Fundamentals – Part Two

Professor Olivier Gruber

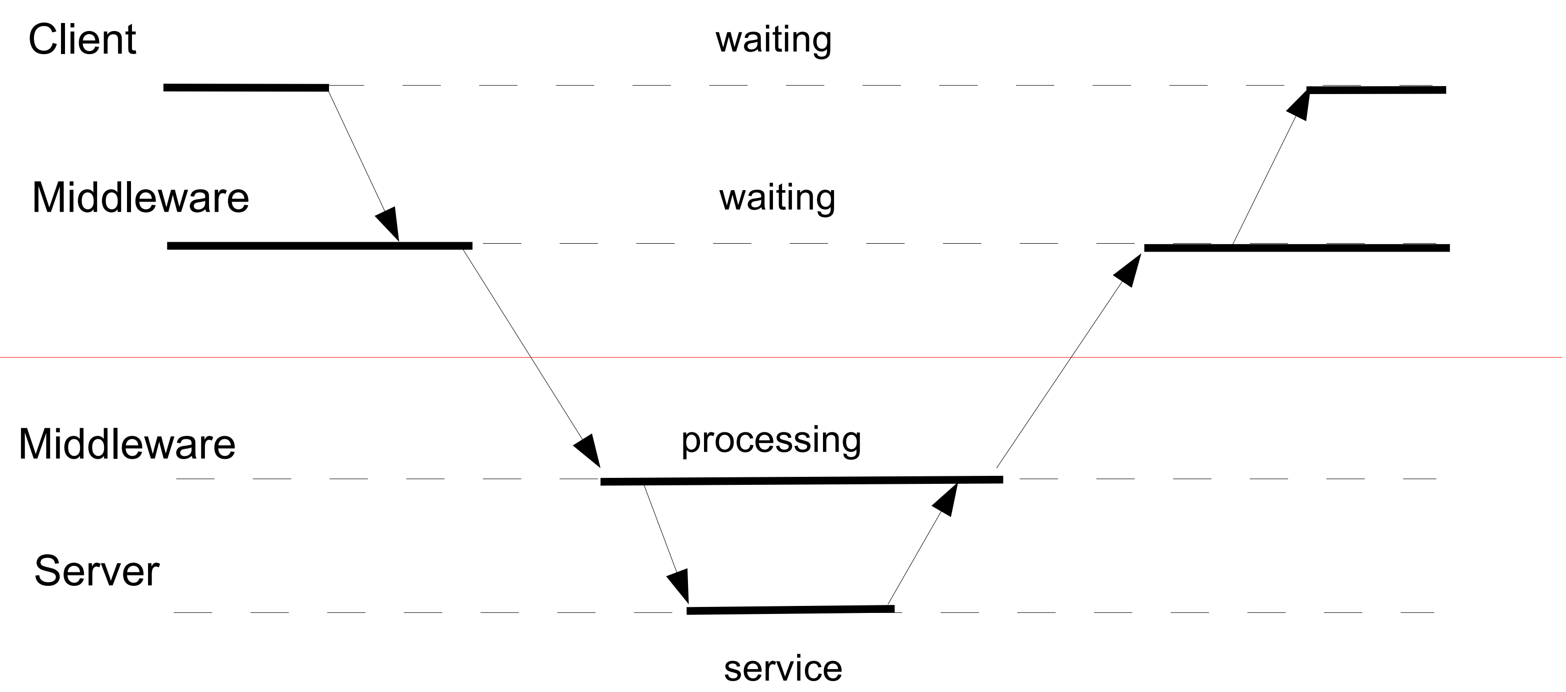
Université Joseph Fourier

Projet SARDES (INRIA et IMAG-LSR)

Olivier.Gruber@inria.fr

Message Fundamentals

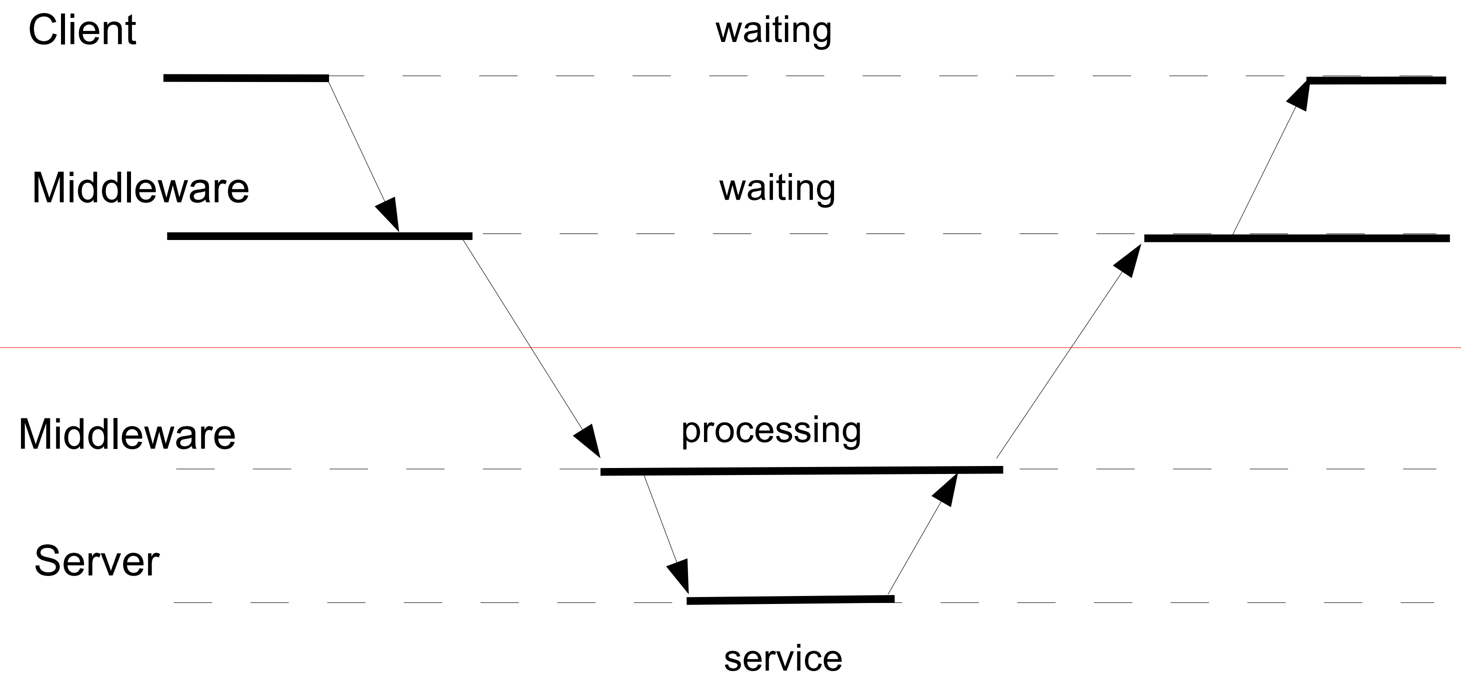
- Last lecture
  - How do we name the destination?
  - How do we route the message?



Olivier.Gruber@inria.fr

Message Fundamentals

- Today's lecture
  - What notion of time do we have?
  - How do we synchronize activities?



Olivier.Gruber@inria.fr

Outline

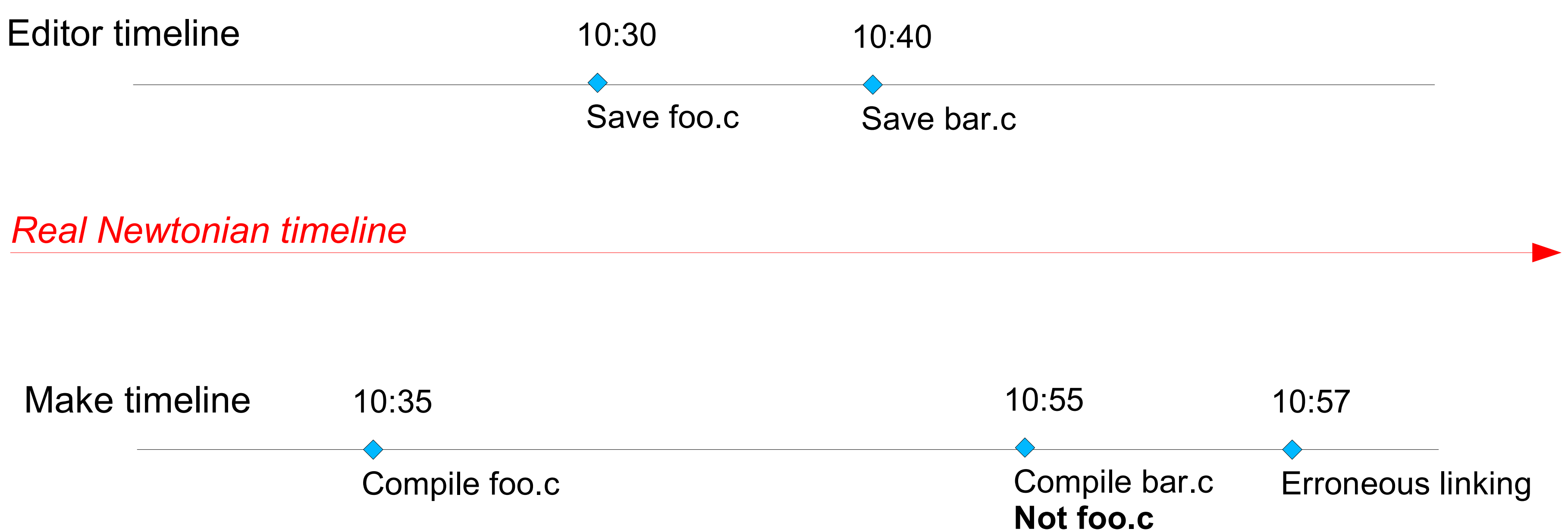
- Discussing time
  - Distributed systems have no concept of a global time
    - Different protocols exist for syncing clocks
    - Good enough for humans, not for synchronization
  - There is no escaping the true nature of distributed time
    - Impacts our execution models
    - Introduces causal order
  - Specific techniques
    - Logical clocks and totally ordered multicast
    - Vector clocks and causally ordered multicast
    - Matrix clocks and causal point-to-point messaging
- Discussing synchronization
  - Mutual exclusion in distributed systems
  - Election in distributed systems

Discussing Time

- Centralized system
  - Time is unambiguous
    - The hardware keeps track of it, the kernel provides access to it
    - It does not matter it is the correct time, it orders local events
  - The concept of time is used in so many places
    - As absolute measure, like with the make program
    - Between events, like mutual exclusion
- Distributed system
  - No concept of global time, time becomes ambiguous
    - Very much like moving from Newton to Einstein physics
      - There no longer a single time, each machine has a notion of time
      - Not everybody agrees about the time of two events or between two events
  - Asynchronous communications
    - Communication delays are unbounded and messages may be lost
    - How to distinguish a slow message from a lost one?

Example: Make

- Unix Make program
  - Relies on time to know what to do
    - Example: compile sources into object files and link them into an executable
  - Running make and editing on different machines
    - They may have different times
    - Yielding linking of incoherent object files



Olivier.Gruber@inria.fr

Physical Clocks

- Real timers
  - Ticks a certain number of times per seconds
  - Time is the number of ticks since a certain known date
    - Like January first, 1970 for most Unix systems
- Clock skew
  - 60Hz timers do not tick exactly 60 per seconds
    - With modern chips, the skew is about 10^{-5}
    - Instead of 216,000 ticks per hour
    - We get between 215,998 and 216,002 ticks
- Clock synchronization
  - Several clocks therefore need to be synchronized
  - It can be done through different protocols

Discussing Time

- A bit of history
  - 17th Century, time is defined through solar day of 24 hours
  - In 1940, scientists established that the earth rotation is slowing down
    - Due to tidal friction and atmospheric drag
    - About 300 million years ago, a year was about 400 day (shorter days)
  - In 1948, we started measuring time with atomic clocks (Cesium 133)
    - Several clocks are around the world, averaged in Paris
    - ***Temps Atomic International*** : averaged Cesium-133 ticks since Jan. 1, 1958
  - Problem: TAI is 3 ms ahead of the solar time (which is still slowing down)
    - In 1582, Pope Gregory XIII decreed that 10 days be omitted from the calendar…
    - Social instability and riots followed…
  - Introduces Universal Coordinated Time (UTC)
    - Bureau International de l'Heure (in Paris)
    - UTC introduces leap seconds to stay in sync with solar time
    - So far, we introduced about 30 leap seconds (when skew is over 800ms)

Discussing Time

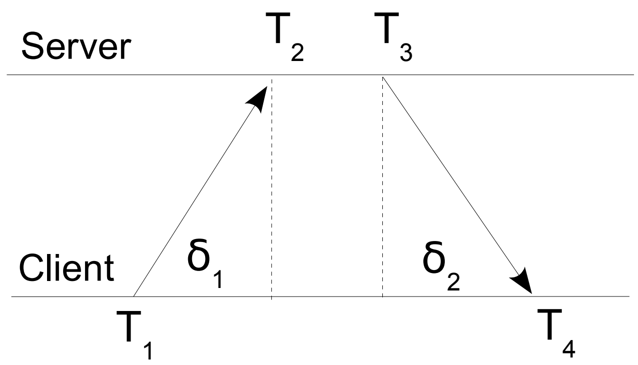
- How do we tell time?
  - Most electric companies keep their frequency in sync with UTC
    - So they raise the current frequency for accounting for leap seconds
    - Accuracy of 1 second is too crude for computer clocks
  - Shortwave radio stations
    - Accuracy is about 1ms, because of atmospheric fluctuations, rarely better than 10ms
  - Geostrationary Satellites
    - Accuracy about 0.5ms, transmission delays have to be taken into account
      - Innacurate satellite position, unknown receiver position, clock skew, atmospheric conditions (ionoshpere effects are changing over time), etc.
    - Claimed accuracy for professional receivers of 20-35 nano-seconds

Discussing Time

- Do we have a solution?
  - Geostrationary Satellites:
    - Claimed accuracy for professional receivers of 20-35 nano-seconds
  - That's pretty good... isn't it?
- Well... it does not solve our problem...
  - Not all networks have such receivers
  - And even if they would...
    - How do we use that time to sync'up others computers?
    - Network delays have to be taken into accounts...

