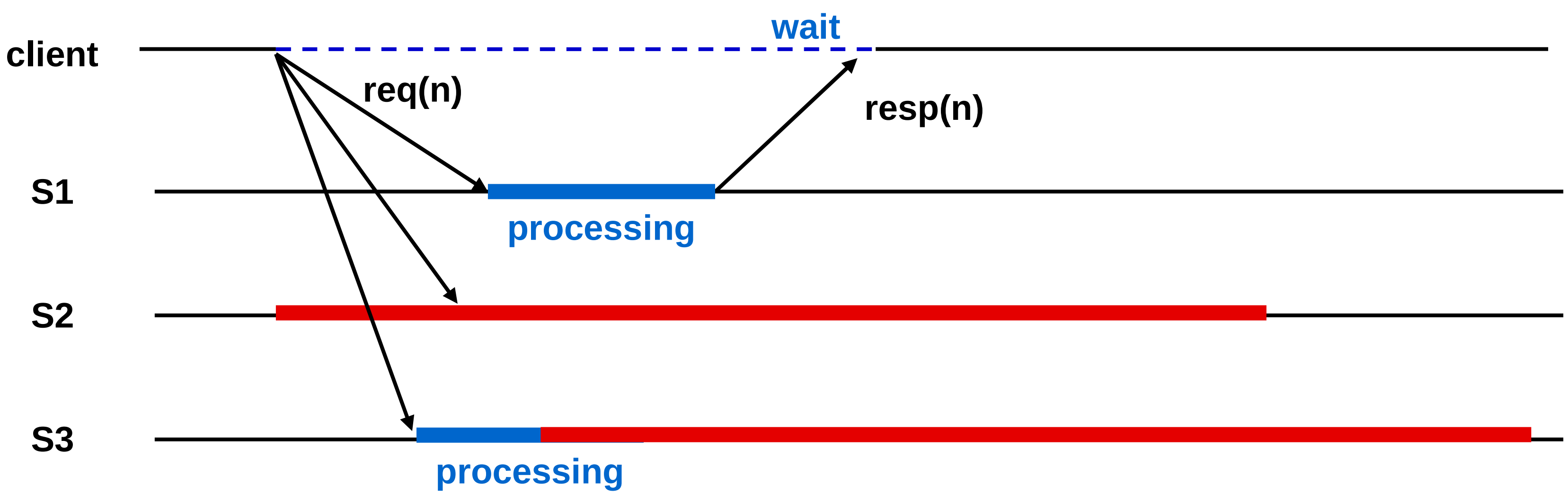
Active Replication

- Each client sends its requests to all servers in parallel
  - Each request has a sequence number (local for each client)
  - For each request, the client waits for the first answer, drops the following ones
- All servers are equal
  - They all process requests, only works with deterministic requests
  - They all possess a copy of the data, all requests must be totally-ordered across servers



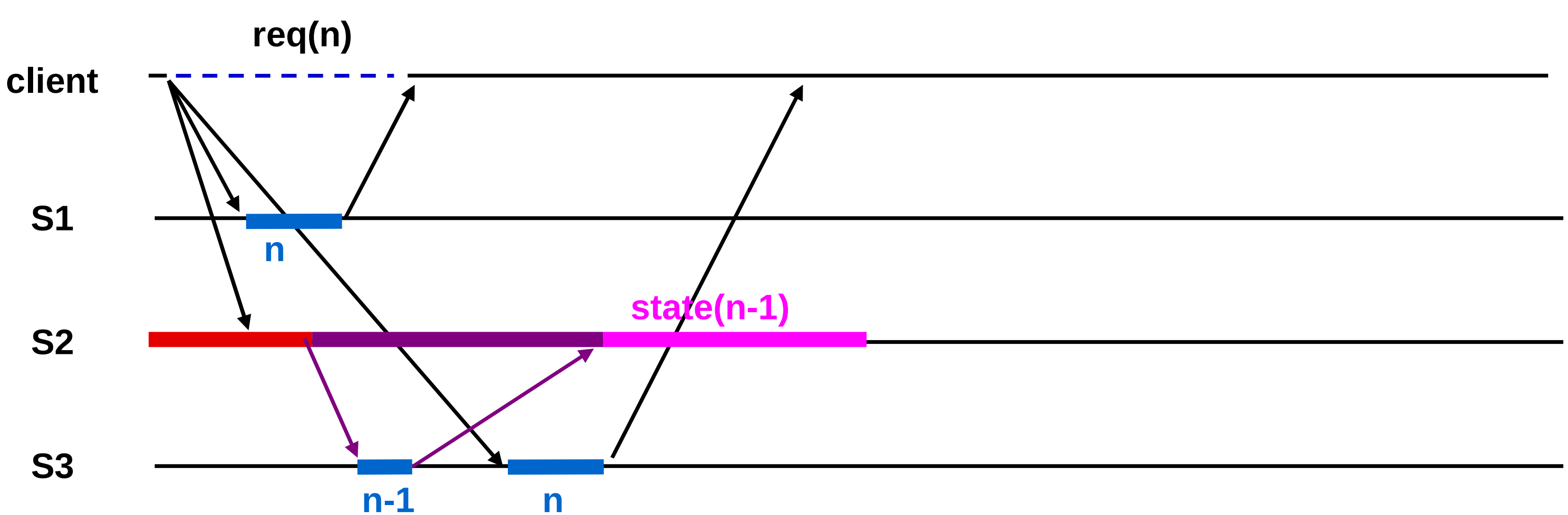
Active Replication

- Fault-tolerance
  - Clients need to receive at least one answer (requires at least one correct server)
    - Consider FIFO and lossless communications between clients and servers
  - Requires fail-stop servers
    - Do not send erroneous answers
  - Repair and reinsert failed servers
    - Required to preserve long-term fault-tolerance



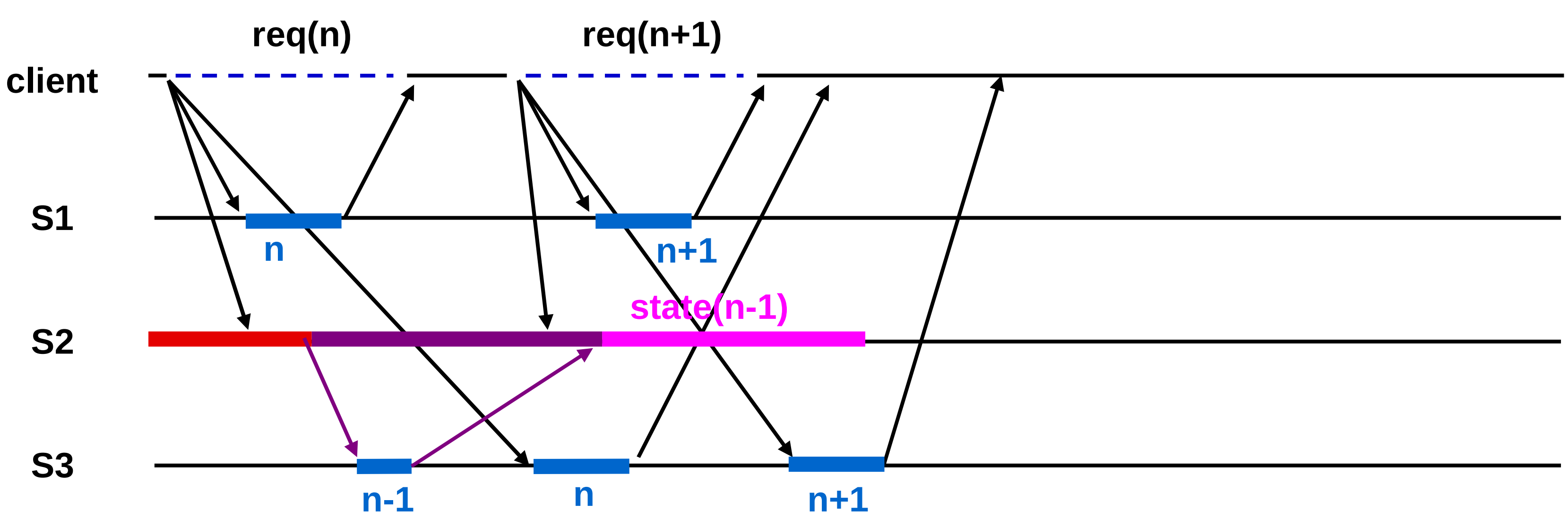
Active Replication

- Repair and reinsert failed servers
  - Detect failures... false-positive may happen
- Recover the state, if lost or corrupted
  - Requests it from another server
- Assert the state level
  - For each client, it will be up to a certain request-id



