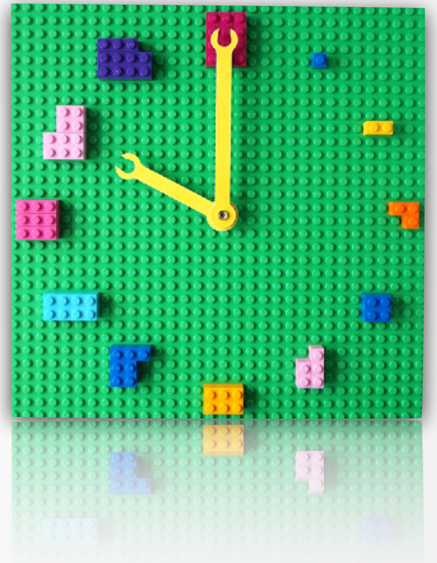
Advanced Course

Distributed Systems

Logical Clocks



Paris Carbone

COURSE TOPICS

- Intro to Distributed Systems
- Fundamental Abstractions and Failure Detectors
- Reliable and Causal Order Broadcast
- Distributed Shared Memory-CRDTs
- Consensus (Paxos)
- Replicated State Machines (OmniPaxos, Raft, Zab etc.)
- Time Abstractions and Interval Clocks (Spanner etc.)
- Consistent Snapshotting (Stream Data Management)
- Distributed ACID Transactions (Cloud DBs)

RECAP: CAUSAL ORDER

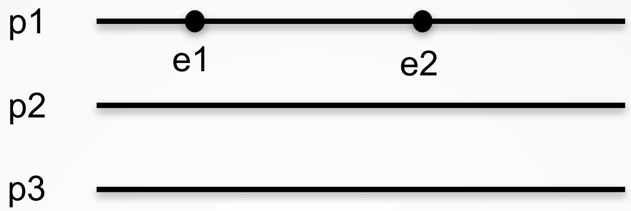
- Given an execution trace β,
  two events $a, b \in \beta$ are causally ordered ($a \rightarrow_{\beta} b$) iff either:
  - a occurs before b on the same process
  - a is a send(m) and b deliver(m) event
  - there exists a sequence of causally ordered events from a to b (transitive) e.g. If $a \rightarrow_{\beta} c$ and $c \rightarrow_{\beta} b$

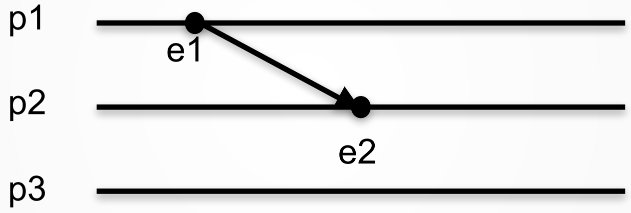
- Two events, a and b, are concurrent if **not** $a \rightarrow_{\beta} b$ and **not** $b \rightarrow_{\beta} a$
  - Concurrent events are denoted as such: $a || b$