Network Time Protocol

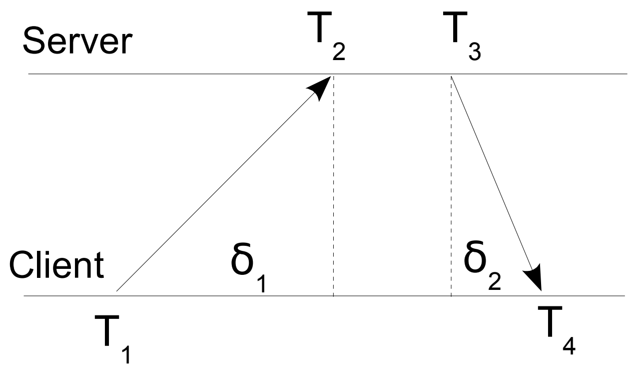
- Cristian algorithm (1989)
  - Use a time server (with a correct UTC)
  - Takes into account message delays
  - Principle
    - All times Ti are local times
    - How do we estimate what T4 should be?
    - We use transmission delays
      - $\delta 1=(T2-T1)$ $\delta 2=(T4-T3)$
    - We assume delays to be roughly constants
      - $\delta 1 \simeq \delta 2$
      - $\delta = (\delta 1 + \delta 2)/2$
    - Correction is Θ
      - $T4 + \Theta = T3 + \delta$



Olivier.Gruber@inria.fr

Network Time Protocol

- Gradual change
  - Correction Θ can be negative or positive
    - Time can't go backward
    - Time should avoid leaps
  - Clocks are slown down or advanced
    - Each interupt is either 9ms or 11 ms instead of 10ms
- Error correction
  - What if δ1 and δ2 differ too much...
    - Average Θs over multiple requests
    - Use multiple time servers and average Θs



Olivier.Gruber@inria.fr

Berkeley Algorithm

- Coordination between nodes
  - No node has UTC, like in disconnected private networks
  - We still want synchronized clocks, even if they are not on UTC
  - Sometimes, agreeing on time is just enough
- Principle
  - A coordinator ask all machines their current time
  - It computes what the time should be
    - It averages received local times, ignoring those with times too far off
  - It sends back time corrections

Discussing Time

- Real time
  - Is just an illusion...
    - Precise enough in some situations, like for humans or for a make program
  - But always some marging of error
    - It cannot be used to reason about a distributed system
    - It cannot be the basis of behavioral proofs

- Example: critical section

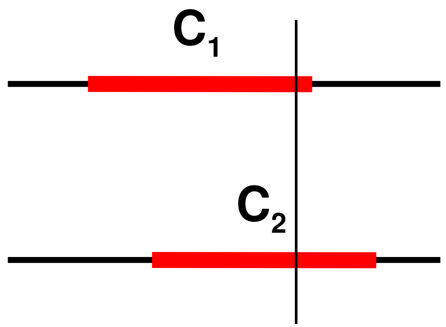
  We either have

  Leave(C2) happens-before Enter(C1)

  Or

  Leave(C1) happens-before Enter(C2)

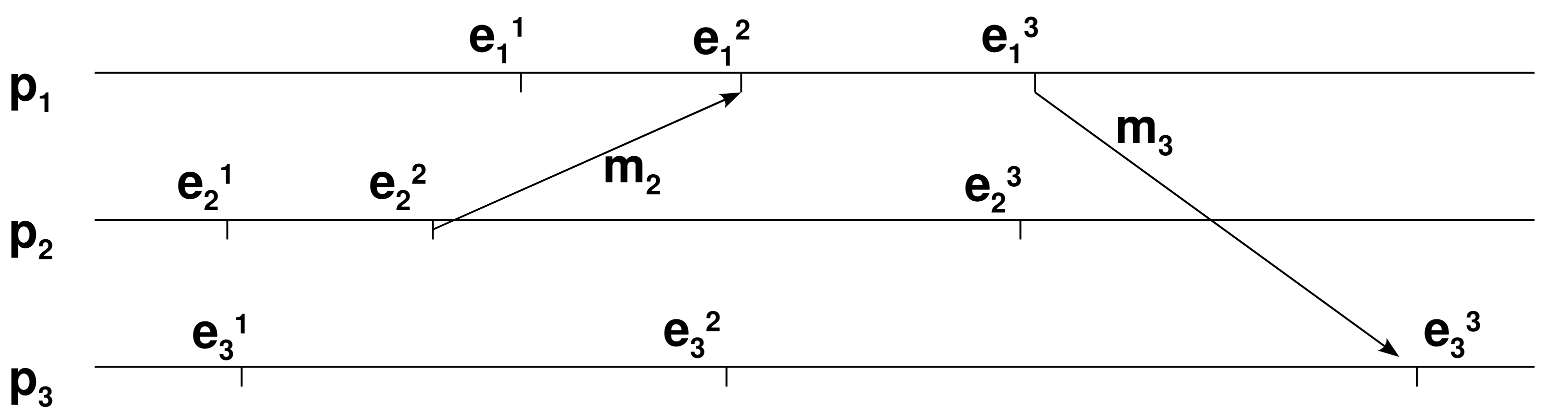
  Without global time, how do we tell?



Olivier.Gruber@inria.fr

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



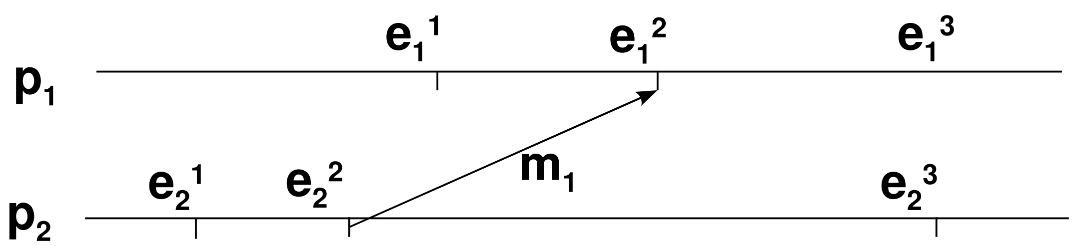
Olivier.Gruber@inria.fr

Causal Order

- Lamport (1978)
  - Causal order between two events is noted
    - **e → e'**
  - It is defined as
    - e *happened-before* e'
  - In our execution model, we have **e → 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 **e → e"** and **e" → e'** then **e → 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

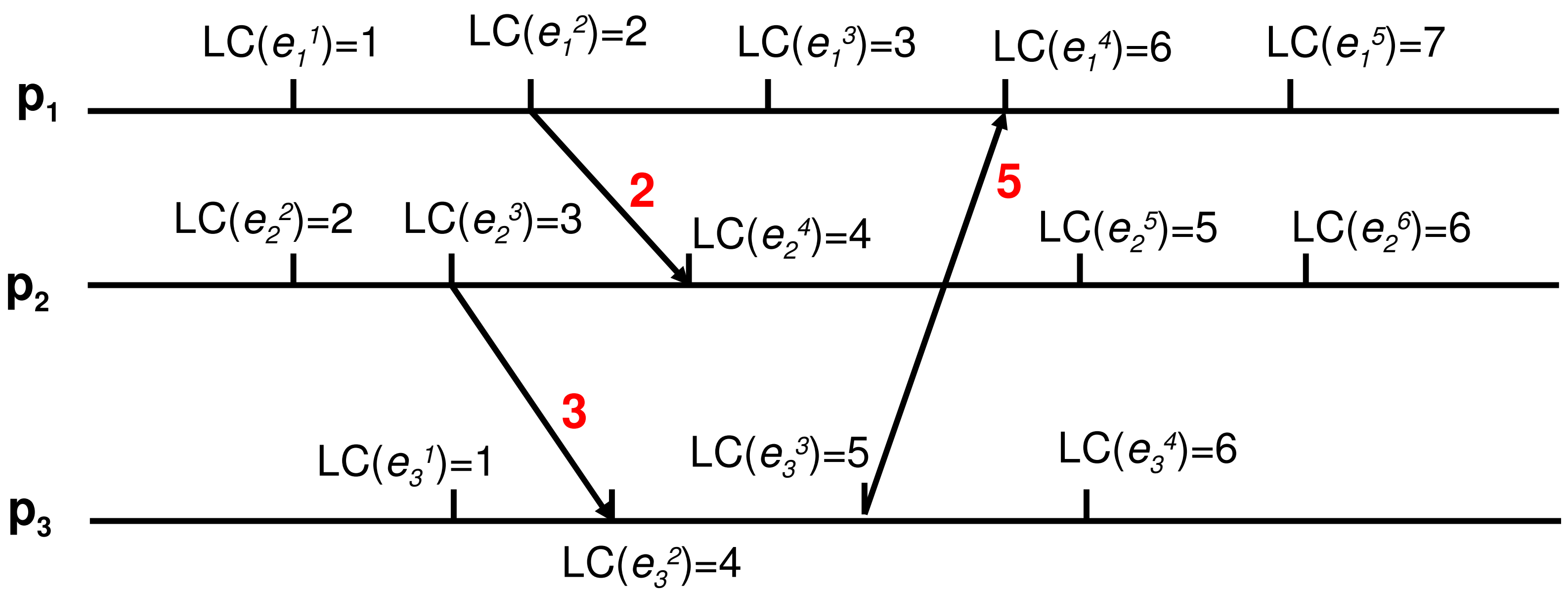
- Logical Clocks
  - Nothing to do with real time
  - Logical clock for an event e_i^k is noted **LC(e_i^k)**
  - Design
    - Logical clocks are maintained as local counters
    - For each new local event e_i^k: $LC(e_i^k) = LC(e_i^{k-1}) + 1$

- Regarding Messages
  - Sending a message M
    - This is a new local event e_i^k: **$LC(e_i^k) = LC(e_i^{k-1}) + 1$**
    - **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
    - **$LC(e_j^r) = max(LC(e_j^{r-1}), LC(e_i^k)) + 1$**