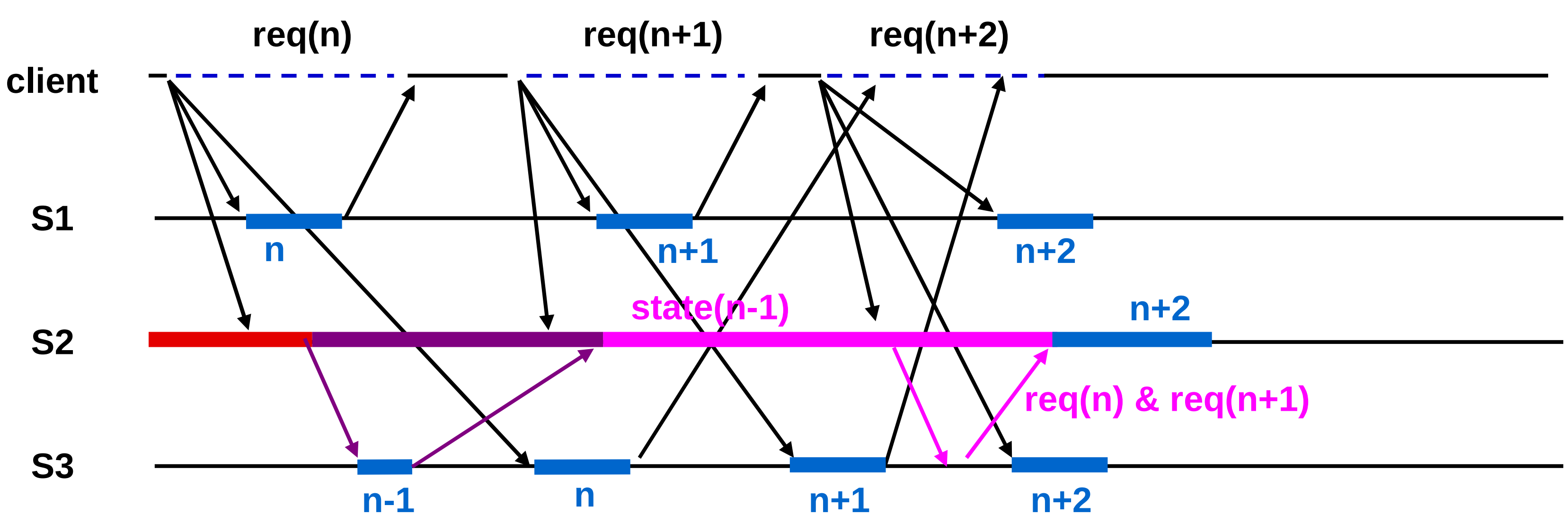
Active Replication

- Repair and reinsert failed servers
  - We lost all requests up to n, but we don't know it
  - We acquired state(n-1)
  - While acquiring state, we lost req(n+1)



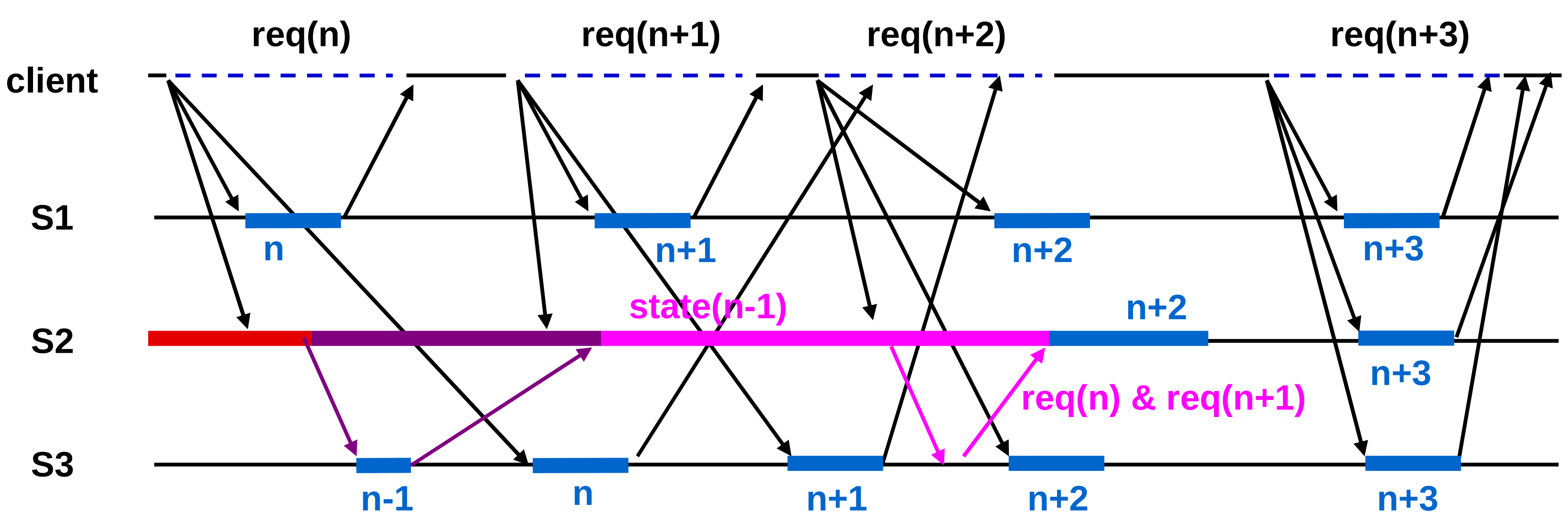
Active Replication

- Repair and reinsert failed servers
  - Having state(n-1), we can't process req(n+2)
    - But we now know which requests we missed: req(n+1) and req(n+2)
  - We request these missed requests from S3
    - We process them on state(n-1)
    - We are up to state(n+2) after that processing



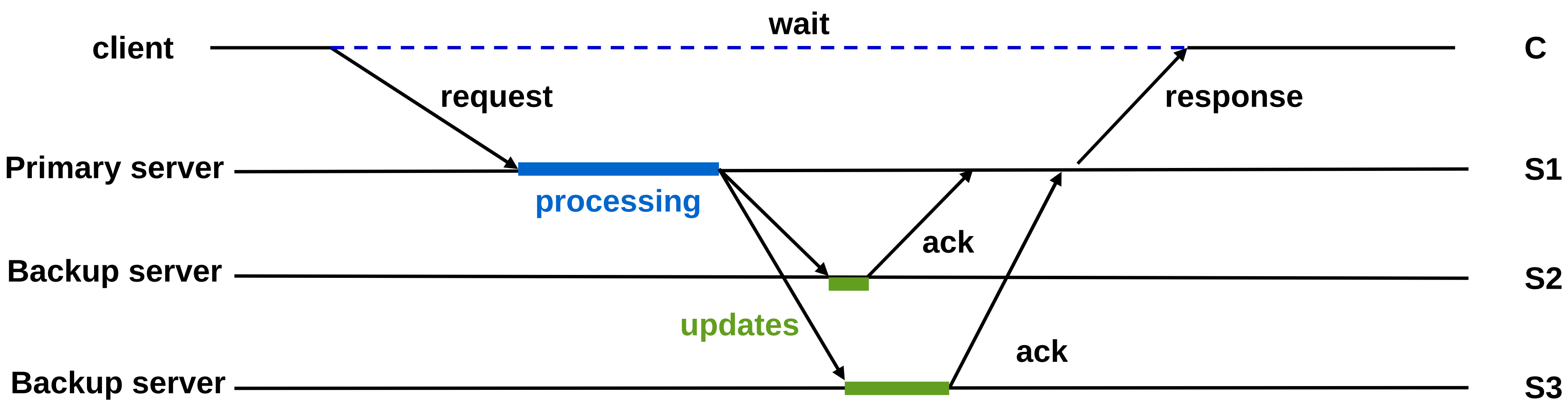
Active Replication

- Repair and reinsert failed servers
  - Back to normal...
    - We receive req(n+3), we have state(n+2)
  - **WARNING: there can be multiple clients...**
    - So we manage vectors of sequence numbers from clients
    - So we need to totally order the requests on replicas
    - We are only back to normal when we have received all request logs that we missed

