Time and Clock Synchronization in Cloud Data Centers

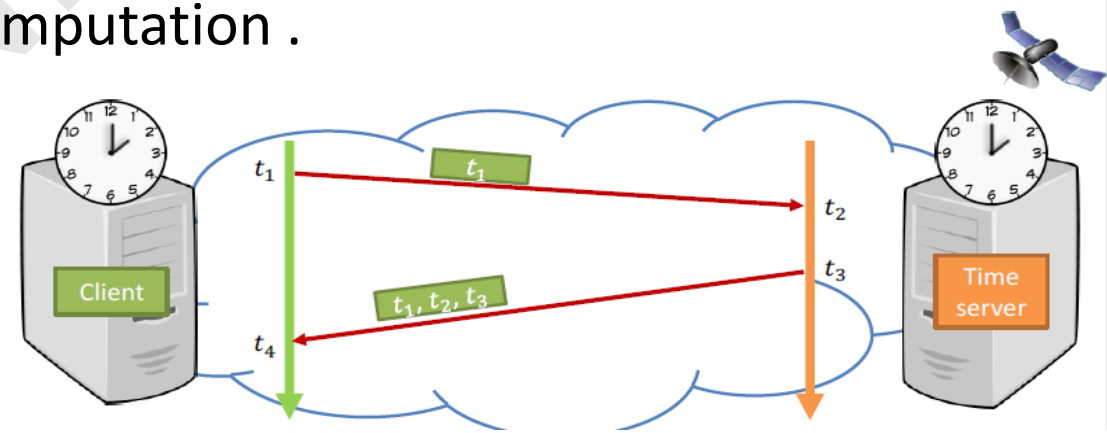
Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss the fundamentals of clock synchronization in cloud and its different algorithms.

- We will also discuss the causality and a general framework of logical clocks and present two systems of logical time, namely, lamport and vector, timestamps to capture causality between events of a distributed computation .



Need of Synchronization

- **You want to catch a bus at 9.05 am, but your watch is off by 15 minutes**
  - What if your watch is Late by 15 minutes?
    - You'll miss the bus!
  - What if your watch is Fast by 15 minutes?
    - You'll end up unfairly waiting for a longer time than you intended

- **Time synchronization is required for:**
  - **Correctness**
  - **Fairness**

Time and Synchronization

- **Time and Synchronization**

  ("There's is never enough time…")

- **Distributed Time**
  - The notion of time is well defined (and measurable) at each single location
  - But the relationship between time at different locations is unclear

- **Time Synchronization is required for:**
  - **Correctness**
  - **Fairness**

Synchronization in the cloud

Example: Cloud based airline reservation system:

- Server X receives, a client request, to purchase the last ticket on a flight, say PQR 123.
- Server X timestamps the purchase using its local clock as **6h:25m:42.55s**. It then logs it. Replies ok to the client.
- That was the very last seat, Server X sends a message to Server Y saying the "flight is full".
- Y enters, "Flight PQR 123 is full" + its own local clock value, (which happens to read **6h:20m:20.21s**).
- Server Z, queries X's and Y's logs. Is confused that a client purchased a ticket at X after the flight became full at Y.
- **This may lead to full incorrect actions at Z**

Key Challenges

- **End-hosts in Internet based systems (like clouds)**
  - Each have its own clock
  - Unlike processors (CPUs) within one server or workstation which share a system clock.
- **Processes in internet based systems follow an asynchronous model.**
  - No bounds on
    - Messages delays
    - Processing delays
  - Unlike multi-processor (or parallel) systems which follow a **synchronous** system model

Definitions

- An asynchronous distributed system consists of a number of **processes**.
- Each process has a state (**values of variables**).
- Each process takes **actions** to change its state, which may be an instruction or a communication action (**send, receive**).
- An **event** is the occurrence of an action.
- Each process has a large clock – events within a process can be assigned **timestamps**, and thus ordered linearly.
- But- in a distributed system, we also need to know the time order of events **across** different processes.

Space-time diagram

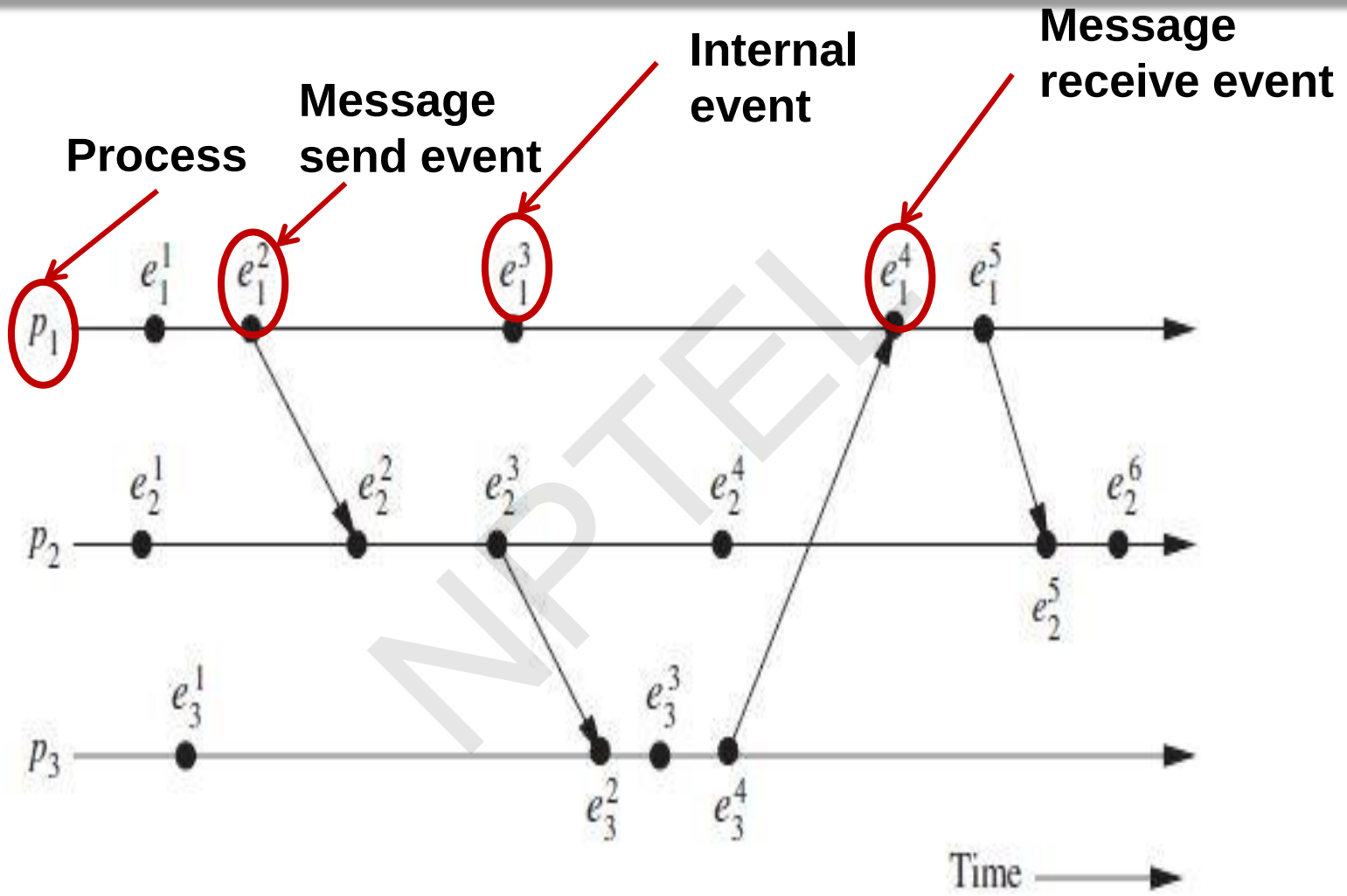


Figure : The space-time diagram of a distributed execution.

Clock Skew vs. Clock Drift

- Each process (running at some end host) has its own clock.
- When comparing two clocks at two processes.
  - **Clock Skew = Relative difference in clock values of two processes.**
    - Like distance between two vehicles on road.
  - **Clock Drift = Relative difference in clock frequencies (rates) of two processes**
    - Like difference in speeds of two vehicles on the road.
- **A non-zero clock skew implies clocks are not synchronized**
- **A non-zero clock drift causes skew increases (eventually).**
  - If faster vehicle is ahead, it will drift away.
  - If faster vehicle is behind, it will catch up and then drift away.

Clock Inaccuracies

- Clocks that must not only be synchronized with each other but also have to adhere to physical time are termed **physical clocks**.

- Physical clocks are synchronized to an accurate real-time standard like **UTC (Universal Coordinated Time).**

- However, due to the clock inaccuracy, **a timer (clock)** is said to be working within its specification if (where **constant ρ** is the **maximum skew rate** specified by the manufacturer)

$$1 - \rho \leq \frac{dc}{dt} \leq 1 + \rho$$

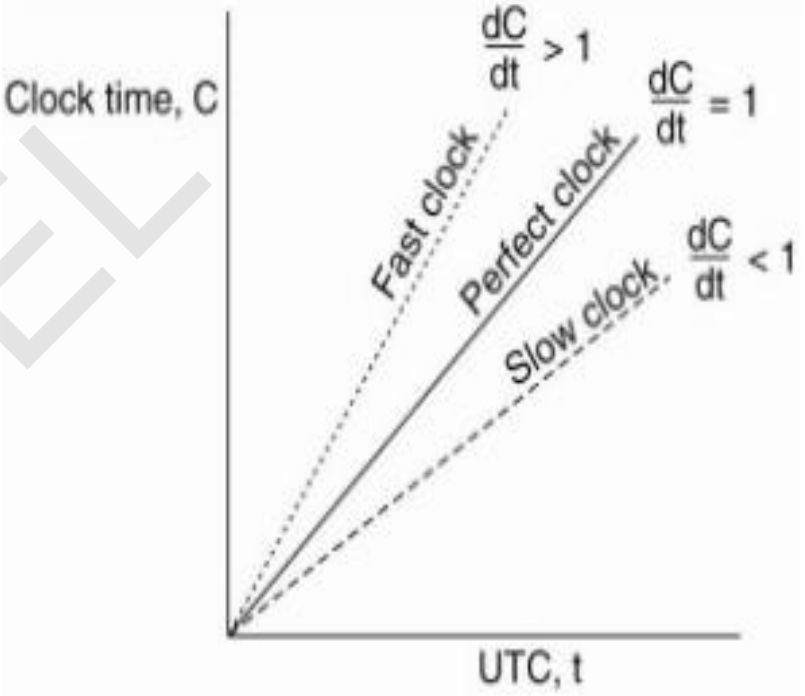


Figure: The behavior of fast, slow, and perfect clocks with respect to UTC.

How often to Synchronize

- **Maximum Drift rate (MDR) of a clock**
- Absolute MDR is defined to relative coordinated universal Time (UTC). UTC is the correct time at any point of time.
    - MDR of any process depends on the environment.
- Maximum drift rate between two clocks with similar MDR is **2*MDR**.
- Given a maximum acceptable skew M between any pair of clocks, need to synchronize at least once every: M/ (2* MDR) time units.
    - Since time = Distance/ Speed.

External vs Internal Synchronization

- Consider a group of processes
- **External synchronization**
  - Each process C(i)'s clock is within a bounded D of a well-known clock S external to the group
  - $|C(i)- S|< D$ at all times.
  - External clock may be connected to UTC (Universal Coordinated Time) or an atomic clock.
  - **Example: Christian's algorithm, NTP**
- **Internal Synchronization**
  - Every pair of processes in group have clocks within bound D
  - $|C(i)- C(j)|< D$ at all times and for all processes i,j.
  - **Example: Berkley Algorithm, DTP**

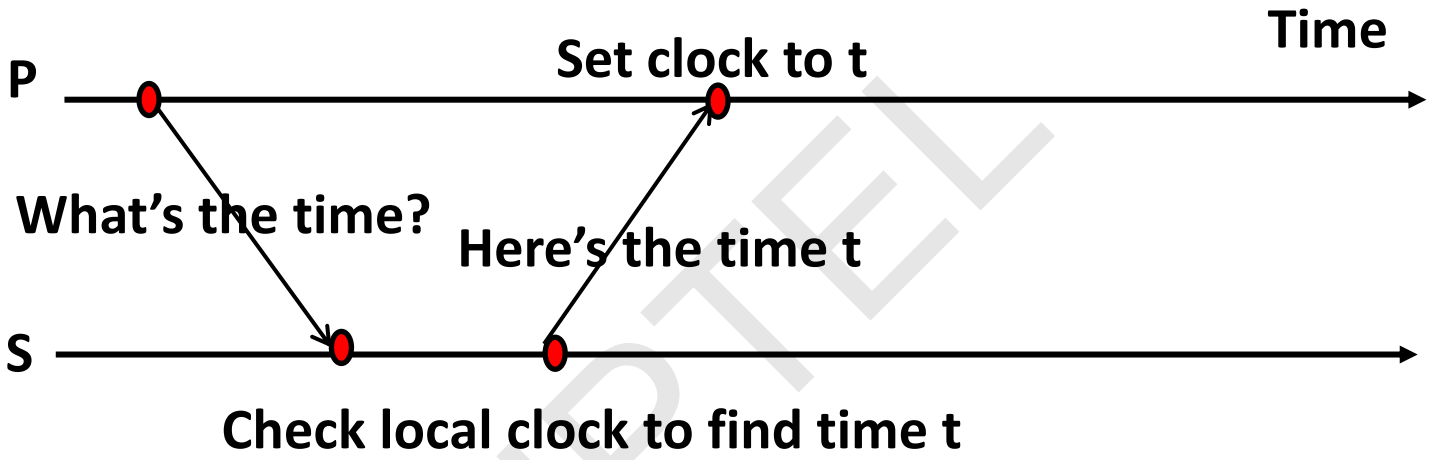
External vs Internal Synchronization

- **External synchronization with D => Internal synchronization with 2*D.**

- **Internal synchronization does not imply External Synchronization.**
  - In fact, the entire system may drift away from the external clock S!

Basic Fundamentals

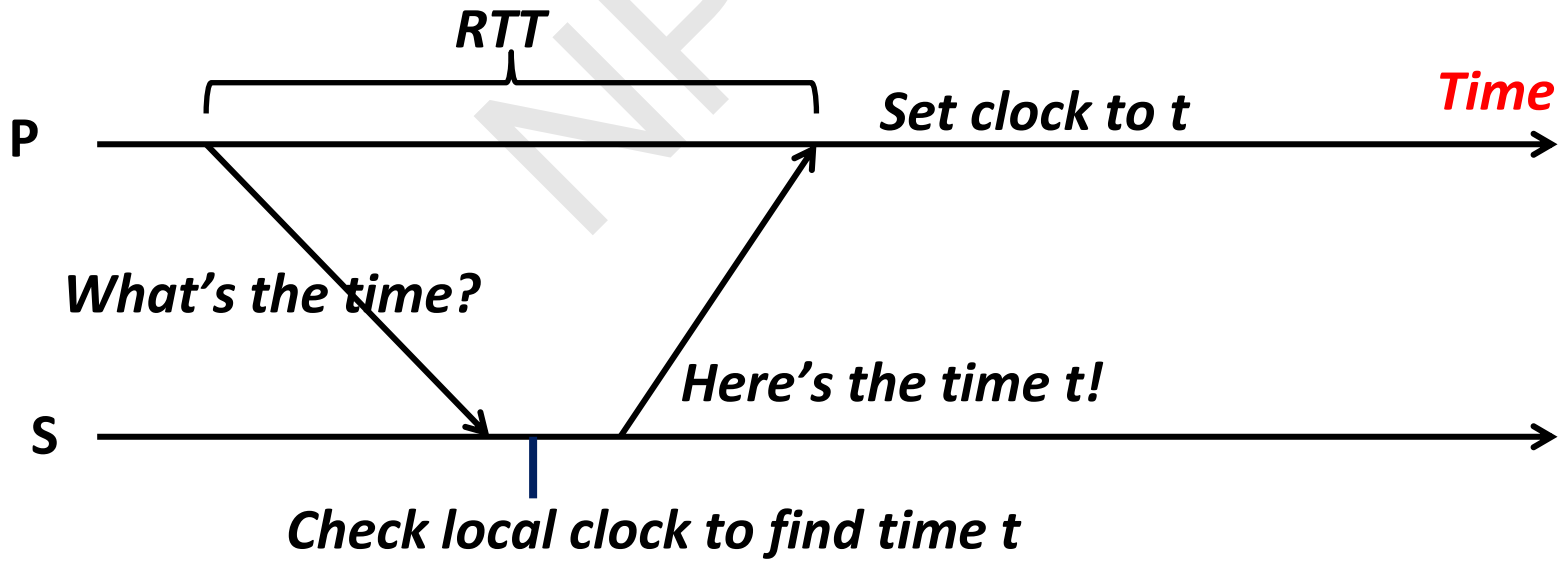
- External time synchronization
- All processes P synchronize with a time server S.



- **What's Wrong:**
  - By the time the message has received at P, time has moved on.
  - P's time set to t is in accurate.
  - Inaccuracy a function of message latencies.
  - Since latencies unbounded in an asynchronous system, the inaccuracy cannot be bounded.

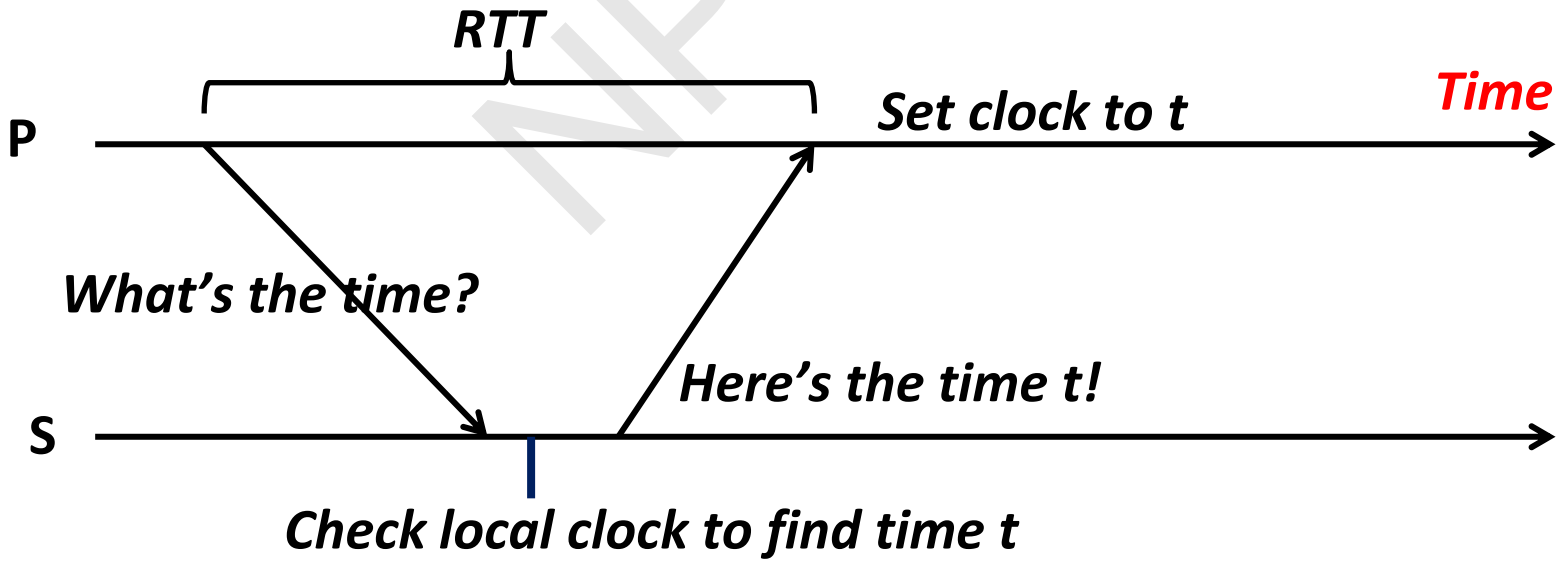
(i) Christians Algorithm

- P measures the round-trip-time RTT of message exchange
- Suppose we know the minimum P → S latency min1
- And the minimum S → P latency min2
  - Min1 and Min2 depends on the OS overhead to buffer messages, TCP time to queue messages, etc.
- The actual time at P when it receives response is between [**t+min2, t + RTT-min1**]

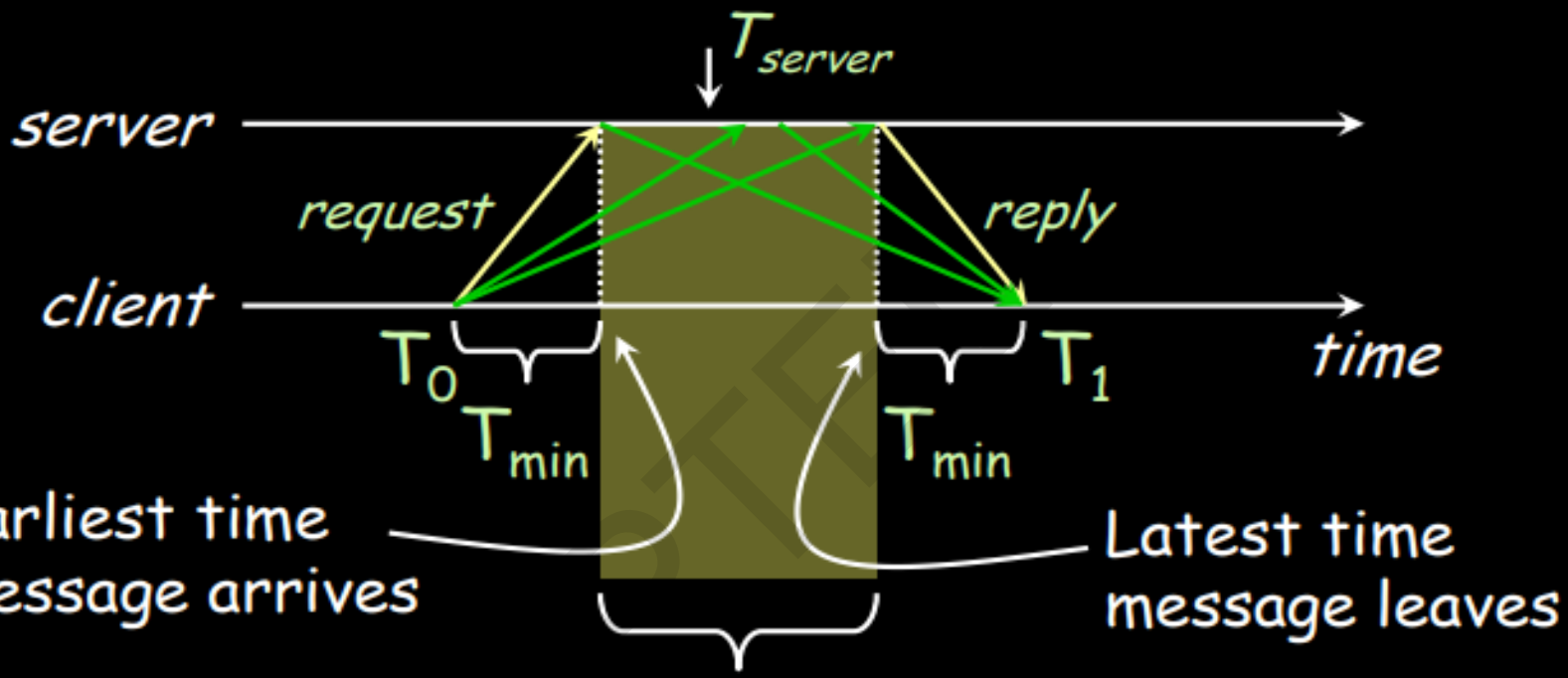


Christians Algorithm

- The actual time at P when it receives response is between [**t+min2, t + RTT-min1**]
- **P sets its time to halfway through this interval**
  - To: t + (RTT+min2-min1)/2
- Error is at most (RTT- min2- min1)/2
  - Bounded



Error Bounds



range = $T_1 - T_0 - 2T_{min}$

accuracy of result = $\pm \frac{T_1 - T_0}{2} - T_{min}$

Error Bounds

- **Allowed to increase clock value but should never decrease clock value**
  - May violate ordering of events within the same process.
- **Allowed to increase or decrease speed of clock**
- **If error is too high, take multiple readings and average them**

Christians Algorithm: Example

- Send request at 5:08:15.100 (T_0)
- Receive response at 5:08:15.900 (T_1)
  - Response contains 5:09:25.300 (T_{server})
- Elapsed time is $T_1 - T_0$
  - 5:08:15.900 - 5:08:15.100 = 800 msec
- Best guess: timestamp was generated
  - 400 msec ago
- Set time to T_{server} + elapsed time
  - 5:09:25.300 + 400 = 5:09.25.700