Olivier.Gruber@inria.fr

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 $(e_3^1 \parallel e_2^3)$

Olivier.Gruber@inria.fr

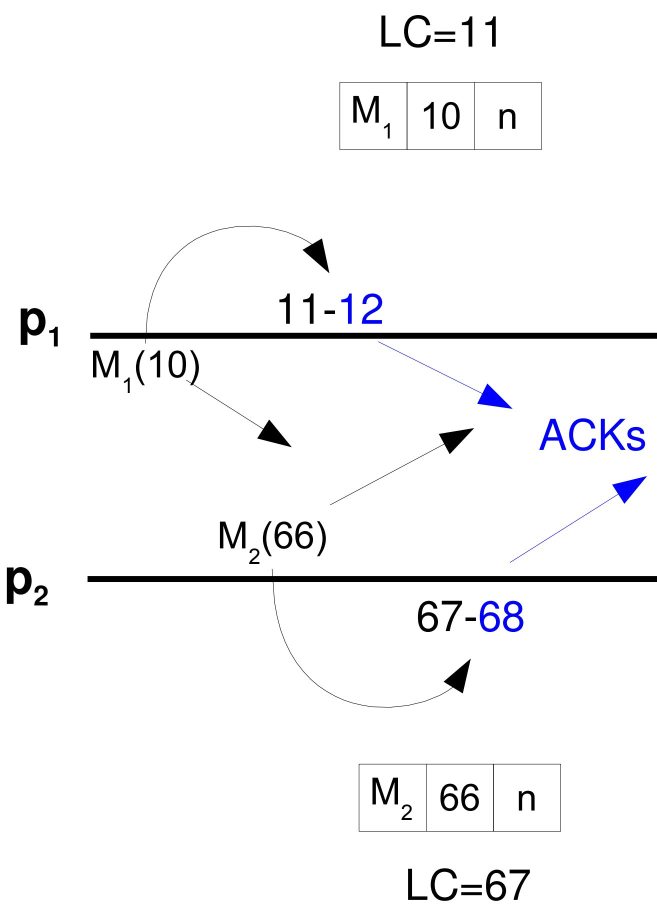
Ordered Multicast

- 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

Totally Ordered Multicast

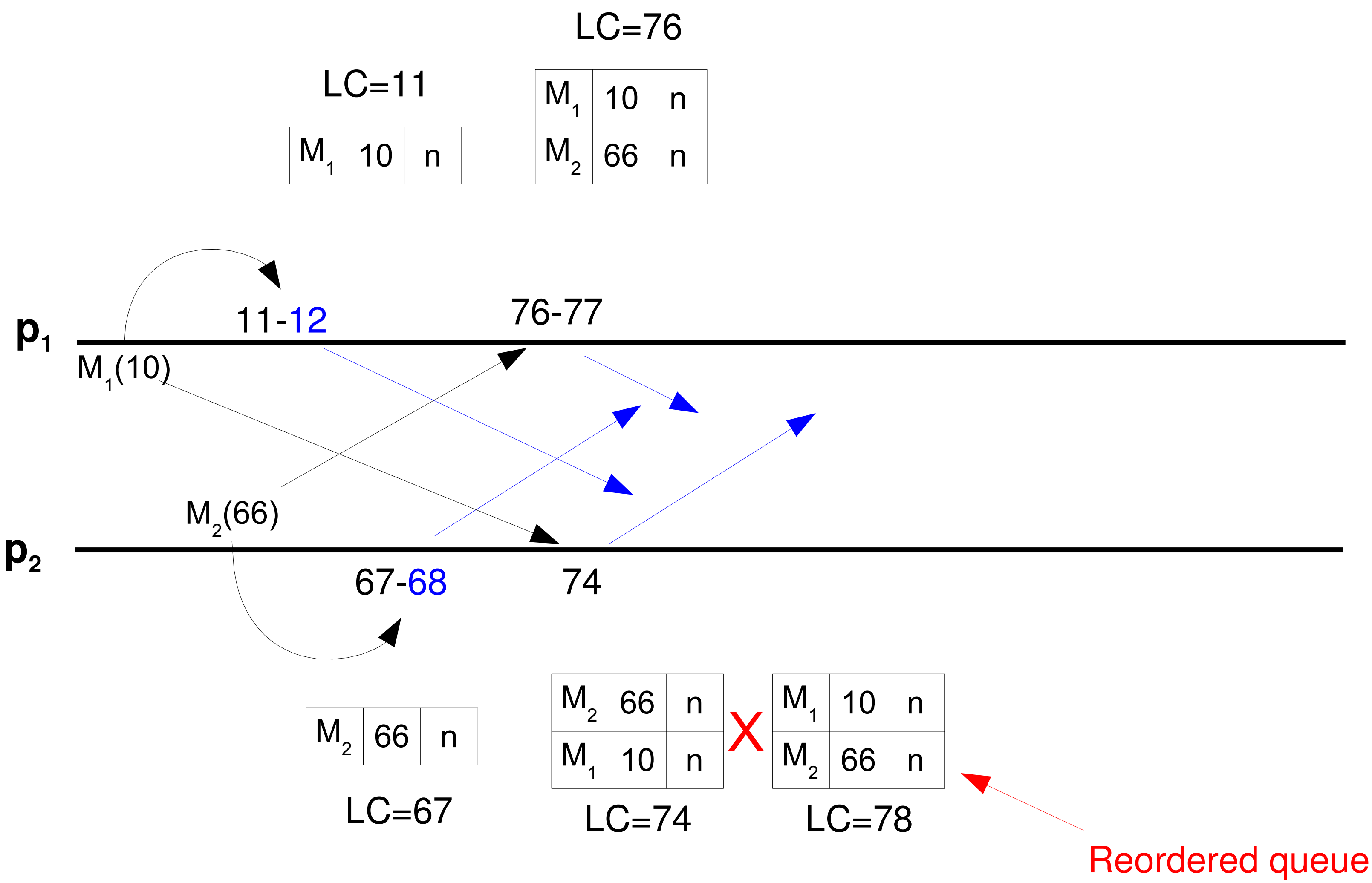
- Totally Ordered Multicast
  - Using Lamport's logical clocks
- Design
  - Between a group of N processes
    - They **must know each others** (concept of a group)
    - Each message from one process is **multicasted to the entire group**
    - We assume FIFO and loss-less communication channels
  - 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



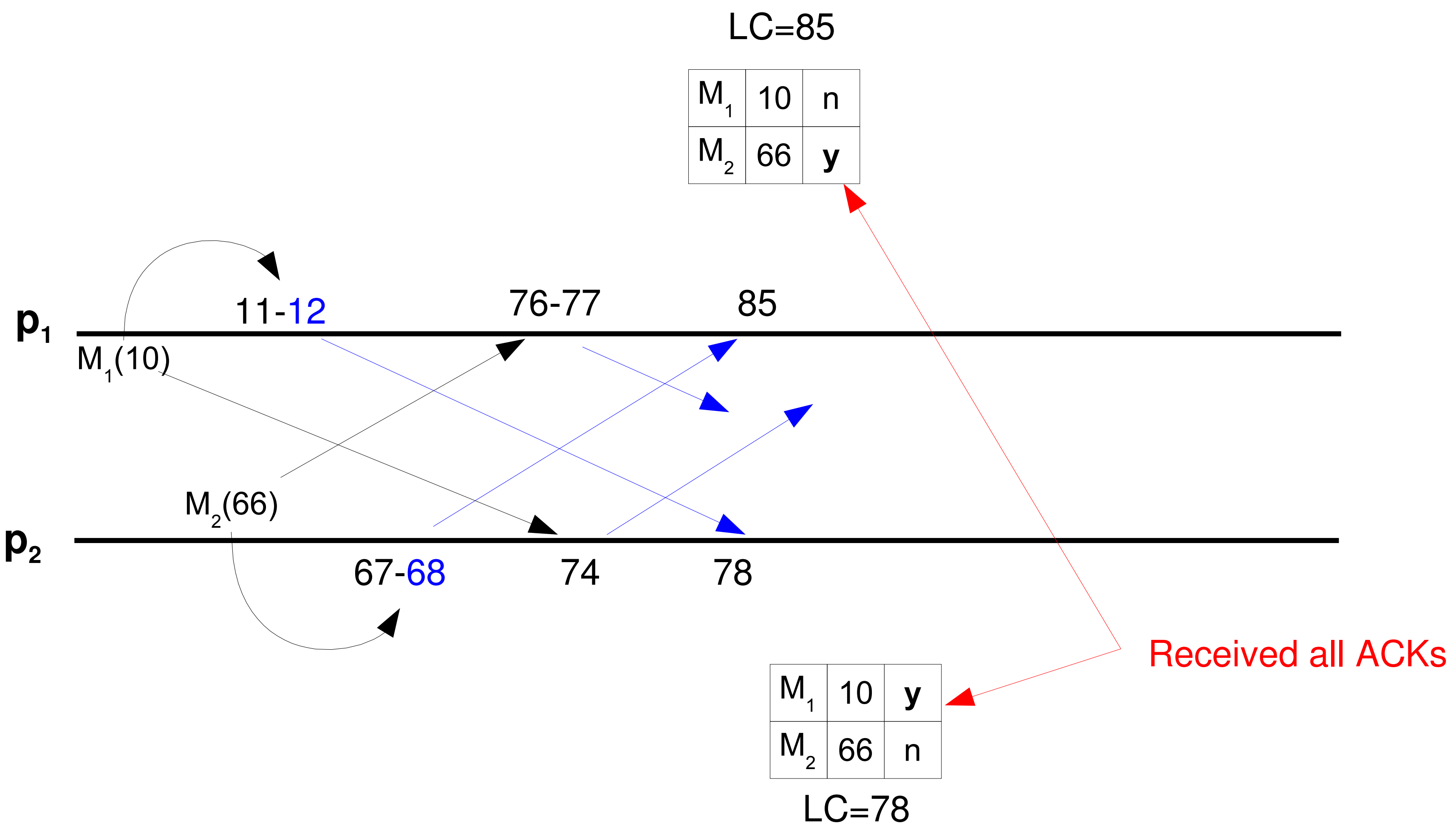
Olivier.Gruber@inria.fr

Totally Ordered Multicast

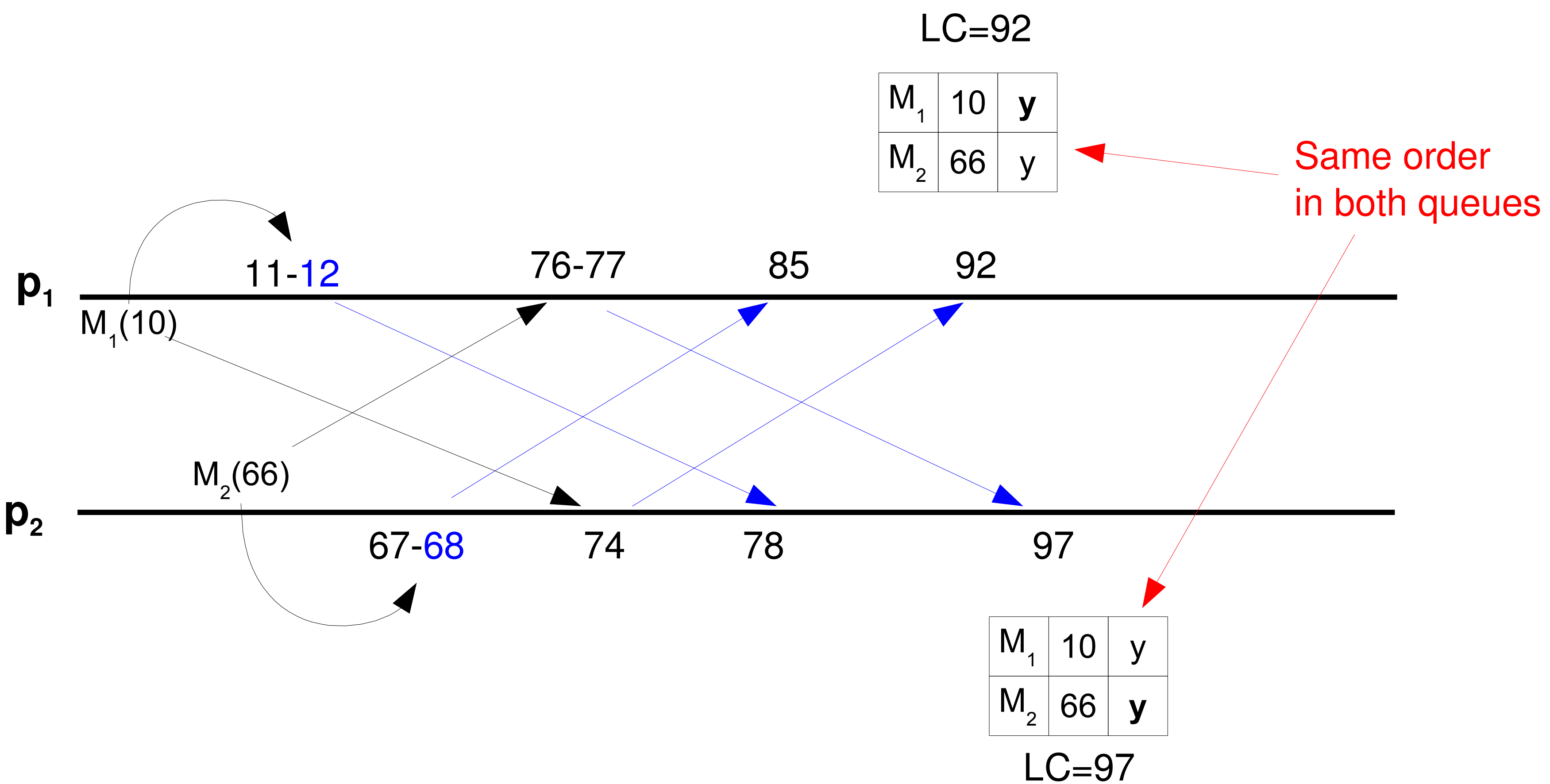


Olivier.Gruber@inria.fr

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

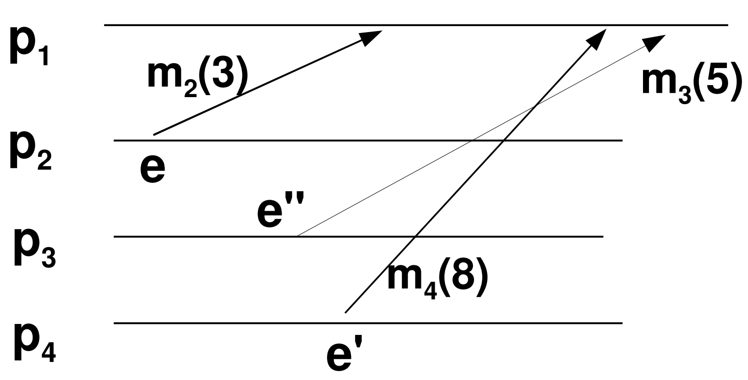
Logical Clock Limits

- 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

- Point-to-Point Causality

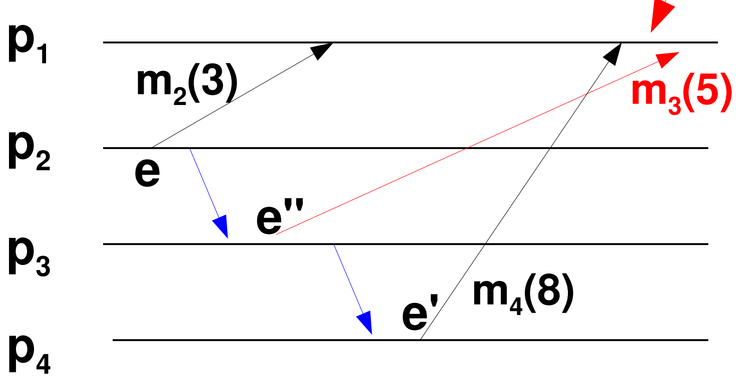
send (m) → send (m')
⇒$deliver_i$(m) → $deliver_i$(m')