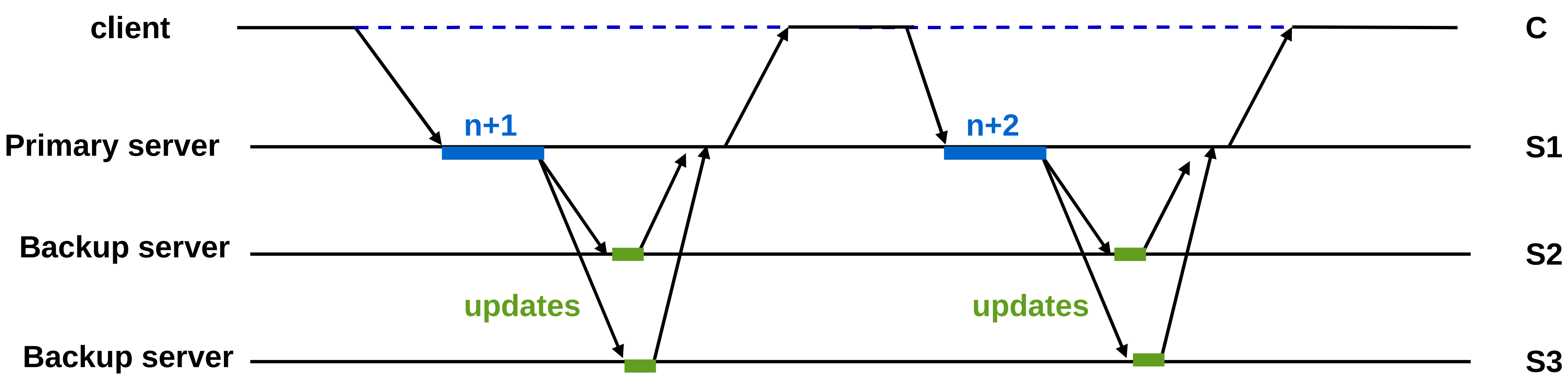


Replicated Servers

- Primary-base Replication
  - One server is the primary, the others are backups
    - The primary executes the client requests
    - It updates locally one or more data items (x, y, ... , z)
    - Updated data items (x,y,...,z) are replicated on backup servers
  - Principle
    - Primary waits for all acknowledgements from replicas
    - All replicas (backup servers) are in the same state

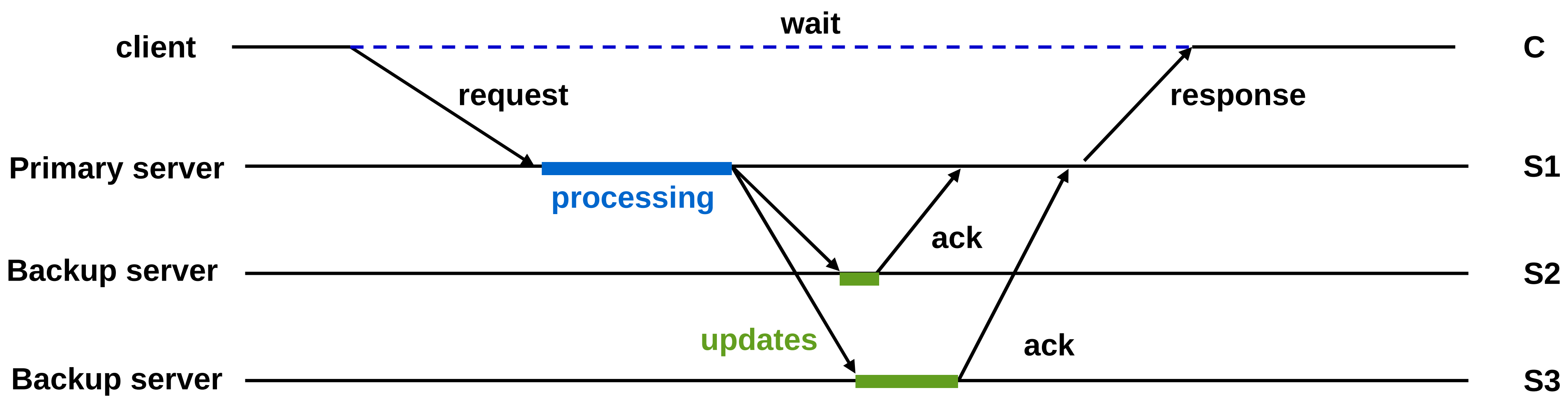


Primary-Based Replication



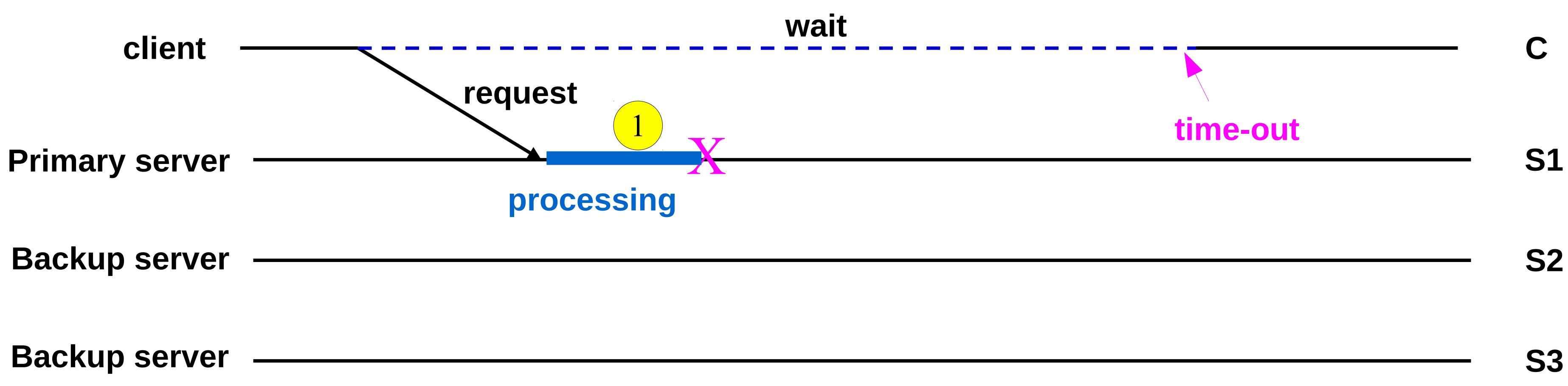
- Consistency Protocol (no failures)
  - Primary sets the execution order
    - Processing order of the requests
  - Communication channels
    - FIFO and loss-less
  - Clients
    - Receive only one response per request (from the primary)

