

Distributed System Models

What we are trying to model?

- **Computation**: Processes, deterministic vs probabilistic executions
- **Interaction**: Processes communicate through **messages**, which result in:
 - Communication, i.e. information flow
 - Coordination, i.e. synchronization and ordering of activities
- **Failure**: Which kind of failures can occur? Processes? Communication channels?
- **Time**: Determining whether we can make any assumption on time bounds on communication and computation speeds.
- **Security**: Security is linked to failures; malicious behavior is a kind of failure. Modeling the possible behavior of attackers, providing a basis for the analysis of threats.
- **Execution**: How distributed algorithms are executed?

Computation

- **Process**: the unit of computation in a distributed system.
Sometimes we may call it node, host, etc.
- **Process set**: denoted by Π , it is composed by a collection of n uniquely identified processes, like p_1, p_2, \dots, p_n .
- Typical assumptions:
 - The set is static (n is well-defined);
 - Processes do know each other
 - All processes run a copy of the same algorithm; the sum of all these copies constitutes the distributed algorithm
- But in P2P systems:
 - Dynamic set
 - Too many, too dynamic to know them all
 - Multiple algorithms

Computation

- **Deterministic algorithms:** the local computation and the messages sent by a process is determined by the current state and the messages previously received.
- **Probabilistic algorithms:** processes may make use of *random* oracles to choose the local computation to be performed or the next message to be sent.

Interaction

- Processes communicate through **messages**
 - $send(m, p)$: sends a message m to p
 - $receive(m)$: receives a messages m (from its sender, whichever it is)
- In some cases, messages may be uniquely identified by
 - Sender of the message
 - A sequence number local to the sender
- **Communication channels**: used by processes to exchange messages
- General assumption: *every pair of processes is connected by a bi-directional channel*
 - Fully connected mesh
 - broadcast medium (like the Ethernet)
 - A ring
 - An “Internet” with routers

This can be normally obtained through routing; specialized algorithms may refine the network view to make use of the underlying topology.

Failures

In a distributed systems, both processes and communication channels may fail, i.e. depart from what is considered its correct or desirable behavior. Hadzilacos and Toueg [1994] provide a taxonomy.

Benign process failures

- **Fail-stop**: A process stops executing events, and other processes may detect this fact.
- **Crash**: A process stops executing events

Malicious process failures

- **Arbitrary failure**, or byzantine: any type of error may occur. This may be caused by:
 - A software bug
 - A malicious behavior inspired by an intelligent adversary

Failures

Comments on process failures

- A process that never fails is **correct**
- A process that eventually fails is **faulty**
- Several protocols are designed to work correctly if the number of failures f is bounded (for example, $f < n/3$).
- In some models, processes may perform a **recovery** action:
 - After some time, a process may resume functioning
 - It suffers *amnesia*: the local state maintained in volatile memory is lost
 - To limit the effects of amnesia, a log can be maintained
 - To avoid the problem completely, every read/write would have to pass through permanent memory; too expensive

Failures

Benign communication failures

How communication works:

- Process p performs `send` of a message m to process q
- Message m is inserted in a local outgoing buffer of p (**Send-omission**)
- Message m is transmitted from p to q (**Omission**)
- Message m is inserted in a local incoming buffer of q (**Receive-omission**)
- Process q performs `receive` of m

Malign communication failures

Messages created out of nothing, duplicated messages, etc.

These problems can easily be solved through encryption techniques.

Failures

Possible causes of message failures:

- Buffer overflow in the operating system
- Congestion, routing errors in routers
- **Partitioning:**
 - Processes are subdivided in disjoint sets called **partitions**
 - Communication inside a partition is possible
 - Communication between partitions is not possible

When a partition disappears, we say that partitions *merge*

Communication Channel Models

Fair-Loss Channels

- **FLC1** (*Fair Loss*): If a message m is sent infinitely often by a process p to a process q and neither p and q crash, then q will receive m infinitely often.
- **FLC2** (*Finite Duplication*): If a message m is sent a finite number of times by a process p to a process q , then m cannot be received by q an infinite number of times.
- **FLC3** (*No creation*): If a message m is delivered by some process p , then m was previously sent by some process q to p .

The idea: the channels cannot systematically drop a specific message. This is the minimum abstraction needed to create reliable channels.

Communication Channel Models

Perfect Channels

- **PC1** (*Reliable delivery*): If p sends a message to q , and neither of p and q crash, then q will eventually receive m .
- **PC2** (*No Duplication*): No message is delivered to a process more than once.
- **PC3** (*No Creation*): If a message m is delivered by some process p , then m was previously sent by some process q to p .

The idea: channels are reliable, messages are never lost. It can be implemented, but there is a price to be paid: asynchrony.

An Example Algorithm

upon event $\langle \textit{init} \rangle$ **do**

$\textit{sent} := \emptyset$

$\textit{delivered} := \emptyset$

$\textit{startTimer}(\textit{timeout})$

upon event $\langle \textit{timeout} \rangle$ **do**

forall $(m, q) \in \textit{sent}$ **do**

$\textit{fairLossSend}(m, q)$

$\textit{startTimer}(\textit{timeout})$

upon event $\langle \textit{perfectSend}(m, q) \rangle$ **do**

$\textit{fairLossSend}(m, q)$

$\textit{sent} := \textit{sent} \cup \{m, q\}$

upon event $\langle \textit{fairLossReceive}(m, q) \rangle$ **do**

if $(m \notin \textit{delivered})$ **then**

$\textit{delivered} := \textit{delivered} \cup \{m\}$

$\textit{perfectReceive}(m, q)$

Safety and liveness

- **Safety Property:** “Something bad will never happen”

In other words, a distributed program should never enter an unacceptable state.

- No message is delivered to a process more than once.

- **Liveness Property:** “Something good eventually does happen”

In other words, a distributed program eventually enters a desirable state.

- If p sends a message to q , and neither of p and q crash, then eventually q will receive m .