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



violates causality

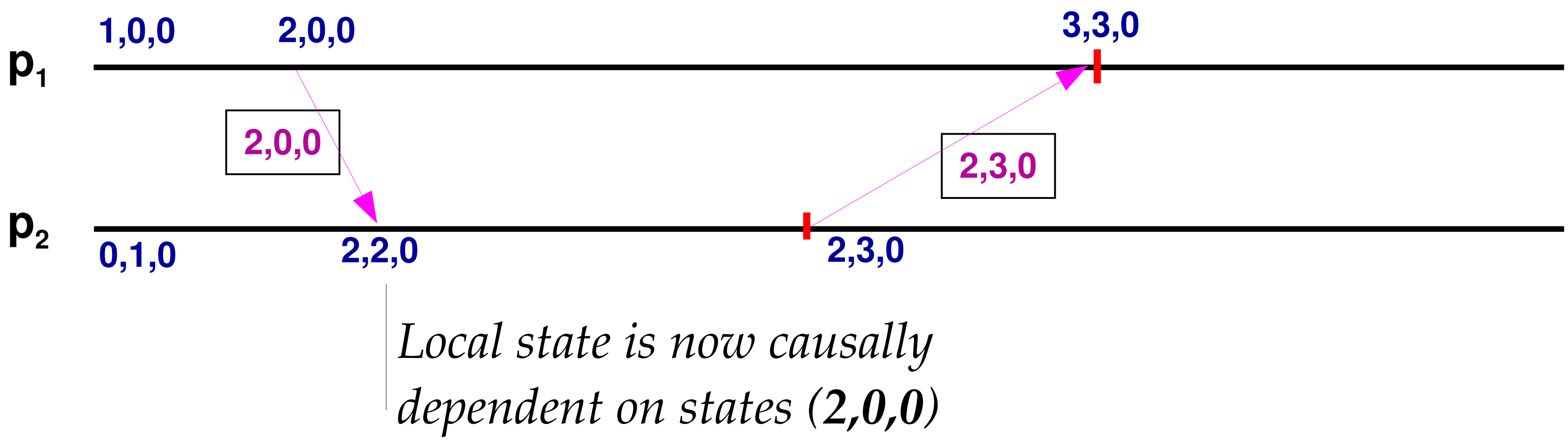
$(e_2 \rightarrow e''_3)$ and $(e''_3 \rightarrow e'_4)$



Olivier.Gruber@inria.fr

Vector Clocks

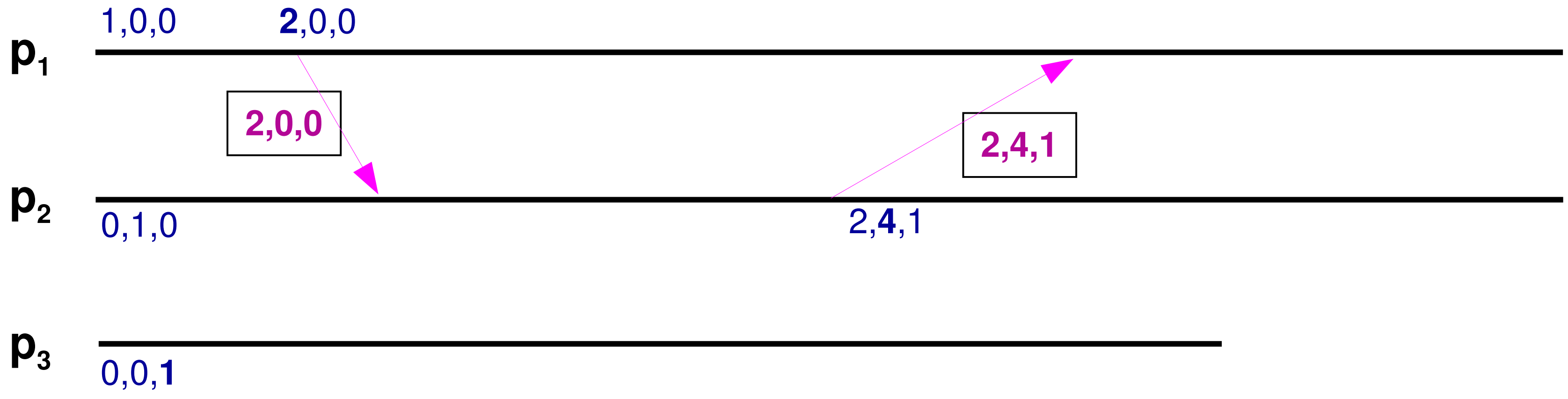
- Vector Clock (Fidge and Mattern, 1988)
  - A vector of logical clocks
    - One entry per known process P_i
    - VC[i] = max value of known LC(P_i)
  - Each event carries a vector clock
    - It gives the history at various processes that the event depends on
  - Each process P_i maintains a vector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



Olivier.Gruber@inria.fr

Vector Clock Management

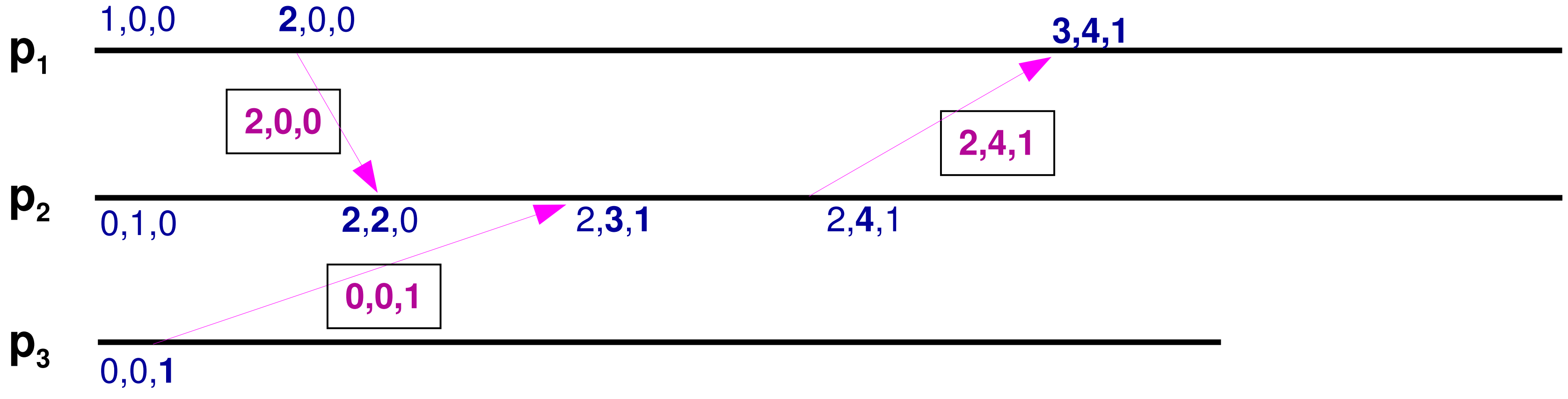
- For each local event, increment local logical clock
  - $VC_i[i] = VC_i[i] + 1$
- Sending messages
  - It is a local event, so increment local logical clock
  - Timestamp messages with its VC_i



Olivier.Gruber@inria.fr

Vector Clock Management

- For each local event, increment local logical clock
  - $VC_i[i] = VC_i[i] + 1$
- Receiving messages with a vector clock VC_m
  - $VC_i[k] = \max(VC_i[k], VC_m[k])$ for all $k \neq i$
  - Increment local logical clock $VC_i[i]$

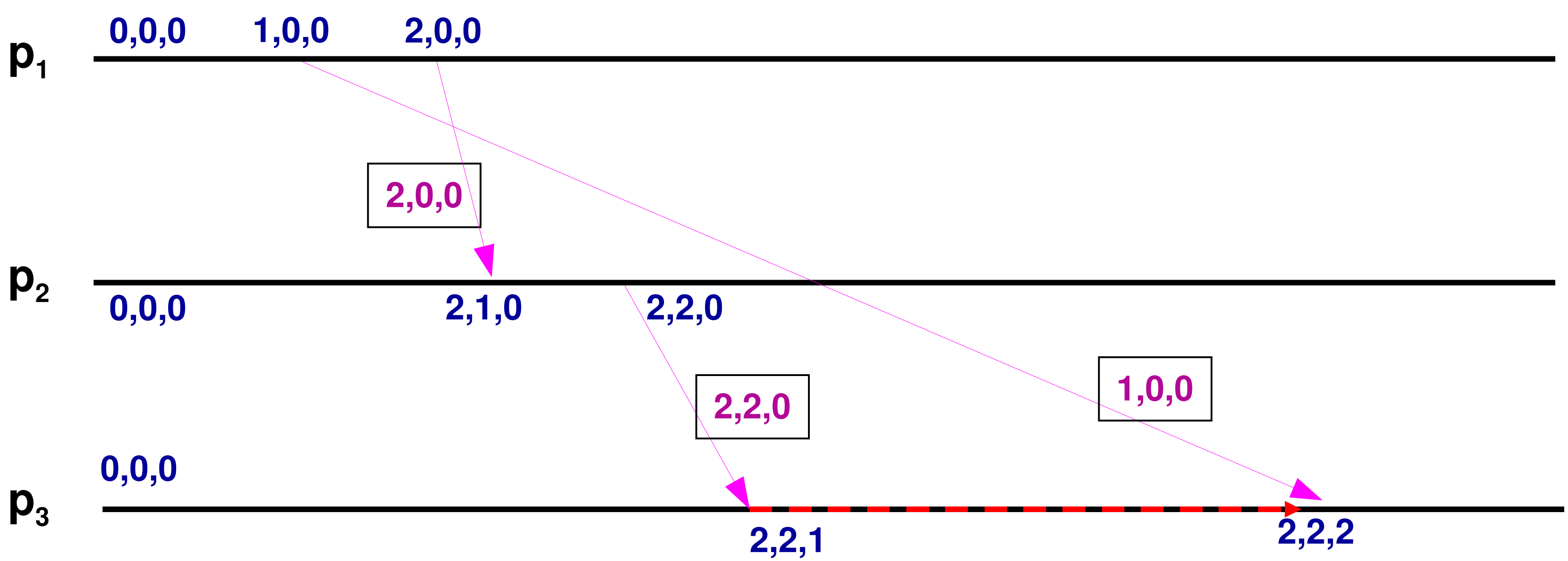


Olivier.Gruber@inria.fr

Vector Clocks

- Unicast (point-to-point messages)
  - Vector clocks are not enough to capture point-to-point causality

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

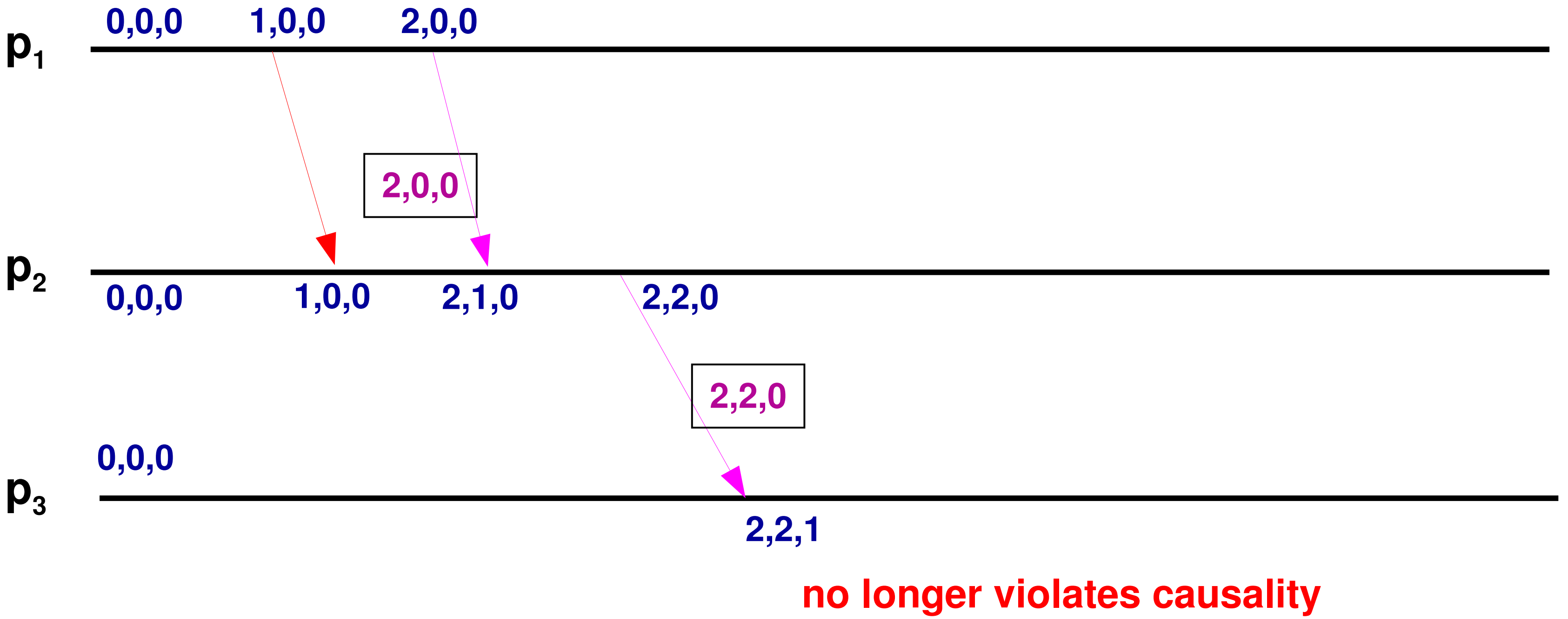


violates causality
the vector clock does not carry any knowledge of late messages