Primary-Based Replication



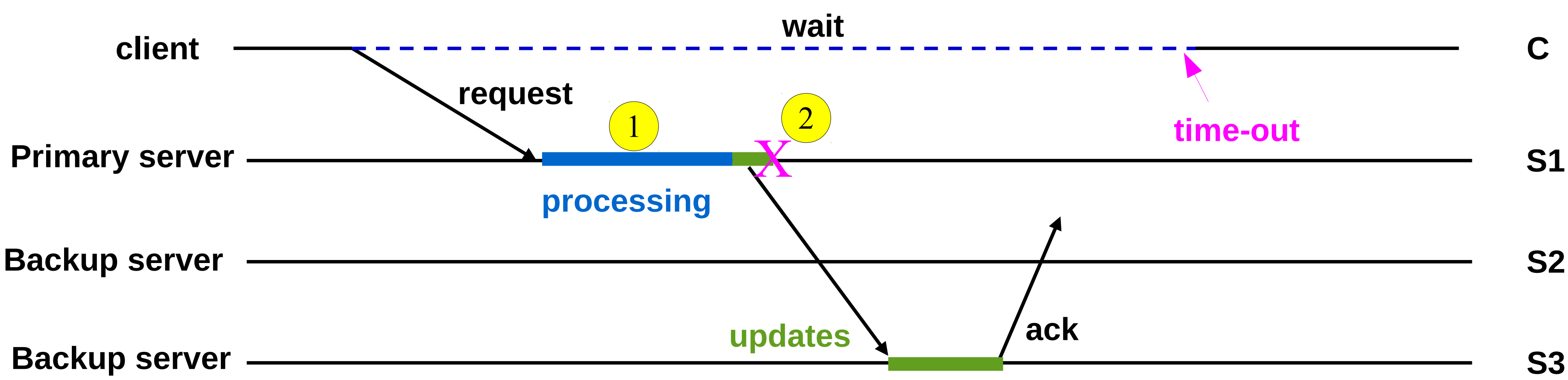
- Introducing Failures
  - We keep FIFO and loss-less channels
  - Both primary server and backup servers may fail
  - We consider only fail-stop servers
- Overall Goal
  - Keep all replicas consistent, despite failures

Primary-Based Replication



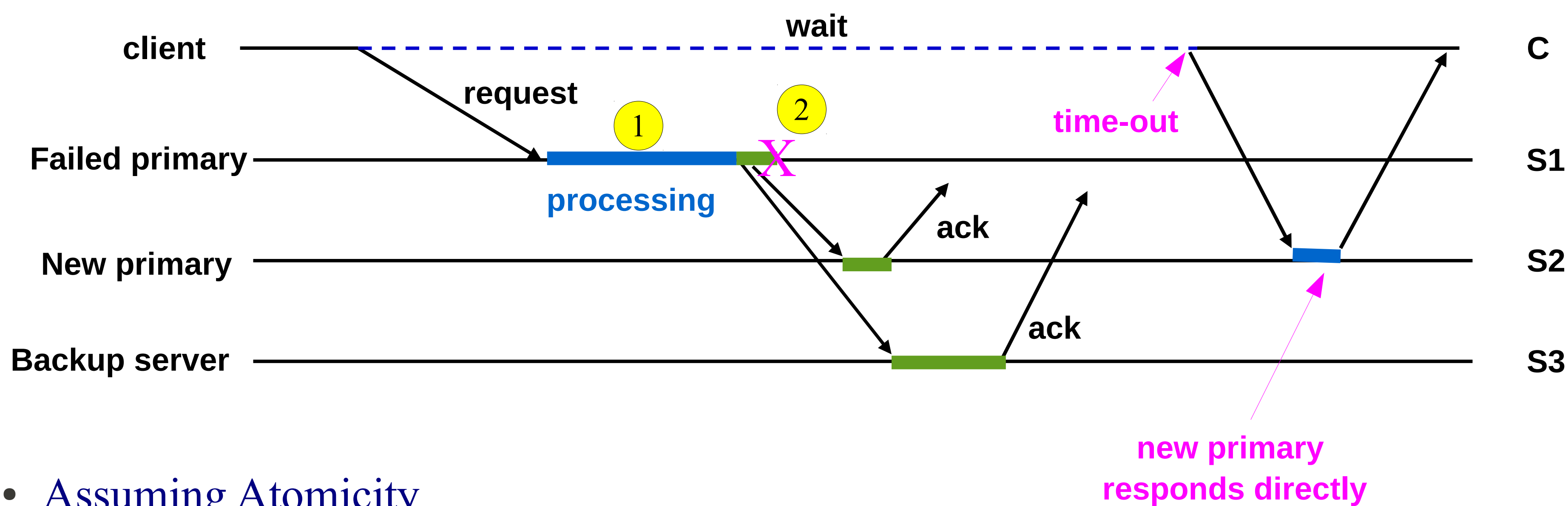
- Primary Failure in 1
  - Crash happens before the processing is over
    - The client will time-out waiting for the response
    - The client will lookup the new primary and retry
  - This requires electing a new primary
    - Which requires to know the group of live servers

Primary-Based Replication



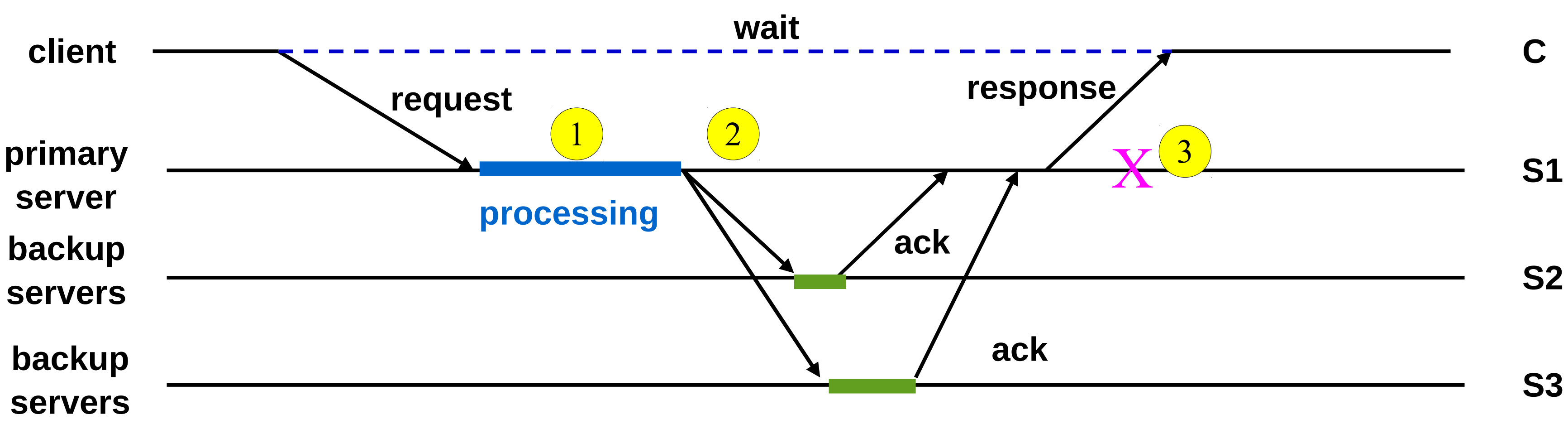
- Primary Failure in 2
  - Crash happens while sending out the updates to replicas
    - The problem is that some replicas might see the updates, while others wont
  - **Atomicity has to be ensured**
    - Must get all updates or none
    - All replicas get all the updates or none of them get any update
  - If no replica received the updates
    - It is equivalent to a failure in 1

Primary-Based Replication



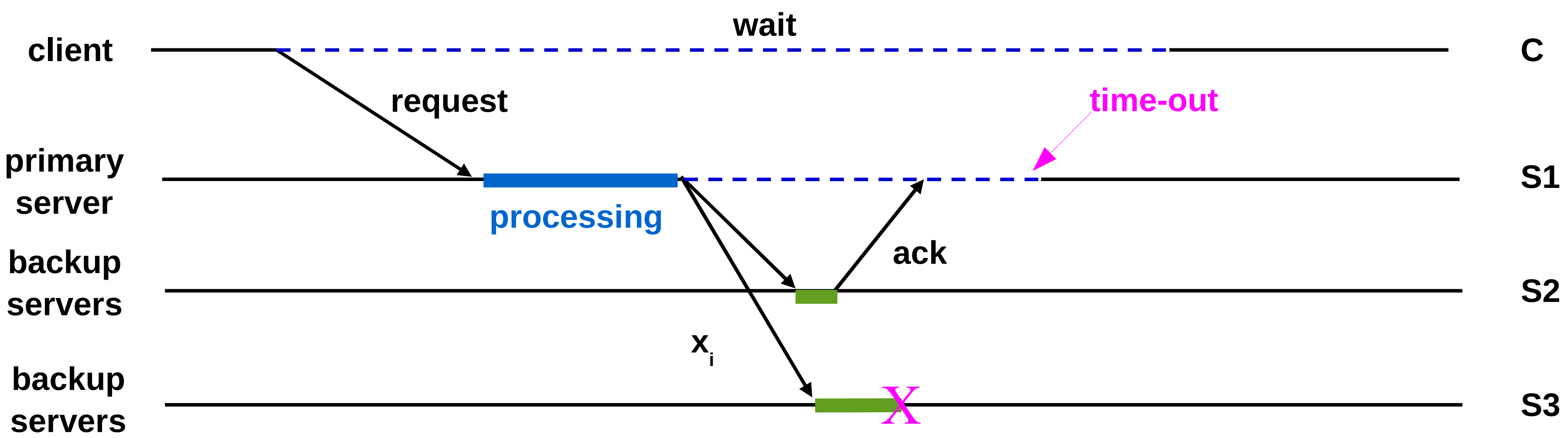
- Assuming Atomicity
  - All replicas received the updates
    - All replicas are up-to-date
    - Any replica may be elected as the new primary
  - Client will still time-out and try again
    - We must detect that the request has been processed already
    - Each request needs a unique identity (sequence number on the failed primary)
    - We need to remember the response for each request