$$Error = \pm\frac{900-100}{2} - 200 = \pm\frac{800}{2} - 200 = \pm 200$$

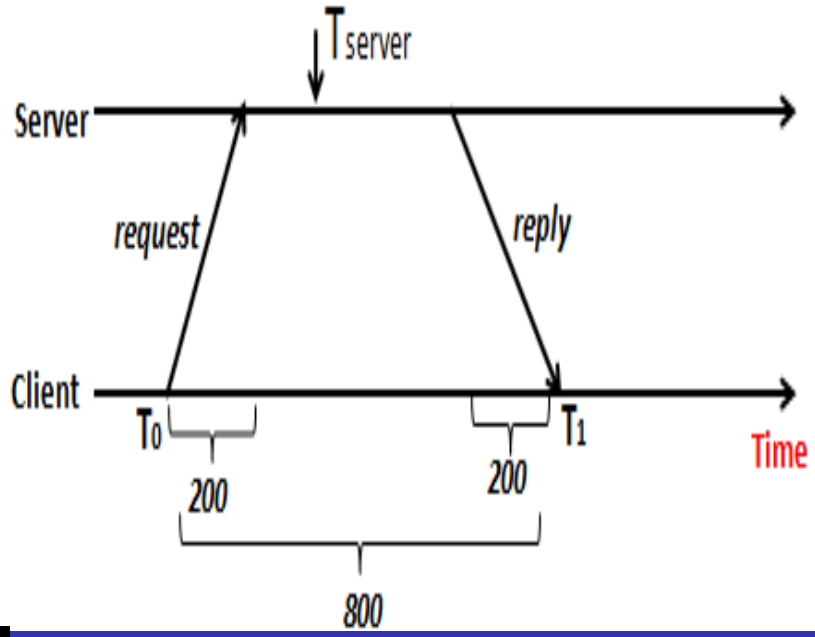
- If best-case message time=200 msec

T_0 = 5:08:15.100

T_1 = 5:08:15.900

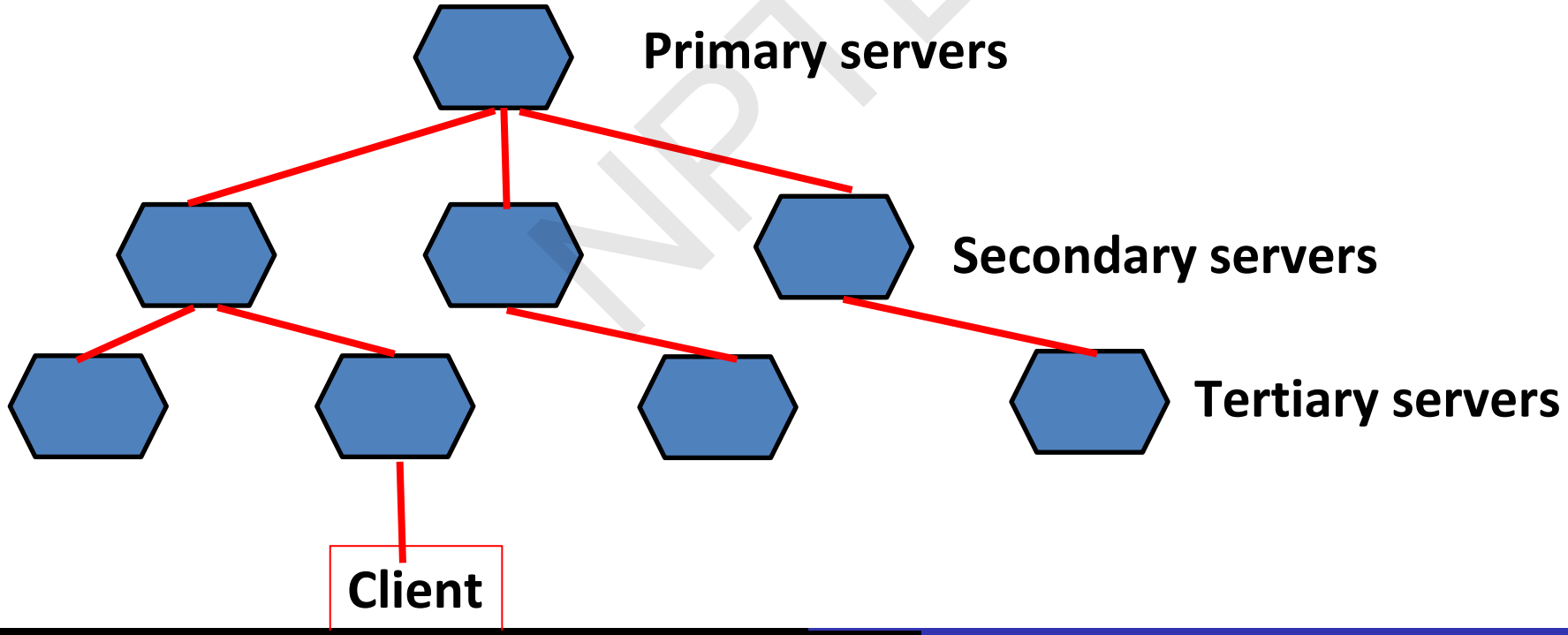
T_{server}= 5:09:25:300

Tmin = 200msec

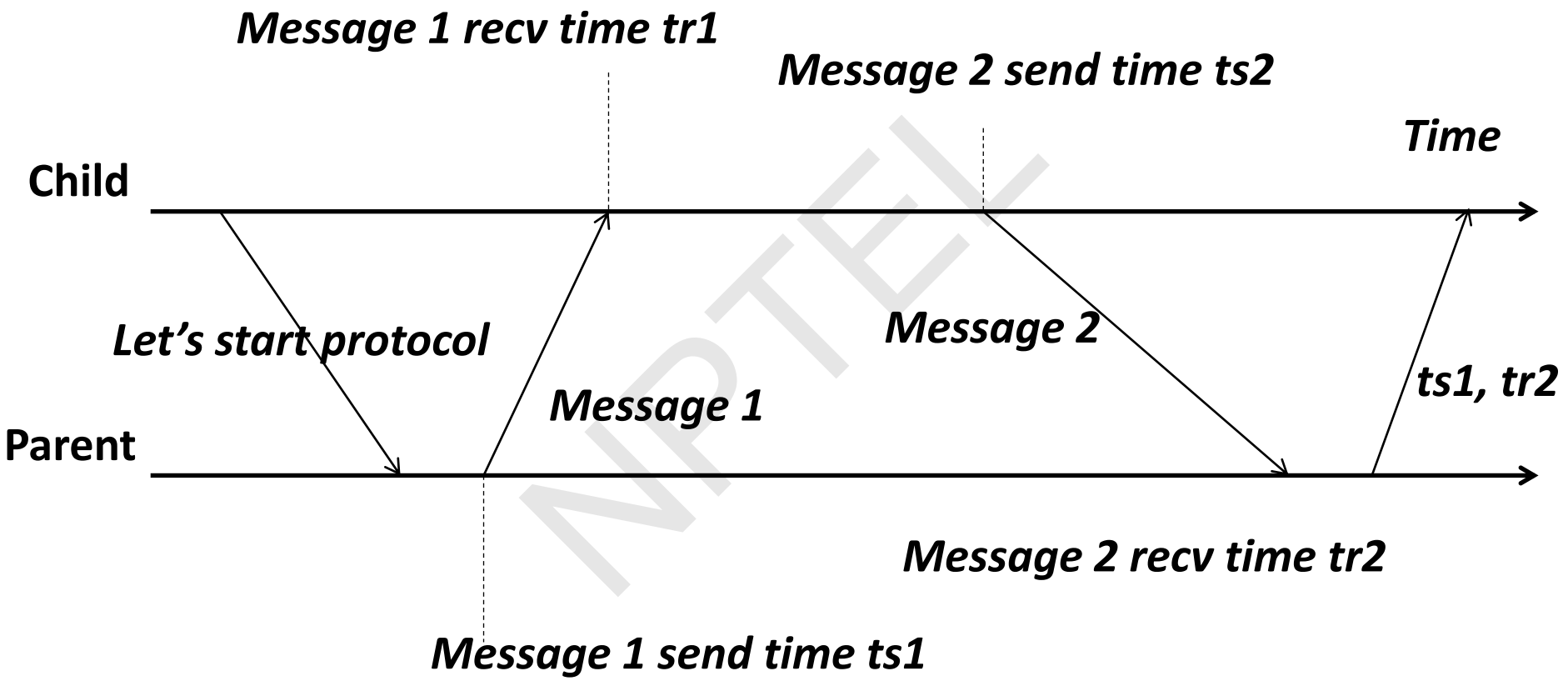


(ii) NTP: Network time protocol

- (1991, 1992) Internet Standard, version 3: RFC 1305
- **NTP servers organized in a tree.**
- Each client = a leaf of a tree.
- Each node synchronizes with its tree parent



NTP Protocol



Why o = (tr1-tr2 + ts2- ts1)/2 ?

- Offset o **= (tr1-tr2 + ts2- ts1)/2**
- Let's calculate the error.
- Suppose real offset is **oreal**
  - Child is ahead of parent by oreal.
  - Parent is ahead of child by –oreal.
- Suppose one way latency of Message 1 is L1. (L2 for Message 2)
- No one knows L1 or L2!
- **Then**
  - tr1 = ts1 + L1 + oreal
  - tr2 = ts2 + L2 – oreal

Why o = (tr1-tr2 + ts2- ts1)/2 ?

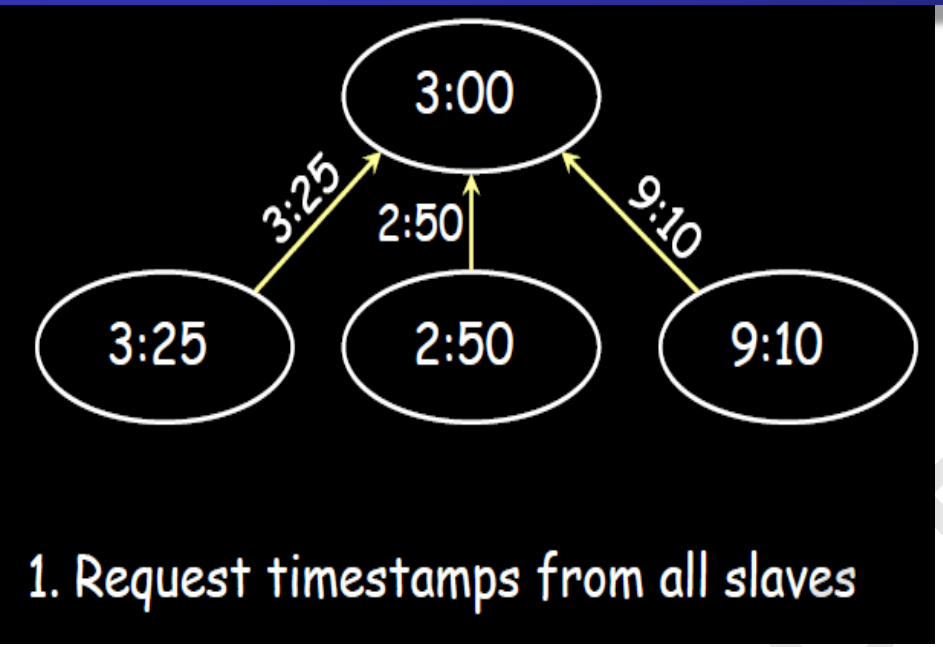
- **Then**
  - tr1 = ts1 + L1 + oreal.
  - tr2 = ts2 + L2 – oreal.

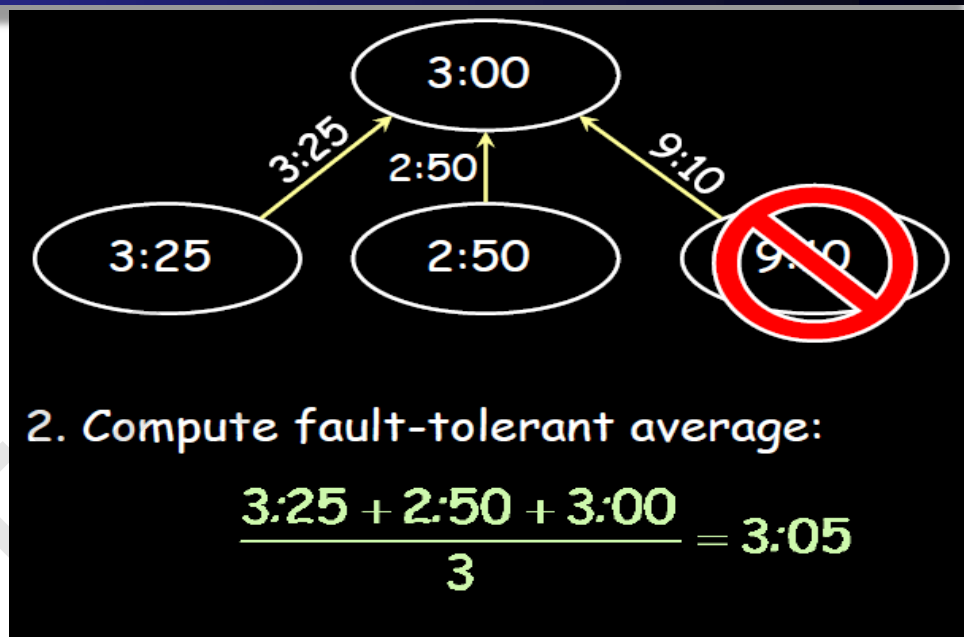
- **Subtracting second equation from first**
  - oreal = (tr1-tr2 + ts2- ts1)/2 – (L2-L1)/2
  - => oreal = o + (L2-L1)/2
  - => |oreal – o|< |(L2-L1)/2| < |(L2+L1)/2|
    - Thus the error is bounded by the round trip time (RTT)

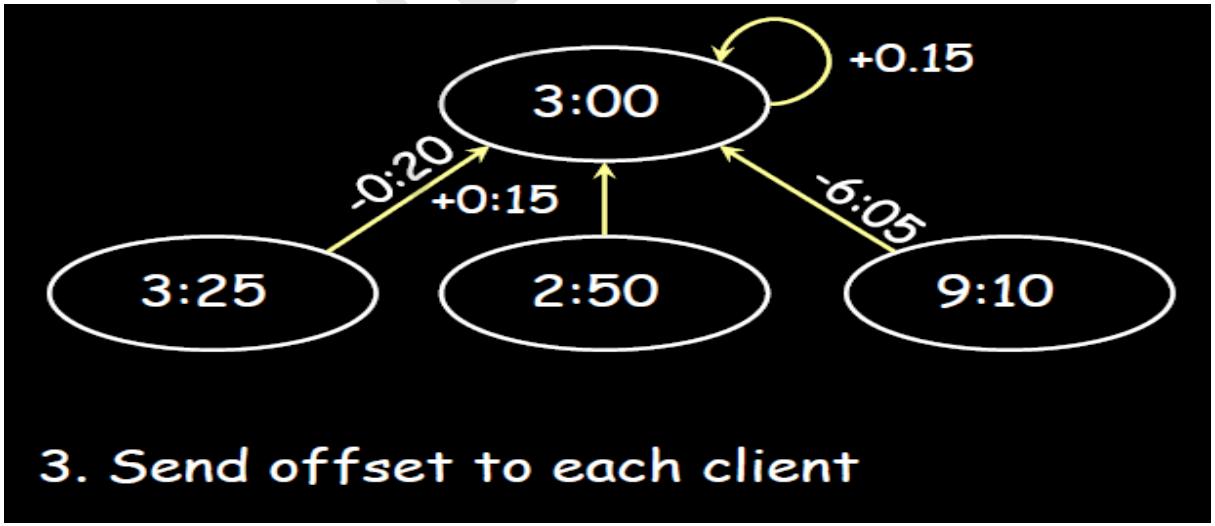
(iii) Berkley's Algorithm

- **Gusella & Zatti, 1989**
- Master poll's each machine periodically
  - Ask each machine for time
    - Can use Christian's algorithm to compensate the network's latency.
- When results are in compute,
  - Including master's time.
- **Hope: average cancels out individual clock's tendency to run fast or slow**
- Send offset by which each clock needs adjustment to each slave
  - Avoids problems with network delays if we send a time-stamp.

Berkley's Algorithm : Example





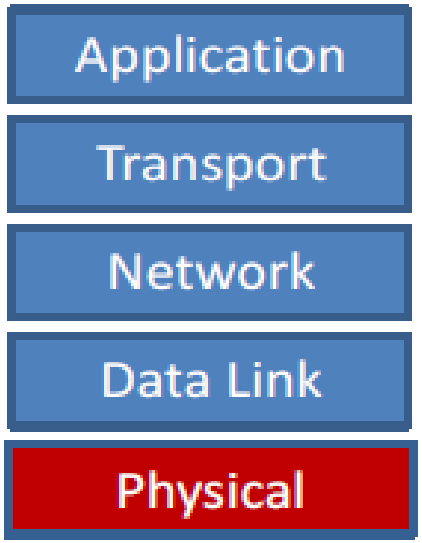


(iv) DTP: Datacenter Time Protocol

Globally Synchronized Time via Datacenter Networks

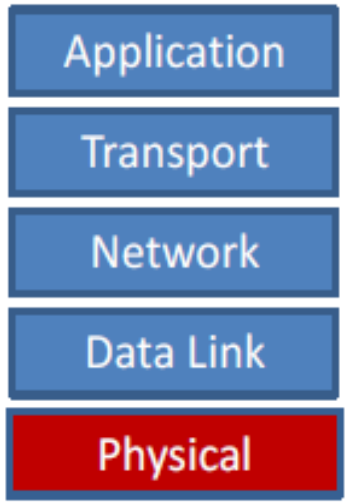
Ki Suh Lee, Han Wang, Vishal Shrivastav, Hakim Weatherspoon
Computer Science Department
Cornell University
kslee,hwang,vishal,hweather@cs.cornell.edu

ACM SIGCOMM 2016

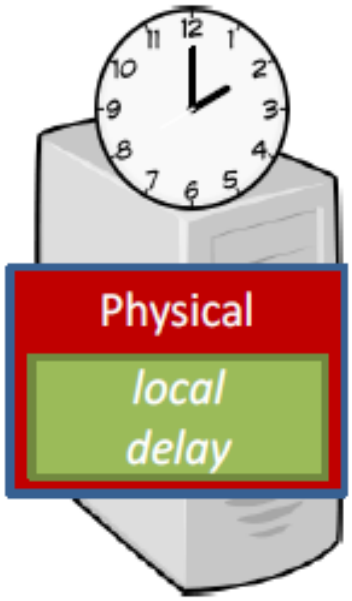


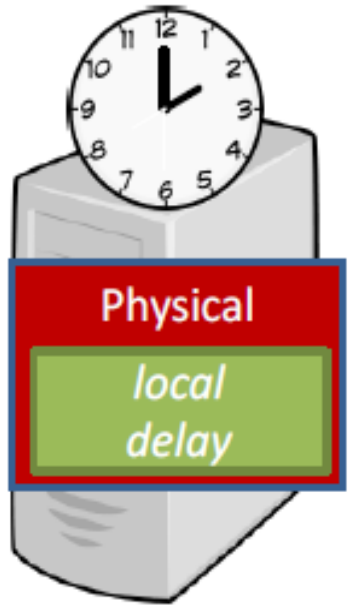
- DTP uses the physical layer of network devices to implement a decentralized clock synchronization protocol.
- ***Highly Scalable with bounded precision!***
  - ~25ns (4 clock ticks) between peers
  - ~150ns for a datacenter with six hops
  - No Network Traffic
  - *Internal Clock Synchronization*
- End-to-End: ~200ns precision!

DTP: Phases



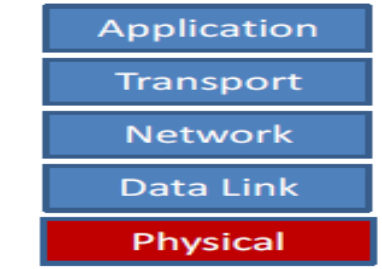
- Runs in two phases between two peers
  - Init Phase: Measuring OWD (one-way delay)
  - Beacon Phase: Re-Synchronization



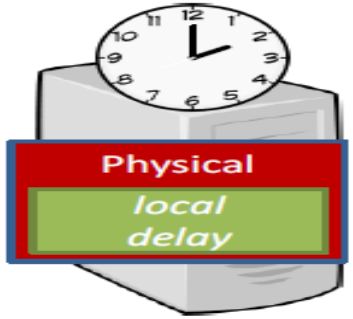


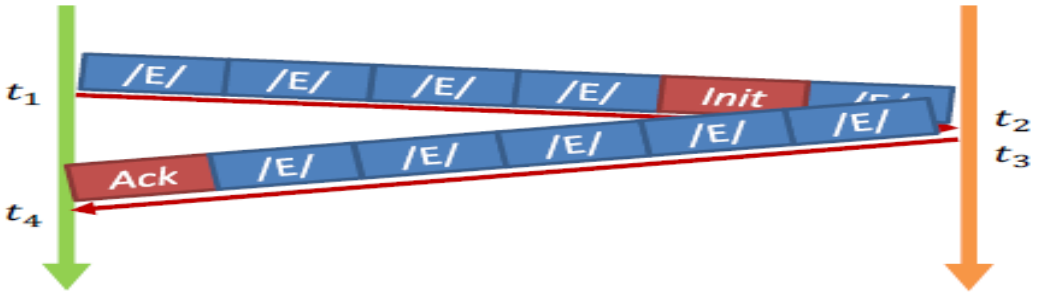
DTP: (i) Init Phase

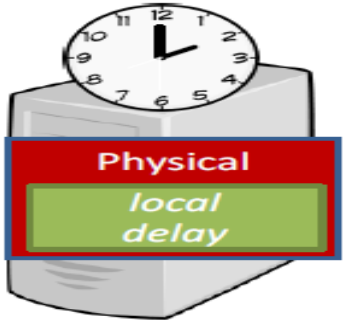
- **INIT phase:** The purpose of the INIT phase is to measure the one-way delay between two peers. The phase begins when two ports are physically connected and start communicating, i.e. when the link between them is established.
- Each peer measures the one-way delay by measuring the time between sending an INIT message and receiving an associated INIT-ACK message, i.e. measure RTT, then divide the measured RTT by two.



- $delay = (t_4 - t_1 - \alpha)/2$
  - α=3: Ensure *delay* is always less than actual delay
- Introduce 2 clock tick errors
  - Due to oscillator skew, timing and Sync FIFO

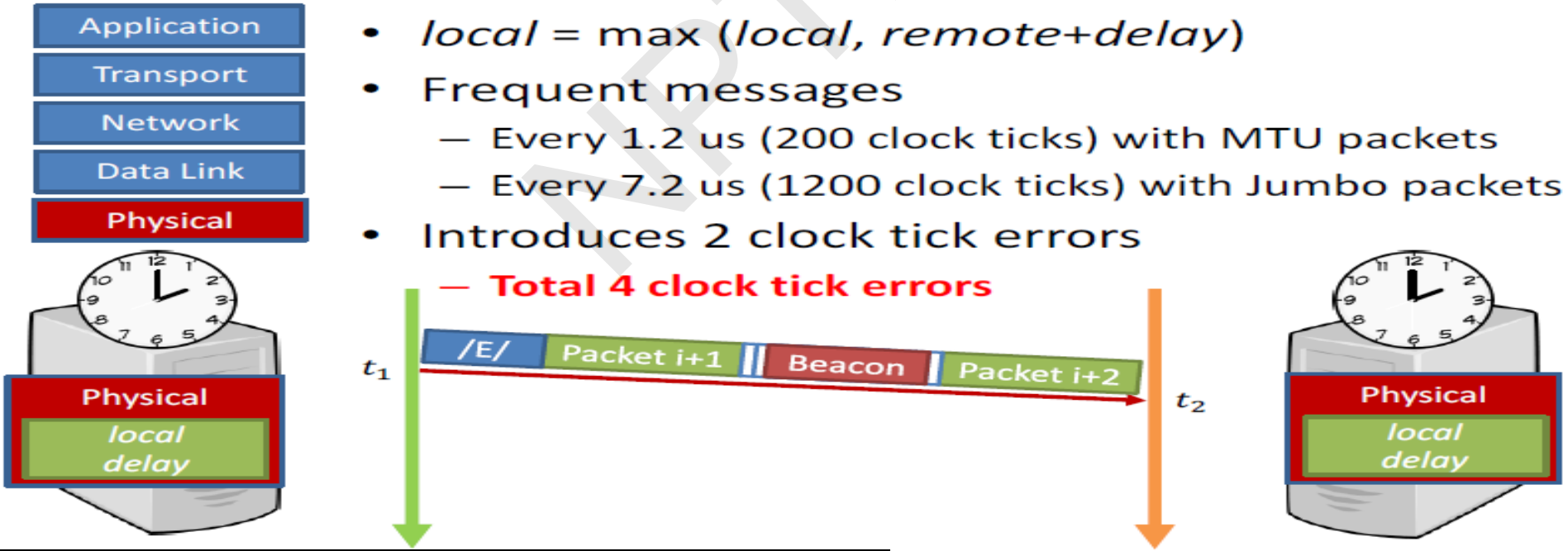




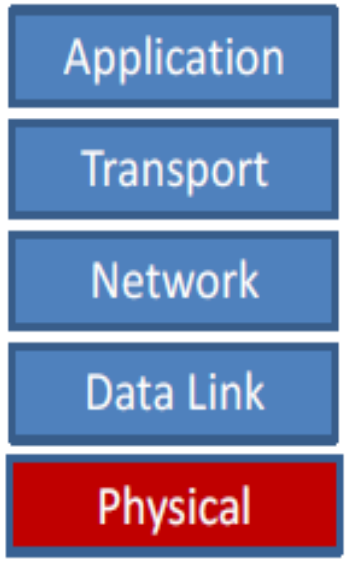


DTP: (ii) Beacon Phase

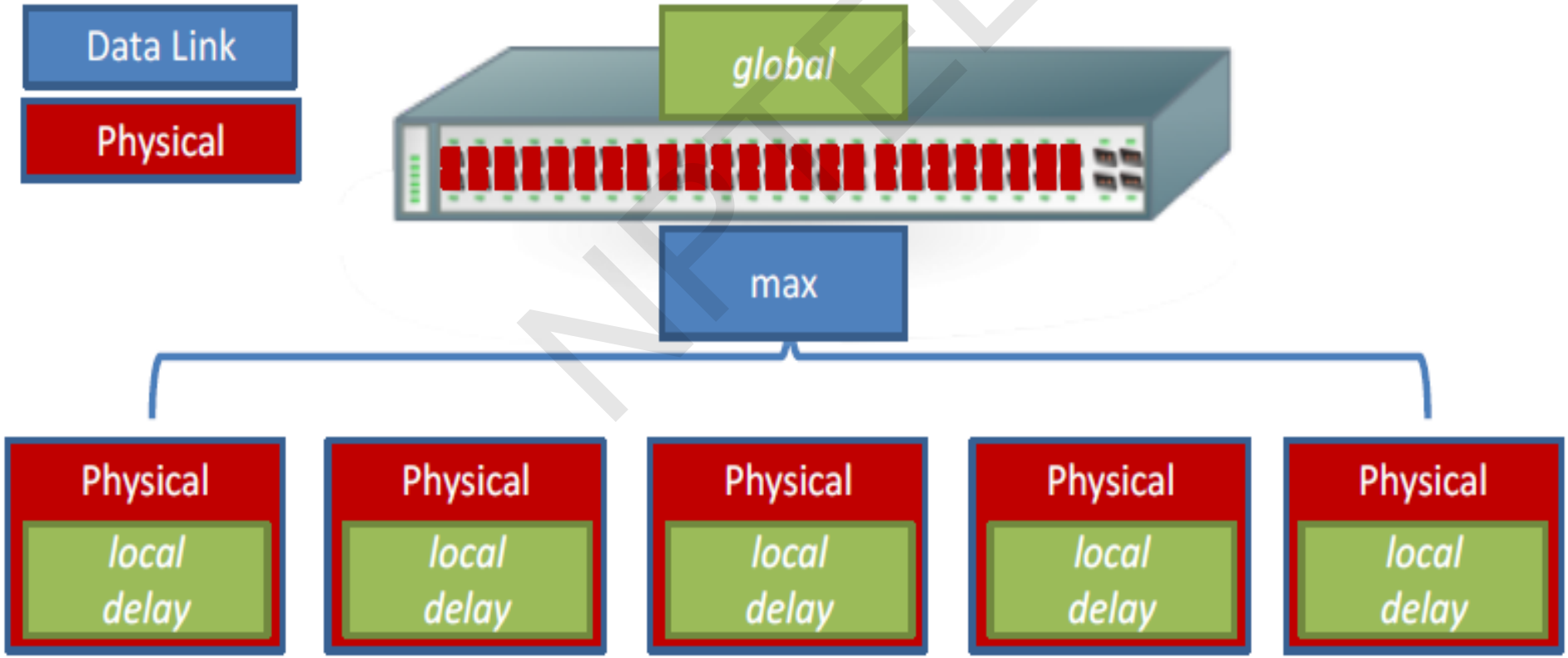
- **BEACON phase:** During the BEACON phase, two ports periodically exchange their local counters for resynchronization. Due to oscillator skew, the offset between two local counters will increase over time. A port adjusts its local counter by selecting the maximum of the local and remote counters upon receiving a BEACON message from its peer. Since BEACON messages are exchanged frequently, hundreds of thousands of times a second (every few microseconds), the offset can be kept to a minimum.



DTP Switch



- *global* = max(*local* counters)
- Propagates *global* via Beacon messages



DTP Property

- **DTP provides bounded precision and scalability**
- **Bounded Precision in hardware**
  - Bounded by 4T (=25.6ns, T=oscillator tick is 6.4ns)
  - Network precision bounded by 4*TD*

    D is network diameter in hops
- **Requires NIC and switch modifications**

But Yet…

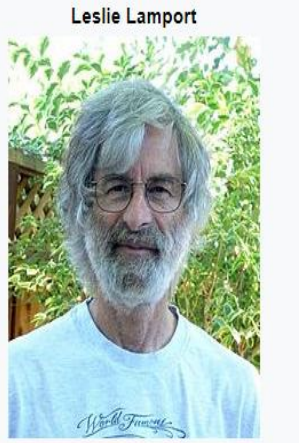
- **We still have a non-zero error!**
- **We just can't seem to get rid of error**
  - Can't as long as messages latencies are non-zero.
- **Can we avoid synchronizing clocks altogether, and still be able to order events ?**

Ordering events in a distributed system

- To order events across processes, trying to synchronize clocks is an approach.
- What if we instead assigned timestamps to events that were not absolute time ?
- As long as those timestamps obey **causality**, that would work
  - If an event A causally happens before another event B, then timestamp(A) < timestamp (B)
  - Example: Humans use causality all the time
    - I enter the house only if I unlock it
    - You receive a letter only after I send it

Logical (or Lamport) ordering

- **Proposed by Leslie Lamport in the 1970s.**
- Used in almost all distributed systems since then
- Almost all cloud computing systems use some form of logical ordering of events.



- **Leslie B. Lamport** (born February 7, 1941) is an American computer scientist. Lamport is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX. Leslie Lamport was the winner of the **2013 Turing Award** for imposing clear, well-defined coherence on the seemingly chaotic behavior of distributed computing systems, in which several autonomous computers communicate with each other by passing messages.

Lamport's research contributions

- **Lamport's research contributions have laid the foundations of the theory of distributed systems. Among his most notable papers are**
  - "Time, Clocks, and the Ordering of Events in a Distributed System", which received the PODC Influential Paper Award in 2000,
  - "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs",which defined the notion of Sequential consistency,
  - "The Byzantine Generals' Problem",
  - "Distributed Snapshots: Determining Global States of a Distributed System" and
  - "The Part-Time Parliament".
- These papers relate to such concepts as logical clocks (and the *happened-before* relationship) and Byzantine failures. They are among the most cited papers in the field of computer science and describe algorithms to solve many fundamental problems in distributed systems, including:
  - the Paxos algorithm for consensus,
  - the bakery algorithm for mutual exclusion of multiple threads in a computer system that require the same resources at the same time,
  - the Chandy-Lamport algorithm for the determination of consistent global states (snapshot), and
  - the Lamport signature, one of the prototypes of the digital signature.