Olivier.Gruber@inria.fr

Vector Clocks

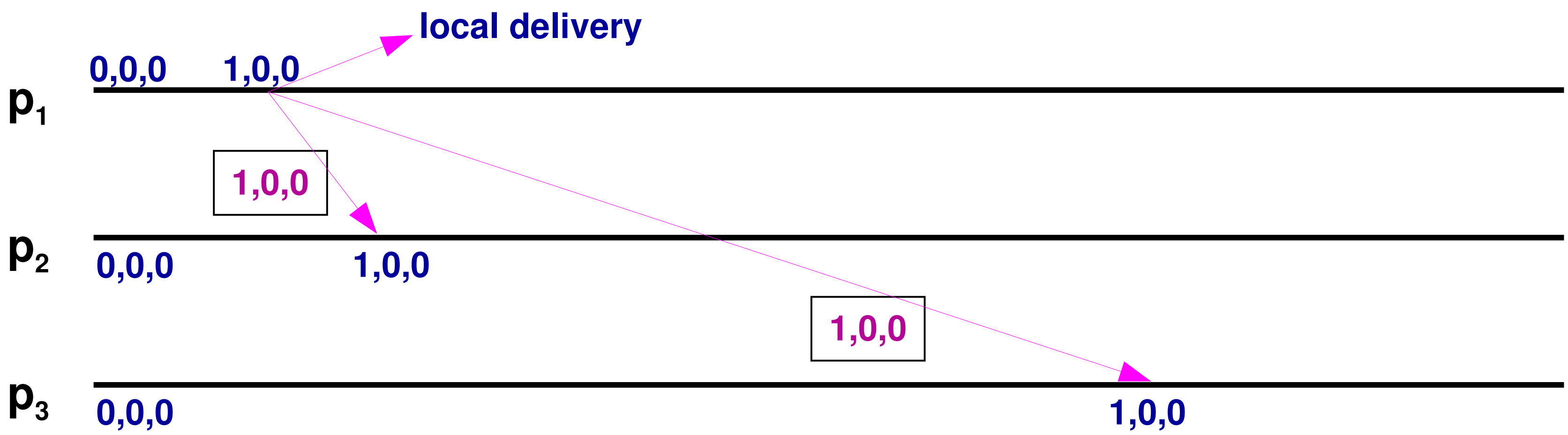
- Unicast (point-to-point messages)
  - Correct execution if P_1 sent the first message to another process than P_3
  - Non-distinguishable from P_3 perspective



Olivier.Gruber@inria.fr

Causally-Ordered Multicast

- Causally Ordered Multicast
  - Vector clocks are not enough to capture point-to-point causality
  - But they can be used for causally-ordered multicast
    - Use vector clocks to know how long to delay message delivery
  - Causally ordered multicast imposes a weaker order than the totally ordered multicasting with logical clocks
    - ***Thus, it performs better ! No ACKs***
    - ***Immediate local delivery of a message when multicasting it***

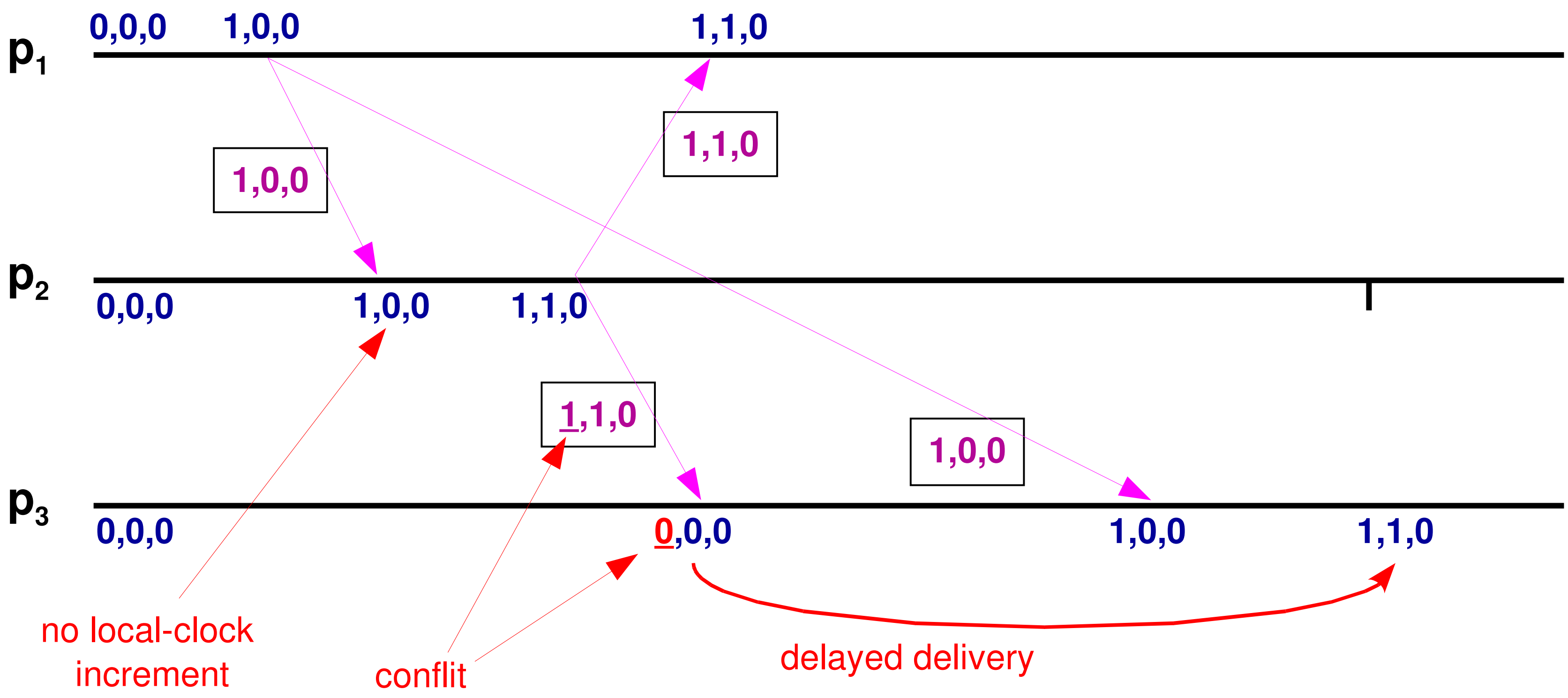


Olivier.Gruber@inria.fr

Causally-Ordered Multicast

- Modified Vector Clock Design
  - 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

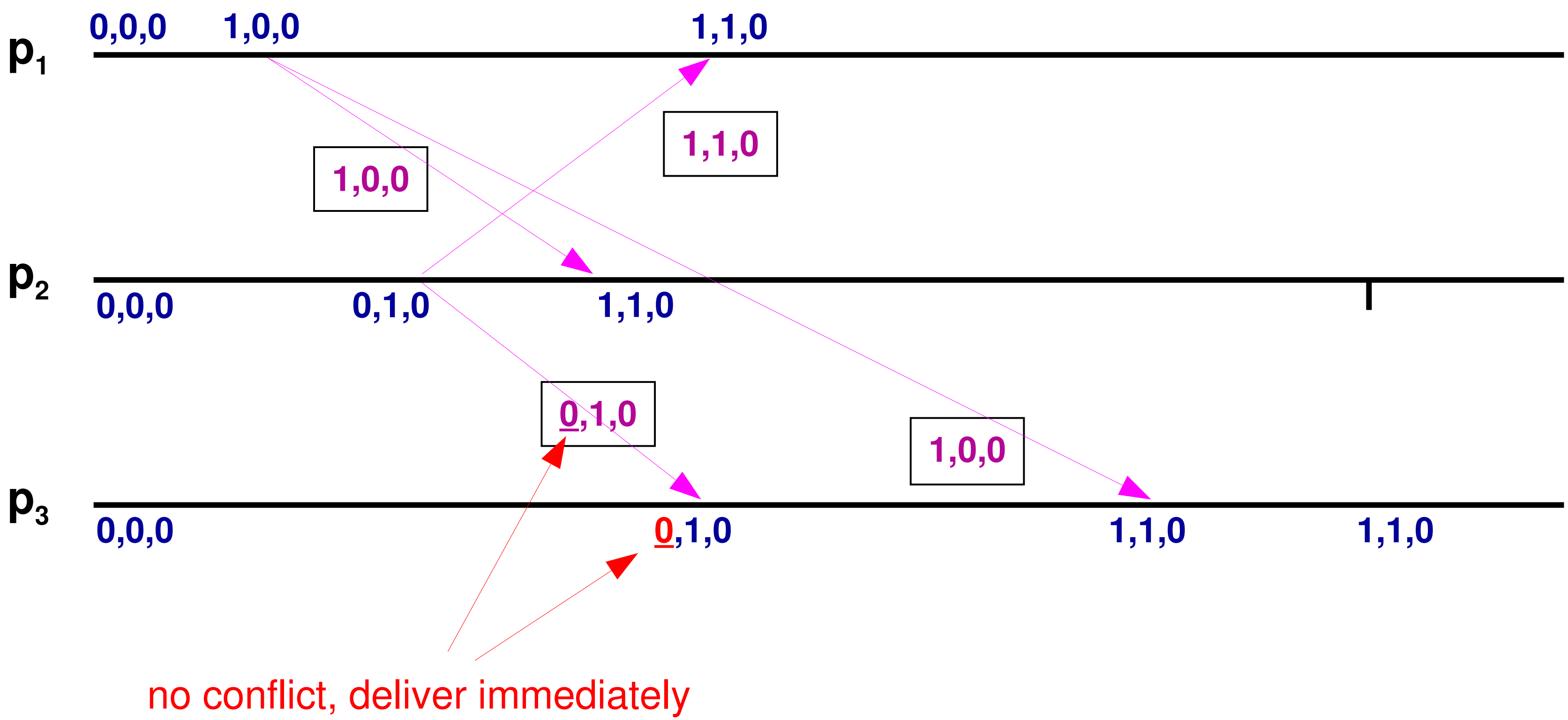
Delay delivery until

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

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

Olivier.Gruber@inria.fr

Causally-Ordered Multicast

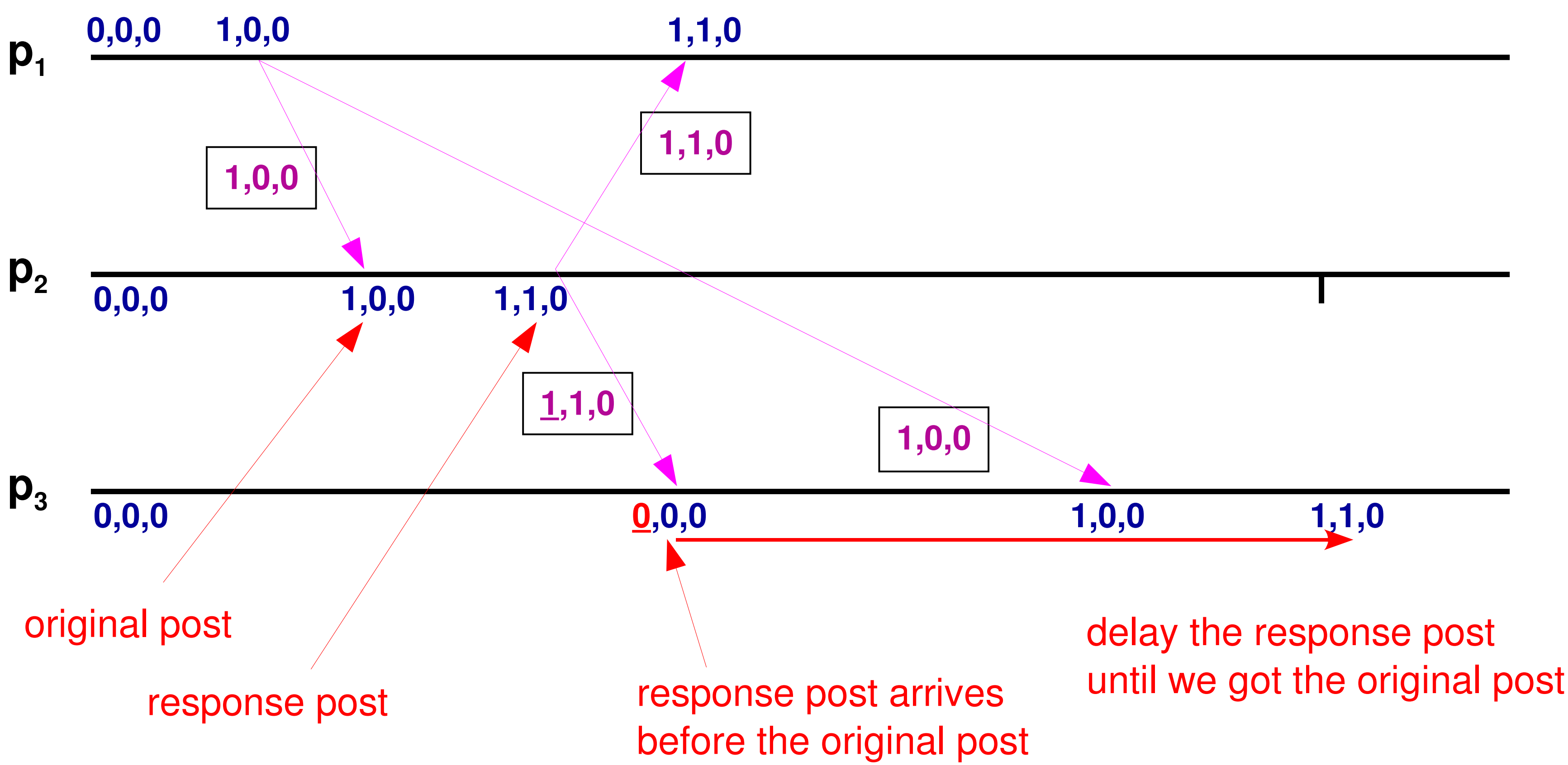


Notice that
we avoided all the acknowledgment messages
of the totally-ordered multicast