Primary-Based Replication



- Primary Failure in 3
  - The client has received the response
  - It will fail to contact the primary upon its next request
    - It will time-out and lookup the newly **elected** primary
  - Eventually back to a normal situation
    - When the new primary is elected

Primary-Based Replication



- Backup Failures
  - How many acknowledgements should a primary wait for?
  - We need a way to detect that a node failed

Does it work?

Discussing Fault Detection

- Synchronous Systems
  - There is a bound on message delivery
    - We have a **Perfect Failure Detector** (PFD)
  - So we can say
    - If the primary detects a backup has failed, it is failed, 100% sure
    - But this is hardly the reality (example: network partitioning)
- Asynchronous Systems
  - There is no bound on message delivery
  - **Impossibility proved by Fischer, Lynch and Paterson (FLP)**
    - In an asynchronous system with fail-stop processes
    - There is no deterministic protocol to reach a consensus

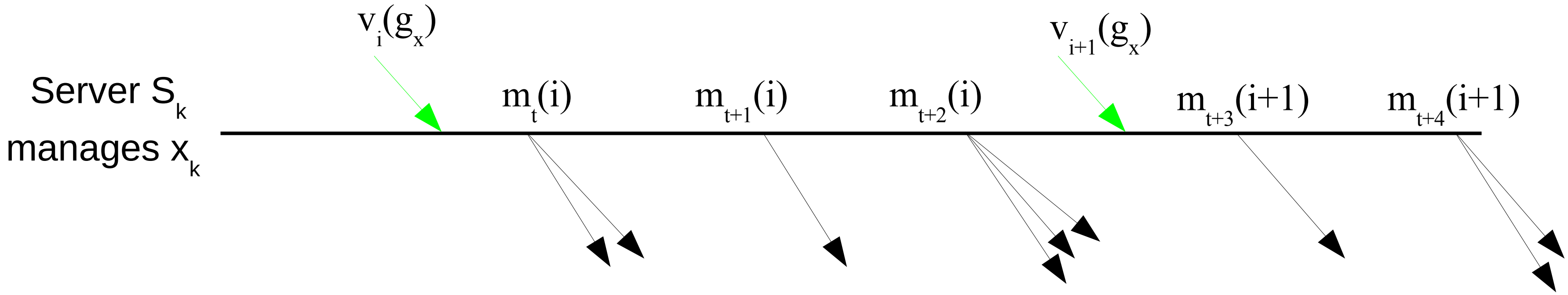
M. J. Fischer, N. Lynch, M. S. Paterson. Impossibility of Distributed Consensus with one Faulty Process, *Journal of the Association for Computing Machinery*, 32(2), pp. 374-382, April 1985 (publication initiale : *Proc. 2nd ACM Principles of Database Systems Symposium*, March 1983)

View Synchronous Multicast

- Basic idea
  - **A view is a consensus about live replicas**
  - New views are created as replicas may join, leave or fail
  - Every one or no one in a view receives each message (atomicity guarantee)
- What for?
  - So we can finish the design of primary-based replication
  - This multicast is not specific to replication, it can be used for other purposes
- Next Steps
  - Explain what is the View Synchronous Multicast
  - Explain how to use it for primary-based replication
  - Discuss consensus that is the foundation of the view mechanism

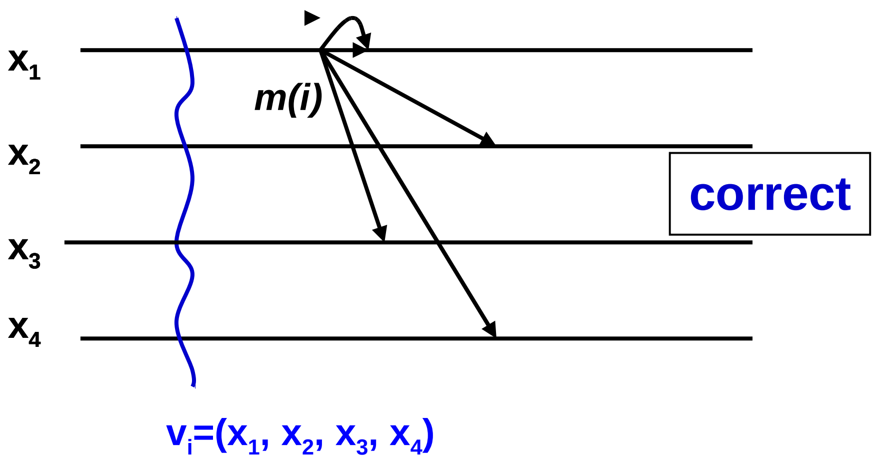
View Synchronous Multicast

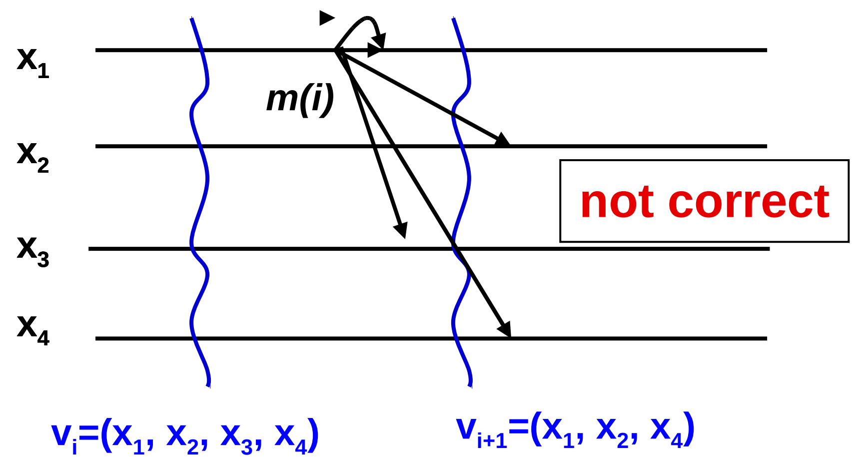
- Principles
  - Consider a group of replicas x_i, for a data item x, noted g_x
  - Consider a sequence of views $v_i(g_x)$, $v_{i+1}(g_x)$, ... $v_{i+n}(g_x)$
    - Each view represents a new state of the group
    - A new view is created everytime a node joins or leaves (includes failure)
  - Assume a node timestamps its messages with the current view
    - Let $t^k(i)$ be the local time at which replica x_k delivers the view $v_i(g_x)$
    - From $t^k(i)$, any message that x_k sends is timestamped with i, noted m(i)
    - This remains true until x_k delivers the view $v_{i+1}(g_x)$