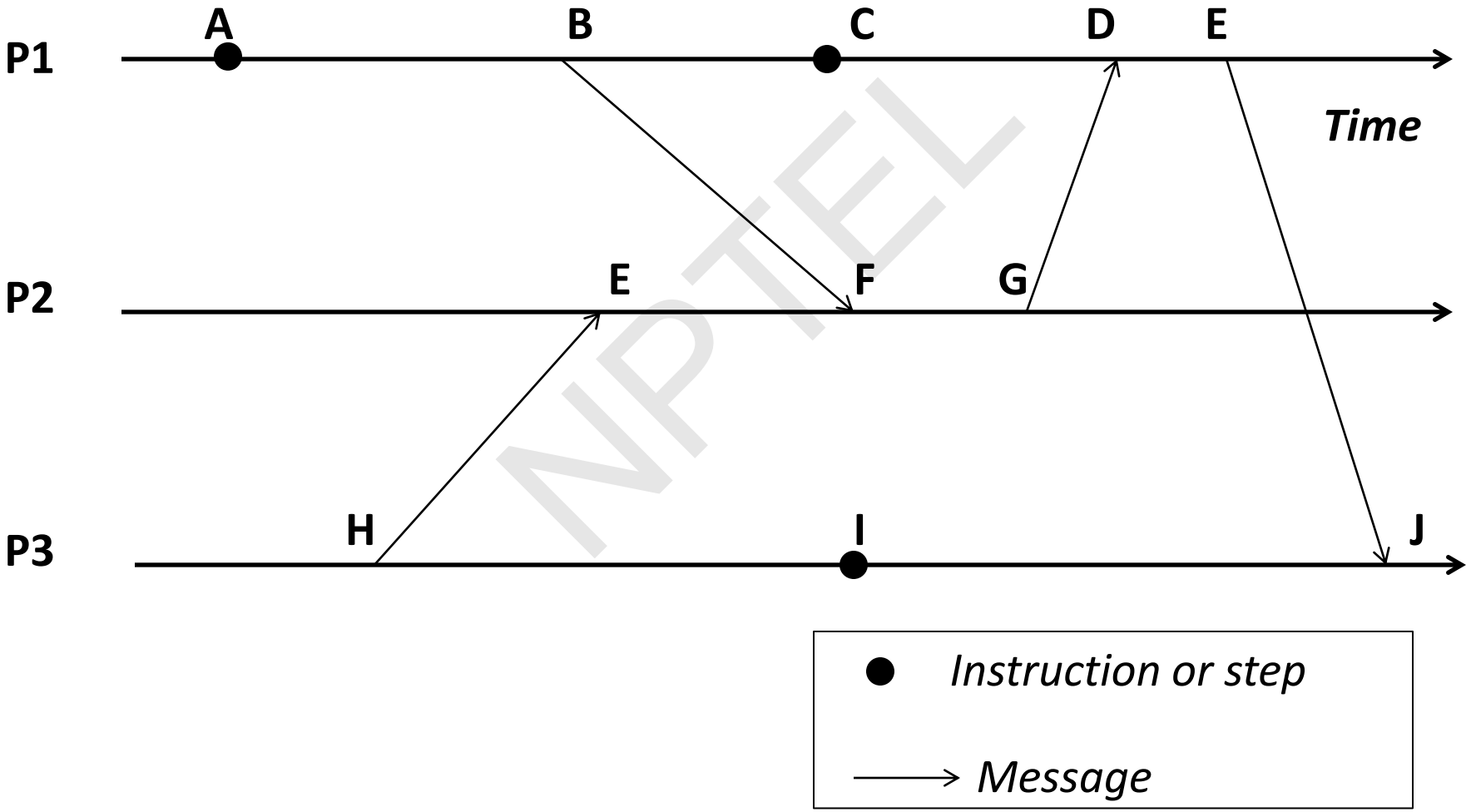
Logical (or Lamport) Ordering(2)

- Define a logical relation ***Happens-Before*** among pairs of events
- **Happens-Before denoted as $\rightarrow$**
- **Three rules:**

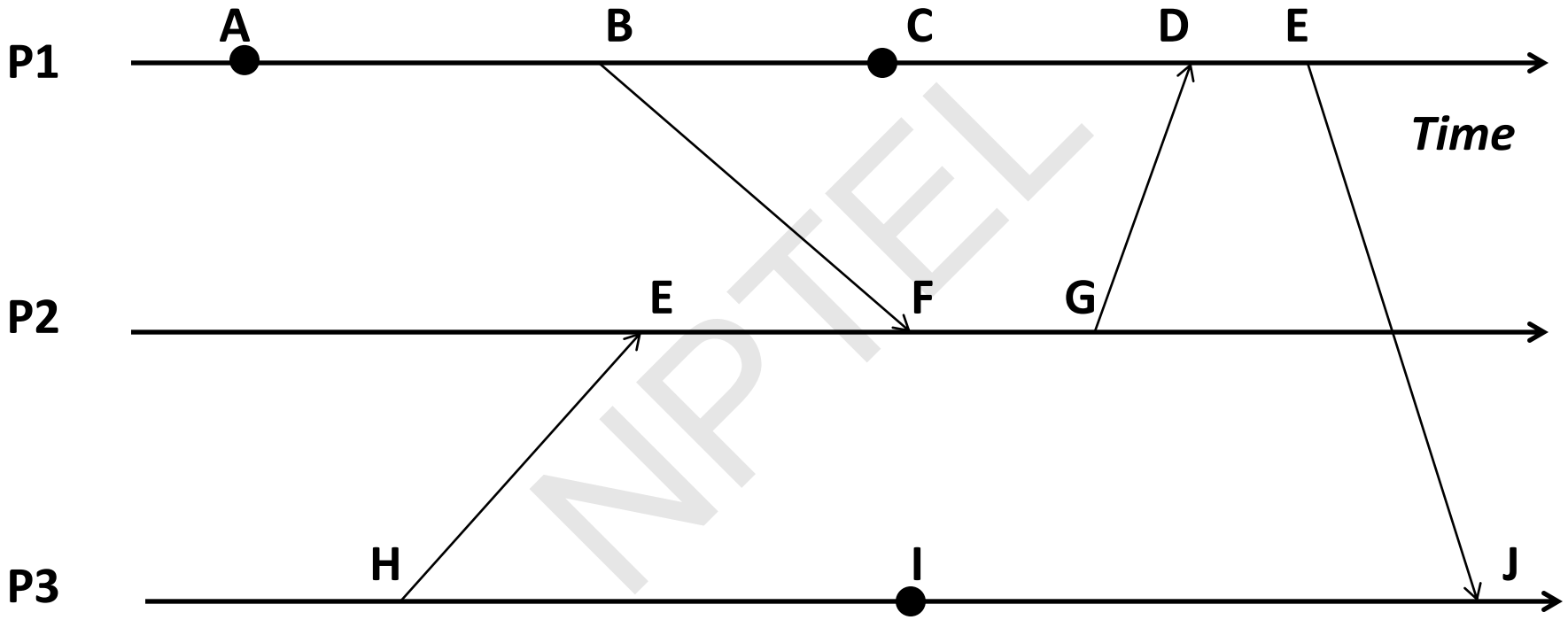
1. On the same process: $a \rightarrow b$, if $time(a) < time(b)$ (using the local clock)
2. If p1 sends *m* to p2: $send(m) \rightarrow receive(m)$
3. **(Transitivity)** If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$

- Creates a *partial order* among events
  - Not all events related to each other via $\rightarrow$

Example 1:

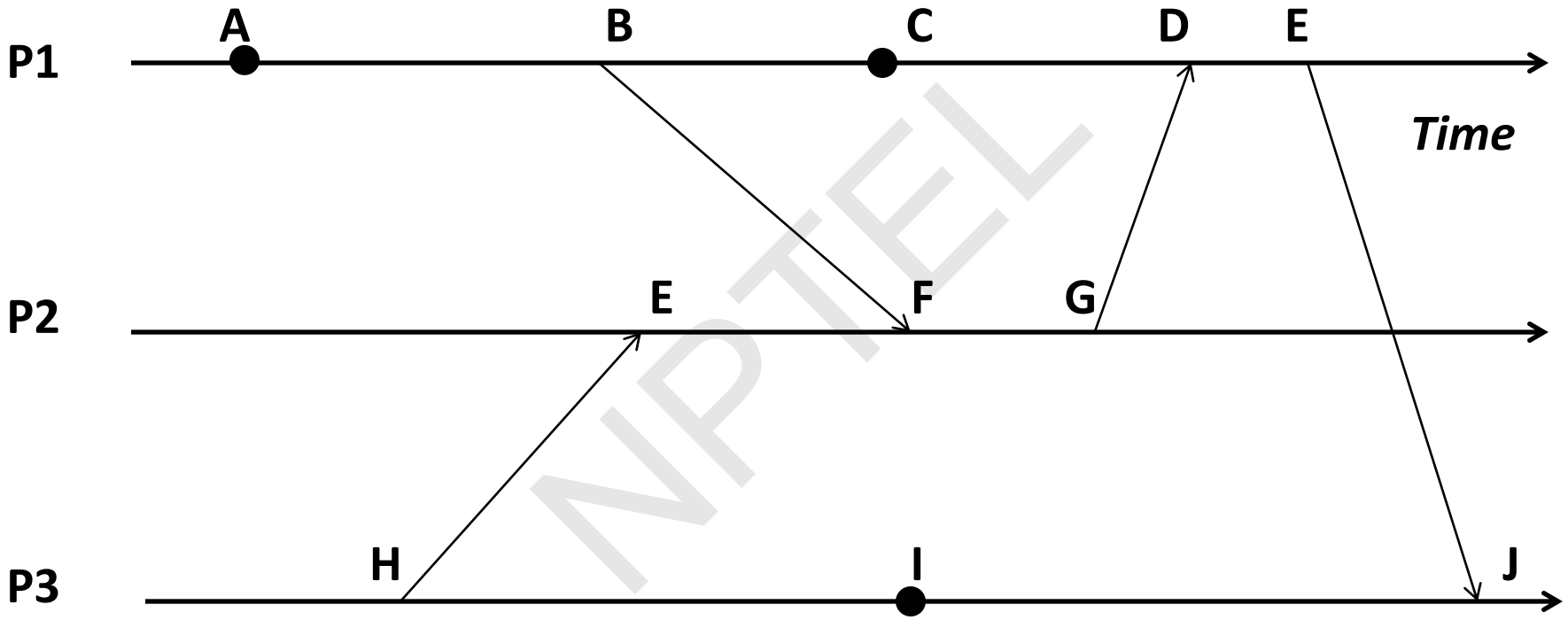


Example 1: Happens-Before

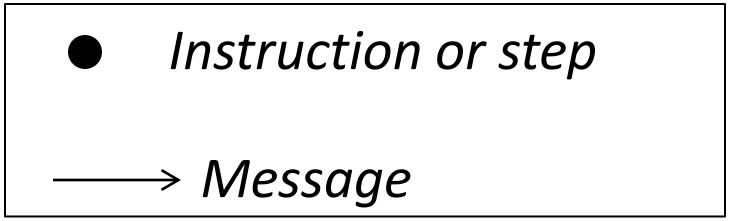


- A → B
- B → F
- A → F

Example 2: Happens-Before



- H → G
- F → J
- H → J
- C → J

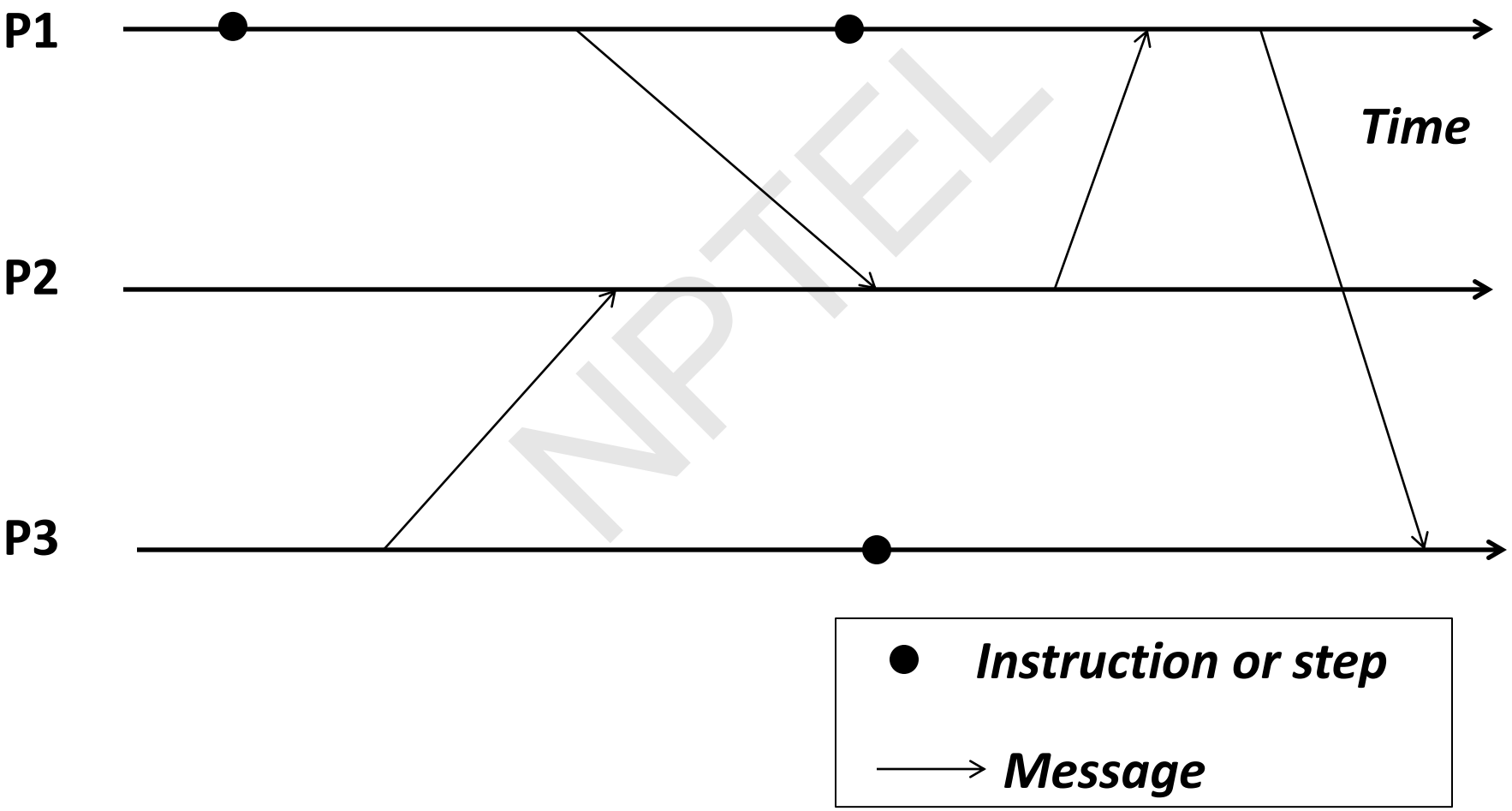


Lamport timestamps

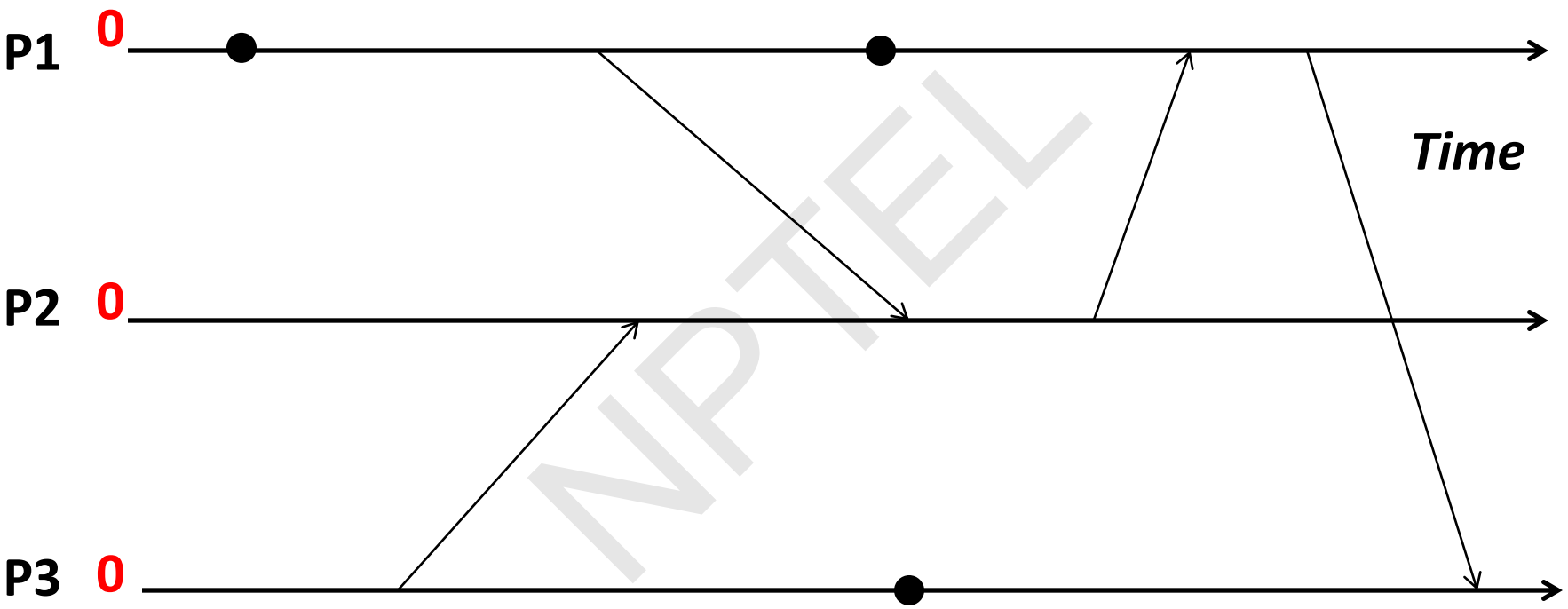
- **Goal: Assign logical (Lamport) timestamp to each event**
- **Timestamps obey causality**
- **Rules**
  - Each process uses a local counter (clock) which is an integer
    - initial value of counter is zero
  - A process increments its counter when a **send** or an **instruction** happens at it. The counter is assigned to the event as its timestamp.
  - A **send (message)** event carries its timestamp
  - For **a receive (message)** event the counter is updated by

max(local clock, message timestamp) + 1

Example

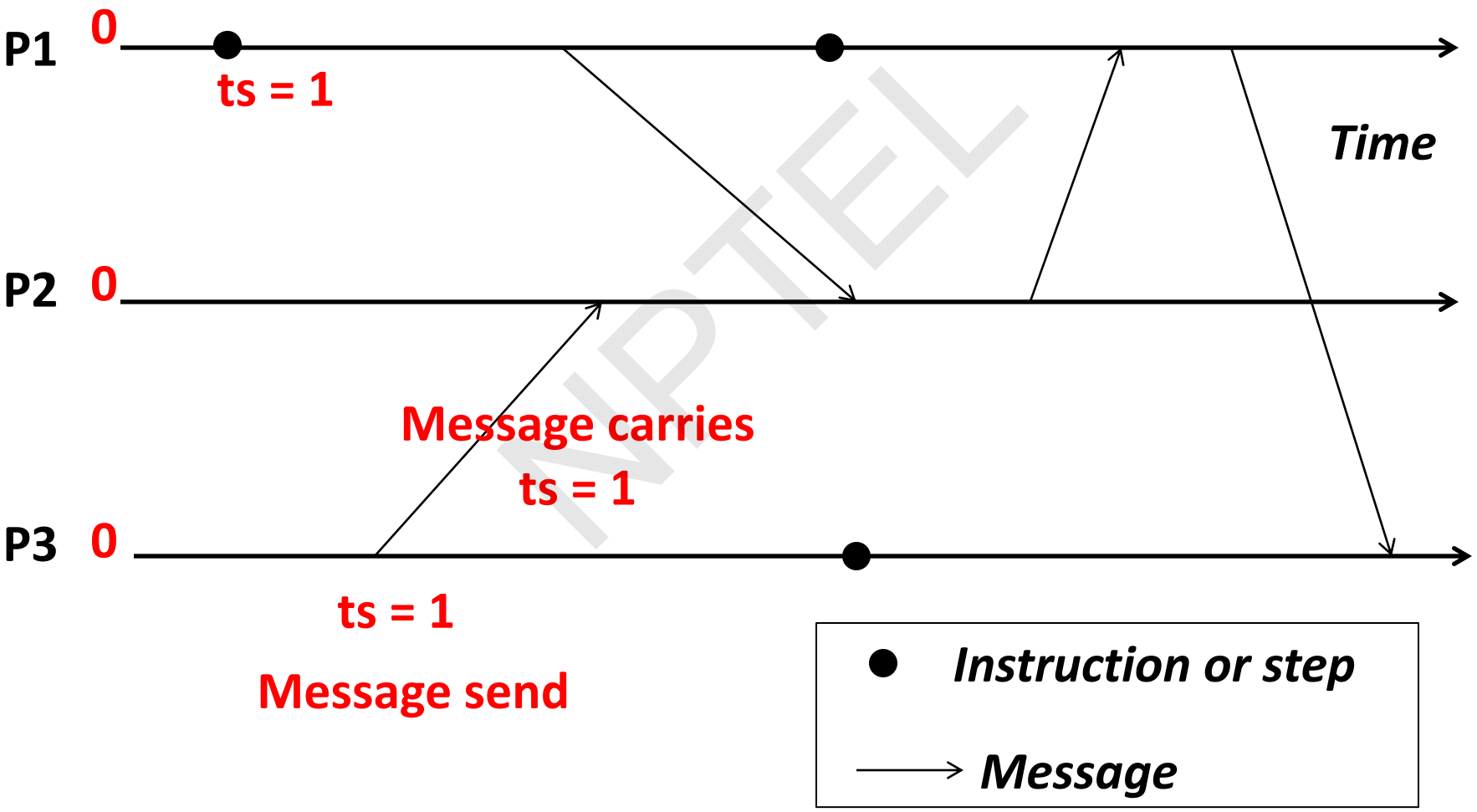


Lamport Timestamps

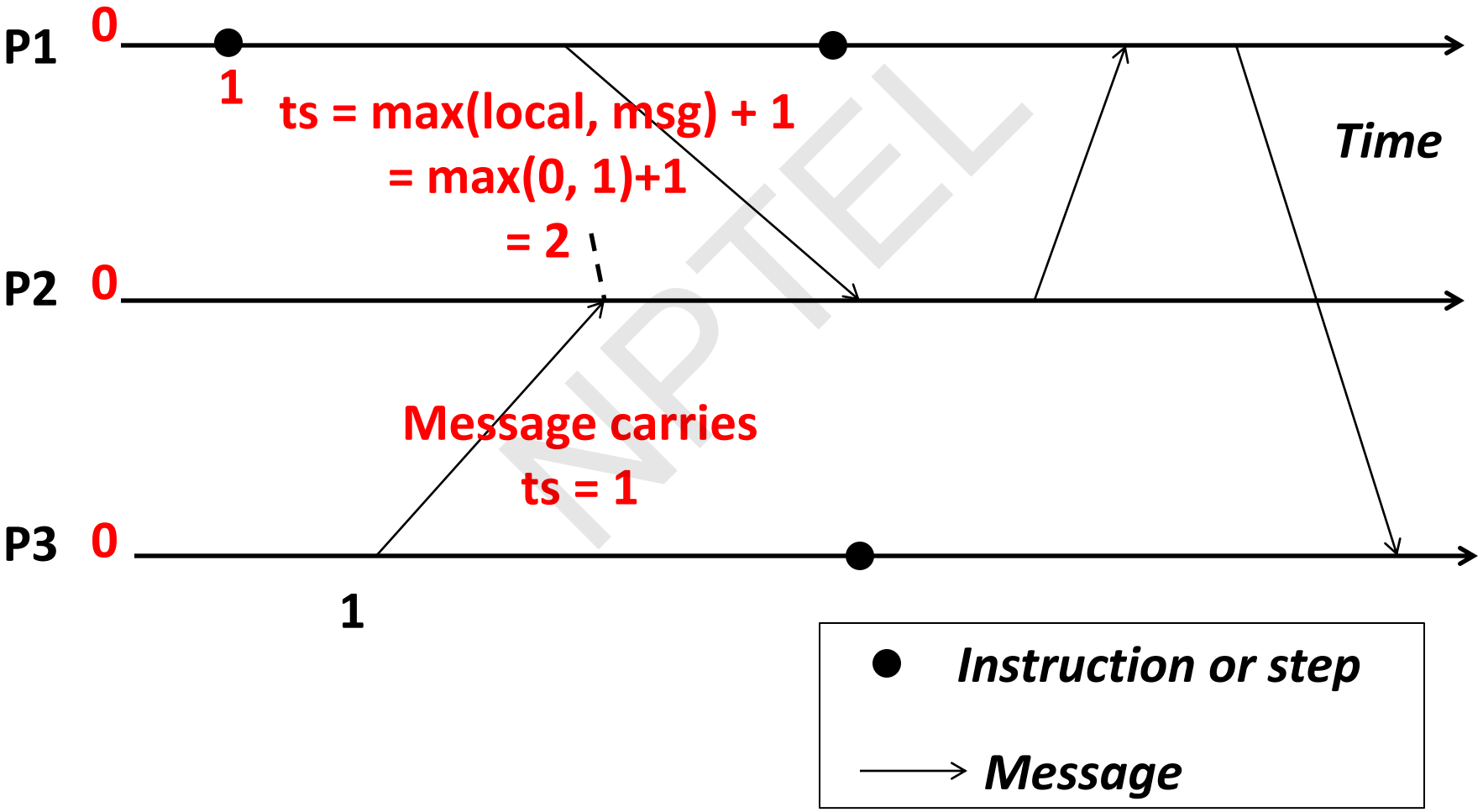


Initial counters (clocks)