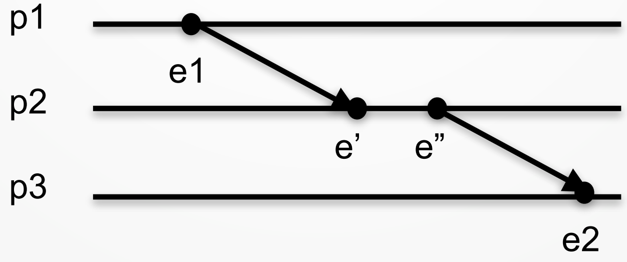
ID2203

KTH-2023

RECAP: CAUSAL ORDER



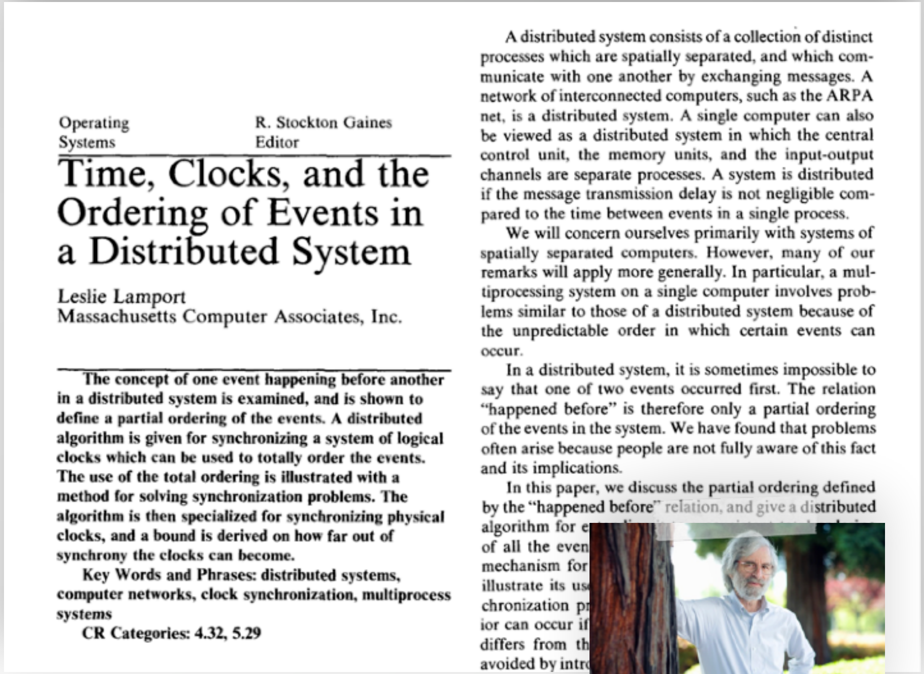




Key Observations

- We cannot order any two events in a D.S. using physical time.
  - Multiple Physical Clocks cannot keep precise physical time.

- We can order any two events in a D.S. using logical time, i.e., causal order.
  - Logical Clocks can capture causality.


Operating Systems — R. Stockton Gaines Editor

Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.

Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems
CR Categories: 4.32, 5.29

A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for e
of all the even
mechanism for
illustrate its us
chronization p
ior can occur if
differs from th
avoided by intro

Logical Clock Intuition

- A logical clock is :
  - an algorithm that assigns a timestamp to each event occurring in a distributed system. $t(a), t(b)$, etc.
- Timestamps can be used to derive a relation between events.

- We are interested in preserving the happen-before relation.

  If $a \rightarrow_\beta b$ then $t(a) < t(b)$

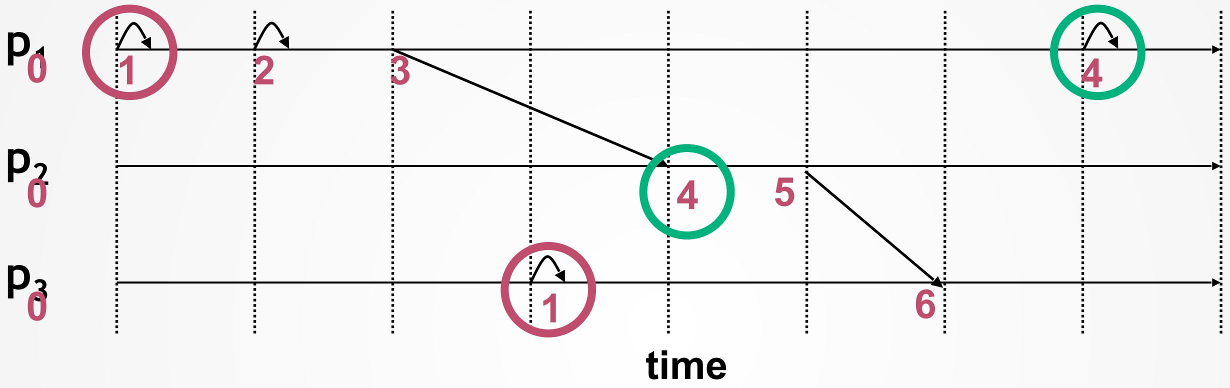
- **Two types of clocks: Lamport and Vector clocks**