Olivier.Gruber@inria.fr

Causally-Ordered Multicast

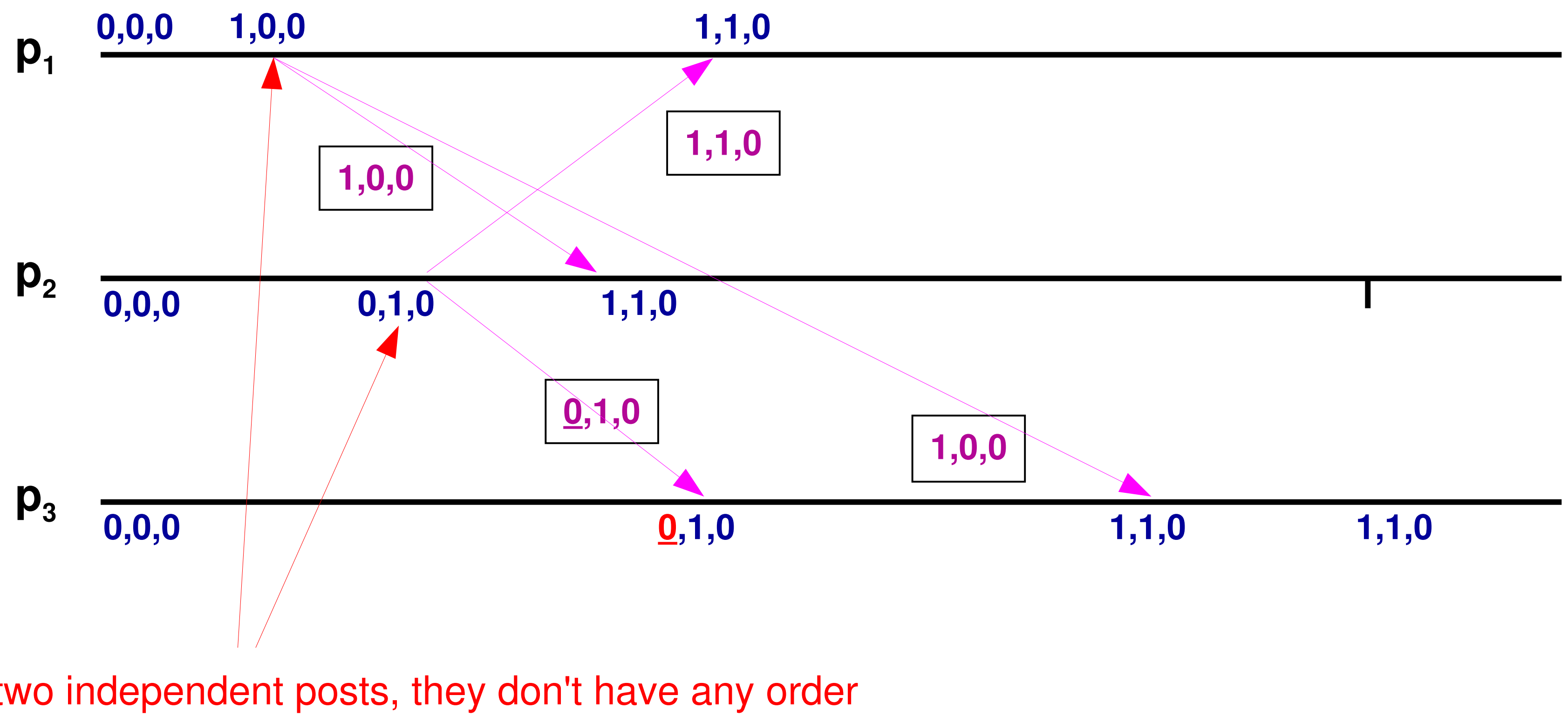
- Example: newgroups
  - We want to avoid response posts to appear before the original posts



Olivier.Gruber@inria.fr

Causally-Ordered Multicast

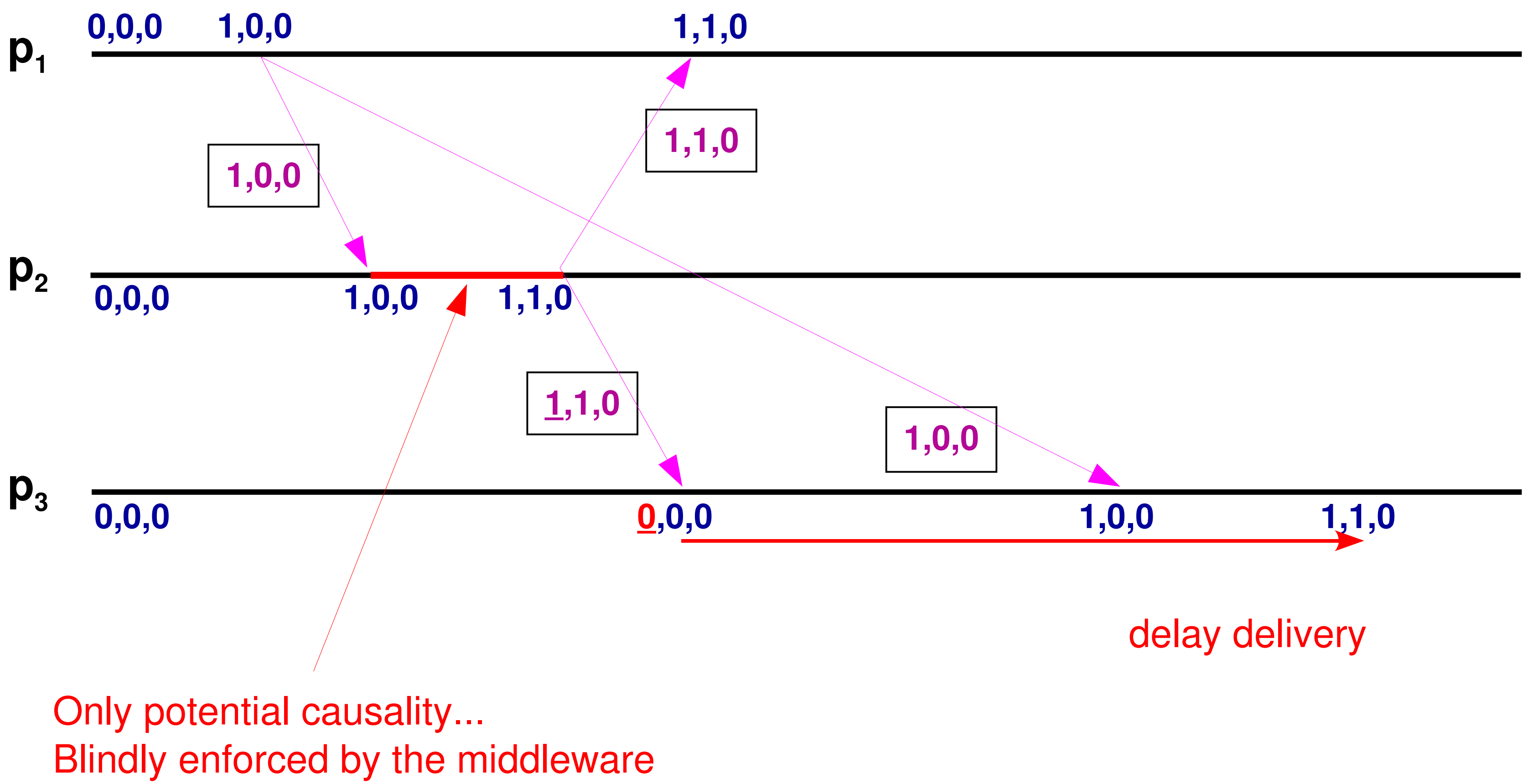
- Example: newsgroup
  - But we don't need to order original posts...



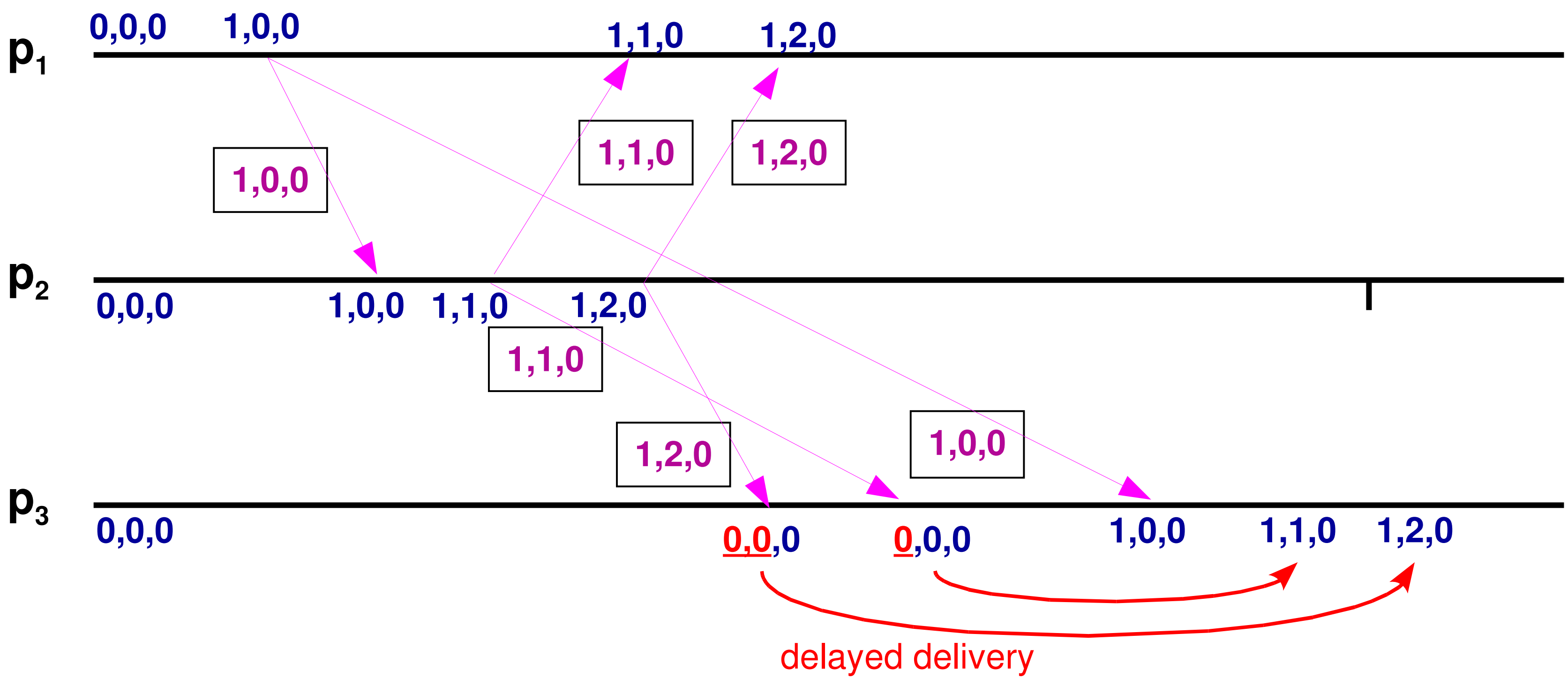
Olivier.Gruber@inria.fr

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 $k \neq s$

Olivier.Gruber@inria.fr

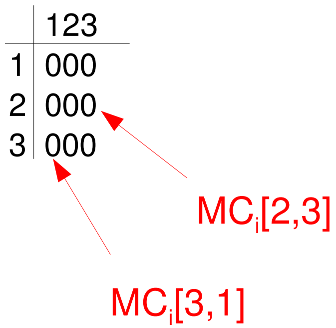
Matrix Clocks

- Towards 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

Olivier.Gruber@inria.fr

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

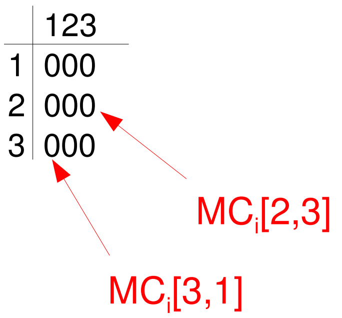


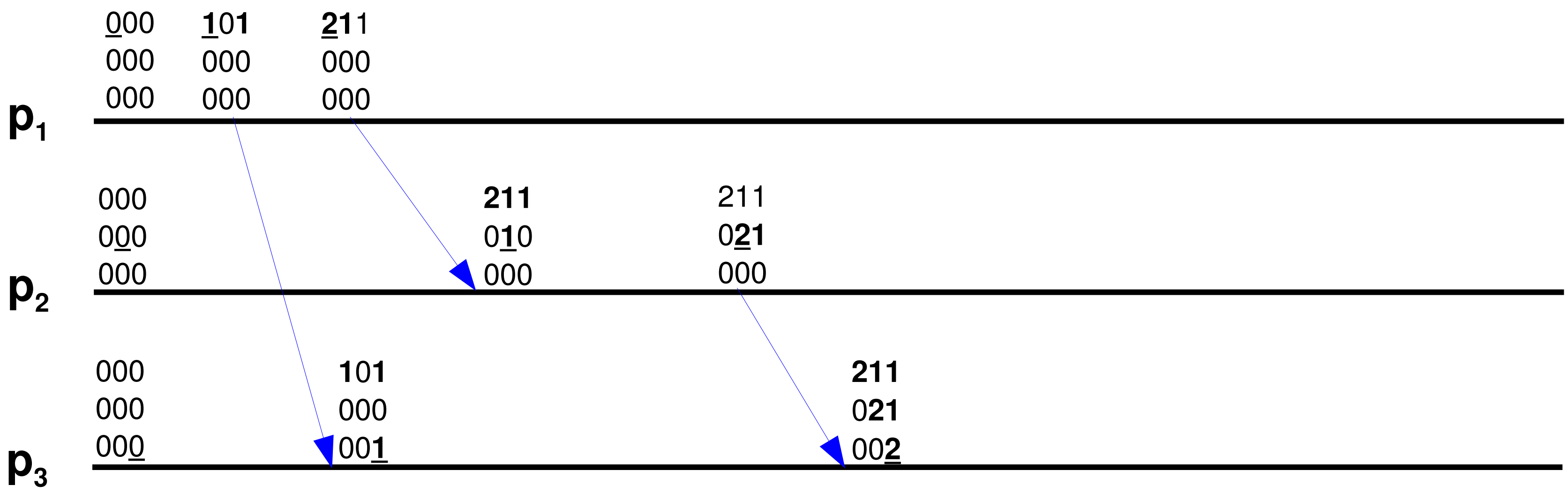
- Matrix definition
  - $MC_i[j,k]$ = number of messages sent by P_j to P_k that P_i causally knows about
    - A column k represents what a process P_k has received from other processes P_j that P_i knows about
  - $MC_i[i,i]$ = local events (local logical clock)