Time

- **Global clock:** For presentation simplicity, it may be convenient to assume the presence of a *global real-time clock*, outside the control of processes.
- This can be used to provide a global ordering of steps in a distributed systems
- **In reality:**
 - Each process is associated with a **local clock**
 - Local clocks may not report the perfect time
 - **Clock drift rate:** refers to the *relative* amount that a computer clock differs from a perfect reference clock.
- Synchronization is possible, but expensive:
 - Atomic clocks
 - GPS
 - * does not work into buildings
 - * cost not justified

Time

Time measures associated to communication:

- **Latency**: The delay between the start of message sending from one process and the beginning of its receipt by another. Possible causes:
 - the actual time for bit transmission (e.g., satellite link)
 - the delay for accessing the network, especially in case of congestion
 - the time taken by the operating system to service the message both at sender and receiver
- **Bandwidth**: Total amount of information that can be transmitted over a communication channel in a given time.
- **Jitter**: Variation in the time taken to deliver a series of messages. Mostly related with multimedia data.

Time

Distributed systems make difficult to reason about time, not only for lack of clock synchronization. It is also difficult to pose time bounds on events and communication.

We may think about several different models:

- **Asynchronous** distributed systems
 - No assumptions can be made.
 - Most of the problems cannot be solved
- **Synchronous** distributed systems
 - Precise assumptions are possible on computation, communication time and clocks.
 - Not really realistic / difficult to implement
- **Partially synchronous** distributed systems
 - Some assumptions can be made, others not, OR
 - Assumptions can be made statistically, OR
 - Assumptions hold for arbitrarily long periods of time

Time

Asynchronous Distributed System

- There are no bounds on the relative speed of process execution.
- There are no bounds on message transmission delays.
- There are no bounds on clock drift.
 - OR, since we cannot count on their precision at all, *there are no clocks*.

Time

Comments

- These are not assumptions! These are “lack of assumptions”!
- The worst possible model: services as simple as:
 - failure detection
 - time-based coordinationare not possible
- Advantages:
 - simple semantics
 - easier to port to more “powerful” models
 - More realistic: several sources of asynchrony are present in a large-scale network (like the Internet)

Time

Synchronous Distributed Systems

- *Synchronous computation:*

There is a known upper bound on the relative speed of process execution.

- *Synchronous communication:*

There is a known upper bound on message transmission delays.

- *Synchronous clocks:*

Processes are equipped with local clocks. There is a known upper bound on the drift rates of local clocks with respect to a global real-time clock.

Time

Comments

- The best possible model. Can be built, but not with standard hardware/software.
(e.g., synchronous Ethernet vs CSMA/CD Ethernet)
- Many interesting properties:
 - Timed failure detection (e.g., ping)
 - Coordination based on time (e.g., lease)
 - Worst-case performance analysis
 - Synchronized clocks

Time

Partial synchrony

For most systems we know of, it is relatively easy to define physical time bounds that are respected *most of the time*. There are however periods where the timing assumptions do not hold.

Delays on processes:

- Machines may run out of memory, slowing down processes
- A typical case of “*no bound on relative speeds of processes*”

Delays on messages:

- Network may congested, and messages may be dropped.
- Re-transmission protocols can ensure reliability, but at the price of asynchrony
Messages may be re-transmitted an arbitrary number of times.

In this sense, practical systems are *partially synchronous*

Time

How to express partial synchrony? A possibility is the following:

Timing assumptions only hold eventually (without stating exactly when)

Theoretically, it means:

- There is a time after which the system is synchronous forever
- The system is initially asynchronous and only after a long time becomes synchronous

How to read it:

- The system is not always synchronous
- There is no known bound to the period in which it is asynchronous
- We expect that there are periods during which the system is synchronous
- Some of these periods are long enough to terminate protocol execution

Execution

- **Distributed algorithm**: a collection of distributed automata, one per process;
- **Execution** of a distributed algorithm: the sequence of **events** executed by the processes
 - **Partial execution**: a finite sequence of events
 - **Infinite execution**: a infinite sequence of events
- Possible events:
 - $send(m, p)$: sends a message m to process p ;
 - $receive(m)$: receives a message m
 - *local events* that change the local state
- **Local state**:
 - at each point in time, each process is associated to a local state, containing all the data items accessible by that process
 - local state is completely private to the process

Local history

- The **local history** of a process p_i is a (possibly infinite) sequence of events
$$h_i = e_i^0 e_i^1 e_i^2 \dots e_i^{m_i}.$$
- Events are locally numbered using the *canonical enumeration*.
- The **partial history** up to event e_i^k is denoted h_i^k and is given by the prefix of the first k events of h_i .

Global History

- The **global history** of the computation is a set $H = h_1 \cup h_2 \cup \dots \cup h_n$, containing all its events (local history are seen as sets here, not sequences).

Note:

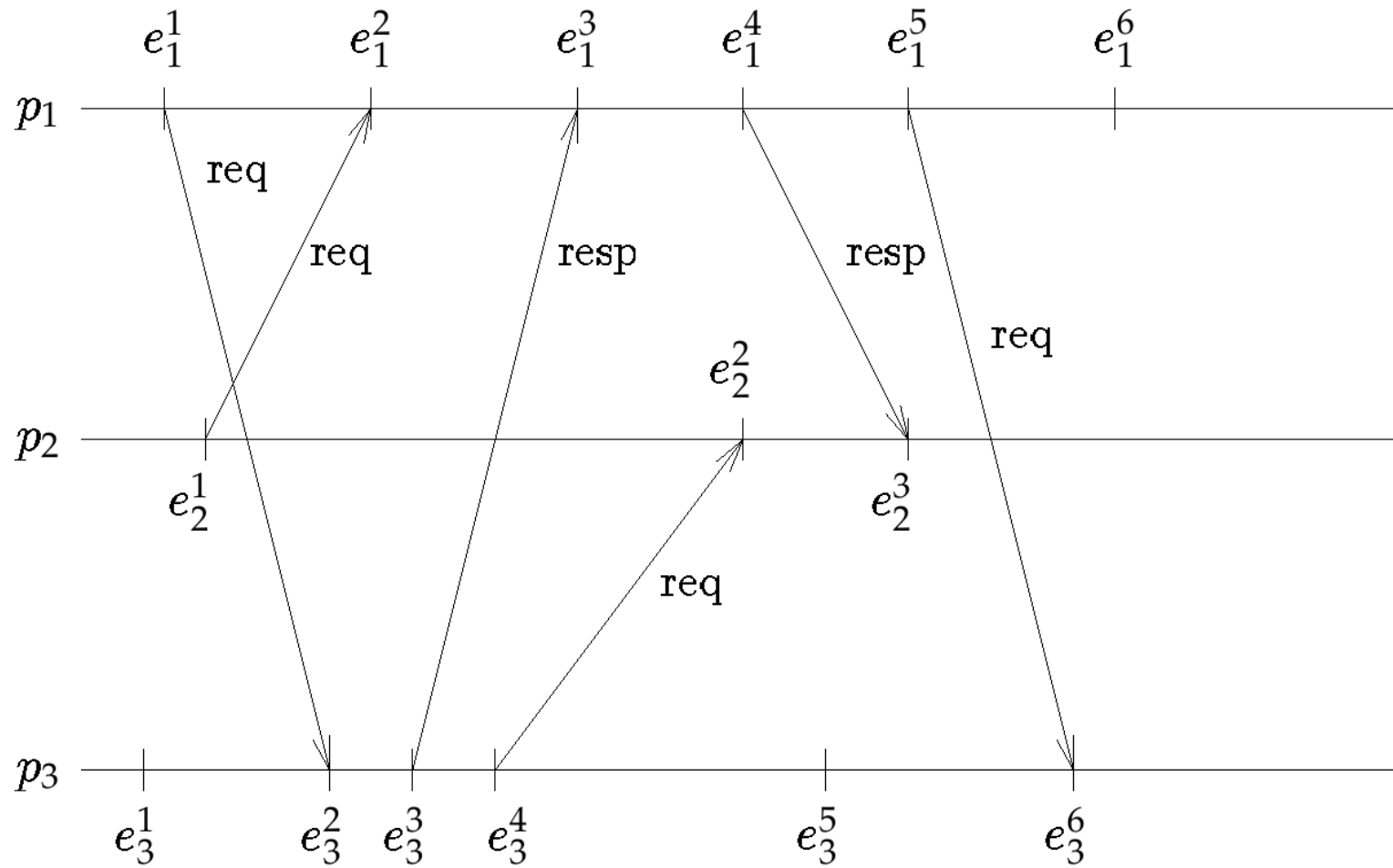
- Global history does not specify any relative timing between events.
- We need a notion of ordering between events, that could help us in deciding whether two events are actually concurrent, or one occurs before another.

Happen-Before

Definizione 1 (Happen-before). We say that an event e happens-before an event e' , and write $e \rightarrow e'$, if one of the following three cases is true:

1. $\exists p_i \in \Pi : e = e_i^r, \quad e' = e_i^s, \quad r < s$
(if e and e' are executed by the same process, e before e')
2. $e = \text{send}(m) \wedge e' = \text{receive}(m)$
(if e is the send event of a message m and e' is the corresponding receive event)
3. $\exists e'' : e \rightarrow e'' \rightarrow e'$
(in other words, \rightarrow is transitive)

Space-Time Diagram of a Distributed Computation



Happen-Before

Meaning of happen-before

If $e \rightarrow e'$, this means that we can find a series of events $e^1 e^2 e^3 \dots e^n$, where $e^1 = e$ and $e^n = e'$, such that for each pair of consecutive events e^i and e^{i+1} :

1. e^i and e^{i+1} are executed on the same process, in this order
2. $e^i = \text{send}(m)$ and $e^{i+1} = \text{receive}(m)$
 - *happen-before* captures the concept of **potential causal ordering**
 - *happen-before* captures a flow of data between two events.
 - Two events e, e' that are not related by the happened-before relation ($e \not\rightarrow e' \wedge e' \not\rightarrow e$) are **concurrent**, and we write $e \parallel e'$.

Homework

Prove or disprove that the \parallel relationship is transitive.