Lamport Clocks

LAMPORT CLOCKS

- Each process has a local logical Lamport clock, kept in variable $\mathbf{t_p}$,
- initially $\mathbf{t_p} = 0$
    - A process p piggybacks $\mathbf{t_p}$ on every message sent
- On **internal event** *a*:
    - $\mathbf{t_p} := \mathbf{t_p} + 1$; perform internal event *a*
- **Sending event** of message m:
    - $\mathbf{t_p} := \mathbf{t_p} + 1$; send(m, $\mathbf{t_p}$))
- Receiving/Delivering **event** of message m with timestamp t_q from q:
    - $\mathbf{t_p} := \max(\mathbf{t_p}, \mathbf{t_q}) + 1$; perform delivery event *a*

LAMPORT LOGICAL CLOCKS



Lamport logical clocks guarantee that:

If $a \rightarrow_\beta b$, then $\mathbf{t}(a) < \mathbf{t}(b)$,

if $\mathbf{t}(a) \geq \mathbf{t}(b)$, then not $(a \rightarrow_\beta b)$

Can we do better?

Total Order with Lamport Clocks

- We can timestamp with process identifiers.
- The pair (t, p) is unique

Total Order Relation $(\prec)$: $(t_p, p) \prec (t_q, q)$ iff either

- $t_p < t_q$
- $t_p = t_q \wedge p < q$

i.e. break ties using process identifiers

e.g. $(5,p_5) < (7,p_2)$, $(4,p_2) < (4,p_3)$

LAMPORT CLOCKS

- Each process has a local logical lamport clock, kept in variable $\mathbf{t_p}$,
- initially $\mathbf{t_p} = 0$
  - A process p piggybacks ($\mathbf{t_p}$, p) on every message sent
- On **internal event** *a*:
  - $\mathbf{t_p} := \mathbf{t_p} + 1$; perform internal event *a*
- **Sending event** of message m:
  - $\mathbf{t_p} := \mathbf{t_p} + 1$; send(m, ($\mathbf{t_p}$, p))
- Receiving/Delivering **event** of message m with timestamp (t_q, q) from q:
  - $\mathbf{t_p} := \max(\mathbf{t_p}, \mathbf{t_q}) + 1$; perform delivery event *a*

REMARKS

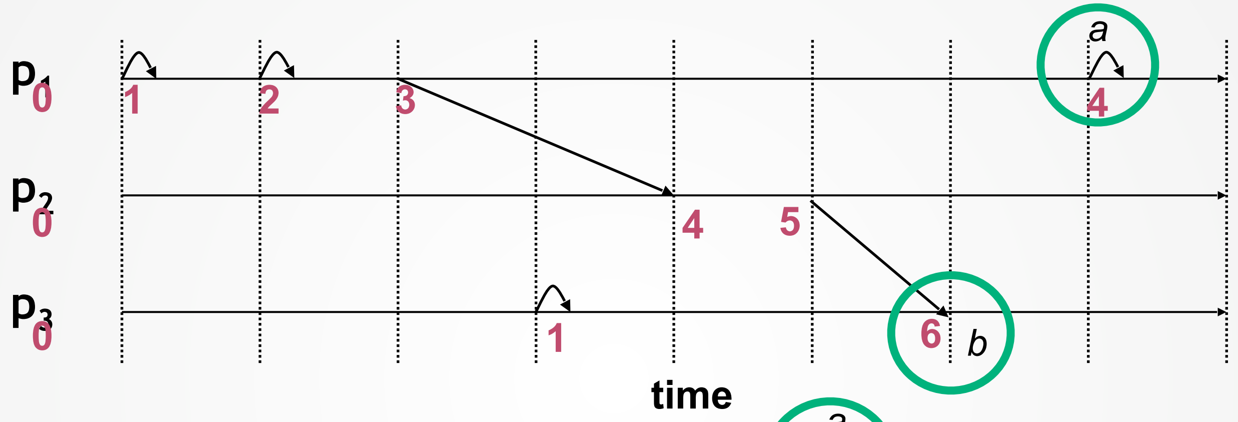
- The total order ($\prec$) defined on (t, p_i) pairs is:
  - A method to deterministically derive a total order of events using local and relative process information.
  - Always respecting causal order.
  - A convention rather than the actual order of events (whatever that means).