Olivier.Gruber@inria.fr

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)

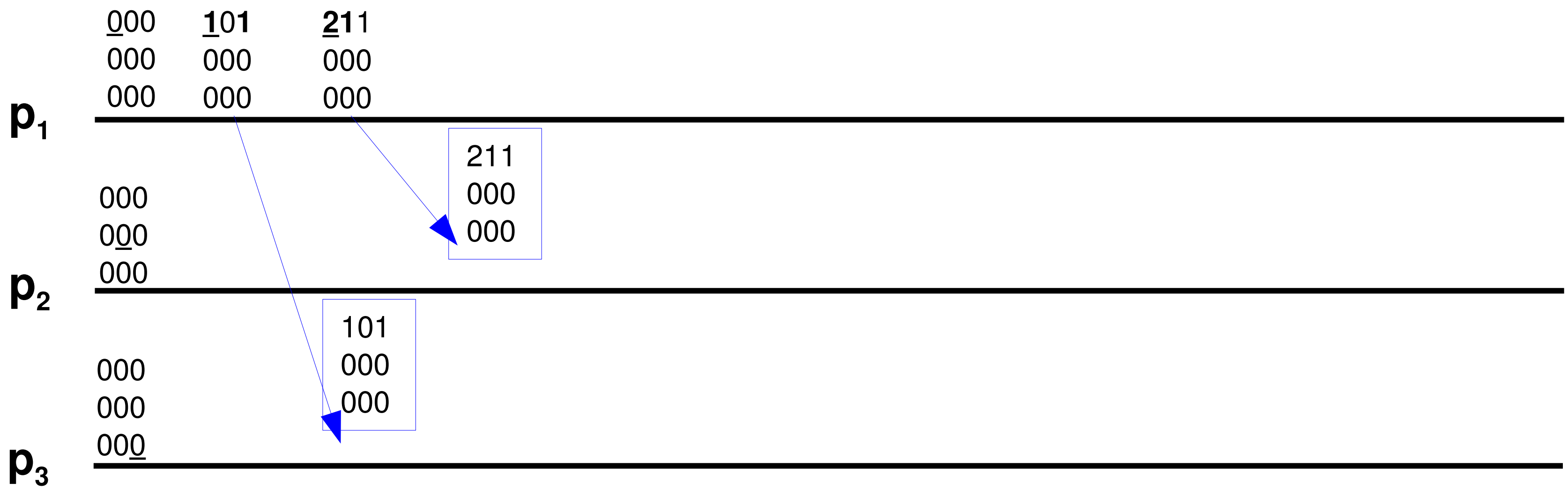




Olivier.Gruber@inria.fr

Matrix Clocks – Rules

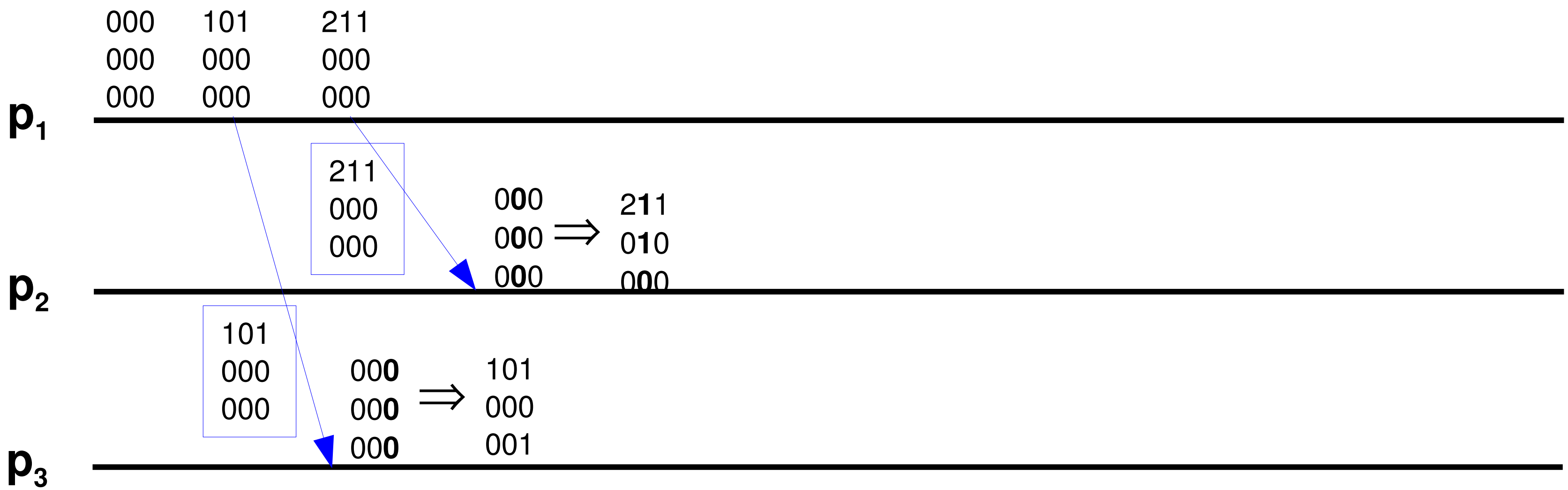
- Local Event:
  - $MC_i[i,i] = MC_i[i,i] + 1$
- Sending a message from P_i towards P_k
  - $MC_i[i,k] = MC_i[i,k] + 1$
  - $MC_i[i,i] = MC_i[i,i] + 1$



Olivier.Gruber@inria.fr

Matrix Clocks – Rules

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

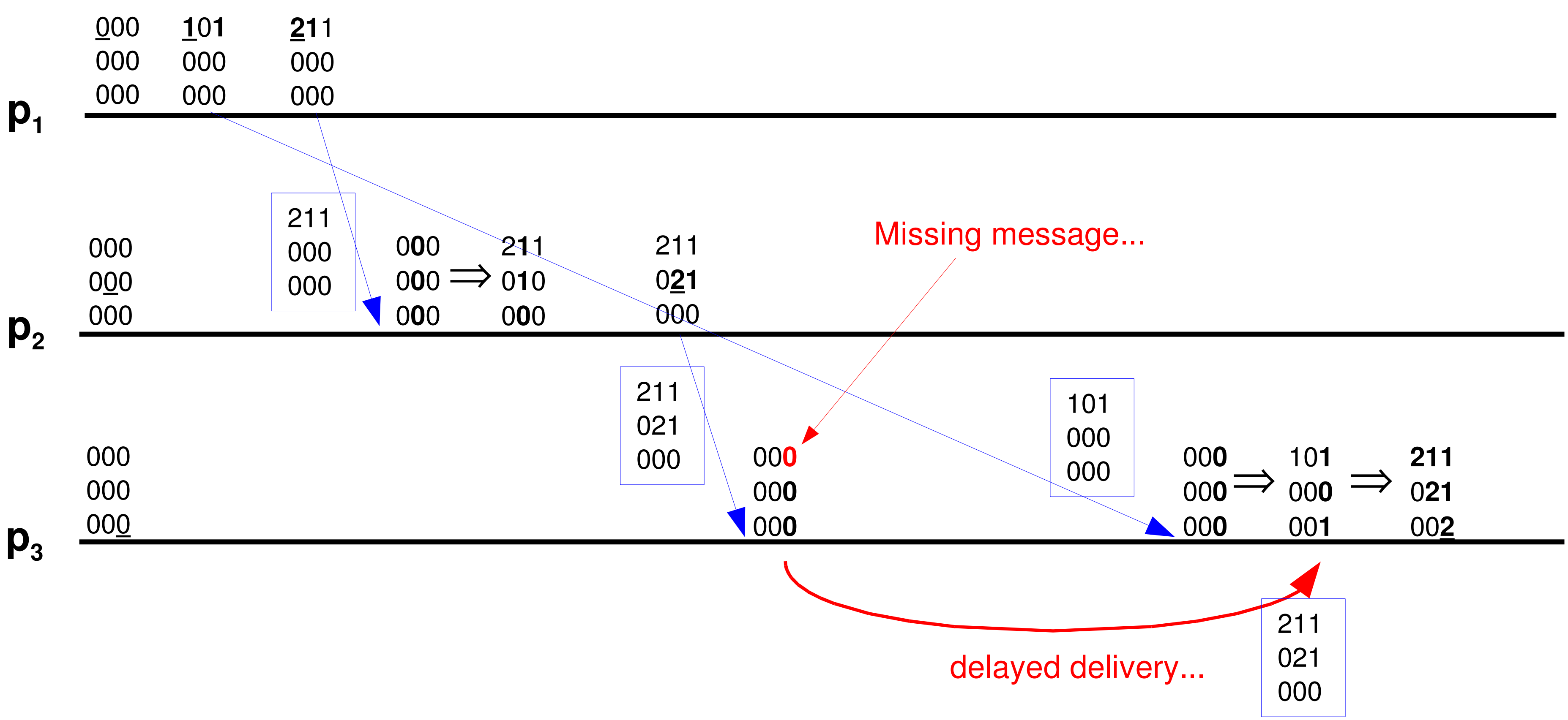


Olivier.Gruber@inria.fr

Matrix Clock

- Point-to-Point causality

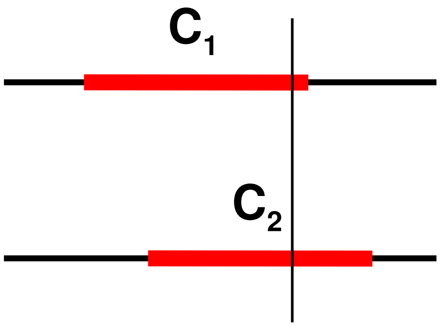
send (m) $\rightarrow$ send (m')
$\Rightarrow$ deliver$_i$ (m) $\rightarrow$ deliver$_i$ (m')



Olivier.Gruber@inria.fr

Mutual Exclusion

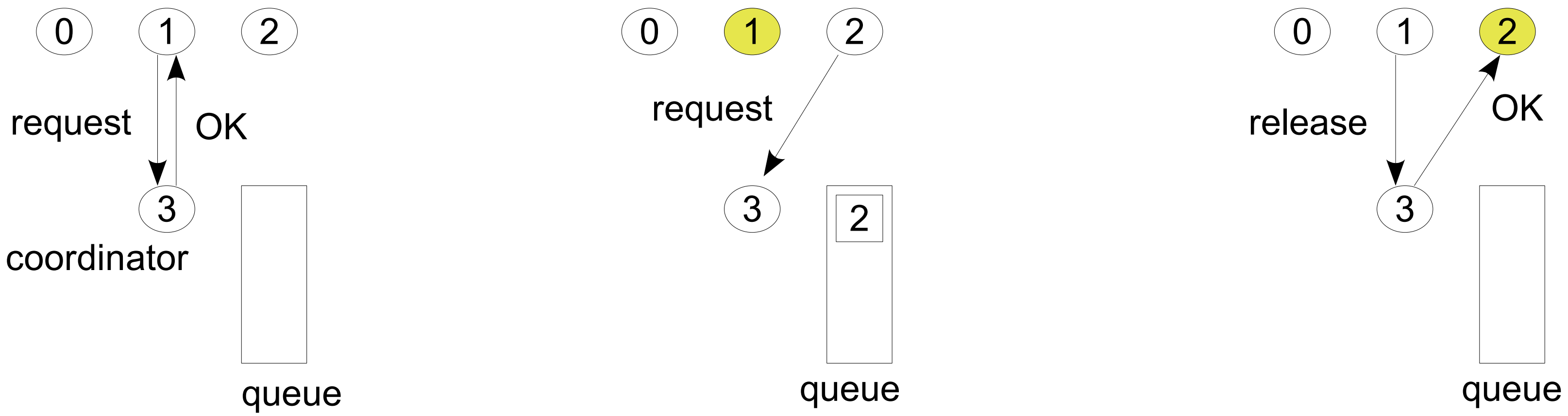
- Critical Section
  - Leave(C2) happens-before Enter(C1)
  - Leave(C1) happens-before Enter(C2)
  - Without global time, how do we tell?
- Can we do it know?
  - We will look at a centralized version
  - Then a distributed one using logical clocks
  - Finally, one using a token



Olivier.Gruber@inria.fr

Mutual Exclusion

- Centralized approach
  - Simulate what happens in one-processor system
    - Elect one process as a coordinator
  - Principle
    - The coordinator grants the critical section if available
    - When not available, it queues the requesting processes
    - When critical section is freed, it schedules the first process in the queue