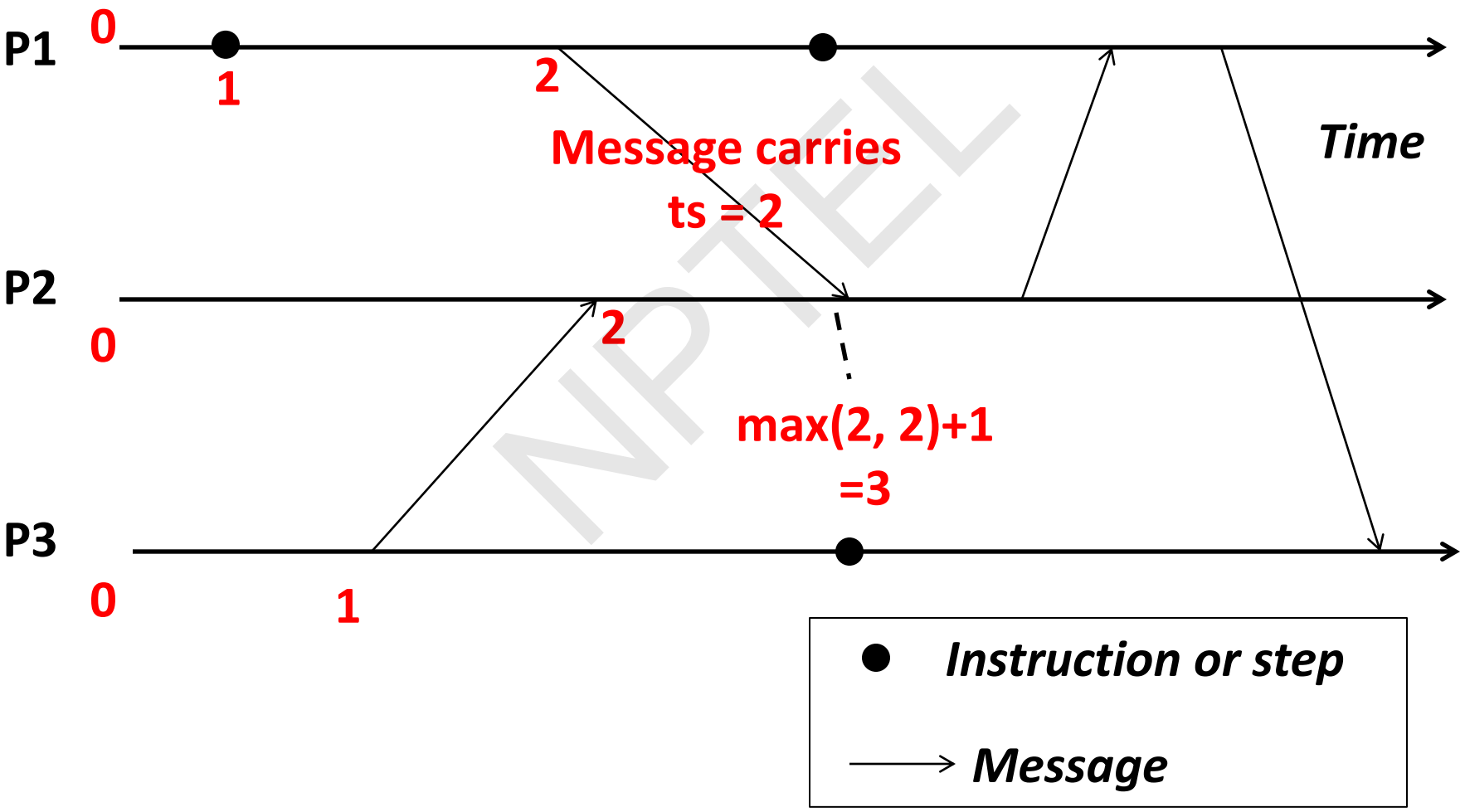
Lamport Timestamps



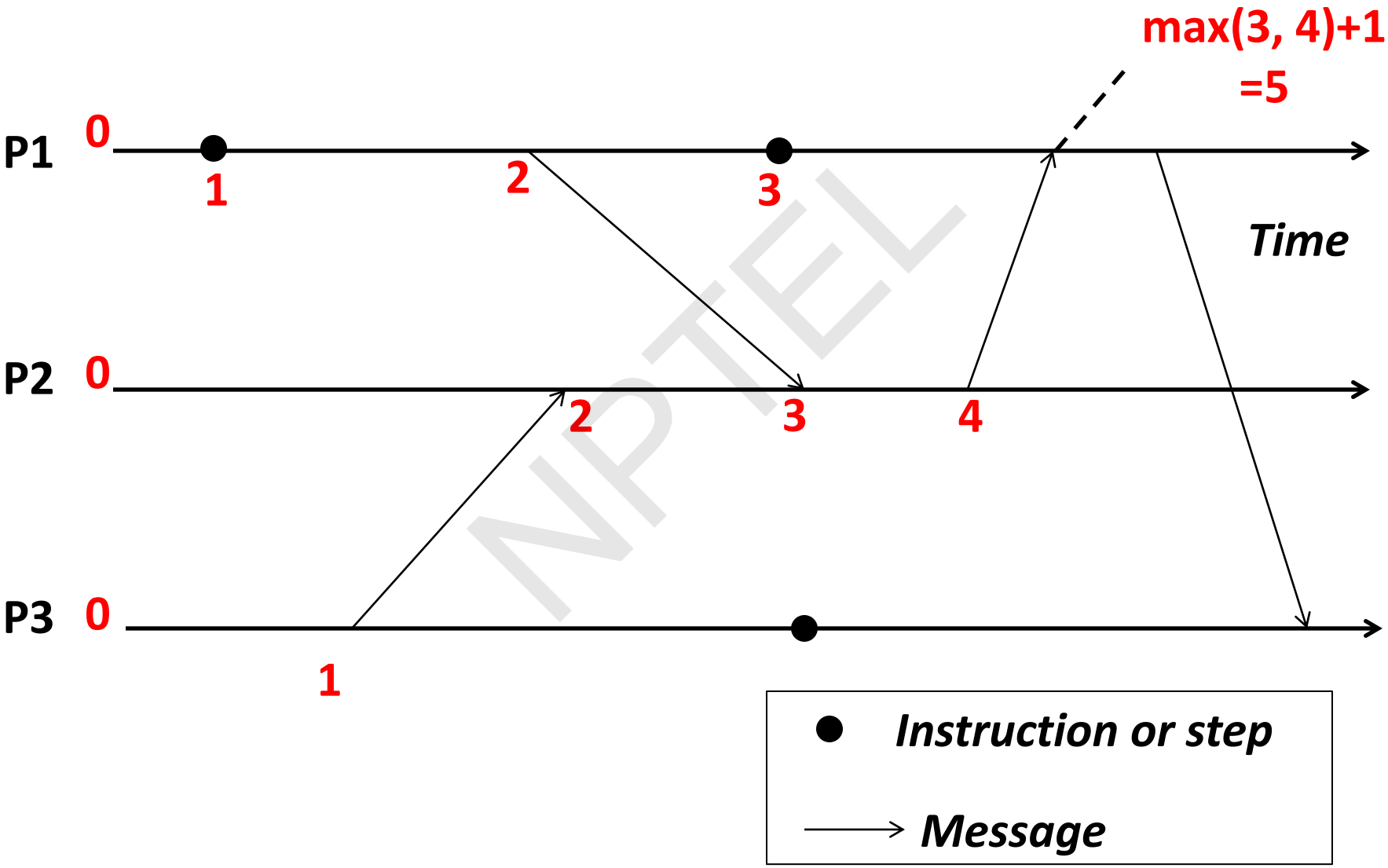
Lamport Timestamps



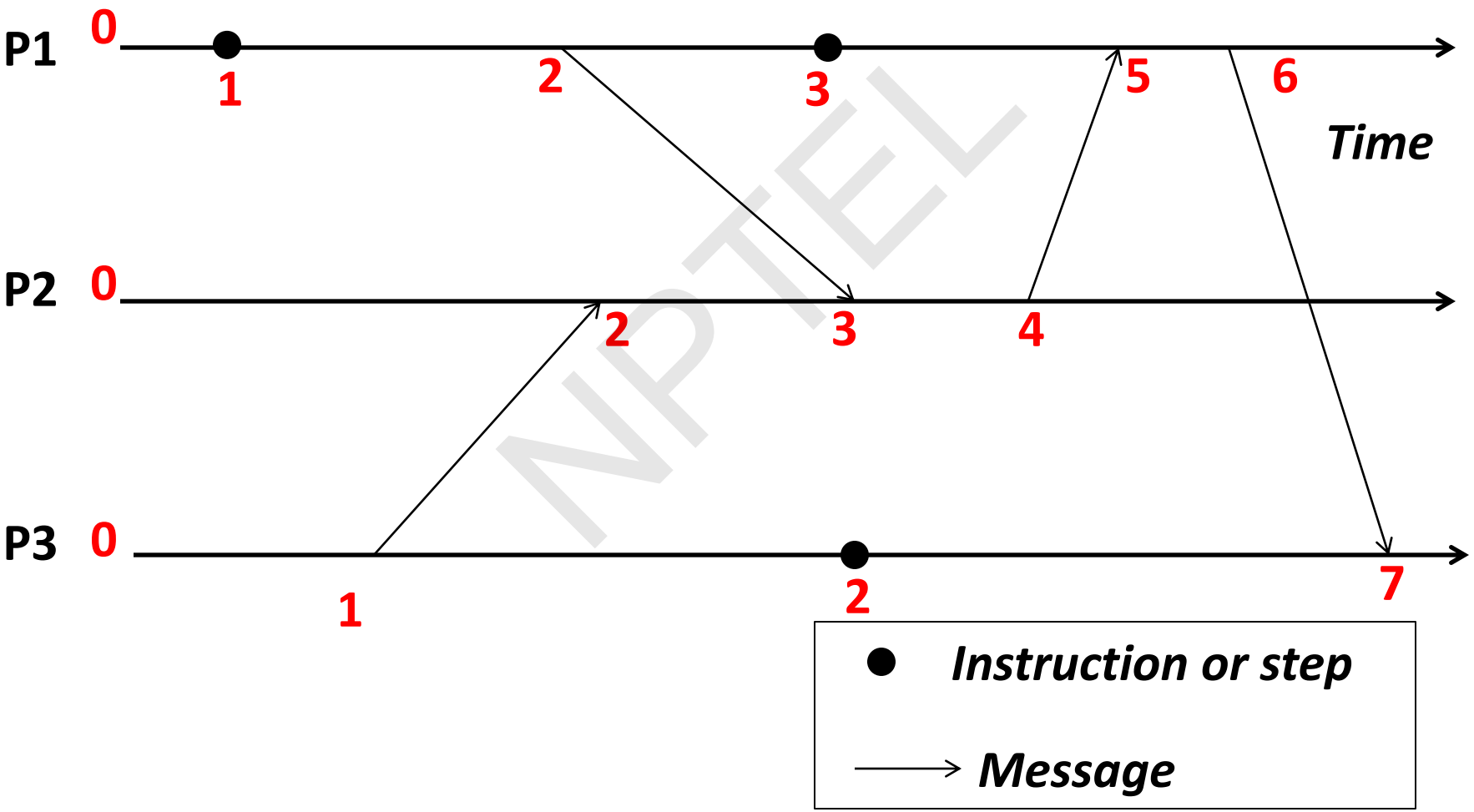
Lamport Timestamps



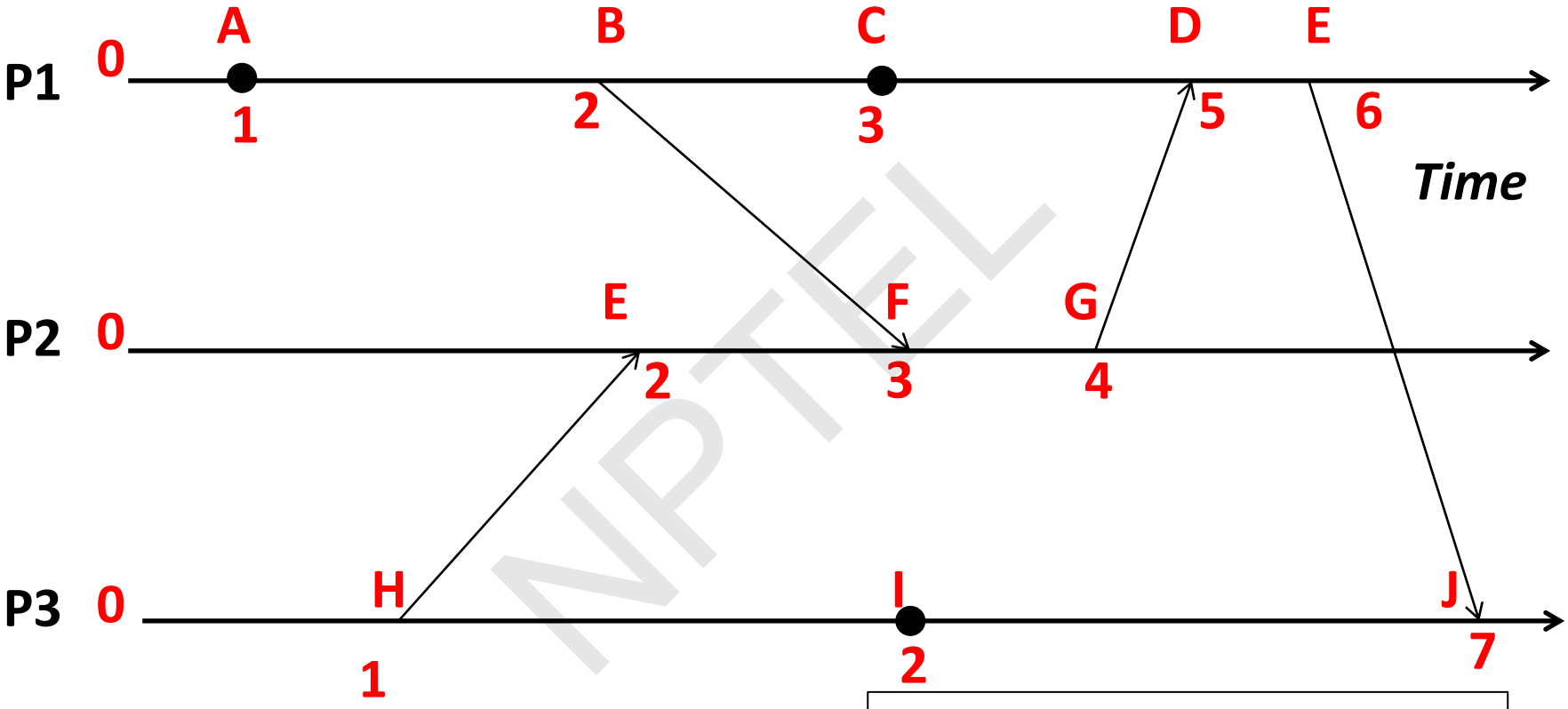
Lamport Timestamps



Lamport Timestamps

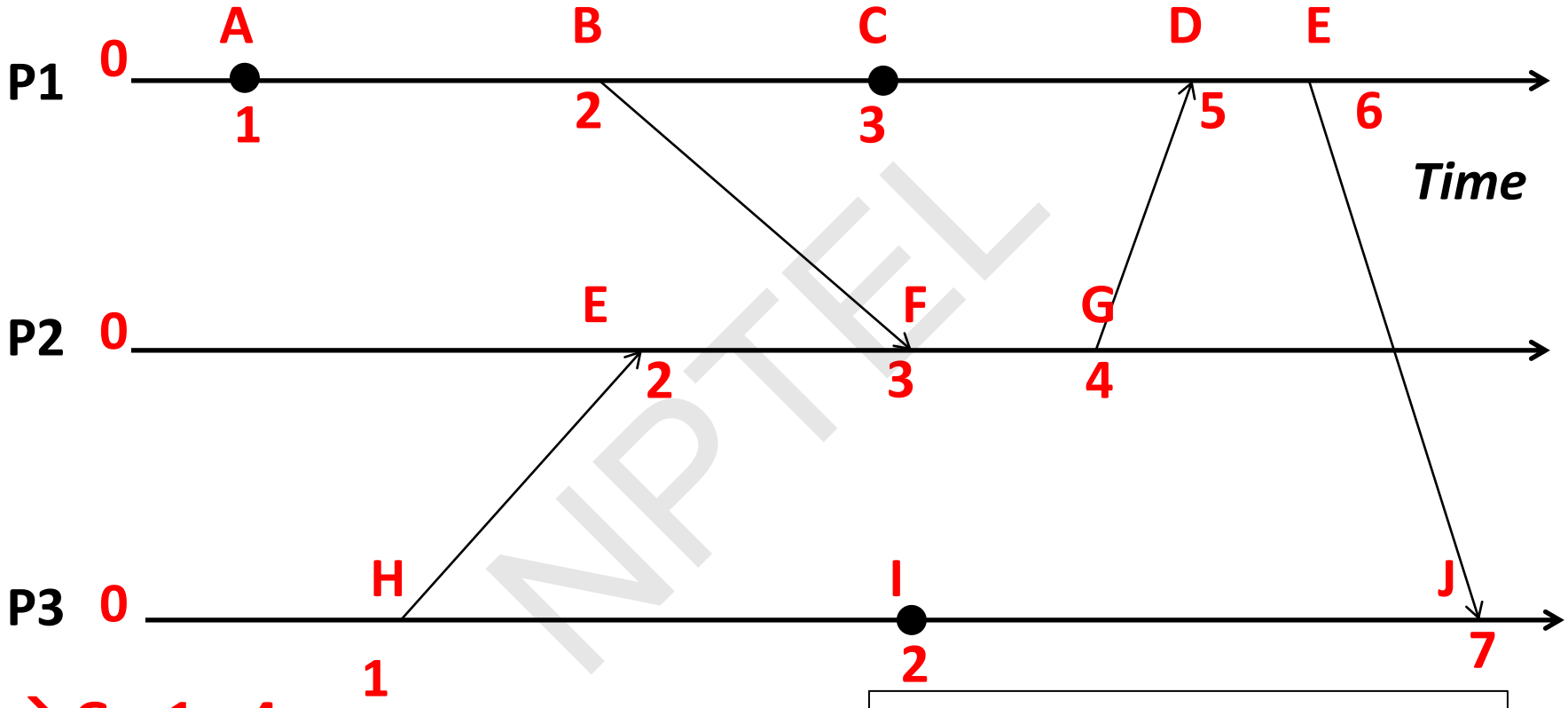


Obeying Causality



- A → B :: 1 < 2
- B → F :: 2 < 3
- A → F :: 1 < 3

Obeying Causality (2)



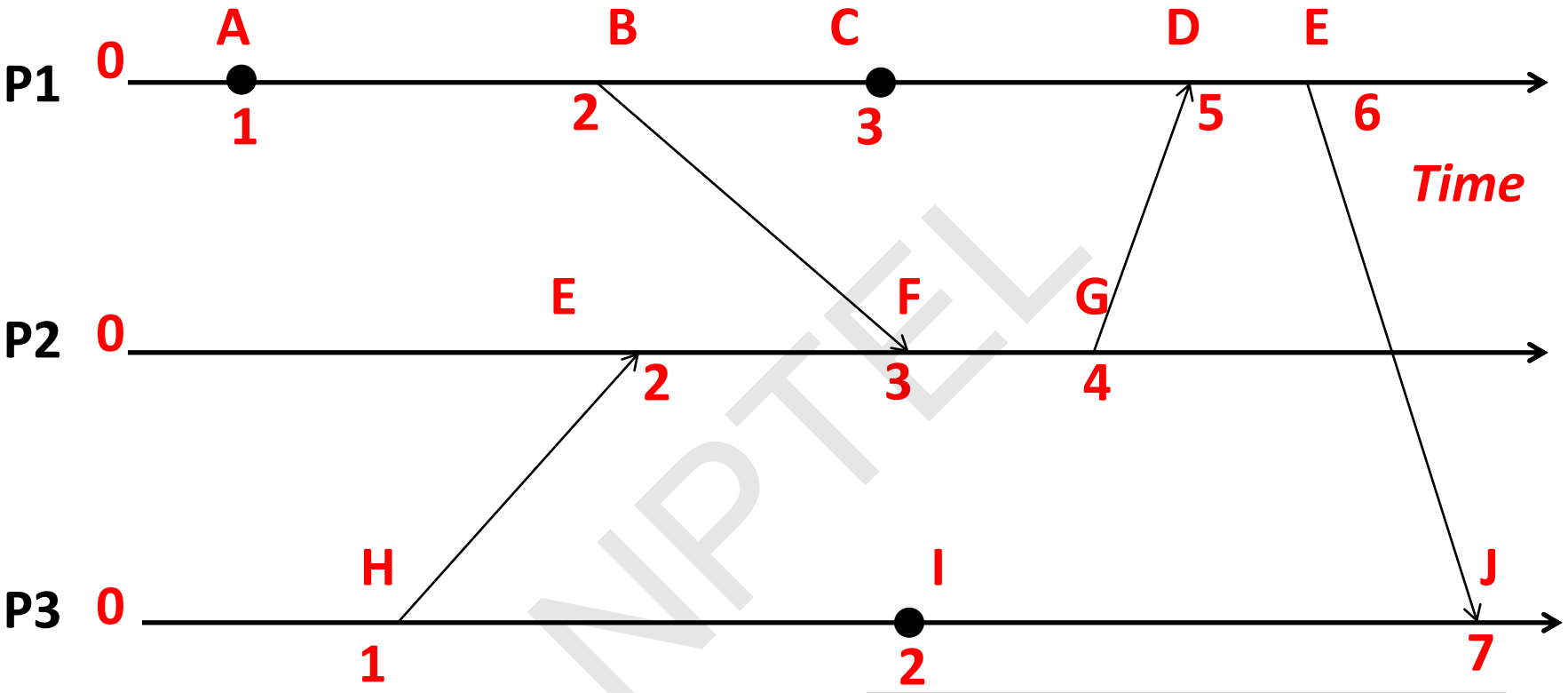
H → G :: 1 < 4

F → J :: 3 < 7

H → J :: 1 < 7

C → J :: 3 < 7

Not always *implying* Causality



- ? C → F ? :: 3 = 3
- ? H → C ? :: 1 < 3
- (C, F) and (H, C) are pairs of *concurrent* events

Concurrent Events

- A pair of concurrent events doesn't have a causal path from one event to another (either way, in the pair)
- Lamport timestamps not guaranteed to be ordered or unequal for concurrent events
- Ok, since concurrent events are not causality related!

- Remember:

$E1 \rightarrow E2 \Rightarrow$ timestamp(E1) < timestamp (E2), BUT

timestamp(E1) < timestamp (E2) $\Rightarrow$

{E1 $\rightarrow$ E2} OR {E1 and E2 concurrent}

Vector Timestamps

- Used in key-value stores like Riak
- Each process uses a vector of integer clocks
- Suppose there are N processes in the group 1…N
- Each vector has N elements
- Process *i maintains vector* $V_i[1…N]$
- *j*th element of vector clock at process *i*, $V_i[j]$, is *i*'s knowledge of latest events at process *j*

Assigning Vector Timestamps

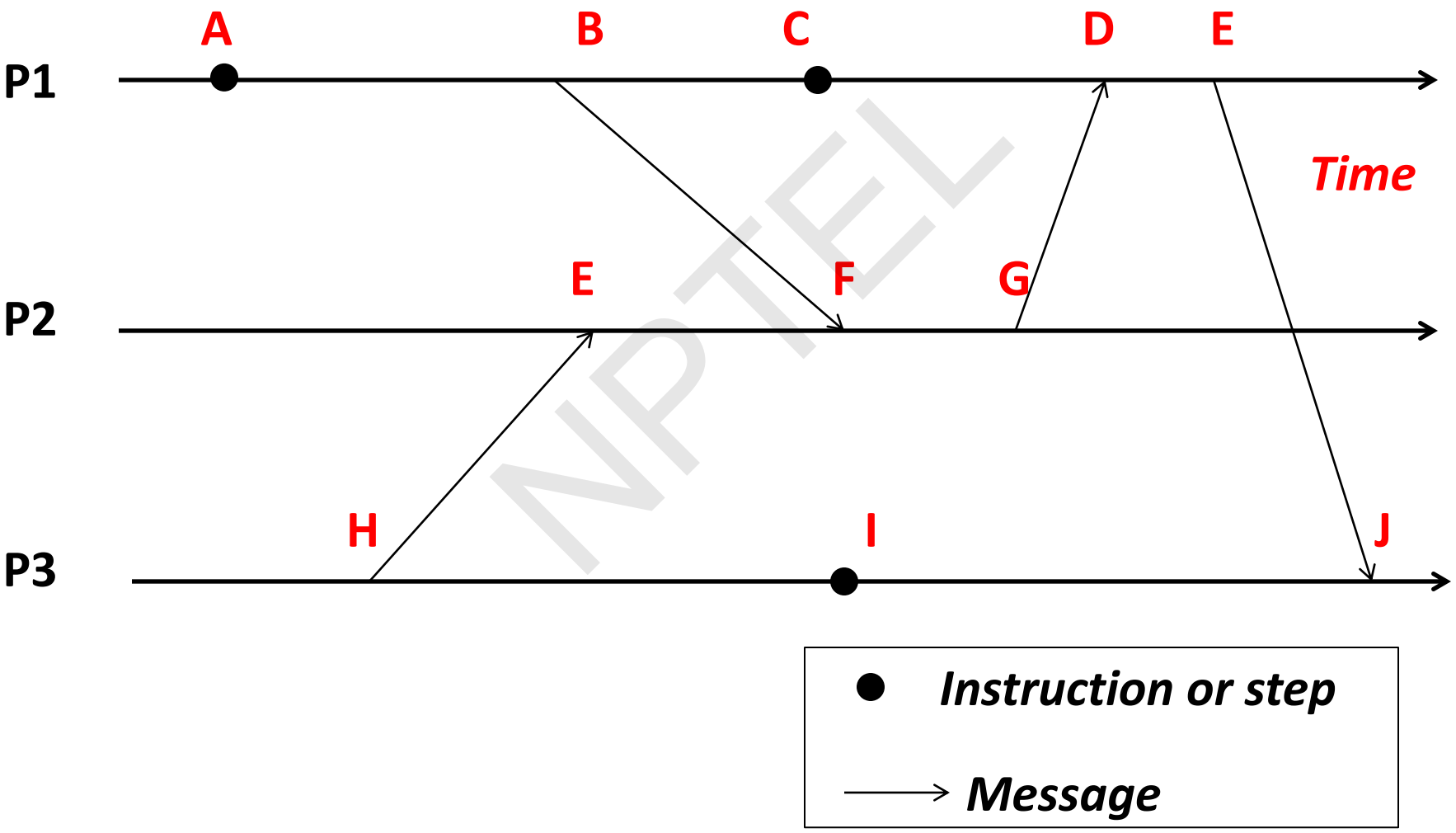
Incrementing vector clocks

1. On an instruction or send event at process *i*, it increments only its *i*th element of its vector clock
2. Each message carries the send-event's vector timestamp $V_{message}[1...N]$
3. On receiving a message at process *i*:

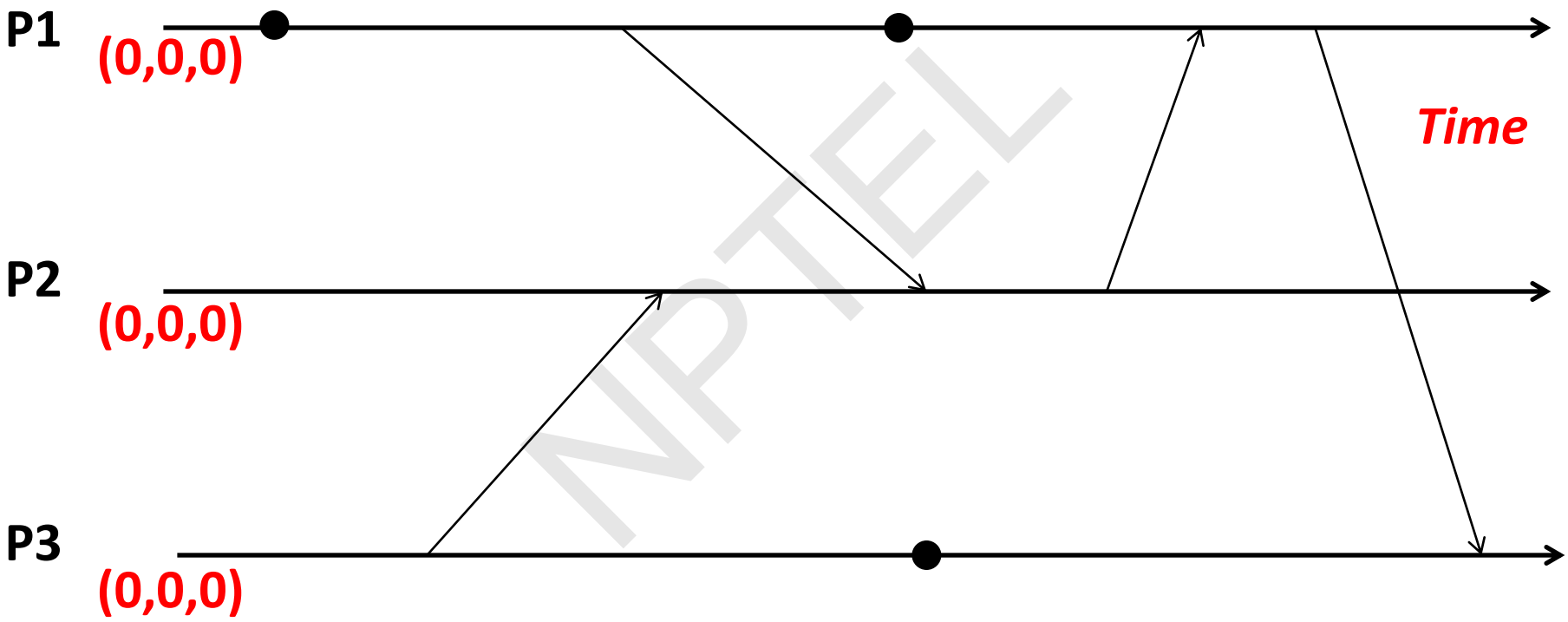
$$V_i[i] = V_i[i] + 1$$

$$V_i[j] = \max(V_{message}[j], V_i[j]) \text{ for } j \neq i$$

Example

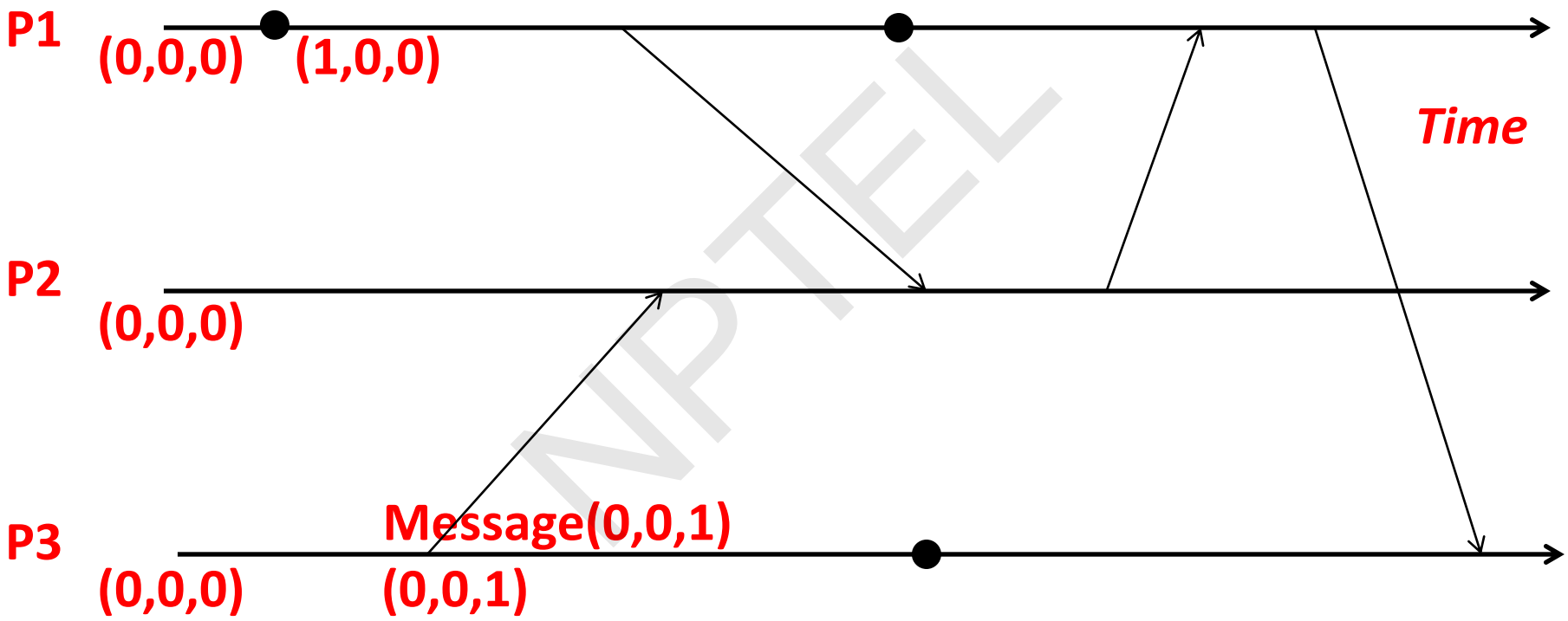


Vector Timestamps

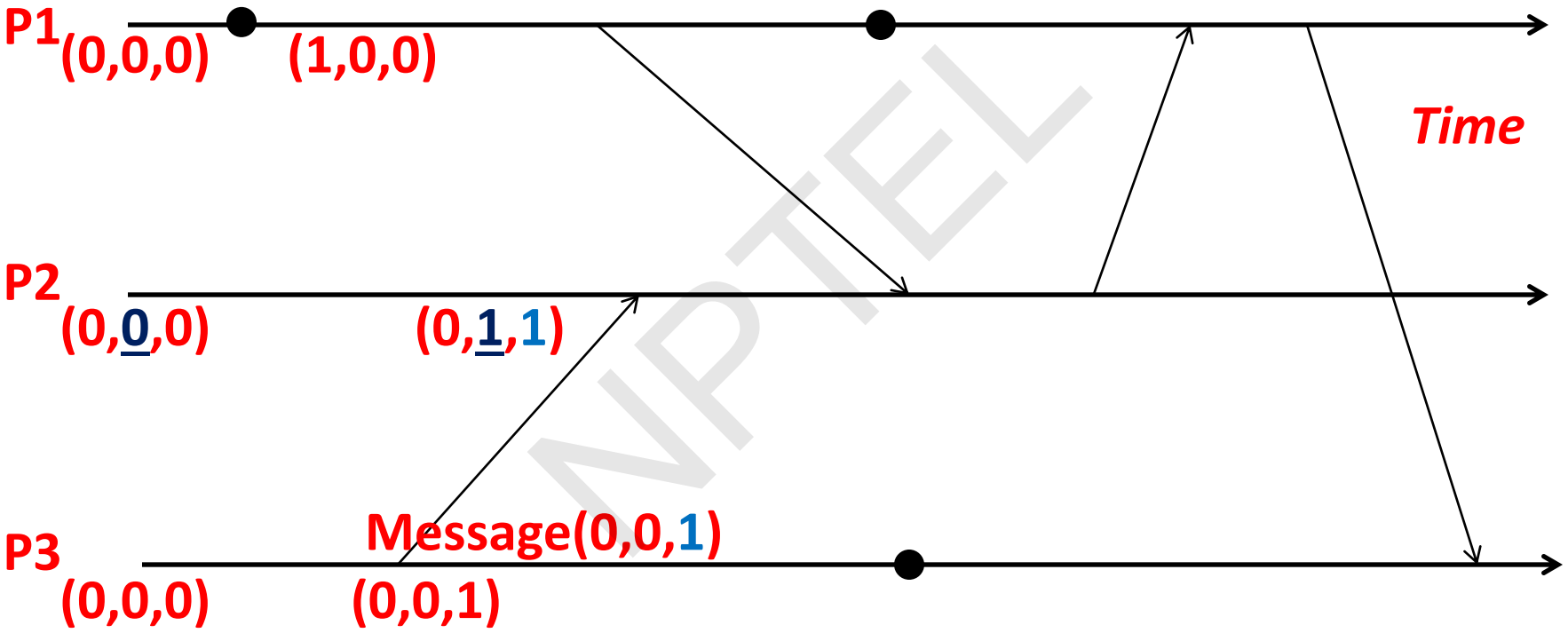


Initial counters (clocks)

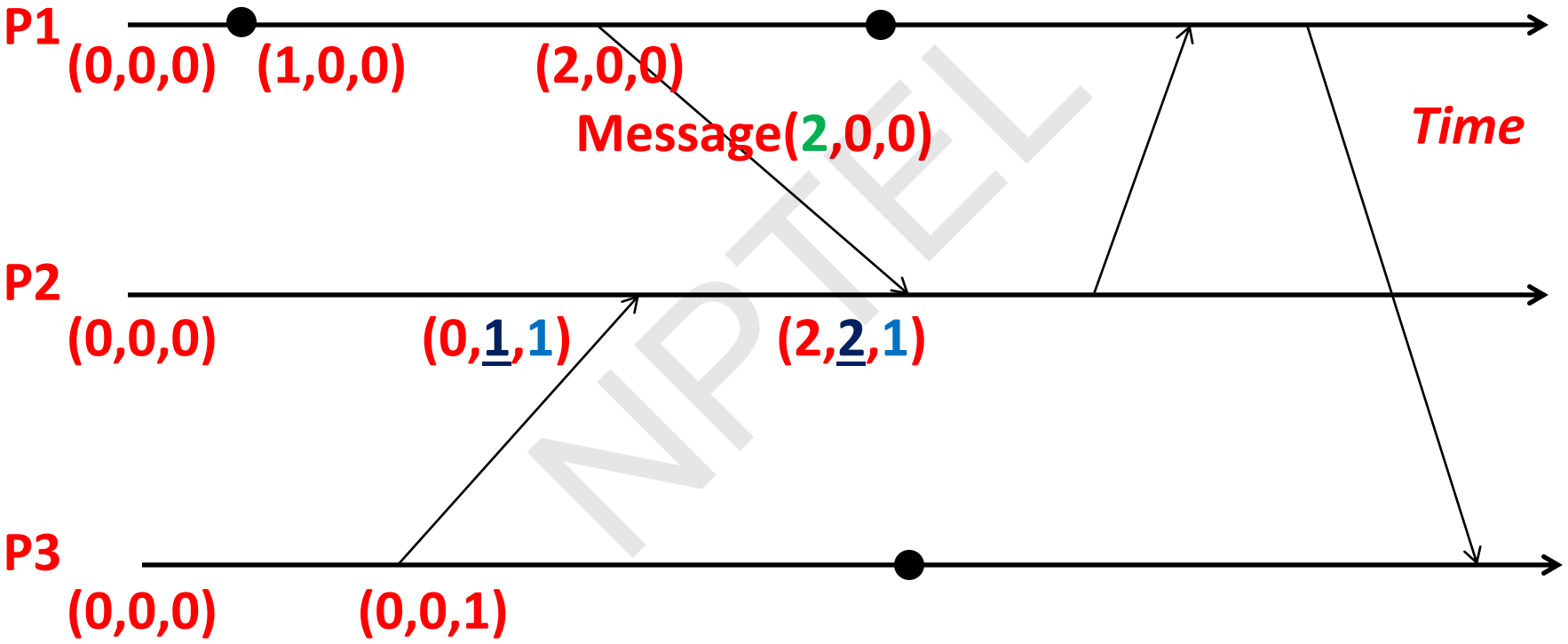
Vector Timestamps



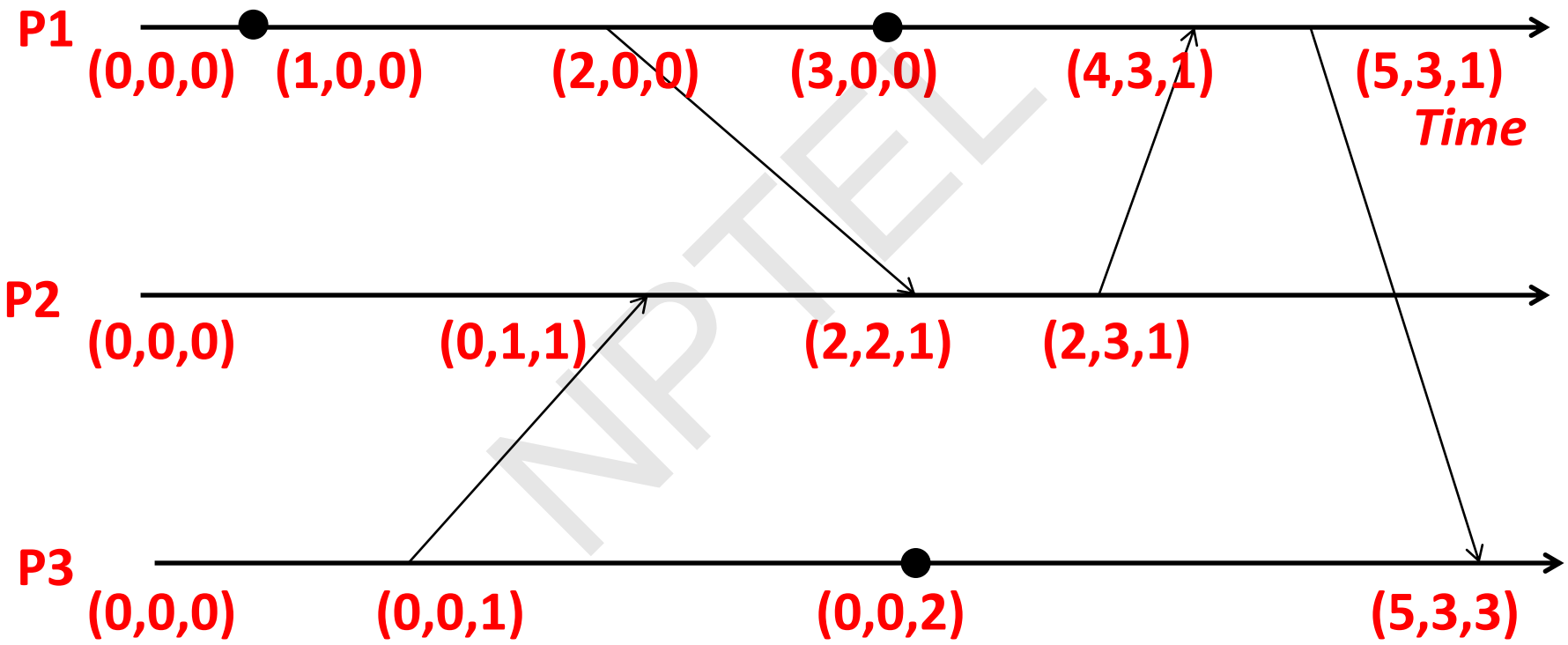
Vector Timestamps



Vector Timestamps



Vector Timestamps



Causally-Related

- $VT_1 = VT_2$,

  iff (if and only if)

  $VT_1[i] = VT_2[i]$, for all $i = 1, \ldots, N$

- $VT_1 \leq VT_2$,

  iff $VT_1[i] \leq VT_2[i]$, for all $i = 1, \ldots, N$

- **Two events are causally related *iff***

  $VT_1 < VT_2$, i.e.,

  iff $VT_1 \leq VT_2$ &

  there exists j such that

  $1 \leq j \leq N$ & $VT_1[j] < VT_2[j]$

… or Not Causally-Related

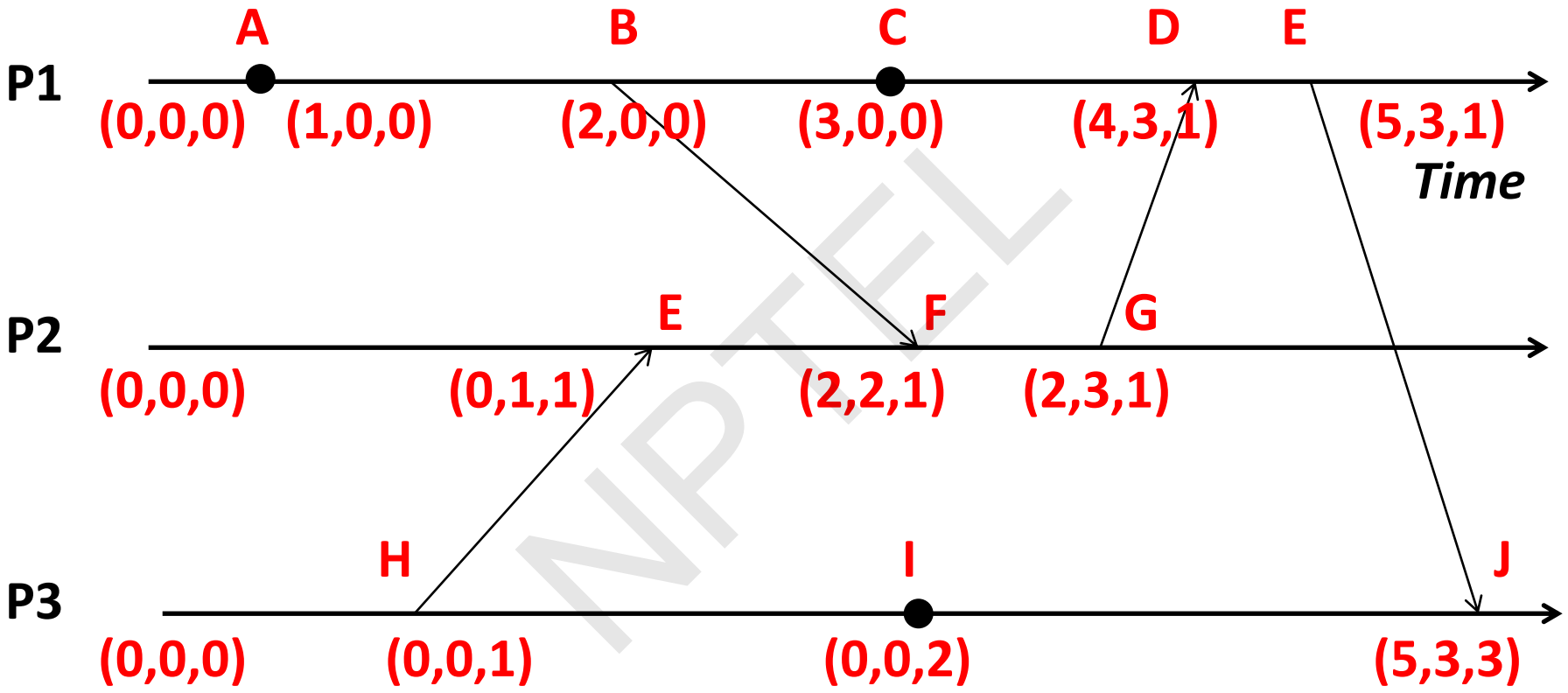
- Two events VT_1 and VT_2 are **concurrent**

  iff

  $$\text{NOT } (VT_1 \leq VT_2) \text{ AND NOT } (VT_2 \leq VT_1)$$

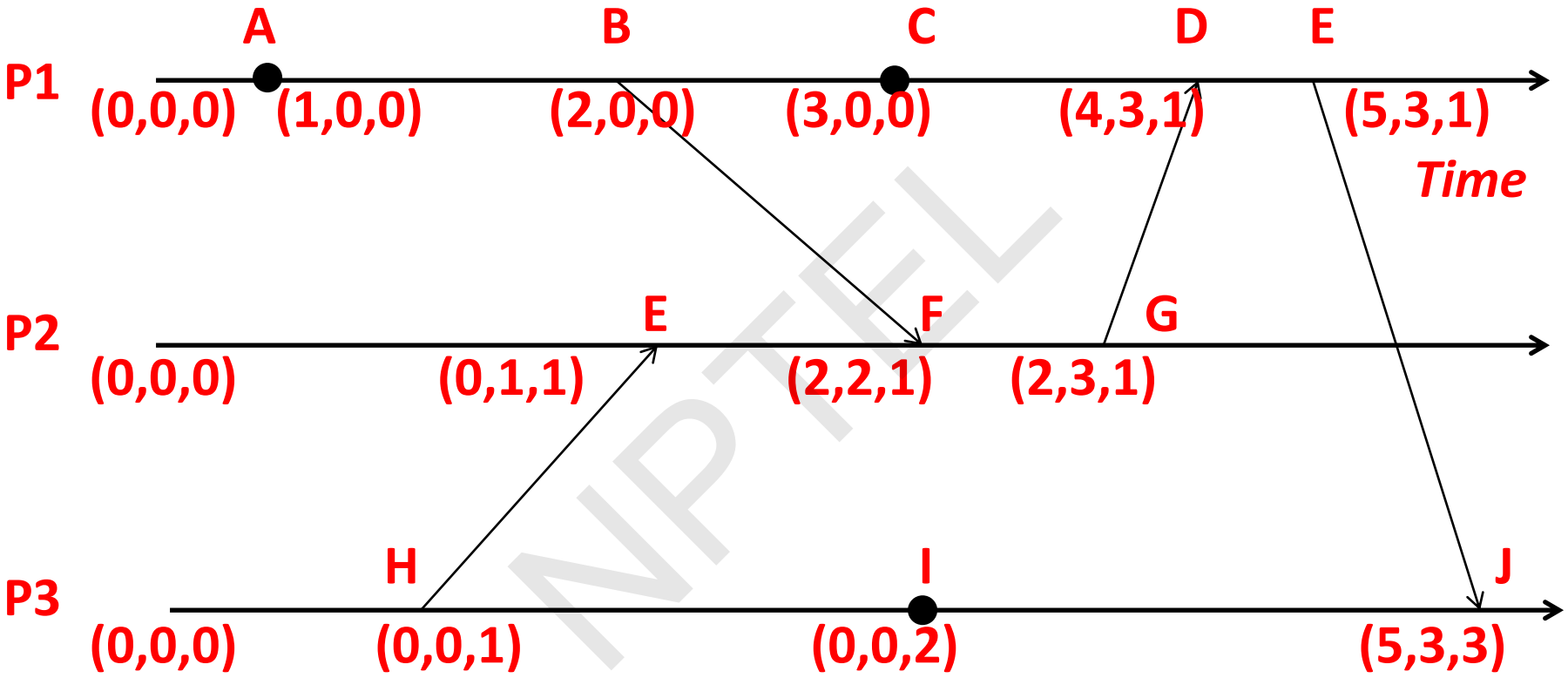
  We'll denote this as $VT_2 \,||\, VT_1$

Obeying Causality



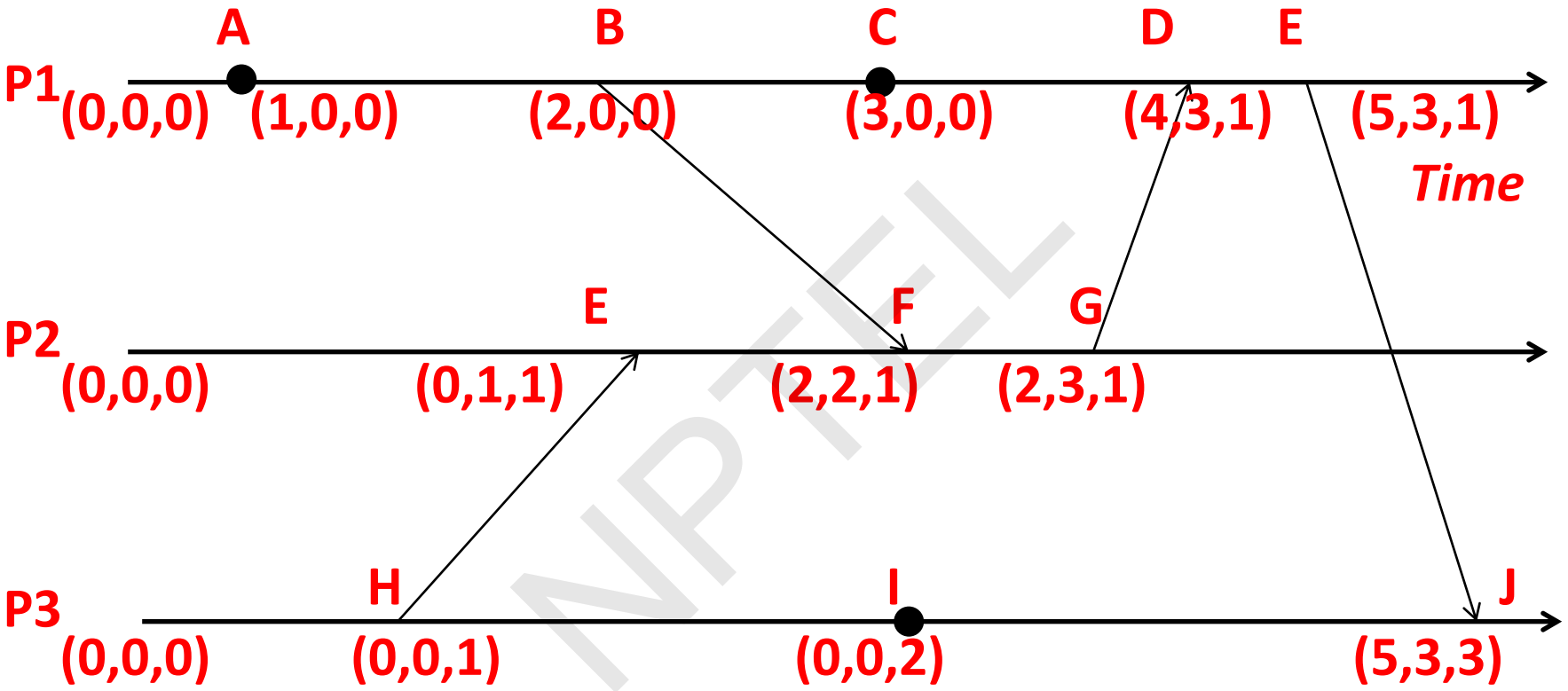
- A → B :: (1,0,0) < (2,0,0)
- B → F :: (2,0,0) < (2,2,1)
- A → F :: (1,0,0) < (2,2,1)

Obeying Causality (2)



- H → G :: (0,0,1) < (2,3,1)
- F → J :: (2,2,1) < (5,3,3)
- H → J :: (0,0,1) < (5,3,3)
- C → J :: (3,0,0) < (5,3,3)

Identifying Concurrent Events



- C & F :: (3,0,0) ||| (2,2,1)
- H & C :: (0,0,1) ||| (3,0,0)
- (C, F) and (H, C) are pairs of *concurrent* events

Summary : Logical Timestamps

- **Lamport timestamp**
  - Integer clocks assigned to events.
  - Obeys causality
  - Cannot distinguish concurrent events.

- **Vector timestamps**
  - Obey causality
  - By using more space, can also identify concurrent events

Conclusion

- Clocks are unsynchronized in an asynchronous distributed system
- But need to order events across processes!

- **Time synchronization:**
  - Christian's algorithm
  - Berkeley algorithm
  - NTP
  - DTP
  - But error a function of RTT

- **Can avoid time synchronization altogether by instead assigning logical timestamps to events**

Global State and Snapshot Recording Algorithms

Dr. Rajiv Misra
Associate Professor
Dept. of Computer Science & Engg.
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss about the Global states (i.e. consistent, inconsistent), Models of communication and Snapshot algorithm *i.e.* Chandy-Lamport algorithm to record the global snapshot.

Snapshots

Here's Snapshot: Collect at a place

Distributed Snapshot

How do you calculate a "global snapshot" in this distributed system? What does a "global snapshot" even mean?

In the Cloud: Global Snapshot

- In a cloud each application or service is running on multiple servers
- Servers handling concurrent events and interacting with each other
- The ability to obtain a "global photograph" or "Global Snapshot" of the system is important
- Some uses of having a global picture of the system
  - ***Checkpointing*:** can restart distributed application on failure
  - ***Garbage collection* of objects:** objects at servers that don't have any other objects (at any servers) with pointers to them
  - **Deadlock detection:** Useful in database transaction systems
  - **Termination of computation:** Useful in batch computing systems

Global State: Introduction

- **Recording the global state** of a distributed system on-the-fly is an important paradigm.
- The **lack of globally shared memory, global clock and unpredictable message delays** in a distributed system make this problem non-trivial.
- This lecture first defines consistent global states and discusses issues to be addressed to compute consistent distributed snapshots.
- Then the algorithm to determine on-the-fly such snapshots is presented.

System Model

- The system consists of a collection of n processes $p_1, p_2, \ldots, p_n$ that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- C_{ij} denotes the channel from process p_i to process p_j and its state is denoted by SC_{ij}.
- The actions performed by a process are modeled as three types of events: Internal events, the message send event and the message receive event.
- For a message m_{ij} that is sent by process p_i to process p_j, let $send(m_{ij})$ and $rec(m_{ij})$ denote its send and receive events.

System Model

- At any instant, the state of process p_i, denoted by LS_i, is a result of the sequence of all the events executed by p_i till that instant.
- For an event e and a process state LS_i, $e \in LS_i$ iff e belongs to the sequence of events that have taken process p_i to state LS_i.
- For an event e and a process state LS_i, $e \notin LS_i$ iff e does not belong to the sequence of events that have taken process p_i to state LS_i.
- For a channel C_{ij}, the following set of messages can be defined based on the local states of the processes p_i and p_j

Transit: $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \bigwedge rec(m_{ij}) \notin LS_j\}$

Consistent Global State

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state *GS* is defined as,

$$GS = \{\cup_i LS_i, \cup_{i,j} SC_{ij}\}$$

- A global state *GS* is a ***consistent global state*** iff it satisfies the following two conditions :

C1: $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$
($\oplus$ is Ex-OR operator)

C2: $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$

Global State of a Distributed System

- In the distributed execution of Figure 6.2:
- A global state GS_1 consisting of local states $\{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$ is **inconsistent** because the state of p_2 has recorded the receipt of message m_{12}, however, the state of p_1 has not recorded its send.
- On the contrary, a global state GS_2 consisting of local states $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$ is **consistent**; all the channels are empty except c_{21} that contains message m_{21}.

Global State of a Distributed System

- A global state GS = $\{U_i LS_i^{xi}, U_{j,k} SC_{jk}^{yj,zk}\}$ is transitless iff
$$\forall i, \forall j : 1 \leq i, j \leq n :: SC_{jk}^{yj,zk} = \emptyset$$
- Thus, all channels are recorded as empty in a transitless global state. A global state is ***strongly consistent*** iff it is transitless as well as consistent. Note that in figure 6.2, the global state of local states $\{LS_1^2, LS_2^3, LS_3^4, LS_4^2\}$ *is* ***strongly consistent***.
- Recording the global state of a distributed system is an important paradigm when one is interested in analyzing, monitoring, testing, or verifying properties of distributed applications, systems, and algorithms. Design of efficient methods for recording the global state of a distributed system is an important problem.