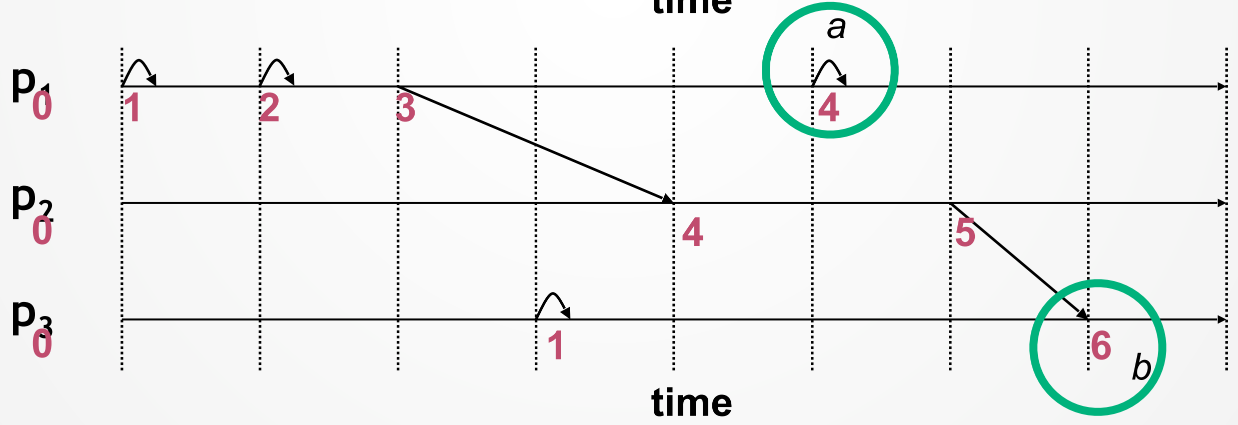
Vector Clocks

Non-causality and Concurrent events

- Two events a and b are concurrent ($a \,||_\beta\, b$) in an execution E (trace(E) = β) if
  - **not** $a \rightarrow_\beta b$ and **not** $b \rightarrow_\beta a$
- Computation theorem implies that if ($a \,||_\beta\, b$) in β then there are two executions (with traces β_1 and β_2) that are similar where a occurs before b in β_1, b occurs before a in β_2