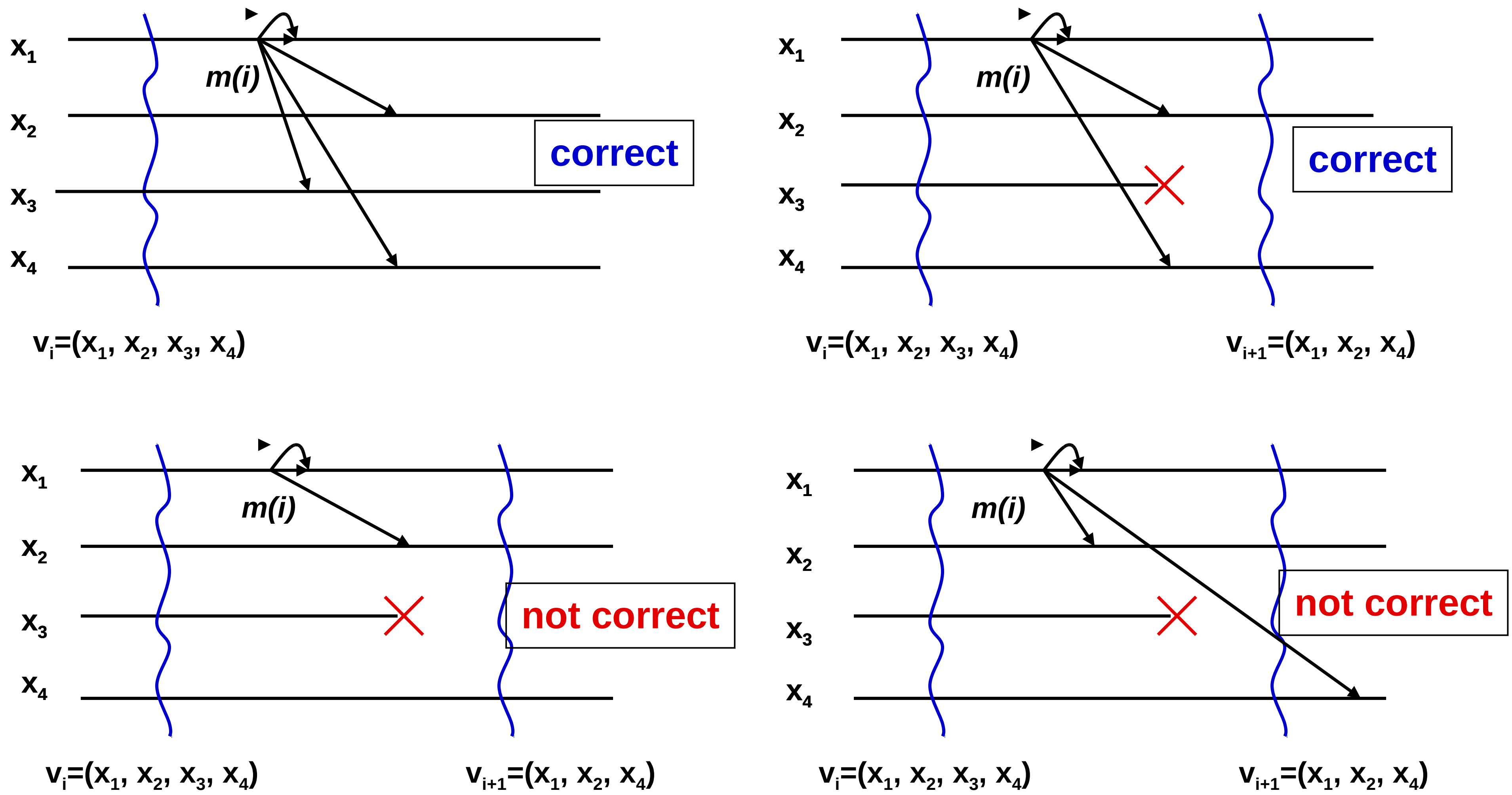
View Synchronous Multicast

- Correctness Rule
  - Given a view $v_i(g_x)$ and a message m(i)
  - All replicas in $v_i(g_x) \cap v_{i+1}(g_x)$ must either
    - all deliver m(i) before delivering $v_{i+1}(g_x)$
    - or none of them delivers m(i)



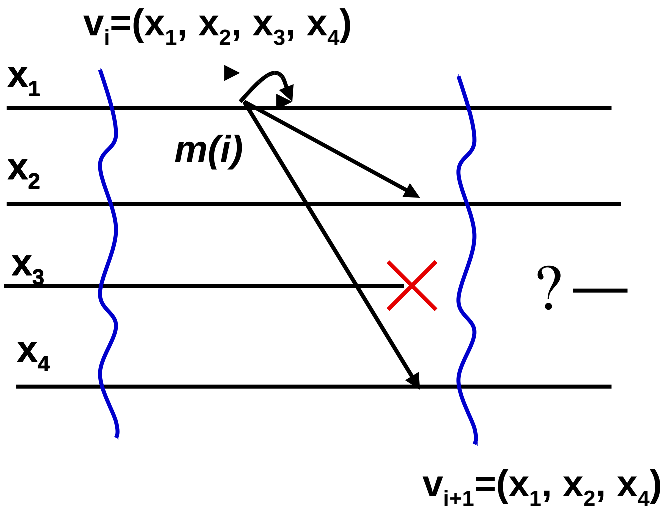


View Synchronous Multicast



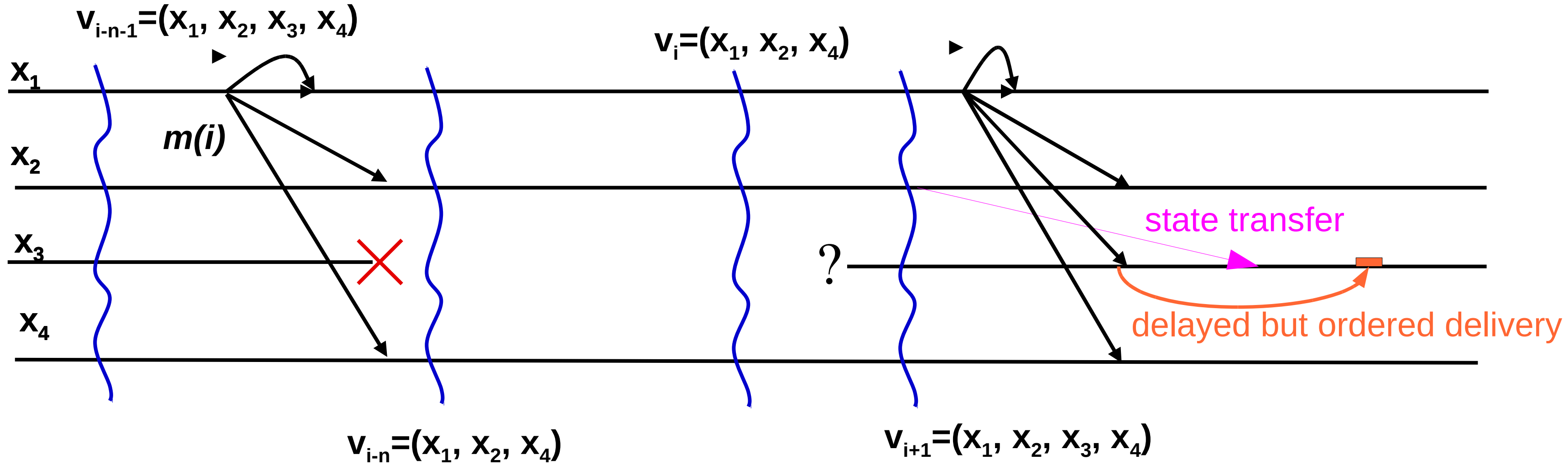
Primary-Based Replication

- Replica Consistency Conditions
  - If we have a failure detector (producing the views)
  - And we have a mechanism to ensure view synchronous multicasts
  - Then we have consistent replicas
- Is that enough?
  - View synchronous multicast is not enough
    - It provides reliable multicast
    - Hence atomic updates across correct replicas
  - After failure at replica x_i
    - Replica x_i is repaired and needs to re-join
    - **Its state needs to be brought up to date**



Primary-Based Replication

- State Transfers
  - To re-join, replica x_p forces a new view $v_{i+1}(g_x)$
    - Replica x_p is added $v_{i+1}(g_x)$
    - Any correct replica x_q can send its state to x_p
      - It sends its state when it delivers the new view $v_{i+1}(g_x)$
    - From the time it delivers $v_{i+1}(g_x)$
      - Replica x_p has to delay delivering all messages m(i+1)
      - Until it receives its new state from x_q