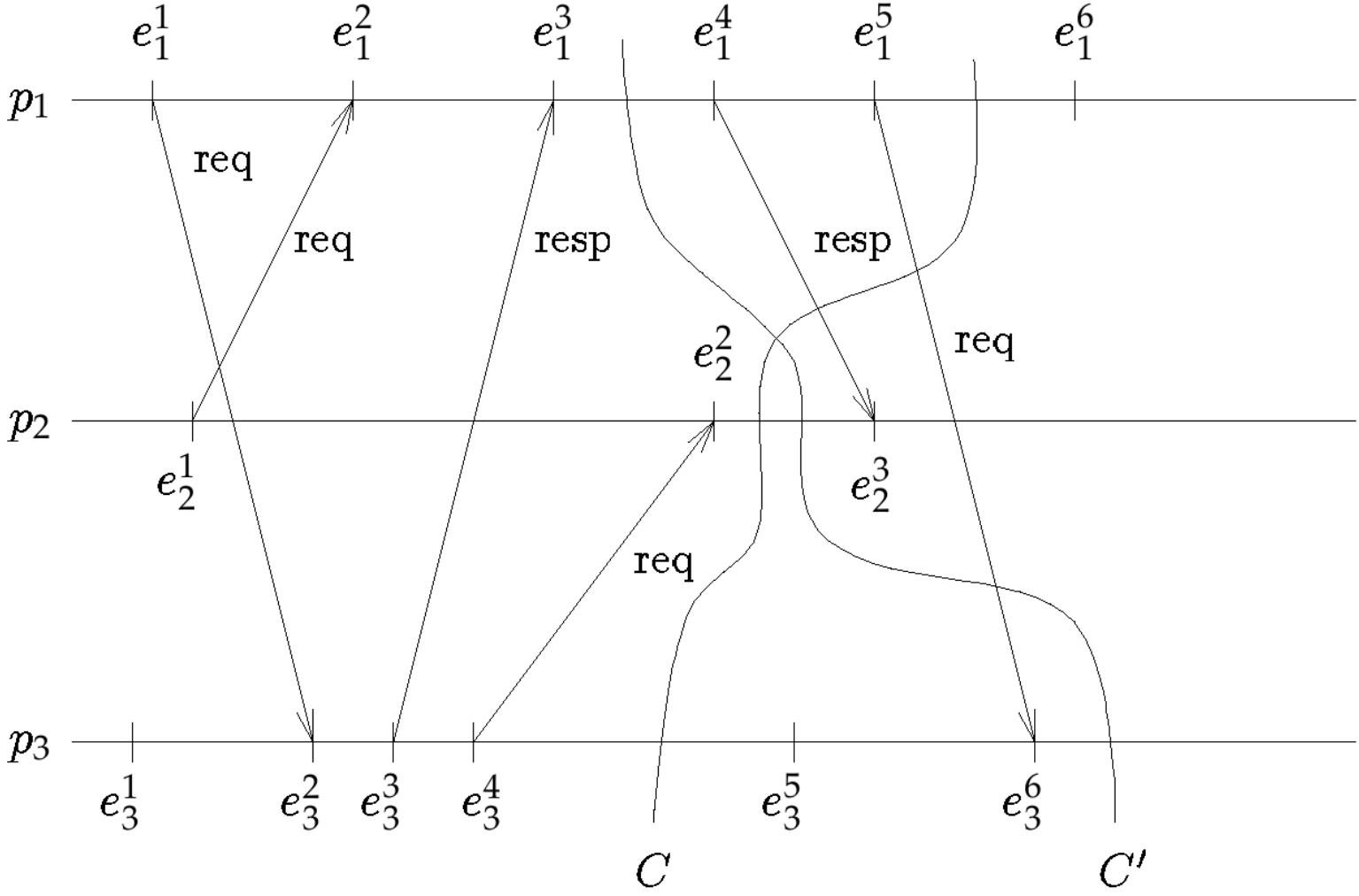
Global States

- The **local state** of a process p_i after the execution of e_i^k is denoted σ_i^k .
- σ_i^0 denotes the **initial state** of p_i
- A **global state** of a distributed computation is an n -tuple of local states $\Sigma = (\sigma_1, \dots, \sigma_n)$, one for each process.
- A **cut** of a distributed computation is the union of n partial histories, one for each process:

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_n^{c_n}$$

- A cut may be described by a tuple (c_1, c_2, \dots, c_n) , identifying the **frontier** of the cut, i.e. the set of last events, one per process.
- Each cut (c_1, \dots, c_n) has a corresponding global state $(\sigma_1^{c_1}, \sigma_2^{c_2}, \dots, \sigma_n^{c_n})$.

Cuts



Inconsistent cut

Consider cuts C' and C in the previous figure.

- Suppose they are generated by a “request for snapshot” message sent to all nodes and received at different times.
- A couple of questions:
 - Is it possible that cut C correspond to a “real” state in the execution of a distributed algorithm?
 - Is it possible that cut C' correspond to a “real” state in the execution of a distributed algorithm?

Consistent Cut

- **Consistent Cut**

A cut C is *consistent* if for all events e and e' ,

$$(e \in C) \wedge (e' \rightarrow e) \Rightarrow e' \in C$$

In other words,

- a consistent cut is left closed with respect to the happen-before relation.
- all the messages that have been received must have been sent before

- **Consistent Global State**

A global state is *consistent* if the corresponding cut is consistent.

Consistent Cut

- In the previous figures, C is consistent and C' is not.
- In the space-time diagram, a cut C is consistent if all the arrows start on the left of the cut and finish on the right of the cut.
- Consistent cuts represent the concept of scalar time in distributed computation: it is possible to distinguish between a “before” and an “after”.
- Predicates can be evaluated in consistent cuts, because they correspond to potential global states that could have taken place during an execution.

The concept of causality

- The concept of causality between events is fundamental to the design and analysis of distributed systems; e.g., it can be used to
 - maintain consistency in replicated databases
 - design correct deadlock detection algorithms
 - reconstruct a consistent state in checkpointing algorithms
- Potential causality is captured by the happen-before relationship
- How can we use the concept of causality to design distributed algorithms?

Causal order

- We want to deliver messages trying to avoid messages “out-of-order” with respect to the happen-before relation

FIFO (First-In, First-Out) Order

Definition Two messages sent by p_i to p_j must be delivered in the same order in which they were sent:

$$\forall m, m' : \text{send}_i(m) \rightarrow \text{send}_i(m') \Rightarrow \text{deliver}_j(m) \rightarrow \text{deliver}_j(m')$$

Causal Order

Definition Two messages sent by p_i and p_j to p_k must be delivered following the happen-before relation:

$$\forall m, m' : \text{send}_i(m) \rightarrow \text{send}_j(m') \Rightarrow \text{deliver}_k(m) \rightarrow \text{deliver}_k(m')$$

Uh? What is “deliver”?

How to implement causal order

- In the “day-to-day” life, potential causality is tracked using physical time
 - we use loosely synchronized watches;
 - Example: I have withdrawn money from an ATM in Trento at 13.00 on 17th May 2006, so I can prove that I’ve not withdrawn money on the same day at 13.20 in Paris.
- In distributed computing systems:
 - the rate of occurrence of events is several magnitudes higher
 - event execution time is several magnitudes smaller
- Consequently, if physical clocks are not precisely synchronized, the causality relation between events may not be accurately captured.

Logical clocks

Instead of using physical clocks, which are impossible to synchronize, we use logical clocks.

- Every process has a **logical clock** that is advanced using a set of rules
- Its value is not required to have any particular relationship to any physical clock.
- Every event is assigned a timestamp, taken from the logical clock
- The causality relation between events can be generally inferred from their timestamps

Logical clocks - Formal definition

Definition

A logical clock LC is a function that maps an event e from a distributed system execution to an element of a time domain T :

$$LC : H \rightarrow T$$

Clock Consistency (Clock Condition)

$$e \rightarrow e' \Rightarrow LC(e) < LC(e')$$

Strong Clock Consistency (Strong Clock Condition)

$$e \rightarrow e' \Leftrightarrow LC(e) < LC(e')$$

How to use logical clocks to guarantee causal order?