Non-causality and Concurrent events



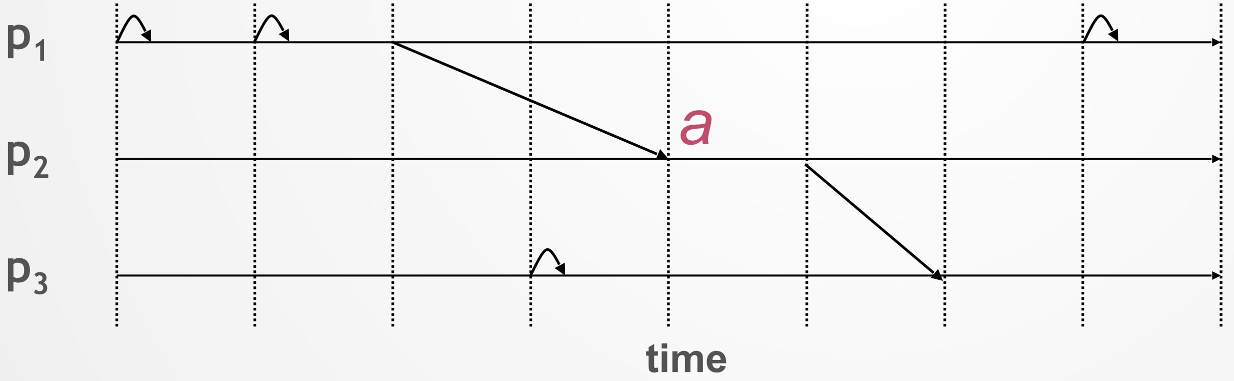


VECTOR CLOCKS

- **Lamport Clock Limitation:** We cannot tell by looking to the timestamps of event *a* and *b* whether there is a causal relation between the events, or they are concurrent.
- **Vector Clocks** guarantee that:
  - 1. if $\mathbf{v}(a) < \mathbf{v}(b)$ then $a \rightarrow_{\beta} b$
  - 2. if $a \rightarrow_{\beta} b$ then $\mathbf{v}(a) < \mathbf{v}(b)$ (same as Lamport Clocks)
    - where $\mathbf{v}(a)$ is a vector clock of event a

VECTOR CLOCK DEFINITION

- Vector clock for an event ***a***
  - $\mathbf{v}(a) = [x_1, \ldots, x_n]$
  - x_i is the number of events at p_i that happened-before ***a***



3 events at p1
1 event at p2
0 events at p3

[3, 1, 0]

Vector Time Implementation

- Processes $p_1, \ldots, p_n \in P$
- Each process p_i has local vector **v** of size **n** (number of processes)
  - **v**[i] = 0 for all i in 1…n
  - Piggyback **v** on every sent message
- For each transition (on each event) update local **v** at p_i:
  - **v**[i] := **v**[i] + 1   (internal, send or deliver)
  - **v**[j] := max(**v**[j], $\mathbf{v}_{\mathbf{q}}$[j]), for all j ≠ i $\in |P|$ (deliver)
    - where $\mathbf{v}_{\mathbf{q}}$ is clock in message received from process q

Comparing Vector Clocks

- $v_p \leq v_q$ iff
  - $v_p[i] \leq v_q[i]$ for $\forall i \in |P|$
- $v_p < v_q$ iff
  - $v_p \leq v_q$ and for some i, $v_p[i] < v_q[i]$
- v_p and v_q are concurrent ($v_p \,||\, v_q$) iff
  - not $v_p < v_q$, and not $v_q < v_p$

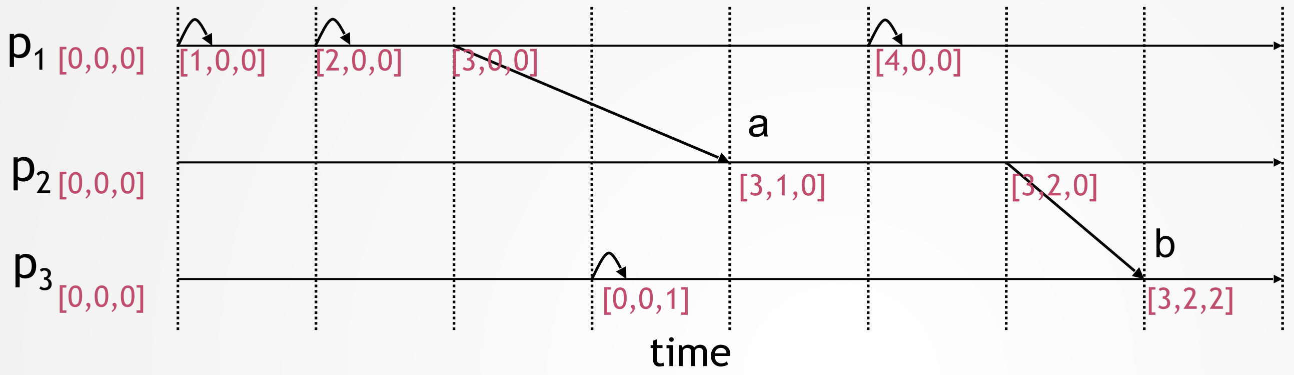
[3,0,0] ≤ [3,1,0]

[3,0,0] < [3,1,0]