Example:

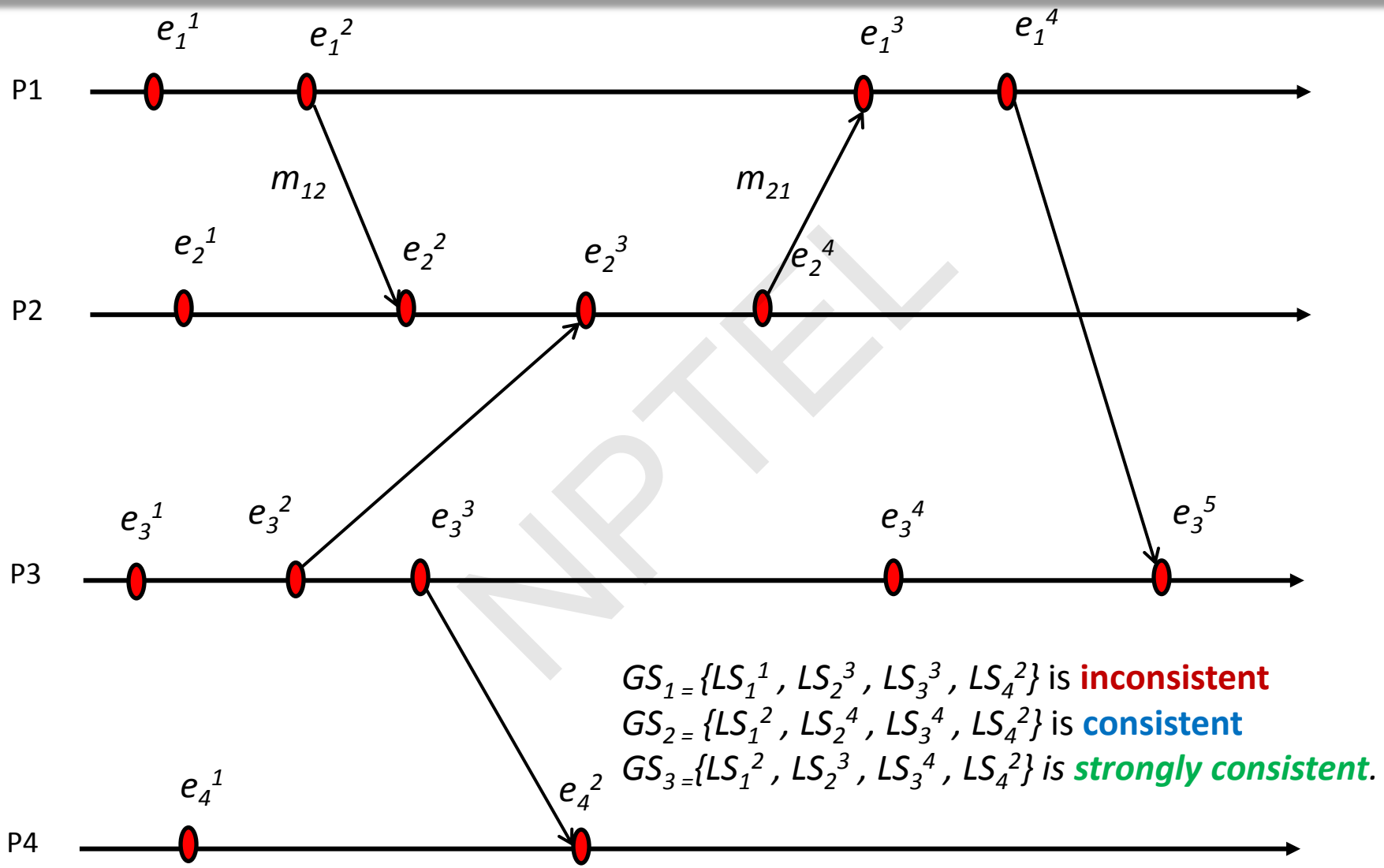


Figure 6.2: The space-time diagram of a distributed execution.

Issues in Recording a Global State

- The following two issues need to be addressed:

I1: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

- -Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
- -Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

I2: How to determine the instant when a process takes its snapshot.

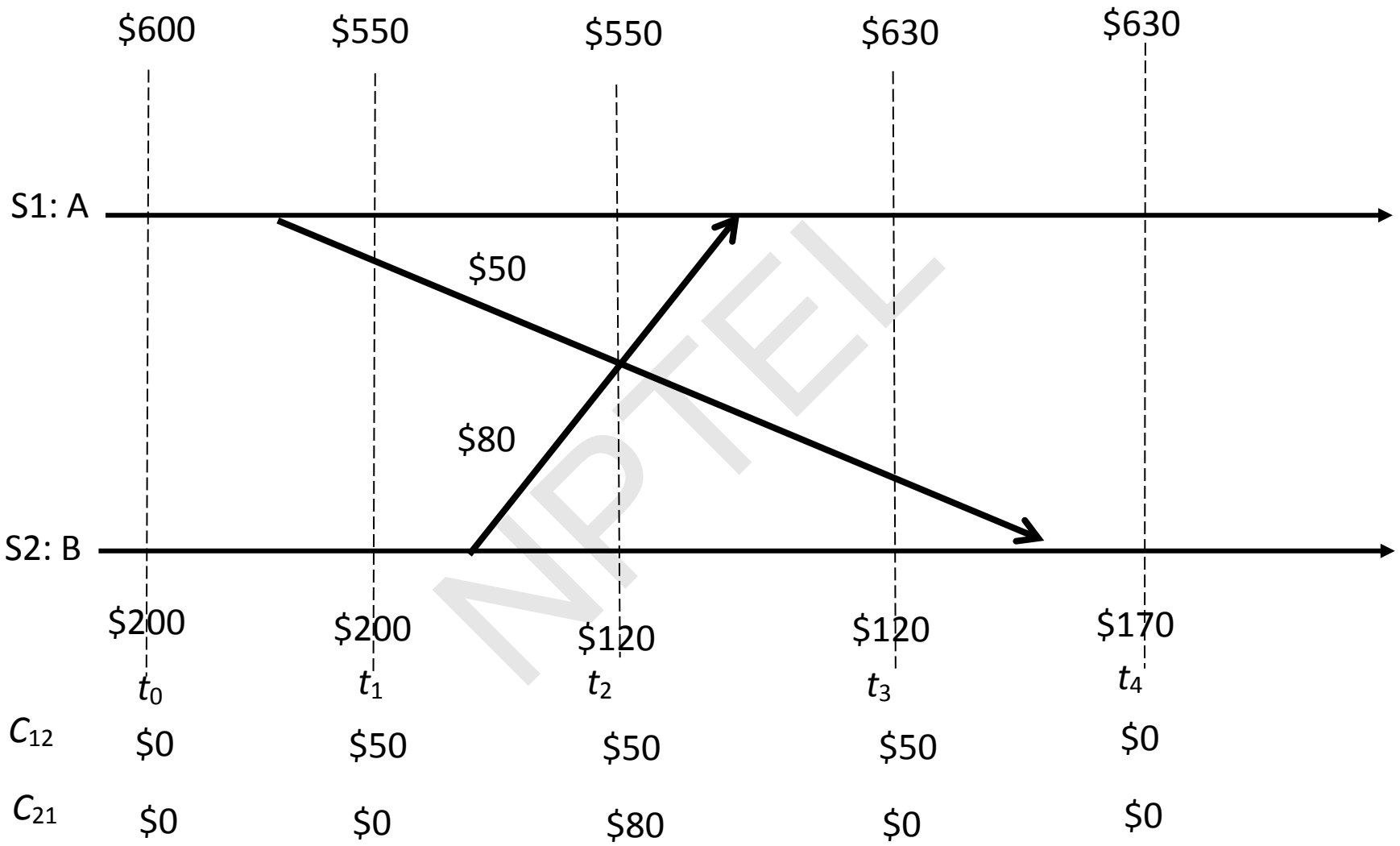
-A process p_j must record its snapshot before processing a message m_{ij} that was sent by process p_i after recording its snapshot.

Example of Money Transfer

- Let S1 and S2 be two distinct sites of a distributed system which maintain bank accounts A and B, respectively. A site refers to a process in this example. Let the communication channels from site S1 to site S2 and from site S2 to site S1 be denoted by C_{12} and C_{21}, respectively. Consider the following sequence of actions, which are also illustrated in the timing diagram of Figure 6.3:
- Time t_0: Initially, Account A=\$600, Account B=\$200, C_{12} =\$0, C_{21} =\$0.
- Time t_1: Site S1 initiates a transfer of \$50 from Account A to Account B.
- Account A is decremented by \$50 to \$550 and a request for \$50 credit to Account B is sent on Channel C_{12} to site S2. Account A=\$550, Account B=\$200, C_{12} =\$50, C_{21} =\$0.

- Time t_2 : Site S2 initiates a transfer of $80 from Account B to Account A.
- Account B is decremented by $80 to $120 and a request for $80 credit to Account A is sent on Channel C_{21} to site S1. Account A=$550, Account B=$120, C_{12} =$50, C_{21} =$80.
- Time t_3: Site S1 receives the message for a $80 credit to Account A and updates Account A.
  Account A=$630, Account B=$120, C_{12} =$50, C_{21} =$0.
- Time t_4: Site S2 receives the message for a $50 credit to Account B and updates Account B.
  Account A=$630, Account B=$170, C_{12} =$0, C_{21} =$0.

T_3: Site S1 receives the message for a $80 credit to Account A and updates



T_4: Site S2 receives the message for a $50 credit to Account B and updates Account B

- Suppose the local state of Account A is recorded at time t_0 to show \$600 and the local state of Account B and channels C_{12} and C_{21} are recorded at time t_2 to show \$120, \$50, and \$80, respectively. Then the recorded global state shows \$850 in the system. An extra \$50 appears in the system.
- **The reason for the inconsistency** is that Account A's state was recorded before the \$50 transfer to Account B using channel C_{12} was initiated, whereas channel C_{12}'s state was recorded after the \$50 transfer was initiated.
- This simple example shows that recording a consistent global state of a distributed system is not a trivial task. Recording activities of individual components must be coordinated appropriately.

Model of Communication

- Recall, there are three models of communication: FIFO, non-FIFO, and Co.

- In **FIFO model**, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.

- In **non-FIFO model**, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

- A system that supports **causal delivery** of messages satisfies the following property: "For any two messages m_{ij} and m_{kj}, if $send(m_{ij}) \rightarrow send(m_{kj})$, then $rec(m_{ij}) \rightarrow rec(m_{kj})$"

Snapshot algorithm for FIFO channels

Chandy-Lamport algorithm:

- The **Chandy-Lamport** algorithm uses a **control message**, called a ***marker*** whose role in a **FIFO system** is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a ***marker***, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

Chandy-Lamport Algorithm

- The algorithm can be initiated by any process by executing the "**Marker Sending Rule**" by which it records its local state and sends a marker on each outgoing channel.
- A process executes the "**Marker Receiving Rule**" on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the "Marker Sending Rule" to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.

Chandy-Lamport Algorithm

Marker Sending Rule for process *i*

1) Process *i* records its state.

2) For each outgoing channel C on which a marker has not been sent, *i* sends a marker along C before *i* sends further messages along C.

Marker Receiving Rule for process *j*

On receiving a marker along channel C:

if *j* has not recorded its state **then**

Record the state of C as the empty set

Follow the "Marker Sending Rule"

else

Record the state of C as the set of messages received along C after *j*'s state was recorded and before *j* received the marker along C

Properties of the recorded global state

- The recorded global state may not correspond to any of the global states that occurred during the computation.

- Consider two possible executions of the snapshot algorithm (shown in Figure 6.4) for the previous money transfer example .

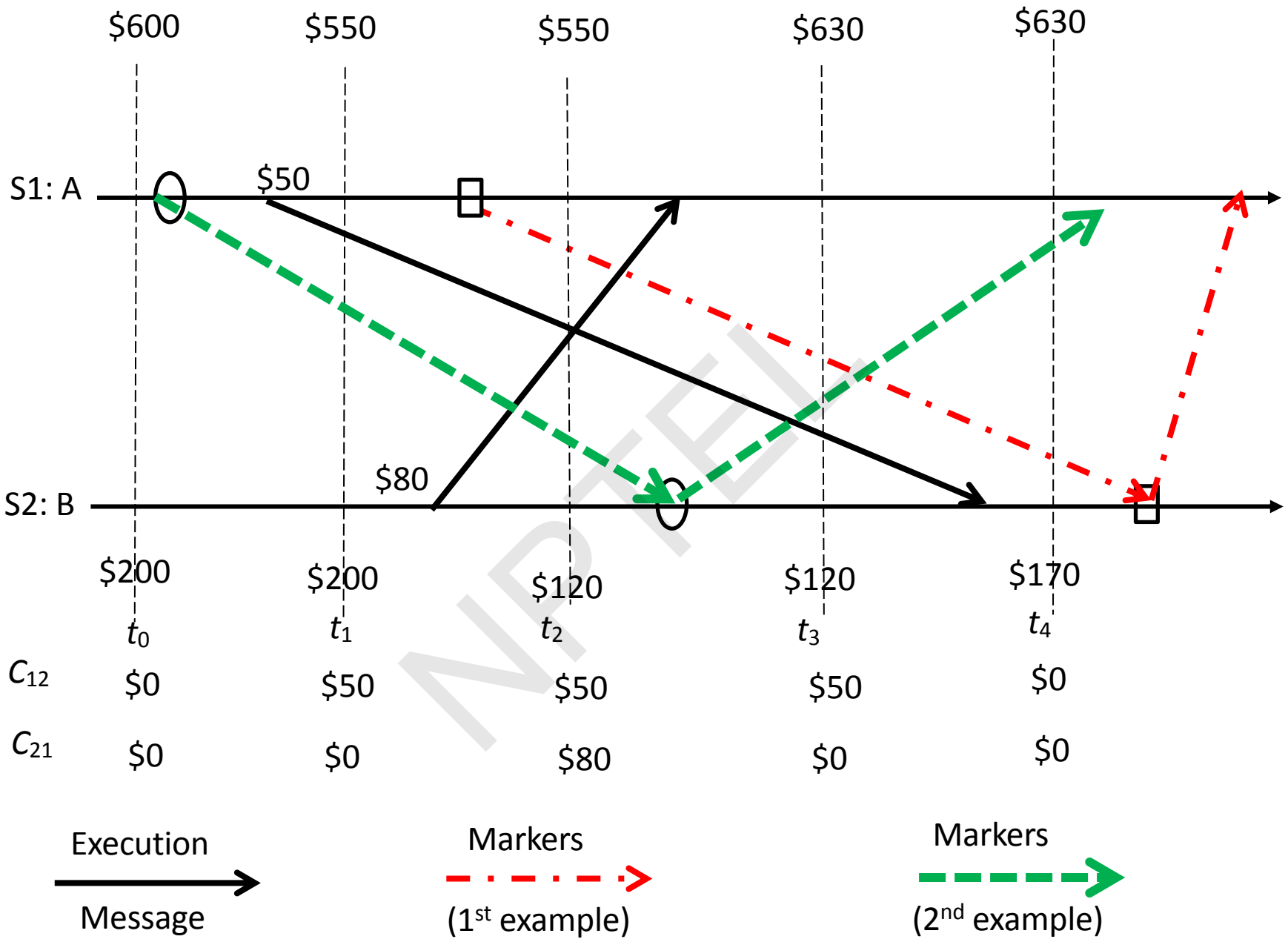


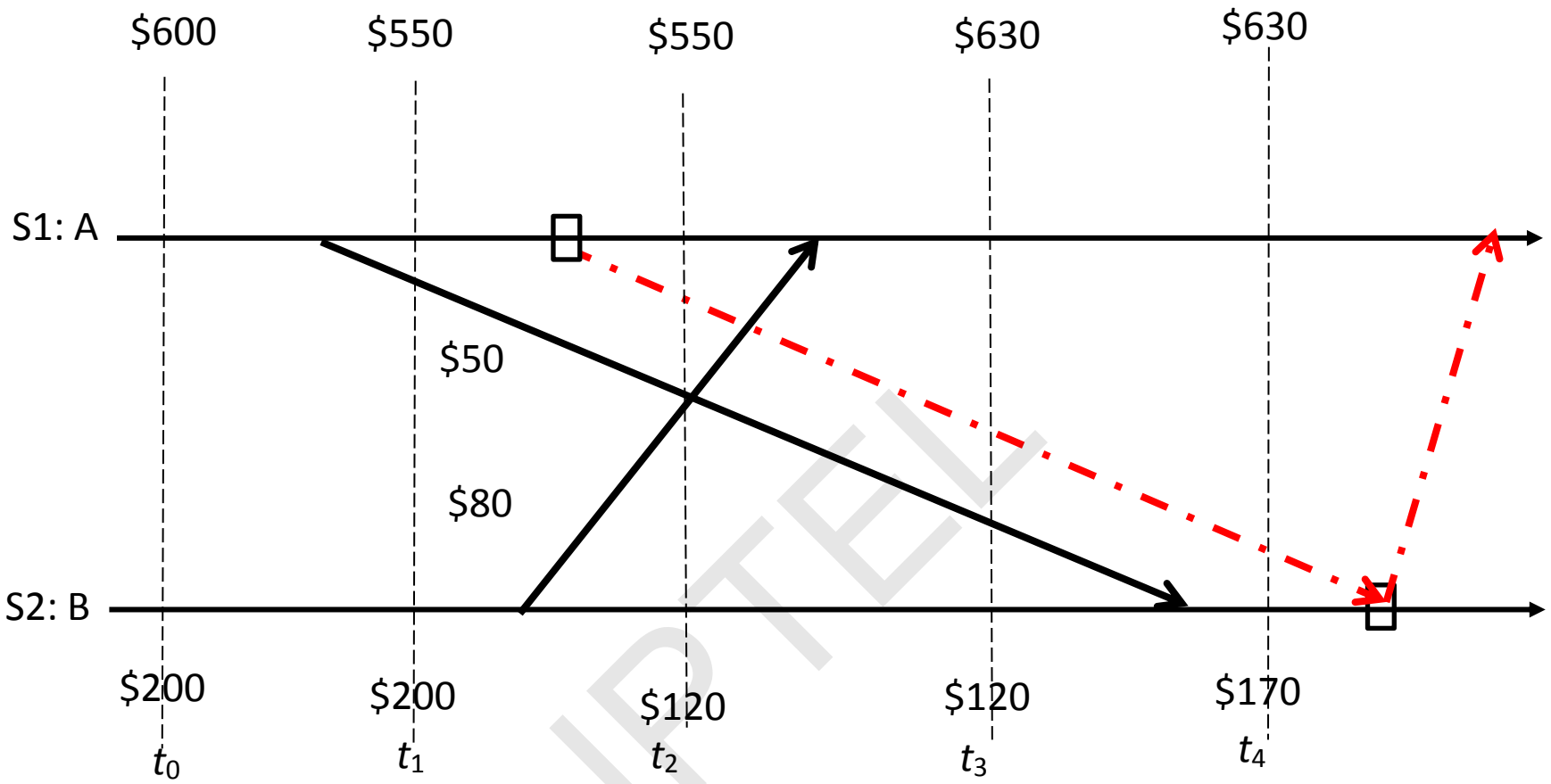
Figure 6.4: Timing diagram of two possible executions of the banking example

Properties of the recorded global state

1. (Markers shown using red dashed-and-dotted arrows.)

Let site S1 initiate the algorithm just after t_1. Site S1 records its local state (account A=$550) and sends a marker to site S2. The marker is received by site S2 after t_4. When site S2 receives the marker, it records its local state (account B=$170), the state of channel C_{12} as $0, and sends a marker along channel C_{21}. When site S1 receives this marker, it records the state of channel C_{21} as $80. The $800 amount in the system is conserved in the recorded global state,

$$A = \$550,\ B = \$170,\ C_{12} = \$0,\ C_{21} = \$80$$



A = $550 **B = $170** **$C_{12}$ = $0** **$C_{21}$ = $80**

The $800 amount in the system is conserved in the recorded global state

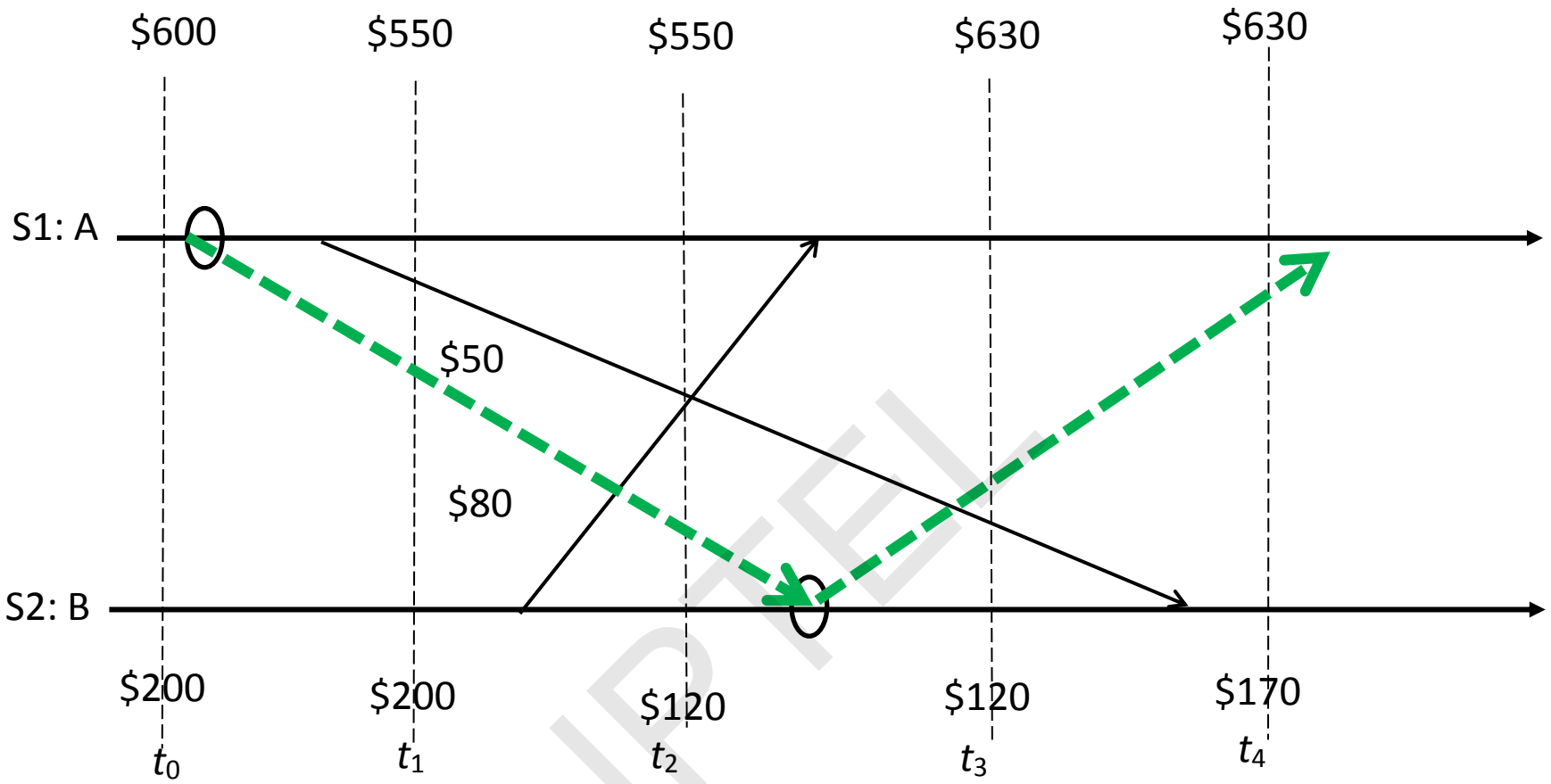
Figure 6.4: Timing diagram of two possible executions of the banking example

Properties of the recorded global state

2. (Markers shown using green dotted arrows.)

Let site S1 initiate the algorithm just after t_0 and before sending the \$50 for S2. Site S1 records its local state (account A = \$600) and sends a marker to site S2. The marker is received by site S2 between t_2 and t_3. When site S2 receives the marker, it records its local state (account B = \$120), the state of channel C_{12} as \$0, and sends a marker along channel C_{21}. When site S1 receives this marker, it records the state of channel C_{21} as \$80. The \$800 amount in the system is conserved in the recorded global state,

$$A = \$600, B = \$120, C_{12} = \$0, C_{21} = \$80$$



A = $600 **B = $120** **$C_{12}$ = $0** **$C_{21}$ = $80**

The $800 amount in the system is conserved in the recorded global state

Figure 6.4: Timing diagram of two possible executions of the banking example

Properties of the recorded global state

- In both these possible runs of the algorithm, the recorded global states never occurred in the execution.
- This happens because a process can change its state asynchronously before the markers it sent are received by other sites and the other sites record their states.
  - But the system could have passed through the recorded global states in some equivalent executions.
  - The recorded global state is a valid state in an equivalent execution and if a stable property (i.e., a property that persists) holds in the system before the snapshot algorithm begins, it holds in the recorded global snapshot.
- Therefore, a recorded global state is useful in detecting stable properties.

Conclusion

- Recording global state of a distributed system is an important paradigm in the design of the distributed systems and the design of efficient methods of recording the global state is an important issue.

- This lecture first discussed a formal definition of the **global state of a distributed system and issues** related to its capture; then we have discussed the **Chandy-Lamport Algorithm** to record a snapshot of a distributed system.

Distributed Mutual Exclusion

Dr. Rajiv Misra

Associate Professor

Dept. of Computer Science & Engg.

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Content of this Lecture:

- In this lecture, we will discuss about the ‘Concepts of Mutual Exclusion’, Classical algorithms for distributed computing systems and Industry systems for Mutual Exclusion.

Need of Mutual Exclusion in Cloud

- **Bank's Servers in the Cloud:** Two customers make simultaneous deposits of 10,000 Rs. into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of 1000 Rs. concurrently from the bank's cloud server
  - Both ATMs add 10,000 Rs. to this amount (locally at the ATM)
  - Both write the final amount to the server
  - **What's wrong? 11000Rs. (or 21000Rs.)**

Need of Mutual Exclusion in Cloud

- **Bank's Servers in the Cloud:** Two customers make simultaneous deposits of 10,000 Rs. into your bank account, each from a separate ATM.
  - Both ATMs read initial amount of 1000 Rs. concurrently from the bank's cloud server
  - Both ATMs add 10,000 Rs. to this amount (locally at the ATM)
  - Both write the final amount to the server
  - You lost 10,000 Rs.!
- The ATMs need ***mutually exclusive*** access to your account entry at the server
  - or, mutually exclusive access to executing the code that modifies the account entry

Some other Mutual Exclusion use

- **Distributed File systems**
  - Locking of files and directories
- **Accessing objects** in a safe and consistent way
  - Ensure at most one server has access to object at any point of time
- **Server coordination**
  - Work partitioned across servers
  - Servers coordinate using locks
- **In industry**
  - Chubby is Google's locking service
  - Many cloud stacks use Apache Zookeeper for coordination among servers

Problem Statement for Mutual Exclusion

- ***Critical Section*** Problem: Piece of code (at all processes) for which we need to ensure there is at most one process executing it at any point of time.
- Each process can call three functions
  - **enter()** to enter the critical section (CS)
  - **AccessResource()** to run the critical section code
  - **exit()** to exit the critical section

Bank Example

ATM1:

```
enter(S);
 // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
exit(S); // exit
```

ATM2:

```
enter(S);
 // AccessResource()
obtain bank amount;
add in deposit;
update bank amount;
// AccessResource() end
exit(S); // exit
```

Approaches to Solve Mutual Exclusion

- **Single OS:**
  - If all processes are running in one OS on a machine (or VM), then
  - Semaphores, mutexes, condition variables, monitors, etc.

Approaches to Solve Mutual Exclusion (2)

- Distributed system:
  - Processes communicating by passing messages

Need to guarantee 3 properties:

- **Safety** (essential): At most one process executes in CS (Critical Section) at any time
- **Liveness** (essential): Every request for a CS is granted eventually
- **Fairness** (desirable): Requests are granted in the order they were made

Processes Sharing an OS: Semaphores

- Semaphore == an integer that can only be accessed via two special functions
- Semaphore S=1; // Max number of allowed accessors

1. **wait(S)** (or **P(S)** or **down(S)**):

enter()

```
while(1) { // each execution of the while loop is atomic
        if (S > 0) {
            S--;
            break;
        }
 }
```

Each while loop execution and S++ are each atomic operations – supported via hardware instructions such as compare-and-swap, test-and-set, etc.

exit() 2. **signal(S)** (or **V(S)** or **up(s)**):

```
S++; // atomic
```

Bank Example Using Semaphores

```
Semaphore S=1; // shared
ATM1:
    wait(S);
     // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    signal(S); // exit
```

```
Semaphore S=1; // shared
ATM2:
    wait(S);
     // AccessResource()
    obtain bank amount;
    add in deposit;
    update bank amount;
    // AccessResource() end
    signal(S); // exit
```

Next

- In a distributed system, cannot share variables like semaphores

- So how do we support mutual exclusion in a distributed system?

System Model

- Before solving any problem, specify its System Model:
  - Each pair of processes is connected by reliable channels (such as TCP).
  - Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
  - Processes do not fail.
    - Fault-tolerant variants exist in literature.