Delivery Rules

- Messages will be re-ordered in order to guarantee casual order.
- To be ordered, each message m carries a **timestamp** $TS(m)$ containing “ordering” information
- The act of providing the process with a message in the desired order is called **delivery**; the event $deliver(m)$ is thus distinct from $receive(m)$.
- The rule describing which messages can be delivered among those received is called **delivery rule**

Before implementing Causal Order..

- Let's practice with FIFO order

FIFO Order - Implementation

- Each process maintains a **local sequence number** incremented at each message sent
- The timestamp of a message corresponds to the local sequence number of the sender at the time of sending

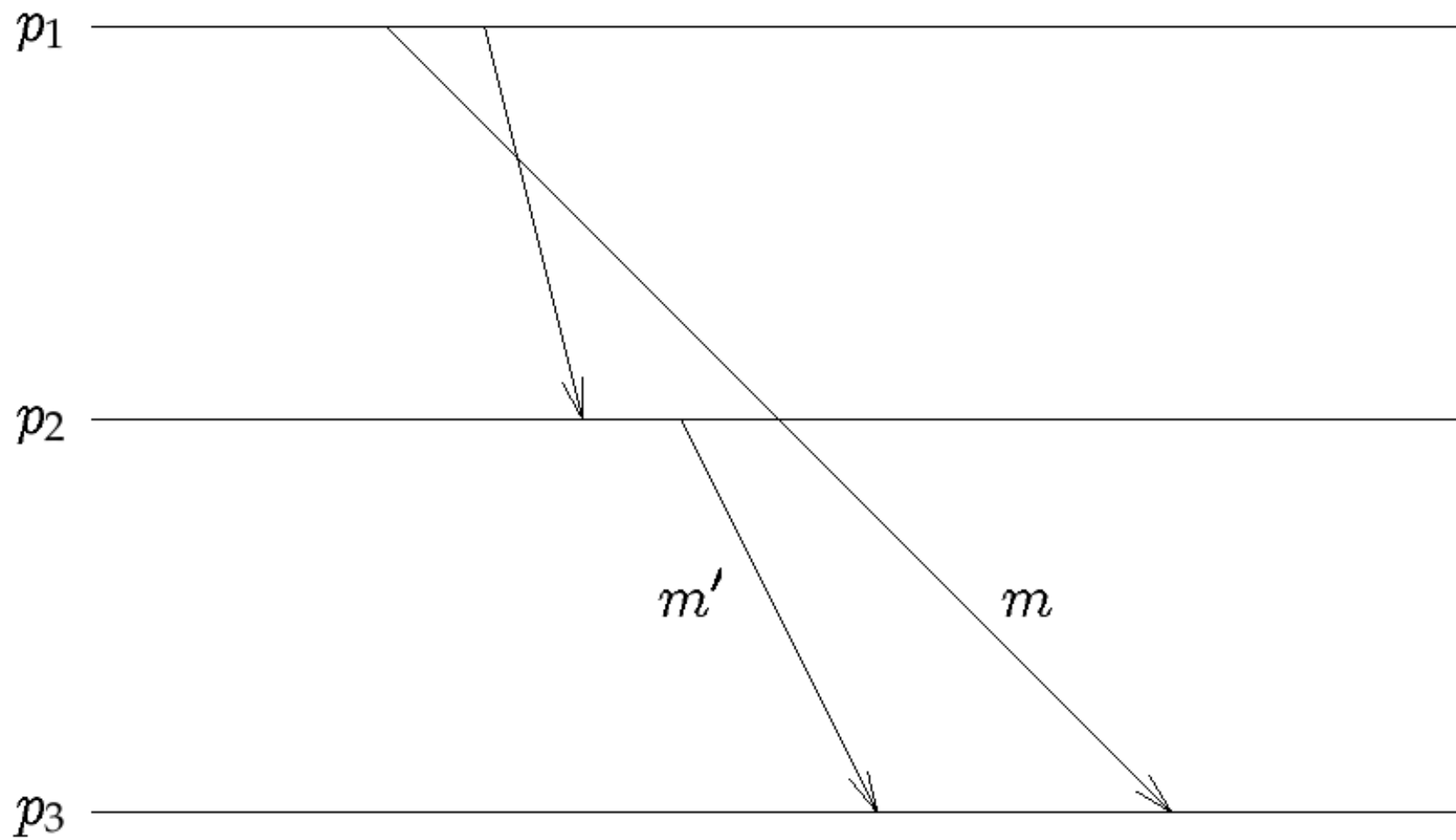
DR0 (FIFO Delivery Rule)

If the last message delivered by p_i from p_j has timestamp s , p_i may deliver “any” message m received from p_j with $TS(m) = s + 1$.

Question FIFO-Delivery among all channels...

Is it sufficient to obtain causal delivery?