Primary-Based Replication Recap

- Dynamic group of servers
  - Any server may fail, but not all of them (at least one must be alive at all time)
  - Failed servers are reinserted
- Primary server is elected
  - Primary server process and answers client requests
  - Backup servers are only sent the updates
  - Work for both deterministic and non-deterministic applications
  - Clients see the primary failures, they have to switch to a new primary
  - Clients do not see failures of backup servers
- Primary uses a view synchronous multicast
  - Based on a consensus of which servers are correct
  - Ensure the atomicity of updates (all backup servers have identical states)
  - Can be built on an imperfect failure detector

Active Replication Recap

- Dynamic group of servers
  - Any server may fail, but not all of them (at least one must be alive at all time)
  - Failed servers are reinserted
- No election is necessary
  - All servers execute the requests and have a complete copy of the data
  - All requests from clients must be totally-ordered on all servers
  - Only works with deterministic computations

Consensus

- Definition
  - Given a set of processes $P_1,\ldots,P_n$
  - Initially, each process P_i proposes a value V_i
  - If the consensus protocol terminates, we have
    - **Agreement**: All ***correct*** processes decide the same value
    - **Integrity**: each process decides at most once
    - **Validity**: the decided value is one of the proposed ones
    - **Decision**: if at least ***one correct process*** starts the consensus, all correct processes ***eventually*** decide a value
  - A process is correct
    - If it is not failed
    - If it has never failed (assuming a failed process may be restarted)
      - A notion that only applies within the start-end bounds of the consensus protocol

Consensus

- Starting a Consensus
  - Not included in the consensus protocol itself
    - Initially, each process P_i proposes a value V_i
  - Different possible approaches
    - It could be at regular intervals or well-know times
      - Beware of clock skewing...
    - It could be by broadcasting to the processes
      - But be really careful about the properties of this broadcast
      - Only those receiving the message will be part of the consensus
- Communication Channels
  - Processes are connected through communication channels
    - Channels are FIFO and loss-less
  - We will consider synchronous and asynchronous systems
    - Delivery time is bounded or not
  - We will consider only fail-stop system
    - Byzantine failures are too complex

Consensus

- Reliable Broadcast
  - A foundation mechanism
  - A process P_i broadcast a message to all processes P_j, including itself
- Reliable Broadcast Properties
  - **Agreement:** if one correct process delivers a message, all correct processes eventually deliver m
  - **Validity**: if one correct process broadcast a message m, all correct processes eventually deliver the message
  - **Integrity**:
    - A broadcasted message is delivered at most once
    - A delivered message must have been broadcasted

Eventually: Delivery <u>will</u> happen in <u>finite</u> time

Reliable Broadcast

- Protocol

Broadcast a message, noted *m*

Timestamp m with a sequence number, noted *seq(m)*

Identify sender, noted *sender(m)*

Send m to all processes, including *sender(m)*

Deliver a broadcasted message at a process P_i

Receive the message m (from the communication channel)

If the message has been delivered, just drop it

If this is the first time P_i receive m and sender(m) is not P_i

Send m to all processes (but process P_i)

Deliver message m

Reliable Broadcast

- Discussion
  - Nothing is said about the order of delivery
  - Atomicity property
    - All correct processes eventually receive a broadcasted message
    - Or none of them receive it
- Remarks
  - It is this atomicity about a global knowledge (the message m) that allows to reason and make progress about a consensus
  - The algorithm is not optimized, better protocols exist, but the protocol shows it is possible to achieve a reliable broadcast under our assumptions

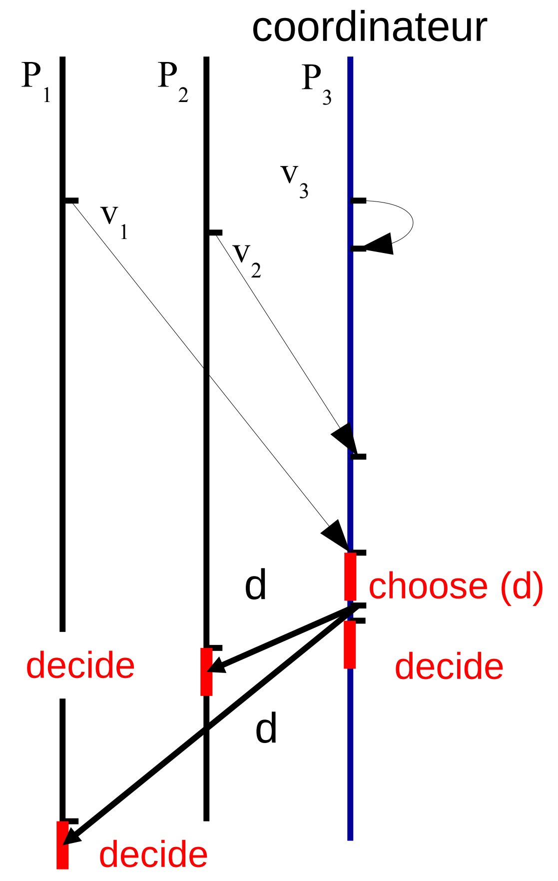
V. Hadzilacos , S. Toueg, Fault-Tolerant Broadcast and Related Problems, in S. Mullender (ed.), *Distributed Systems* (2nd edition), Addison-Wesley, 1993

Reliable Broadcast

- Proof
  - Agreement:
    - If one process delivers a message m, it finished sending the message to all other processes prior to delivering it
    - Since communication channels are loss-less, all correct processes will eventually receive the message and deliver it (unless they crash, in which case they are not correct any more)
  - Validity:
    - If a correct process has broadcasted a message (the broadcast pseudo code was executed) the message was sent to all processes
    - Since the sender is correct (it is not failed and didn't fail), it eventually delivered the message and because of the agreement above all correct processes also delivered the message
  - Integrity
    - By the very structure of the algorithm
    - Only sent messages are received and already delivered messages are ignored

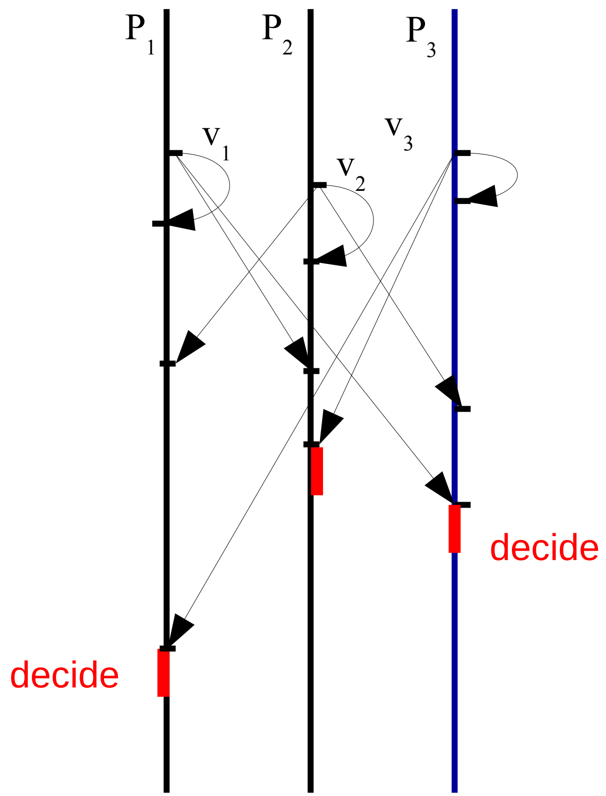
Consensus

- Hypothesis
  - Loss-less communication channels
  - No node and no process failures
- Coordinator Solution
  - Each process sends its value to the coordinator
  - When the coordinator has all the values, it picks one (on whatever criterium) and sends that value to all processes
  - When receiving the value, all processes decide the same value