Central Solution

- Elect a central master (or leader)
  - Use one of our election algorithms!
- Master keeps
  - A **queue** of waiting requests from processes who wish to access the CS
  - A special **token** which allows its holder to access CS
- Actions of any process in group:
  - **enter()**
    - Send a request to master
    - Wait for token from master
  - **exit()**
    - Send back token to master

Central Solution

- Master Actions:
  - On receiving a request from process P*i*

    if (master has token)

        Send token to P*i*

    else

        Add P*i* to queue
  - On receiving a token from process P*i*

    if (queue is not empty)

        Dequeue head of queue (say P*j*), send that process the token

    else

        Retain token

Analysis of Central Algorithm

- Safety – at most one process in CS
  - Exactly one token
- Liveness – every request for CS granted eventually
  - With *N* processes in system, queue has at most *N* processes
  - If each process exits CS eventually and no failures, liveness guaranteed
- FIFO Ordering is guaranteed, in order of requests received at master

Performance Analysis

Efficient mutual exclusion algorithms use **fewer messages**, and make processes **wait for shorter durations** to access resources.

Three metrics:

- ***Bandwidth*:** the total number of messages sent in each *enter* and *exit* operation.
- ***Client delay*:** the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)

  (We will prefer mostly the enter operation.)
- ***Synchronization delay*:** the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)

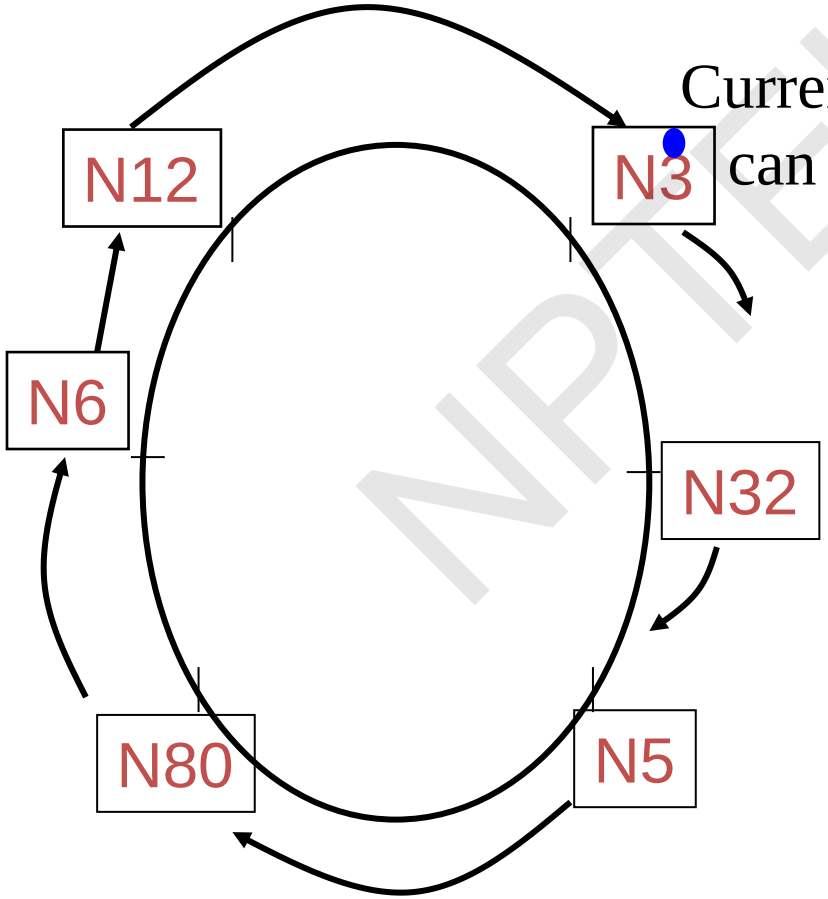
Analysis of Central Algorithm

- ***Bandwidth*:** the total number of messages sent in each *enter* and *exit* operation.
  - 2 messages for enter
  - 1 message for exit
- ***Client delay*:** the delay incurred by a process at each enter and exit operation (when *no* other process is in, or waiting)
  - 2 message latencies (request + grant)
- ***Synchronization delay*:** the time interval between one process exiting the critical section and the next process entering it (when there is *only one* process waiting)
  - 2 message latencies (release + grant)

But…

- The master is the performance bottleneck and SPoF (single point of failure)

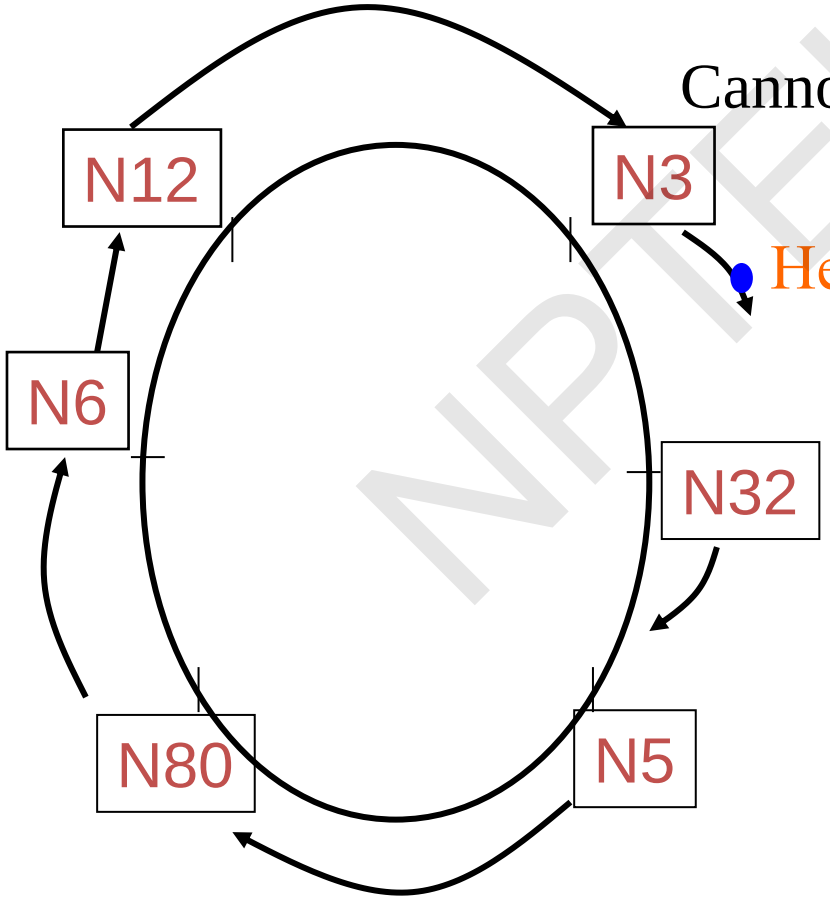
Ring-based Mutual Exclusion



Currently holds token, can access CS

Token: •

Ring-based Mutual Exclusion

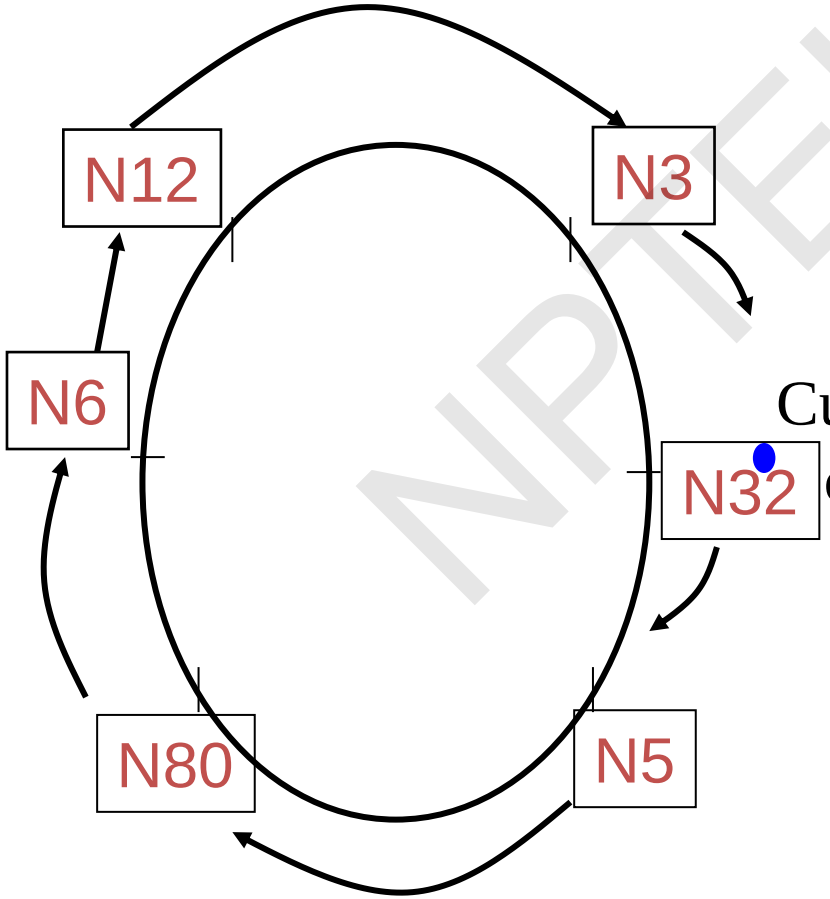


Cannot access CS anymore

Here's the token!

Token:

Ring-based Mutual Exclusion



Currently holds token, can access CS

Token: •

Ring-based Mutual Exclusion

- *N* Processes organized in a virtual ring
- Each process can send message to its successor in ring
- Exactly 1 token
- enter()
  - Wait until you get token
- exit() // already have token
  - Pass on token to ring successor
- If receive token, and not currently in enter(), just pass on token to ring successor

Analysis of Ring-based Mutual Exclusion

- Safety
  - Exactly one token
- Liveness
  - Token eventually loops around ring and reaches requesting process (no failures)
- Bandwidth
  - Per enter(), 1 message by requesting process but up to *N* messages throughout system
  - 1 message sent per exit()

Analysis of Ring-Based Mutual Exclusion (2)

- Client delay: 0 to *N* message transmissions after entering enter()
  - Best case: already have token
  - Worst case: just sent token to neighbor
- Synchronization delay between one process' exit() from the CS and the next process' enter():
  - Between 1 and (*N-1*) message transmissions.
  - Best case: process in enter() is successor of process in exit()
  - Worst case: process in enter() is predecessor of process in exit()

Next

- Client/Synchronization delay to access CS still O(*N*) in Ring-Based approach.
- Can we make this faster?

System Model

- Before solving any problem, specify its System Model:
  - Each pair of processes is connected by reliable channels (such as TCP).
  - Messages are eventually delivered to recipient, and in FIFO (First In First Out) order.
  - Processes do not fail.

Lamport's Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site S_i keeps a queue, ***request_queue***$_i$ which contains mutual exclusion requests ordered by their timestamps.
- This algorithm requires communication channels to deliver messages the **FIFO order**. Three types of messages are used- **Request, Reply and Release**. These messages with timestamps also **updates** logical clock.

The Algorithm

Requesting the critical section:

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$.
  ((ts_i, i) denotes the timestamp of the request.)
- When S_j receives the REQUEST(ts_i, i) message from site S_i, S_j places site S_i's request on $request\_queue_j$ and it returns a *timestamped* REPLY message to S_i.

Executing the critical section: Site S_i enters the CS when the following two conditions hold:

L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i's request is at the top of $request\_queue_i$.

The Algorithm

Releasing the critical section:

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a *timestamped* RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

Correctness

Theorem: Lamport's algorithm achieves mutual exclusion.

Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*.
- This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their *request_queues* and condition L1 holds at them. Without loss of generality, assume that S_i's request has smaller timestamp than the request of S_j.
- From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of S_i must be present in $request\_queue_j$ when S_j was executing its CS. This implies that S_j's own request is at the top of its own *request_queue* when a smaller timestamp request, S_i's request, is present in the $request\ queue_j$ – a contradiction!

Correctness

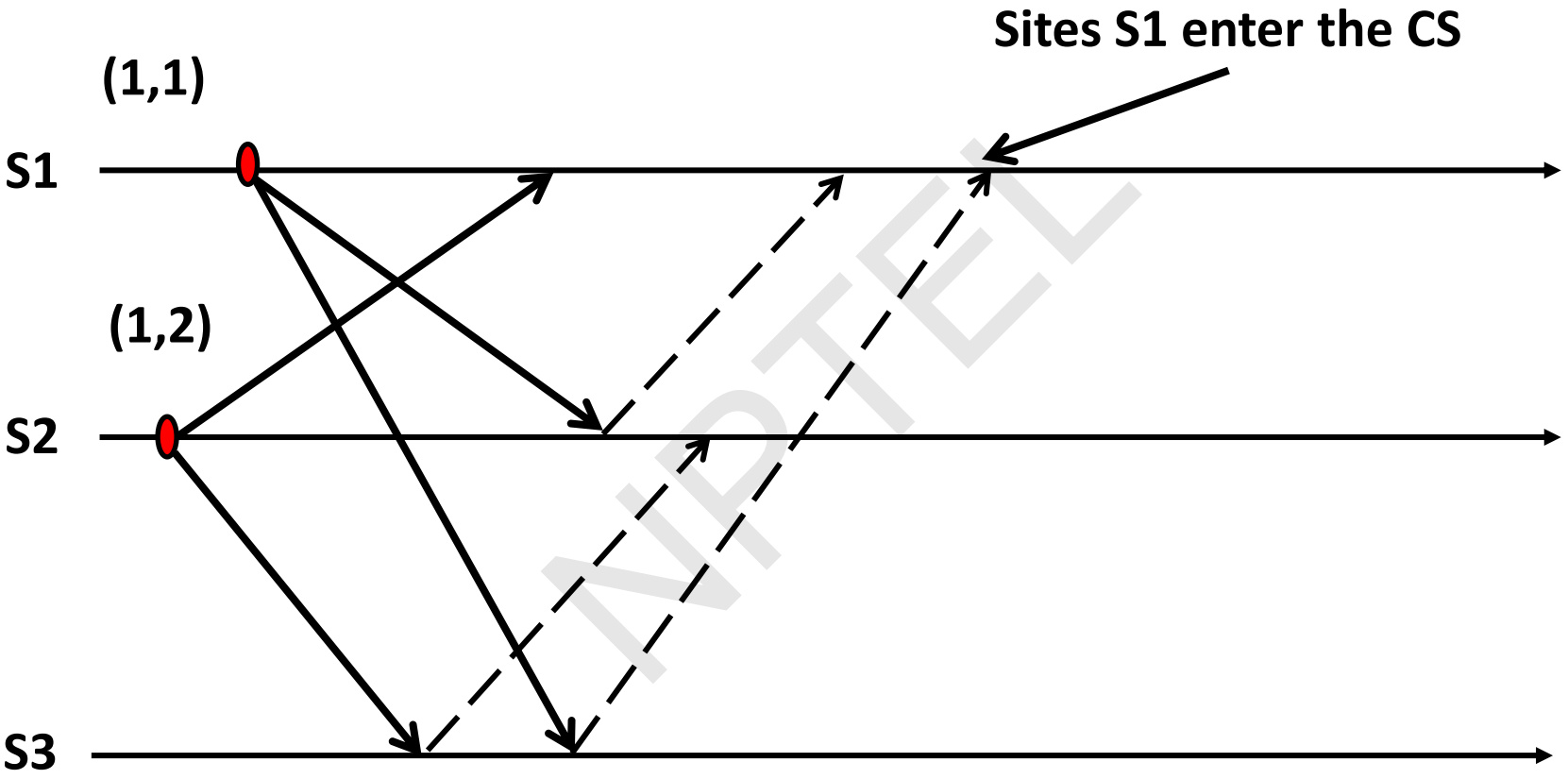
Theorem: Lamport's algorithm is fair.

Proof:

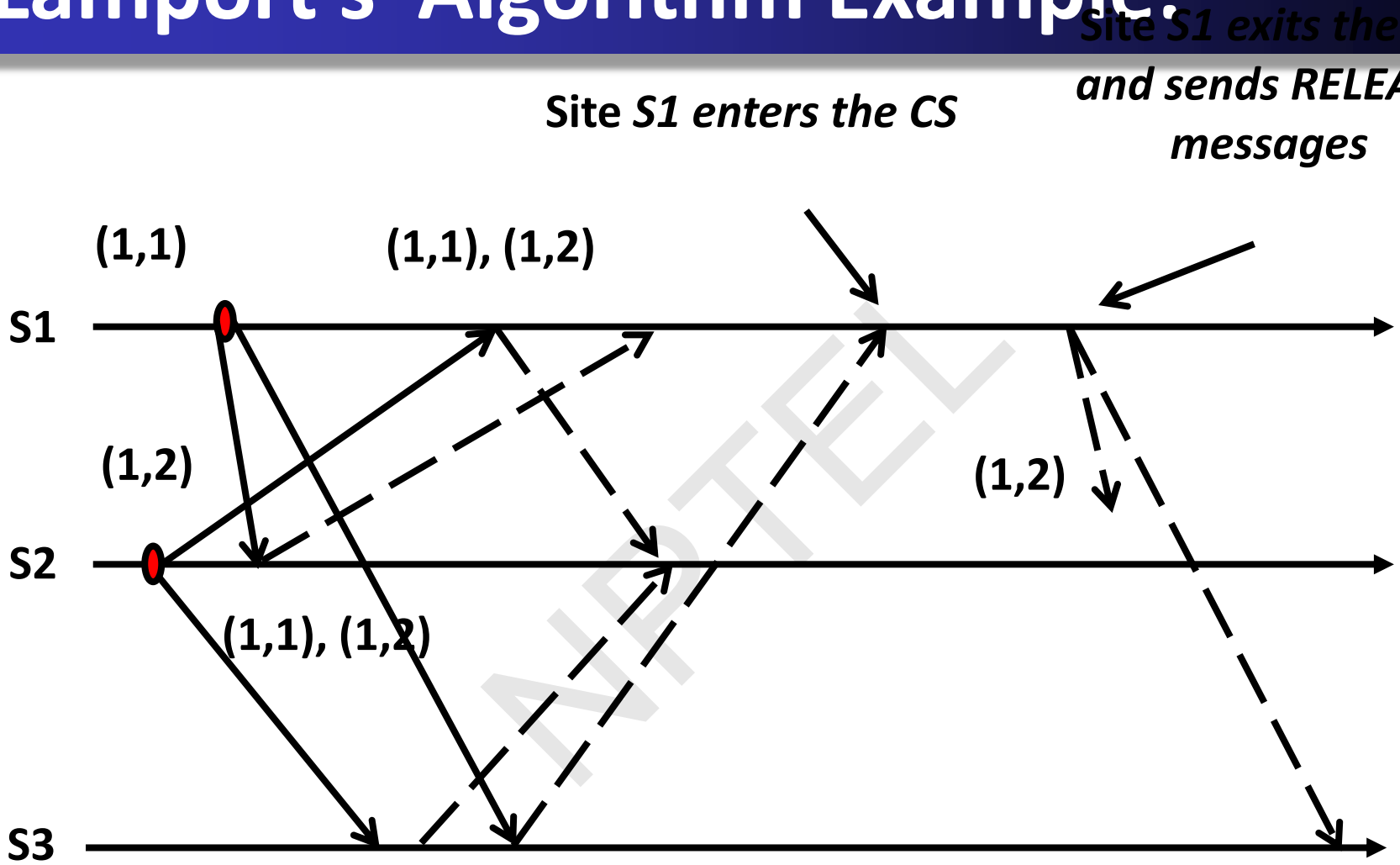
- The proof is by contradiction. Suppose a site S_i's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i.
- For S_j to execute the CS, it has to satisfy the conditions **L1** and **L2**. This implies that at some instant in time say t, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But *request_queue* at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i's request must be placed ahead of the S_j's request in the $request\_queue_j$. This is a contradiction!

Lamport's Algorithm Example:

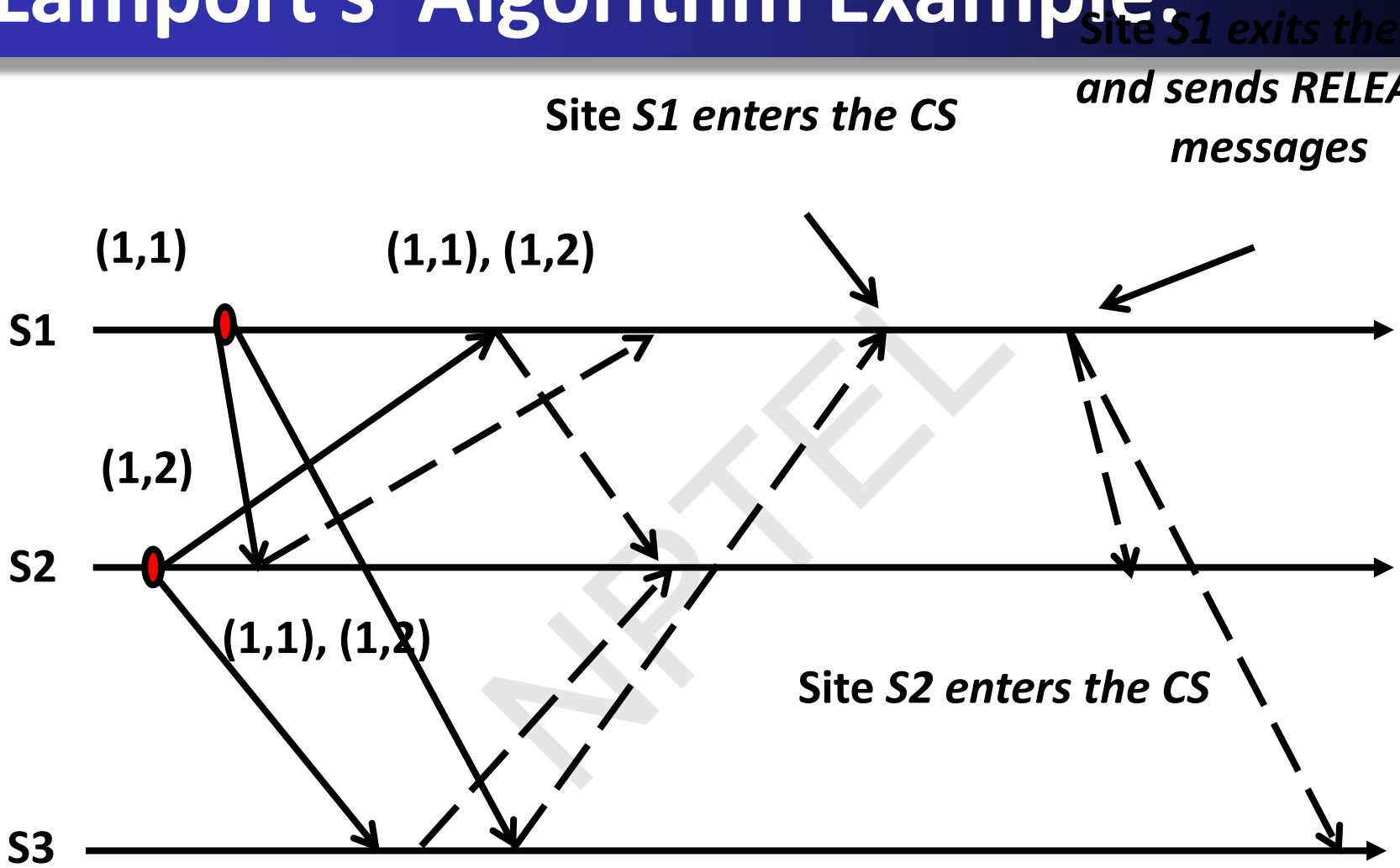
Sites S1 and S2 are Making Requests for the CS



Lamport's Algorithm Example:



Lamport's Algorithm Example:



Performance

- For each CS execution, Lamport's algorithm requires $(N - 1)$ REQUEST messages, $(N - 1)$ REPLY messages, and $(N - 1)$ RELEASE messages.

- Thus, Lamport's algorithm requires $3(N - 1)$ messages per CS invocation.

- Synchronization delay in the algorithm is T .

An Optimization

- In Lamport's algorithm, REPLY messages can be omitted in certain situations. For example, if site S_j receives a REQUEST message from site S_i after it has sent its own REQUEST message with timestamp higher than the timestamp of site S_i's request, then site S_j need not send a REPLY message to site S_i.

- This is because when site S_i receives site S_j's request with timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request which is still pending.

- With this optimization, Lamport's algorithm requires between $3(N-1)$ and $2(N-1)$ messages per CS execution.

Ricart-Agrawala's Algorithm

- Classical algorithm from 1981
- Invented by Glenn Ricart (NIH) and Ashok Agrawala (U. Maryland)

- No token
- Uses the notion of causality and multicast
- Has lower waiting time to enter CS than Ring-Based approach

Key Idea: Ricart-Agrawala Algorithm

- enter() at process P*i*
  - multicast a request to all processes
    - Request: <T, P*i*>, where T = current Lamport timestamp at P*i*
  - Wait until *all* other processes have responded positively to request
- Requests are granted in order of causality
- <T, P*i*> is used lexicographically: P*i* in request <T, P*i*> is used to break ties (since Lamport timestamps are not unique for concurrent events)

Ricart-Agrawala Algorithm

- The Ricart-Agrawala algorithm assumes the communication channels are **FIFO**. The algorithm uses two types of messages: ***REQUEST*** and ***REPLY***.
- A process sends a **REQUEST** message to all other processes to request their permission to enter the critical section. A process sends a **REPLY** message to a process to give its permission to that process.
- Processes use **Lamport-style logical clocks** to assign a timestamp to critical section requests and timestamps are used to decide the priority of requests.
- Each process p_i maintains the Request-Deferred array, RD_i, the size of which is the same as the number of processes in the system.
- Initially, $\forall i \quad \forall j$: $RD_i[j]=0$. Whenever p_i defer the request sent by p_j, it sets $RD_i[j]=1$ and after it has sent a REPLY message to p_j, it sets $RD_i[j]=0$.

Description of the Algorithm

Requesting the critical section:

(a) When a site S_i wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS, or if the site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. Otherwise, the reply is deferred and S_j sets RD_j [i]=1

Executing the critical section:

(c) Site S_i enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

Contd…

Releasing the critical section:

(d) When site S_i exits the CS, it sends all the deferred REPLY messages: $\forall j$ if RD_i [j]=1, then send a REPLY message to S_j and set RD_i [j]=0.

Notes:

- When a site receives a message, it updates its clock using the timestamp in the message.
- When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.

Correctness

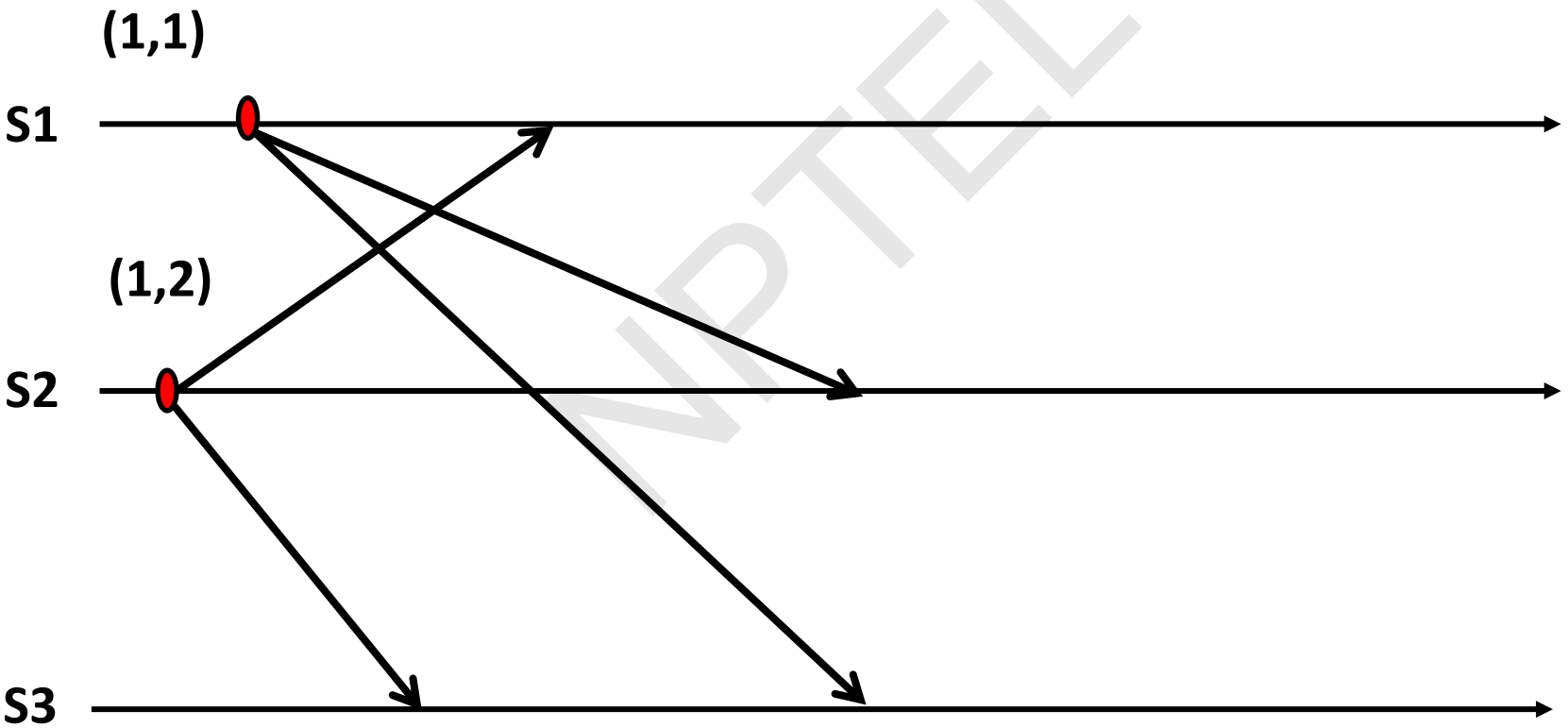
Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.