Example



Scalar Logical Clocks

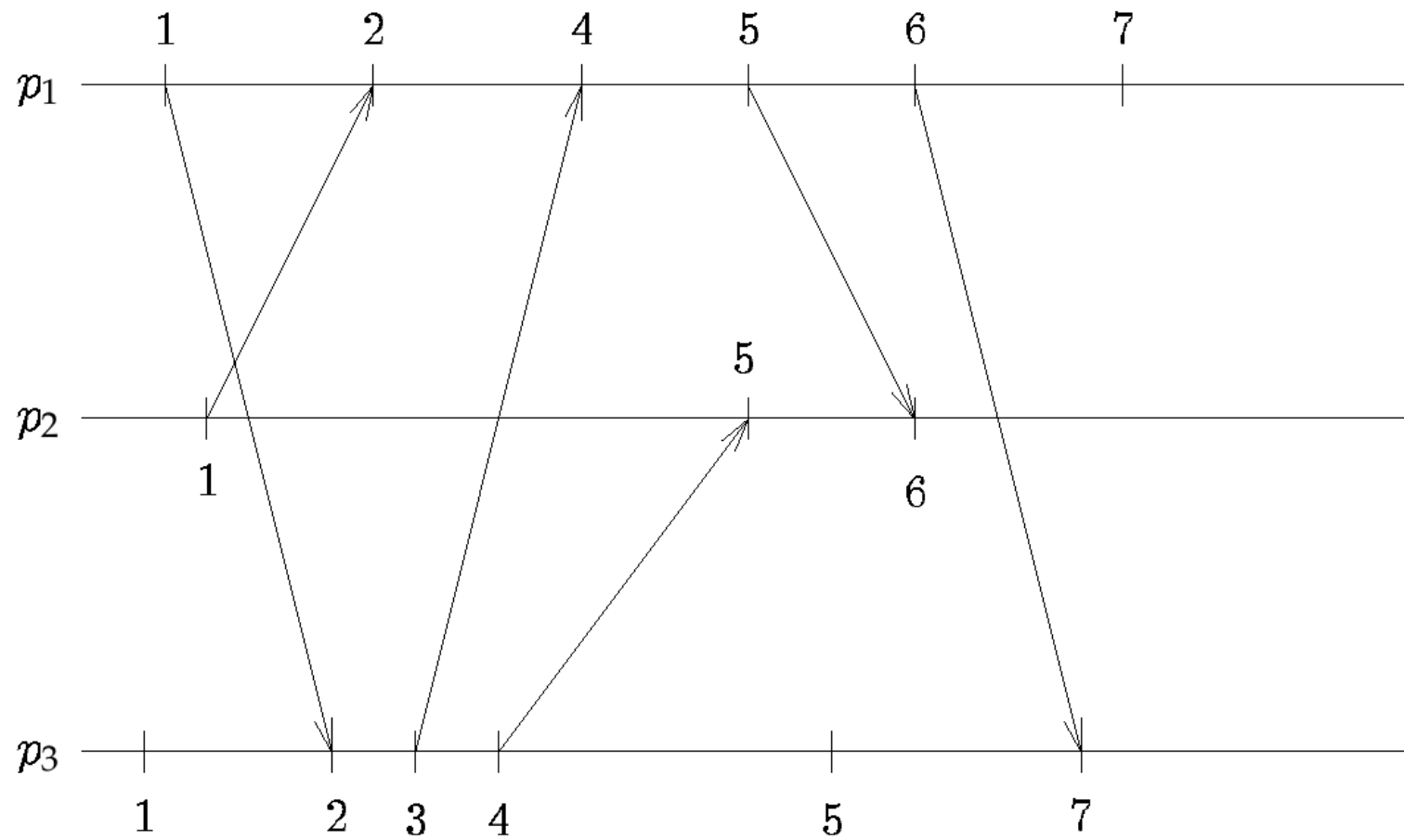
- A Lamport's **scalar** logical clock is a monotonically increasing software counter
- Each process p_i keeps its own logical clock LC_i
- The timestamp of an event e executed at process p_i is denoted $LC_i(e)$;
- Messages carry the timestamp of their *send* event.
- Logical clocks are initialized to 0.

How scalar logical clocks are updated?

Whenever an event e is executed by process p_i , its local logical clock is updated as follows:

$$LC_i = \begin{cases} LC_i + 1 & \text{If } e_i \text{ is an internal or } \textit{send} \text{ event} \\ \max\{LC_i, TS(m)\} + 1 & \text{If } e_i = \textit{receive}(m) \end{cases}$$

Scalar Logical Clocks



Properties

Clock Consistency

It is easy to see that clock consistency is satisfied. This immediately follows from the update rules of the clock.

Strong Clock Consistency

Strong clock consistency is not satisfied; i.e.,

$$LC(e) < LC(e') \not\Rightarrow e \rightarrow e'$$

Can you find an example on the space-time diagram?

Total ordering

It is also possible to provide a total order for events, by sorting based on the process identifier for events with the same timestamp.

Implementing a delivery rule

Good! Are we done? Is this sufficient to write a delivery rule?

Gap Detection for Logical Clock

We say that a logical clock LC satisfies gap detection if, given two events e and e' along with their clock values $LC(e) < LC(e')$, we are able determine whether some other event e'' exists such that $LC(e) < LC(e'') < LC(e')$.

Problem

Logical clocks do not guarantee *gap detection*. So, no message will ever be delivered for fear of receiving a later message with a smaller timestamp.