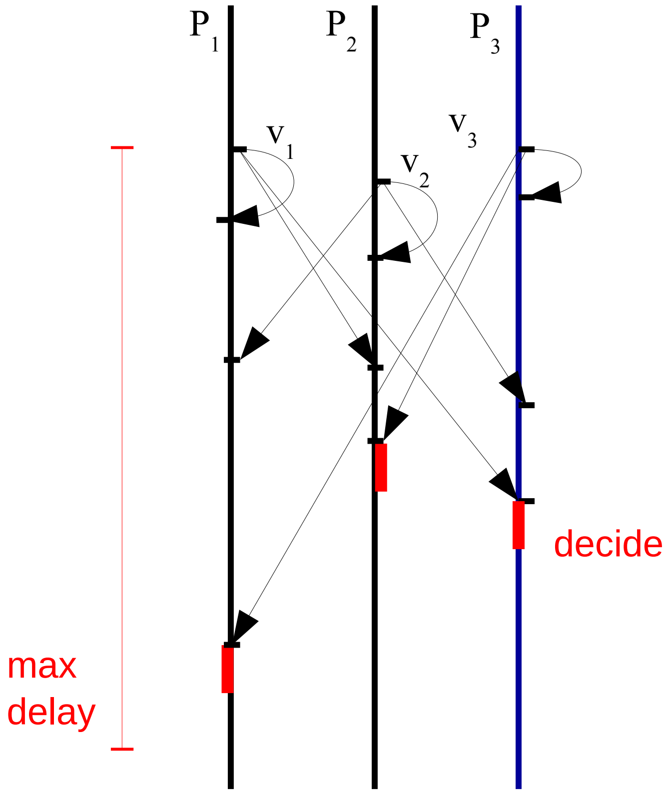
Consensus

- Same Hypothesis
  - Loss-less communication channels
  - No node and no process failures
- Symmetric Solution
  - All processes are equivalent
  - Each process broadcasts its initial value to all other processes
  - When a process has received all the values, it picks one using an agreed upon algorithm
  - All processes have all the same values, the same decision algorithm, they will decide the same value



Consensus

- **New** Hypothesis
  - Loss-less communication channels
  - **Fail-stop processes**
  - **Synchronous system**
- Symmetric Solution
  - Same symetric solution
  - Wait for values only for a maximum delay
  - The maximum delay can be estimated (synchronous system)
  - Passed that delay, we know that if we didn't get a message, the sender has failed
  - **True only if** the sending of the initial values are somewhat coordinated



Consensus

- Moving to Asynchronous Systems
  - We are facing FLP...
- Different Approaches
  - Partially synchronous systems
    - Dwork, Lynch, Stockmeyer (1988)
  - Non-deterministic algorithms
    - Rabin (1983)
  - Best-effort approaches
    - Paxos Algorithm (Lamport 1989), even adaptable to byzantine failures
    - Use imperfect fault-detectors like Chandra and Toueg (1991)

Best-Effort Consensus

- One Study Only
  - Only looking at the use of imperfect fault detectors
  - Basic idea:
    - The FLP impossibility relies on the inability to know if some process has failed or if the message we are waiting for is late, delayed in transit
    - Having a fault detector, even imperfect, is enough to avoid the FLP impossibility and make reaching a consensus possible
  - Remember:
    - Loss-less communication channels (messages will eventually arrive)
    - Asynchronous system (no bound on message delivery time)

Imperfect Failure Detectors

- Completeness
  - **Strong Completeness**: eventually, every process that crashes is permanently suspected by *every* correct process
  - **Weak Completeness**: eventually, every process that crashes is permanently suspected by *some* correct process
- Accuracy
  - **Strong Accuracy:** no process is suspected before it crashes
  - **Eventual Strong Accuracy**: eventually, correct processes are not suspected by any correct process.
  - **Weak Accuracy:** some correct process is never suspected
  - **Eventual Weak Accurary:** eventually, some *correct* process is never suspected by any *correct* process

Imperfect Failure Detectors

- Practical Choice
  - **Strong Completeness**: eventually, every process that crashes is permanently suspected by *every* correct process
  - **Eventual Weak Accurary:** eventually, some *correct* process is never suspected by any *correct* process
- Simple Design
  - Each process q periodically sends a message q-*is-alive*
    - If a process p times-out without receiving anything from q
      - It adds q to a list of suspected processes (failed)
    - If a process p realizes it erroneously suspected q
      - It removes the process q from the suspected list
      - It increments the time-out for that process q
        - Trying to safeguard against the same mistake...
- Does it work?

Imperfect Failure Detectors

- Does it work? Nope.
  - If it really did, FLP impossibility would not stand!
- But it is enough in practice...
  - As we grow the timeout
    - More and more likely that a correct process will be considered live
      - So we achieved eventual weak accuracy...
      - But no theoritical proof, just practical behavior of real systems
    - Longer will be the delay before we consider a failed process
      - So we endanger strong completeness

Strong Completeness: eventually, every process that crashes is permanently suspected by *every* correct process

Eventual Weak Accurary: eventually, some *correct* process is never suspected by any *correct* process

Consensus Protocol

- Hypothesis
  - Strong completeness and eventual weak accuracy
  - Uses a reliable broadcast noted *R-broadcast(m)*
  - Want to resist F failures, we need $(2F+1)$ processes
- Principle
  - Tries to reach a consensus in multiple rounds
  - For each round, we try one process as the coordinator
    - If it reaches a consensus, we are done
    - If not, we try the next process as a coordinator
  - We rotate between correct processes as long as we don't have a consensus
    - Eventually, we will reach one (depending on faults and accuracy of our fault detector)
    - **No guarantee in any bounded time !**

Consensus Protocol

- Per Round
  - We have four phases
    - **Phase 1**: all processes send to the coordinator their estimate of the consensus
    - **Phase 2**: the coordinator **waits** until it has a majority of estimates, picks one as the new estimate and broadcast that new estimate
    - **Phase 3**: all processes **receive** the new estimate and acknowledge that new estimate to the coordinator
    - **Phase 4**: the coordinator **waits** for a majority of acknowledgements and then decide for that last estimate that it **reliably broadcast**
  - If anything fails to happen that way, we go for another round.
    - The coordinator may suspect a majority of processes to have failed
    - A process may suspect the coordinator to have fail and not acknowledge the new estimate
    - The coordinator may suspect a majority of processes to have failed while waiting for the acknowledgements of the last estimate

Consensus Protocol

```
upon propose(v)                          // processus pi
   r:=0                                  // current round
   t:=0                                  // last round where v was updated
   while not decided do
      c := (r mod N) + 1                 // pc is the coordinator
      send (vote, r, v, t) to pc         // N is the number of processes      } phase 1

      if i = c then // only happens at the coordinator
         wait until (receive (vote, r, v', t') from (N+1)/2 non-suspected processes)
         maxt := largest t' received
         v := some v' received with t' = maxt
         send (propose, r, v) to all     // v is the new proposed consensus   } phase 2

      wait until (receive (propose, r, v') from  pc or c is suspected )
      if a (propose, r, v') message was received then
         v := v' ; t := r                // Update proposed consensus
         send (ack) to pc                // Acknowledge proposal
      else send (nack) to pc             // pi suspects the coordinator       } phase 3

      if i = c then // only happens at the coordinator
         wait until (receive ack or nack messages from (N+1)/2 non-suspected processes)
          if all are ack then R-broadcast(decide, v)                          } phase 4
      r := r + 1                         // try another round...
```

```
upon R-deliver (decide, v')              // Deliver procedure of the reliable broadcast
   if not decided then
      decide(v')
      decided := true
```

Consensus Protocol

- Remarks
  - If we want to consider crash-recovery, we need a modified protocol

    M. Aguilera, W. Chen, S. Toueg. Failure detection and consensus in the crash-recovery model, *Proc 12th Int. Symp. on Distributed Computing*, 1998

  - So we achieved consensus with
    - Strong completeness
    - Eventual weak accuracy
    - We tolerate (n/2)-1 failures
    - Number of rounds is finite, but not bounded
  - Can we do better?
    - Nope, our assumptions are the weakest that solves the consensus
    - Chandra, Hadzilacos, Toueg (1996)

Conclusion

- Replication
  - We have seen two basic models (primary-based and active)
  - In synchronous and failure-free systems
    - It is rather easy
    - With fail-stop processes, it is harder
  - In asynchronous system
    - With fail-stop processes, it is complex
  - Byzantine failures are a research topic for all practical purposes
- Consensus
  - Equivalent to View Synchronous Multicast
  - Also equivalent to totally-ordered and reliable multicast
  - So both primary-based and active replication need consensus
  - Consensus is a core challenge of asynchronous distributed systems