Olivier.Gruber@inria.fr

Mutual Exclusion

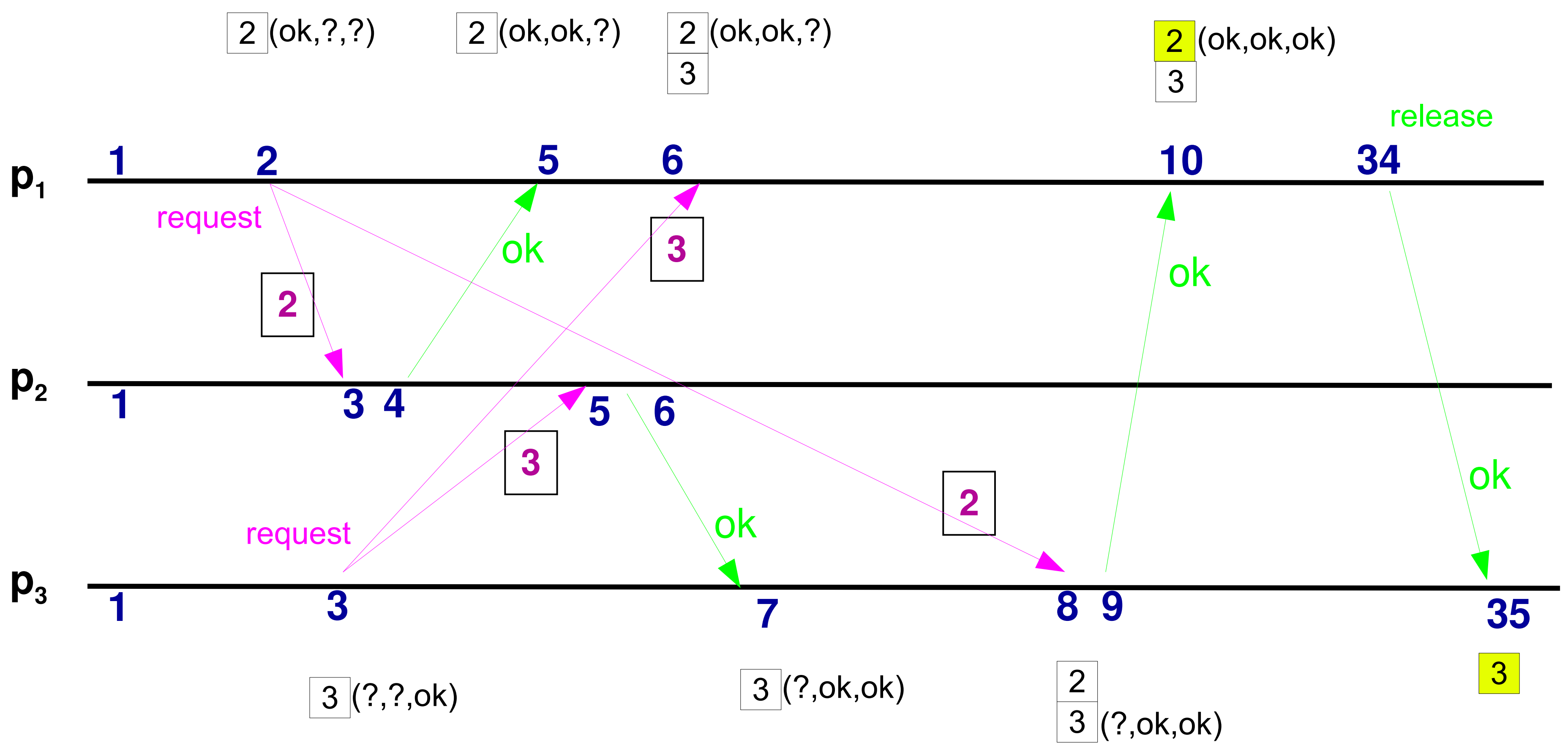
- Ricart and Agrawala (1981)
  - N processes
    - Interconnected with reliable FIFO channels
  - Requires a total ordering of all events
    - We use extended logical clock
      - When we had:
        - $LC(e_{32}^{k}) = 56$ and $LC(e_{24}^{k}) = 56$
      - We now have
        - $LC(e_{32}^{k}) = 56.32$ and $LC(e_{24}^{k}) = 56.24$
  - Basic idea
    - Each access request to a resource has a logical timestamp
    - Processes are granted access in the order of the logical timestamps of their requests
    - Real close to the principle of the totally-ordered multicast

Olivier.Gruber@inria.fr

Mutual Exclusion

- Principle
  - Each process multicast its requests to all other processes
    - Waits for granted access from all processes
    - When it has granted access from all, it has access to the resource
  - Upon receiving a request
    - If the request receiver is not accessing the resource
      - It grants access
    - If the request receiver has already exclusive access to the resource
      - It queues the request with no reply
  - Upon release
    - The owner will grant all pending requests

Mutual Exclusion



Olivier.Gruber@inria.fr

Mutual Exclusion

- Token-based approach
  - Overlay ring, no matters what the real network topology is
  - There is only one token, going around the ring
  - The token represents the granted access to a shared resource
- Principle
  - A site enters the critical section
    - Waits for the token to arrive (granted access)
  - Accesses the resource
    - When done, releases the token onto the ring (next process)

Olivier.Gruber@inria.fr

Mutual Exclusion

- Token-based approach
  - Starvation must be avoided
  - Temptation
    - Allow local reuse of the token if the critical section is locally requested upon its release…
    - Rationale: avoids potentially going around the ring for nothing
  - Danger
    - Potentially leads to starvation
  - Possible solution
    - Limit the re-use of the token locally

Mutual Exclusion

Algorithm	Messages per entry/exit	Delay before entry	Problems
Centralized	3 messages	2 messages	Coordinator crash
Distributed	2(n-1) messages	2(n-1) messages	Crash of any node
Token ring	From one to unbounded	From 0 to n-1	Lost token

no one wants the CS
token goes around and around
but just waste a little bandwidth...

Slower, more expensive, more fragile... why bother?
Shows it is possible to approach it as a distributed design
It is still open research to do better...

Discussing Failures

- Examples of failures
  - Messages may be lost or delayed enormously
  - Machines or processes may fail
  - Impossible to detect the difference in practice
- Difficult problem
  - None of the above algorithms resist failures
    - Messages must be delivered in bounded time
    - Processes and machines must not fail
  - In practice, the centralized approach is the more robust
    - Simple failure detector based on the heart-beart technique
    - Re-elect a coordinator if a failure is detected

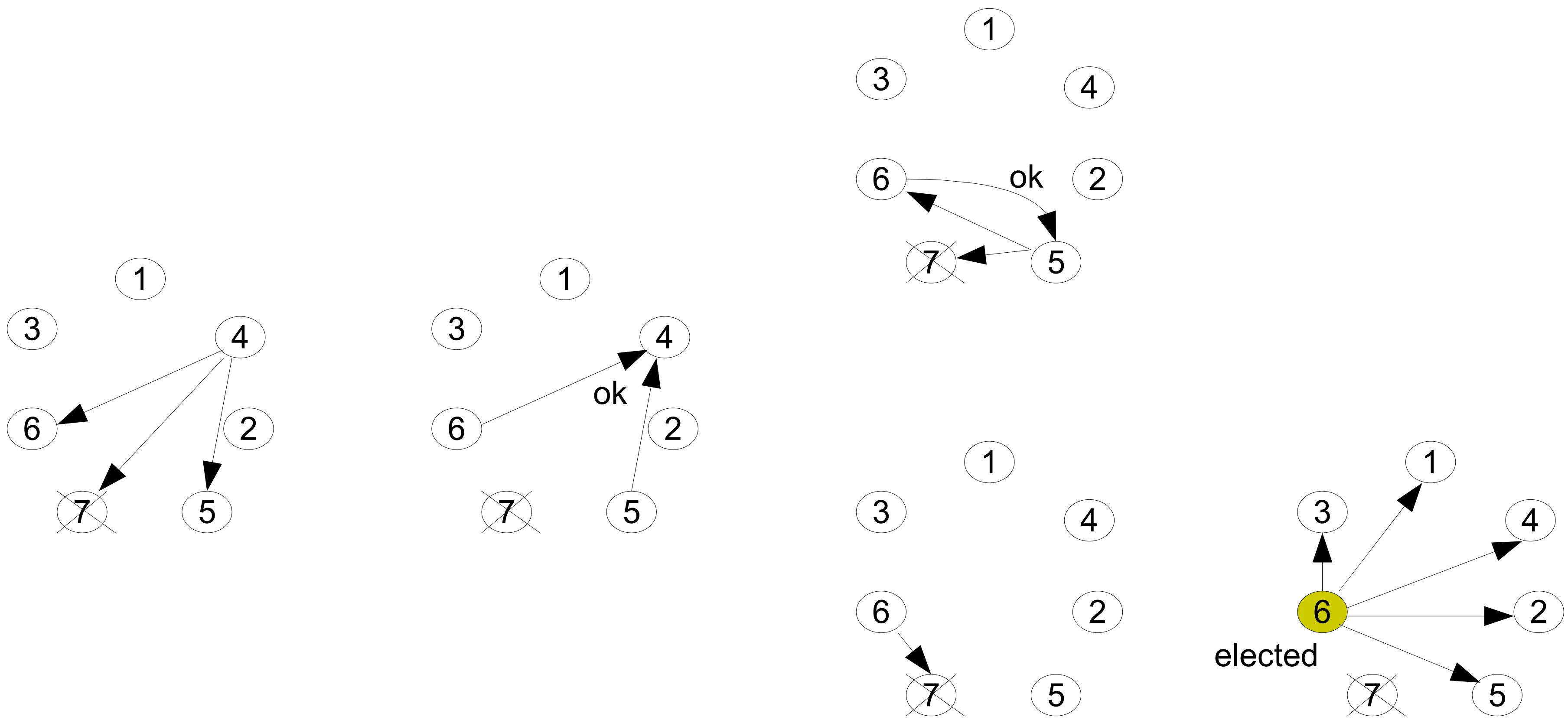
The Election Challenge

- 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

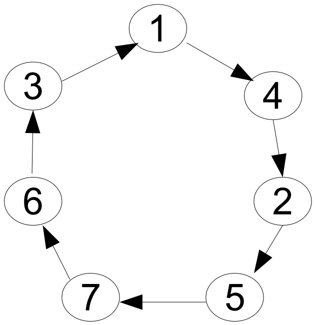
Bully Algorithm



Olivier.Gruber@inria.fr

Election Algorithms

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

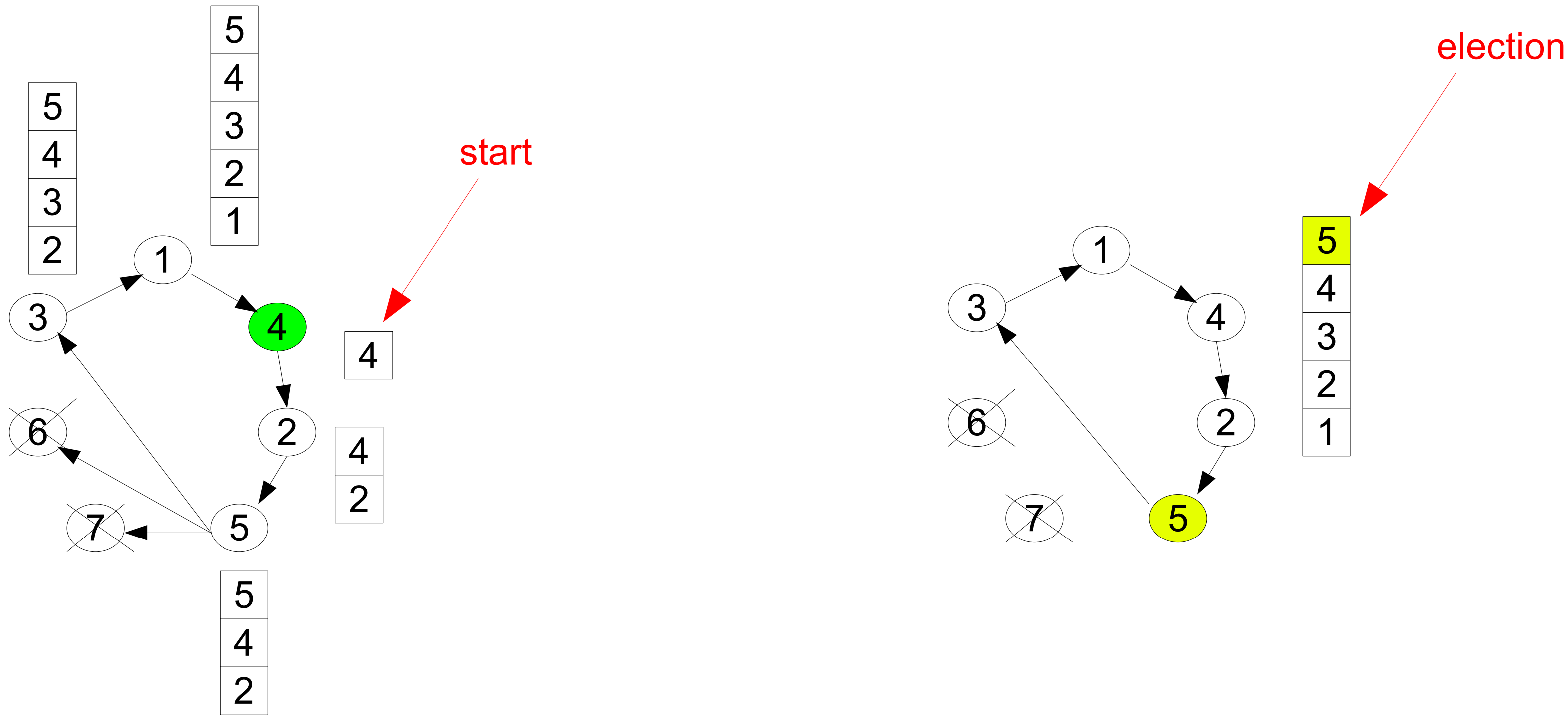


Election Algorithms

- 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)

Olivier.Gruber@inria.fr

Ring Algorithm



Olivier.Gruber@inria.fr

Discussing 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
- Requirements for previous algorithms
  - Messages must be delivered in bounded time (no loss)
  - Processes may only fail-stop

Olivier.Gruber@inria.fr