Causal delivery rule

Causal History

$$\theta(e) = \{e' \in H \mid e' \rightarrow e\} \cup \{e\}$$

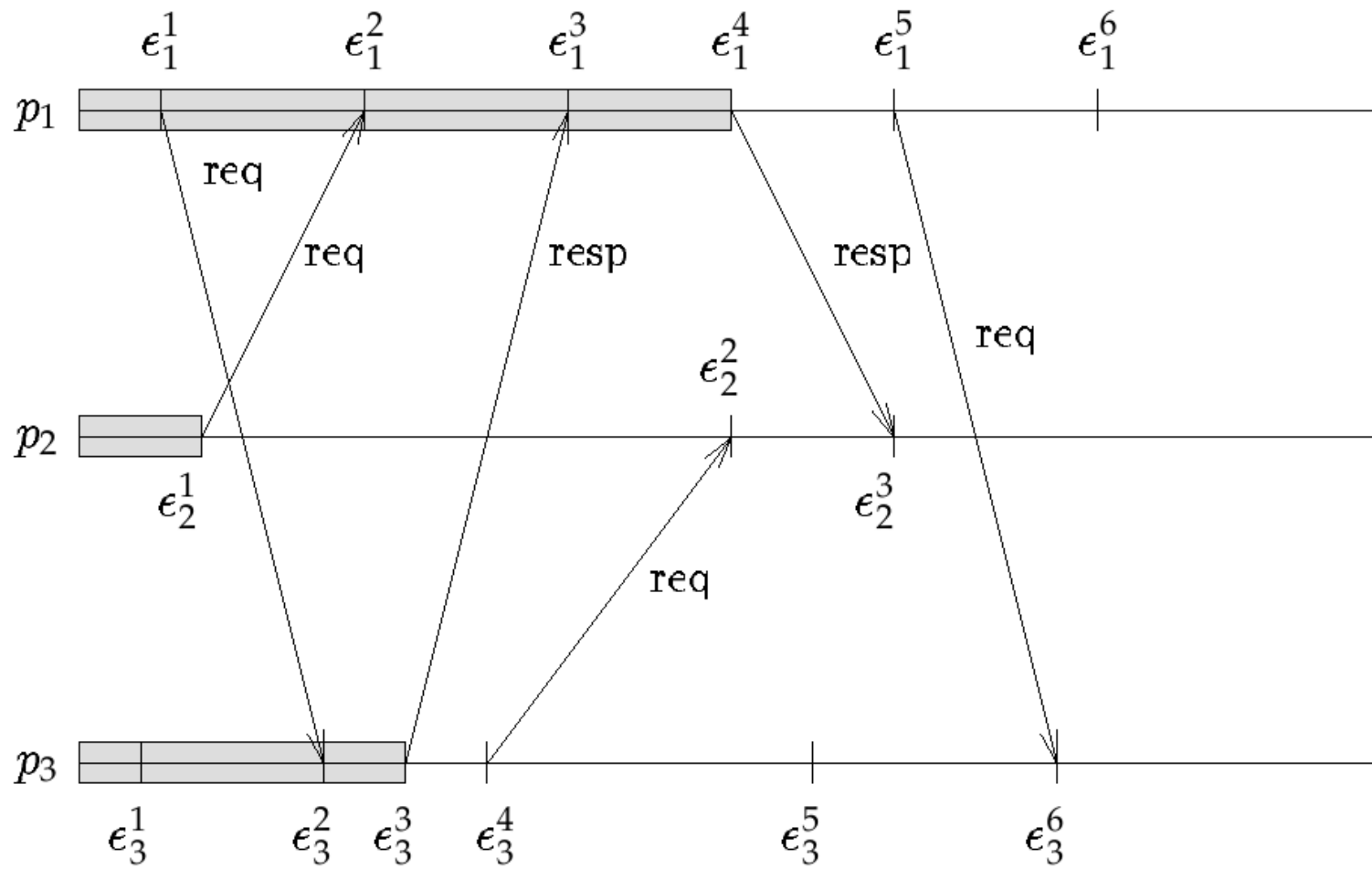
Strong Clock Consistency

$$\forall e \neq e' : e \rightarrow e' \Leftrightarrow e \in \theta(e') \Leftrightarrow \theta(e) \subset \theta(e')$$

Problem

- Causal histories tend to grow too much; they cannot be used as “timestamps” for messages.
- See for example $\theta(e_1^4)$ in the next slide.

Example



Vector Clocks

- $\theta_i(e) = \theta(e) \cap h_i = h_i^{c_i}$
- $\theta(e) = \theta_1(e) \cup \theta_2(e) \cup \dots \cup \theta_n(e) = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_n^{c_n}$
- In other words, $\theta(e)$ is a cut, which happens to be consistent.
- Cuts can be represented by their frontiers: $\theta(e) = (c_1, c_2, \dots, c_n)$

Definition The **vector clock** associated to event e is a n -dimensional vector $VC(e)$ such that

$$VC(e)[i] = c_i \quad \text{where } \theta_i(e) = h_i^{c_i}$$

Vector Clock: Implementation

- Each process maintain a local vector clock VC , initialized at all 0;
- When event e_i is executed, VC assumes the value of $VC(e_i)$;
- If $e_i = send(m)$, the timestamp of m is $TS(m) = VC(e_i)$;
- When an event e_i is executed by process p_i , the VC is updated as follows:
 - If e_i is an internal or *send* event:

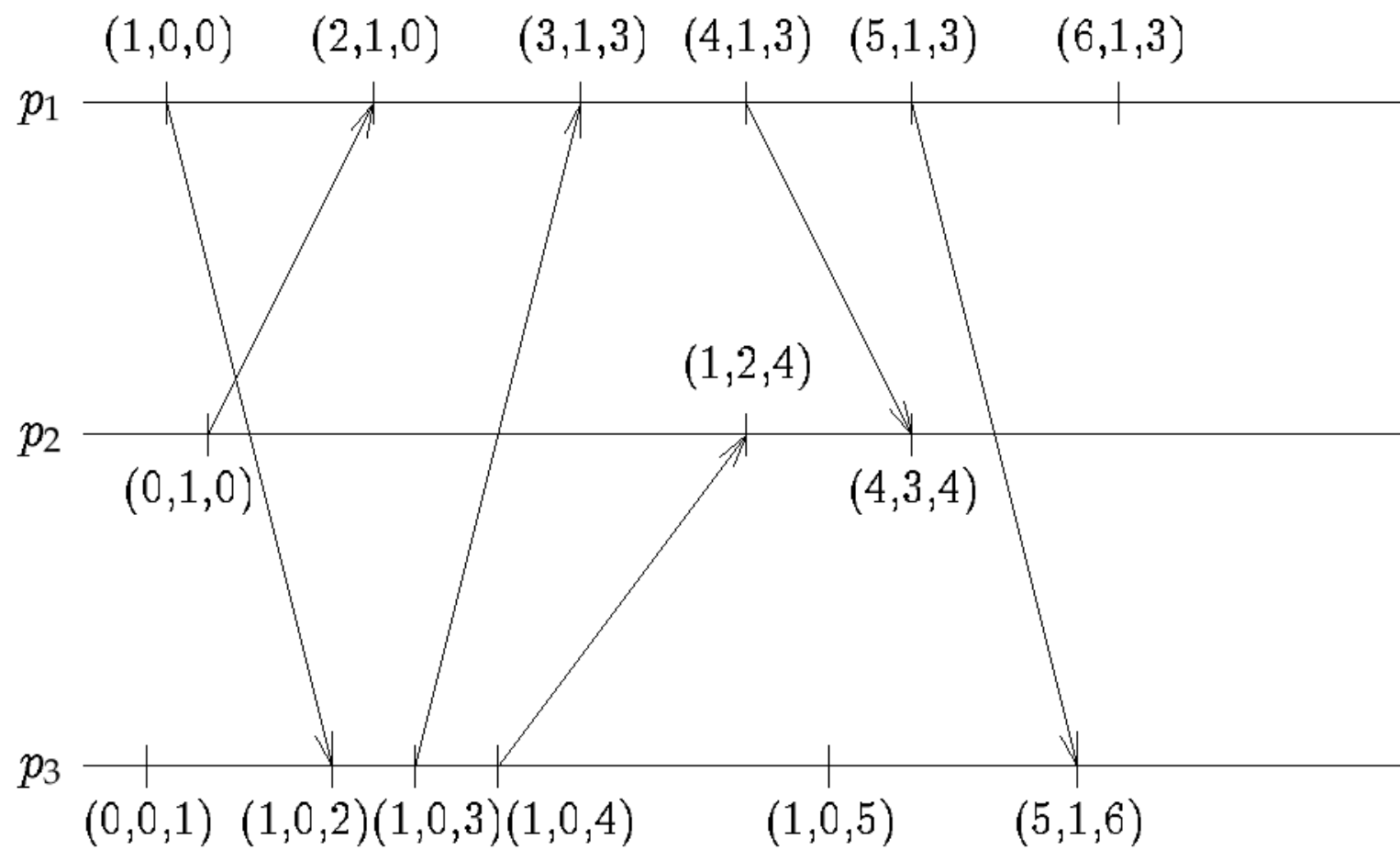
$$VC[i] = VC[i] + 1$$

- If $e_i = receive(m)$:

$$VC[j] = \max\{VC[j], TS(m)[j]\} \quad \forall j \neq i$$

$$VC[i] = VC[i] + 1$$

Example



Properties of Vector Clocks

“Less than” relation for Vector Clocks

$$V < V' \Leftrightarrow (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k])$$

Strong Clock Condition

$$e \rightarrow e' \Leftrightarrow VC(e) < VC(e') \Leftrightarrow \theta(e) \subset \theta(e')$$

Simple Strong Clock Condition

$$e_i \rightarrow e_j \Leftrightarrow VC(e_i)[i] \leq VC(e_j)[i]$$

Properties of Vector Clocks

Concurrent events Events e_i and e_j are *concurrent* (i.e. $e_i || e_j$) if and only if:

$$(VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$

Meaning: event e_i does not happen-before e_j , and e_j does not happen before e_i .

Example: ?

Pairwise Inconsistent Events e_i and e_j with $i \neq j$ are *pairwise inconsistent* if and only if

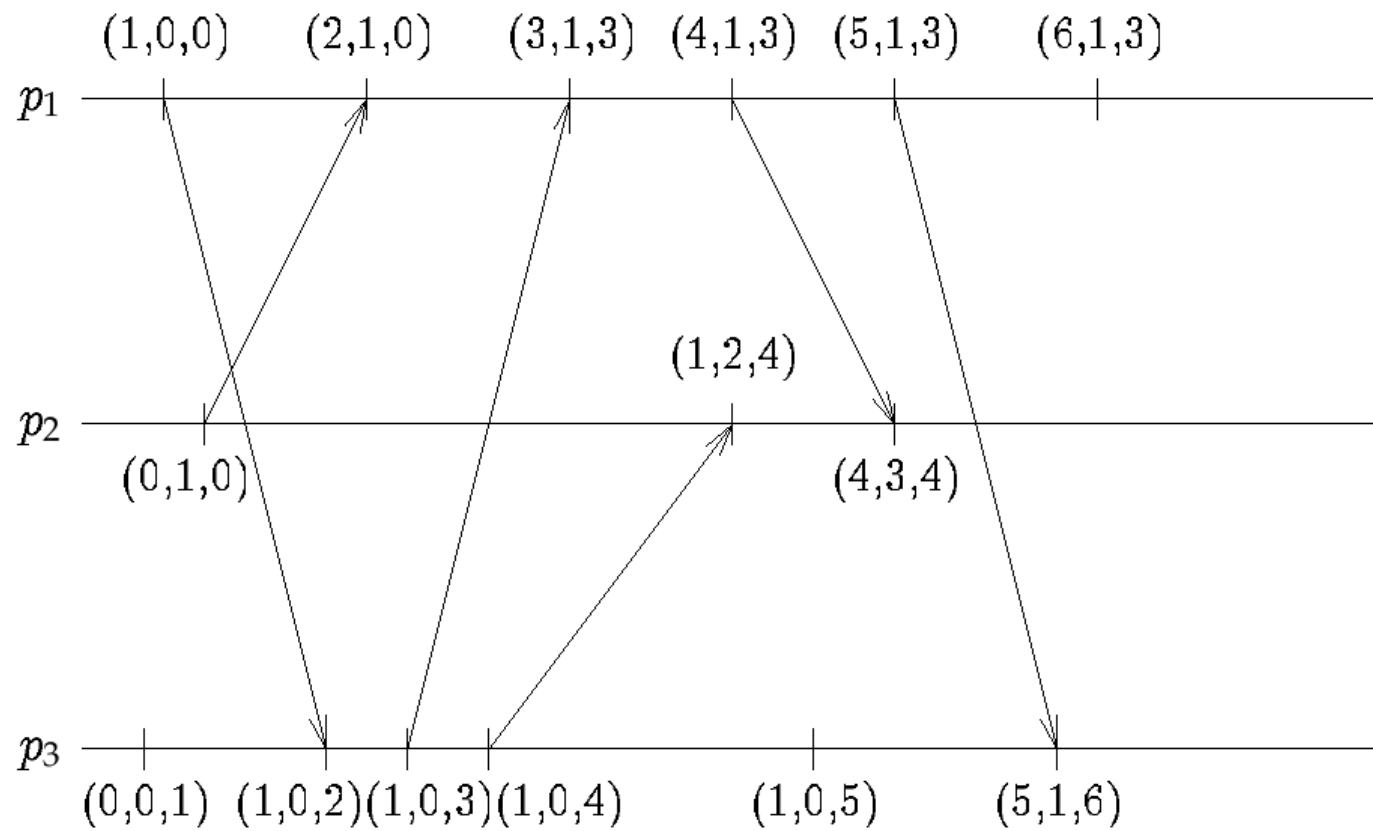
$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$

Meaning: two events are pairwise inconsistent if they cannot belong to the frontier of the same consistent cut. The formula characterize the fact that the cut include a *receive* event without including a *send* event.

Example: ?

Concurrent events

$$(VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$$



Pairwise inconsistent

$$(VC(e_i)[i] < VC(e_j)[i]) \vee (VC(e_j)[j] < VC(e_i)[j])$$