Proof:

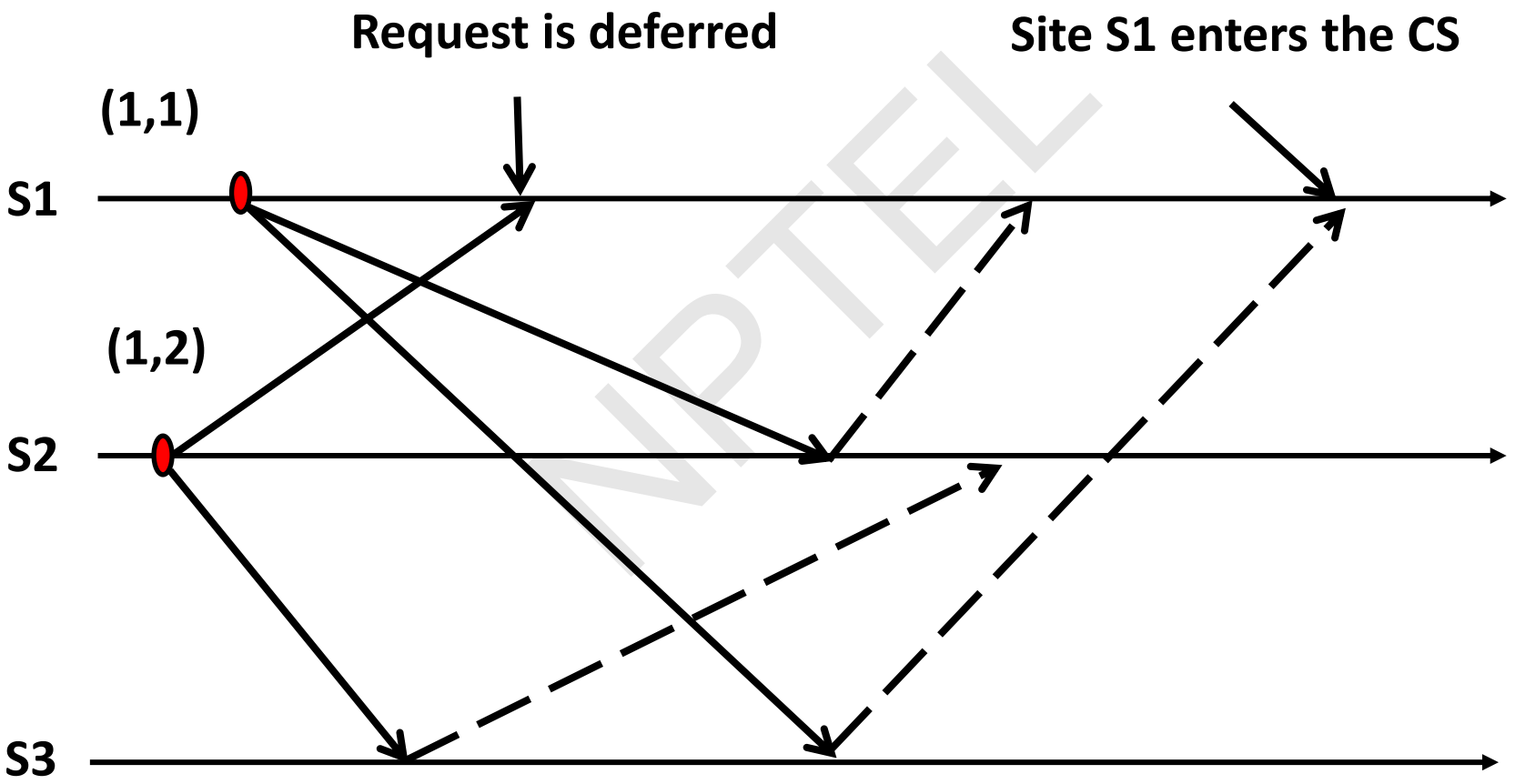
- Proof is by contradiction. Suppose two sites S_i and S_j ' are executing the CS concurrently and S_i's request has higher priority than the request of S_j. Clearly, S_i received S_j's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a REPLY to S_j (in response to S_j's request) before S_i exits the CS.
- However, this is impossible because S_j's request has lower priority.
- Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

Ricart–Agrawala algorithm Example:

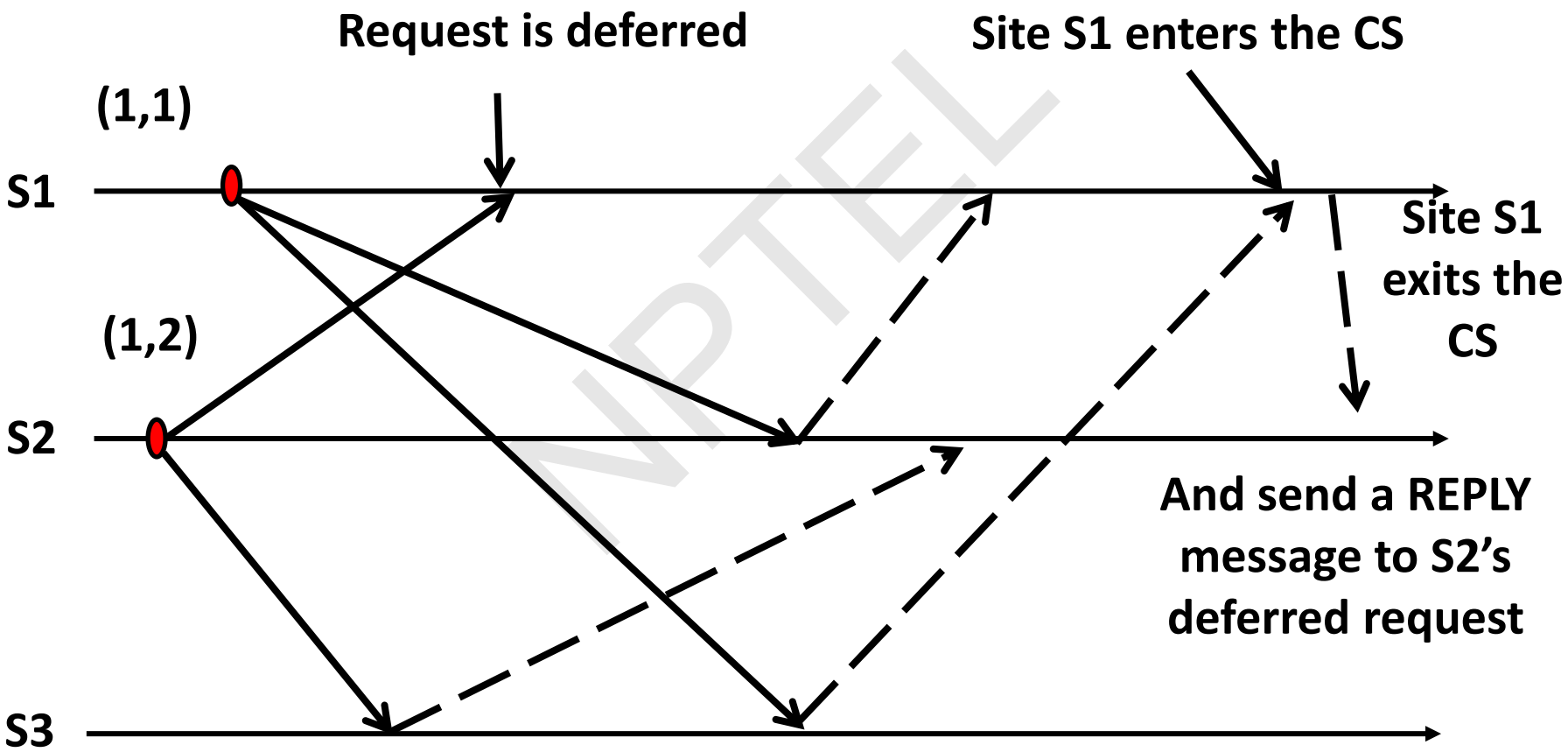
Sites S1 and S2 are Making Requests for the CS



Ricart–Agrawala algorithm Example:



Ricart–Agrawala algorithm Example:



Performance

- For each CS execution, Ricart-Agrawala algorithm requires **(*N* – 1) REQUEST** messages and **(*N* – 1) REPLY** messages.

- Thus, it requires **2(*N* – 1) messages** per CS execution.

- Synchronization delay in the algorithm is ***T*** .

Comparison

- Compared to Ring-Based approach, in Ricart-Agrawala approach
  - Client/synchronization delay has now gone down to O(1)
  - But bandwidth has gone up to O(*N*)
- Can we get *both* down?

Quorum-based approach

- In the '**quorum-based approach**', each site requests permission to execute the CS from a **subset of sites** (called a **quorum**).

- The **intersection property of quorums** make sure that only one request executes the CS at any time.

Quorum-Based Mutual Exclusion Algorithms

Quorum-based mutual exclusion algorithms are different in **two** ways:

1. A site does not request permission from **all other sites**, but only from a **subset of the sites**.

   *The **request set** of sites are chosen such that*

   $\forall i \ \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \Phi.$

   Consequently, every pair of sites has a site which mediates conflicts between that pair.

2. A site can send out only **one REPLY** message at any time.

   A site can send a **REPLY** message only after it has received a **RELEASE** message for the previous **REPLY** message.

Contd…

Notion of '**Coteries**' and '**Quorums**':

A **coterie *C*** is defined as a set of sets, where each set g ∈ C is called a **quorum.**

The following properties hold for quorums in a coterie:

- **Intersection property:** For every quorum g, h ∈ C, g ∩ h ≠ ∅.
  For example, sets {1,2,3}, {2,5,7} and {5,7,9} cannot be quorums in a coterie, because first and third sets **do not have a common element**.

- **Minimality property:** There should be no quorums g, h in **coterie C** such that **g ⊇ h i.e g is superset of h**.
  For example, sets {1,2,3} and {1,3} cannot be quorums in a coterie because the first set is a superset of the second.

Maekawa's Algorithm

Maekawa's algorithm was **first quorum-based mutual exclusion algorithm**.

- The **request sets for sites** (i.e., **quorums**) in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i \ \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$

M2: $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$

M3: $(\forall i : 1 \leq i \leq N :: |R_i| = K)$

M4: Any site S_j is contained in K number of R_is, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that $N = K(K-1) + 1$. This relation gives $|R_i| = \sqrt{N}$

Maekawa's Algorithm

- Conditions **M1** and **M2** are necessary for correctness; whereas conditions **M3** and **M4** provide other desirable features to the algorithm.
- Condition **M3** states that the size of the requests sets of all sites must be equal implying that all sites should have to do an equal amount of work to invoke mutual exclusion.
- Condition **M4** enforces that exactly the same number of sites should request permission from any site, which implies that all sites have **"equal responsibility"** in granting permission to other sites.

The Algorithm

A site S_i executes the following steps to execute the CS.

Requesting the critical section

(a) A site S_i requests access to the CS by sending **REQUEST(*i*)** messages to all sites in its request set R_i .

(b) When a site S_j receives the **REQUEST(*i*)** message, it sends a **REPLY(*j*)** message to S_i provided it hasn't sent a **REPLY** message to a site since its receipt of the last **RELEASE** message. Otherwise, it queues up the **REQUEST(*i*)** for later consideration.

Executing the critical section

(c) Site S_i executes the CS only after it has received a **REPLY** message from every site in R_i .

The Algorithm

Releasing the critical section

(d) After the execution of the CS is over, site S_i sends a **RELEASE(*i*)** message to every site in R_i .

(e) When a site S_j receives a **RELEASE(*i*)** message from site S_i, it sends a **REPLY** message to the next site waiting in the queue and deletes that entry from the queue.

If the queue is empty, then the site updates its state to reflect that it has not sent out any **REPLY** message since the receipt of the last **RELEASE** message.

Correctness

Theorem: ***Maekawa's algorithm achieves mutual exclusion.***

Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are concurrently executing the CS.
- This means site S_i received a **REPLY** message from all sites in R_i and concurrently site S_j was able to receive a **REPLY** message from all sites in R_j.
- If $R_i \cap R_j = \{S_k\}$, then site S_k must have sent **REPLY** messages to both S_i and S_j concurrently, which is a contradiction.

Performance

- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY, and $\sqrt{N}$ RELEASE messages, resulting in $3\sqrt{N}$ messages per CS execution.

- **Synchronization delay** in this algorithm is ***2T*** . This is because after a site S_j exits the CS, it first releases all the sites in R_j and then one of those sites sends a REPLY message to the next site that executes the CS.

Problem of Deadlocks

- **Maekawa's algorithm can deadlock** because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.

Assume three sites **S_i, S_j, and S_k simultaneously invoke mutual exclusion**.

- Suppose $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, and $R_k \cap R_i = \{S_{ki}\}$.

Consider the following scenario:

1. S_{ij} has been locked by S_i (forcing S_j to wait at S_{ij}).
2. S_{jk} has been locked by S_j (forcing S_k to wait at S_{jk}).
3. S_{ki} has been locked by S_k (forcing S_i to wait at S_{ki}).

- This state represents a deadlock involving sites S_i, S_j, and S_k.

Handling Deadlocks

- Maekawa's algorithm handles deadlocks by requiring a site to yield a lock if **the timestamp of its request is larger than the timestamp of some other request** waiting for the same lock.

- A site **suspects a deadlock** (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a **REPLY message to a lower priority request**.

Message types for Handling Deadlocks

Deadlock handling requires **three types of messages**:

FAILED: A FAILED message from site S_i to site S_j indicates that S_i can not grant S_j's request because it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from S_i to S_j indicates that S_i would like to find out from S_j if it has succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site S_i to S_j indicates that S_i is returning the permission to S_j (to yield to a higher priority request at S_j).

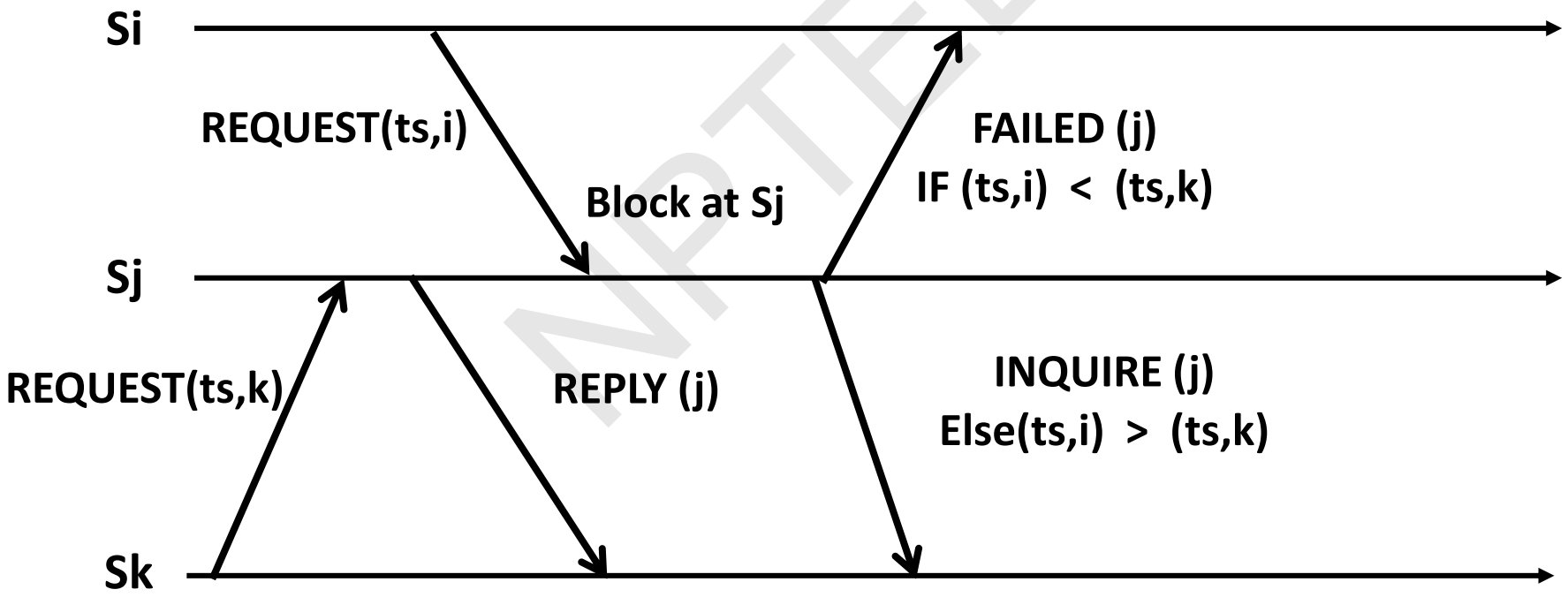
Handling Deadlocks

Maekawa's algorithm handles deadlocks as follows:

- When a **REQUEST(*ts* , *i*)** from site S_i blocks at site S_j because S_j has currently granted permission to site S_k, then S_j sends a **FAILED(*j*)** message to S_i if S_i's request has lower priority. Otherwise, S_j sends an **INQUIRE(*j*)** message to site S_k.
- In response to an **INQUIRE(*j*)** message from site S_j, site S_k sends a **YIELD(*k*)** message to S_j provided S_k has received a **FAILED** message from a site in its request set and if it sent a **YIELD** to any of these sites, but has not received a new **REPLY** from it.
- In response to a **YIELD(*k*)** message from site S_k, site S_j assumes as if it has been released by S_k, places the request of S_k at appropriate location in the request queue, and sends a **REPLY(*j*)** to the top request's site in the queue.
- **Maximum number of messages** required per CS execution in this case is 5 $\sqrt{N}$

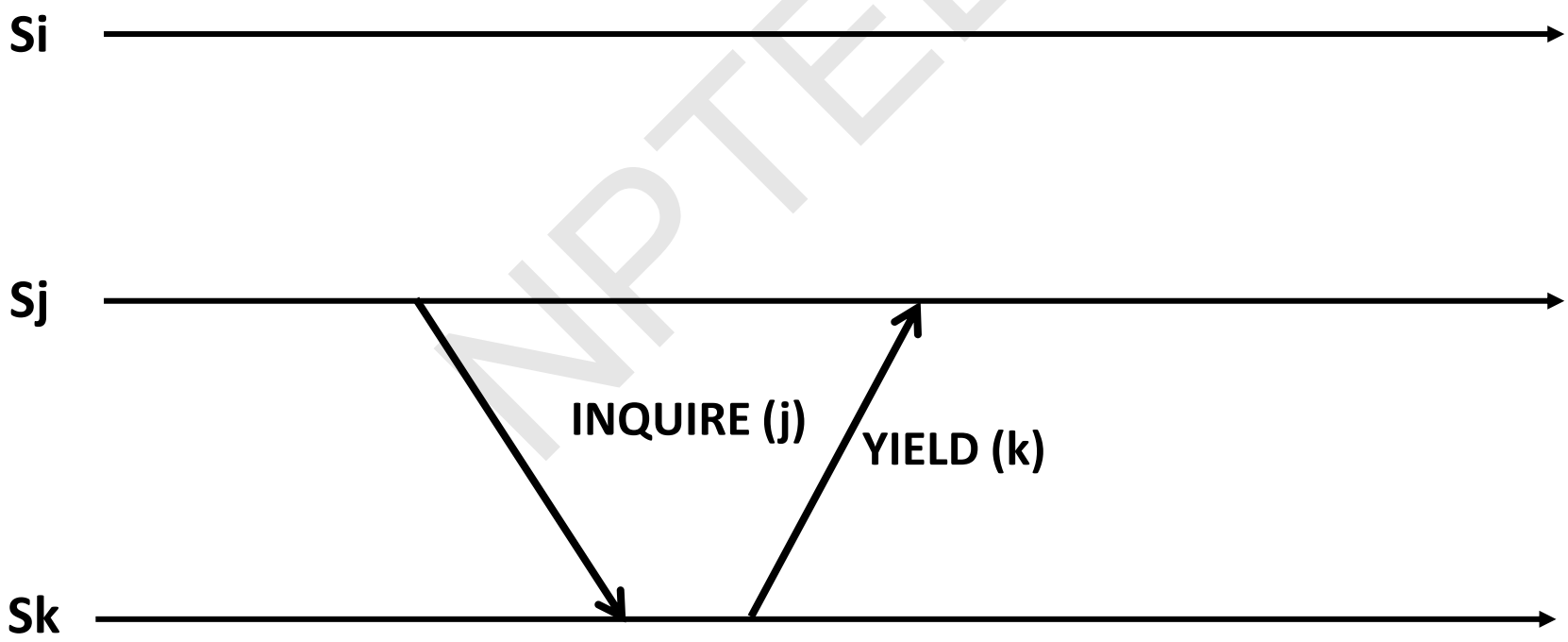
Handling Deadlocks: Case-I

When a **REQUEST(*ts* , *i*)** from site S_i blocks at site S_j because S_j has currently granted permission to site S_k , then S_j sends a **FAILED(*j*)** message to S_i if S_i's request has lower priority. Otherwise, S_j sends an **INQUIRE(*j*)** message to site S_k .



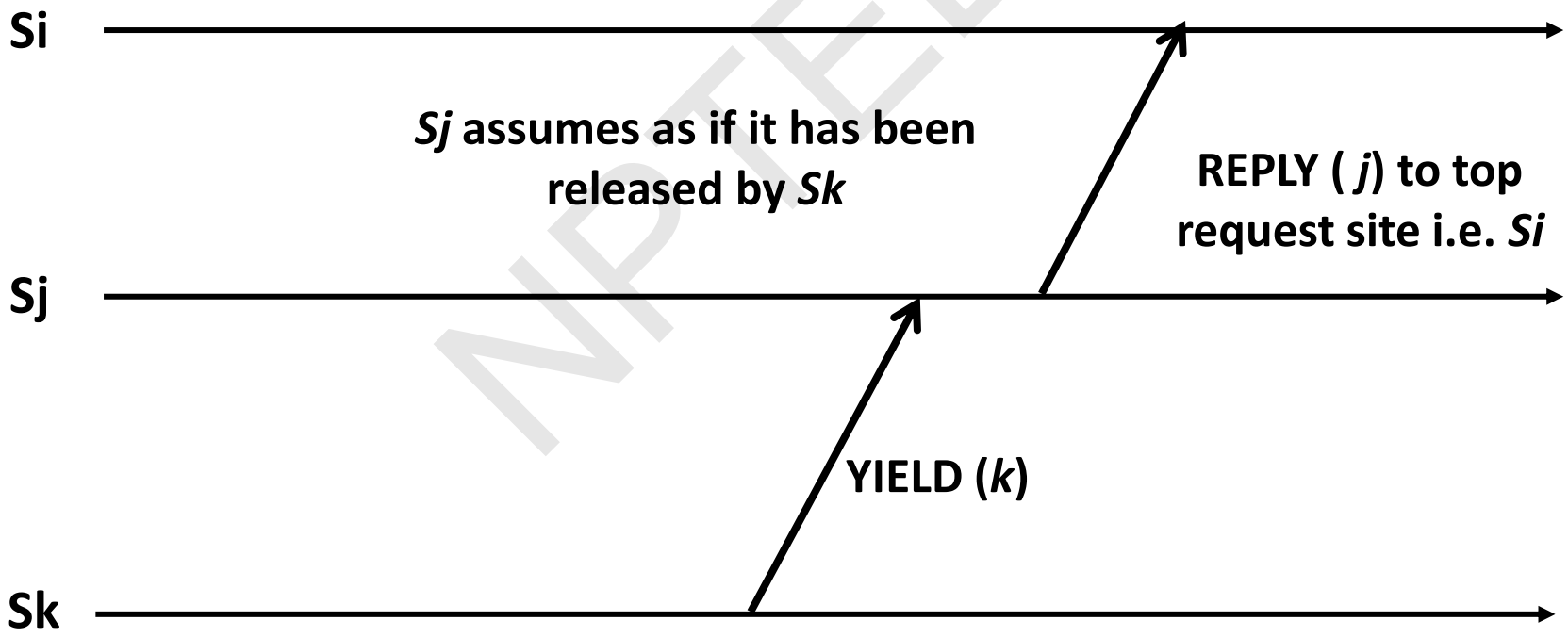
Handling Deadlocks: Case-II

In response to an **INQUIRE(*j*)** message from site S_j, site S_k sends a **YIELD(*k*)** message to S_j provided S_k has received a **FAILED** message from a site in its request set and if it sent a **YIELD** to any of these sites, but has not received a new **REPLY** from it.



Handling Deadlocks: Case-III

In response to a **YIELD(*k*)** message from site S_k , site S_j assumes as if it has been released by S_k , places the request of S_k at appropriate location in the request queue, and sends a **REPLY(*j*)** to the top request's site in the queue.



Failures?

- other ways to handle failures: Use Paxos like!

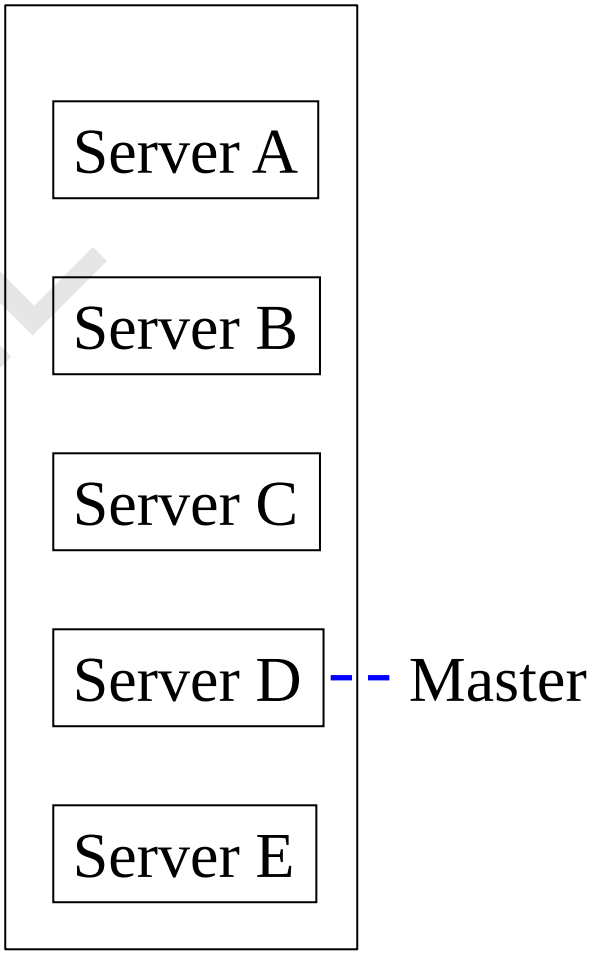
Industry Mutual Exclusion : Chubby

- Google's system for locking
- Used underneath Google's systems like BigTable, Megastore, etc.
- Chubby provides *Advisory* locks only
  - Doesn't guarantee mutual exclusion unless every client checks lock before accessing resource

Reference: *http://research.google.com/archive/chubby.html*

Chubby

- Can use not only for locking but also writing small configuration files
- Relies on Paxos-like (consensus) protocol
- Group of servers with one elected as Master
  - All servers replicate same information
- Clients send **read** requests to Master, which serves it **locally**
- Clients send **write** requests to Master, which sends it to all servers, gets **majority (quorum)** among servers, and then responds to client
- On master failure, run election protocol
- On replica failure, just replace it and have it catch up



Conclusion

- Mutual exclusion important problem in cloud computing systems
- Classical algorithms
  - Central
  - Ring-based
  - Lamport's Algorithm
  - Ricart-Agrawala
  - Maekawa
- Industry systems
  - Chubby: a coordination service
  - Similarly, Apache Zookeeper for coordination