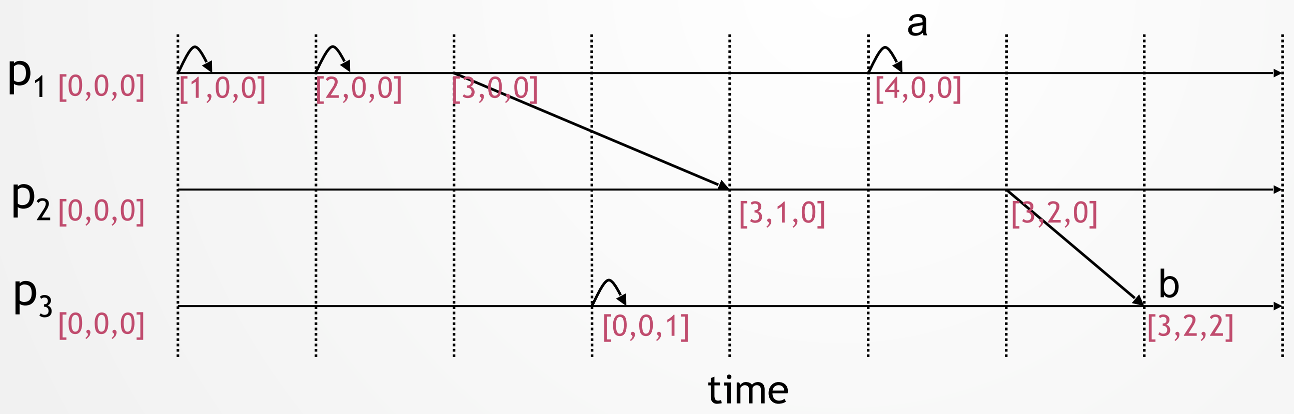
[3,1,0] <> [4,0,0]

- Vector clocks guarantee
  - If v(a) < v(b) then a $\rightarrow$ b, and
  - If a $\rightarrow$ b, then v(a) < v(b)
    - where v(a) is the vector clock of event a

Example of Vector Clocks



v(a) < v(b) implies a → b

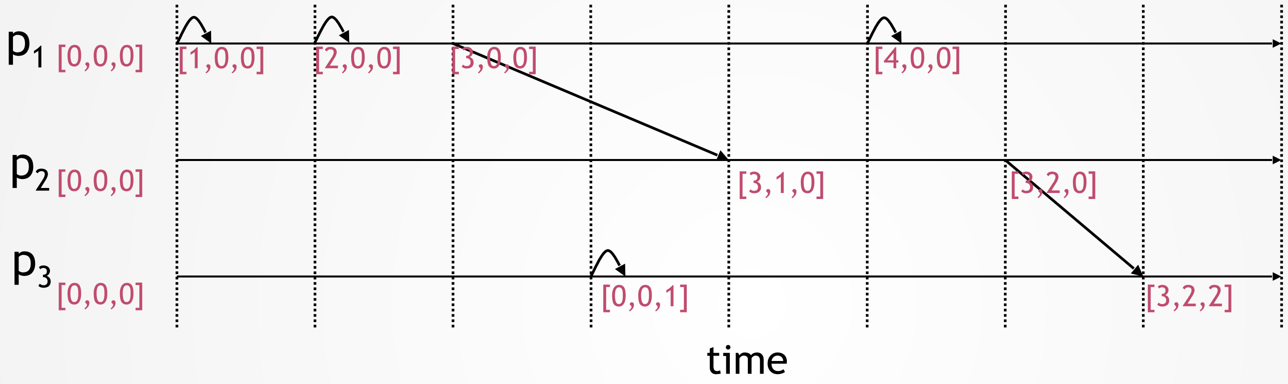


v(a) <> v(b) implies a || b

ID2203

KTH-2023

LIMITATIONS OF VECTOR CLOCKS?



- Vectors need to be statically defined of size n
- Insufficient for problems that require total event ordering

ORDERING - SUMMARY

- the relation $\rightarrow_\beta$ on events in executions
  - Partial: $\rightarrow_\beta$ doesn't order concurrent events
- the relation $\prec$ on Lamport logical clocks
  - Total: any two distinct clock values are ordered (adding pid)
- the relation $\prec$ on vector timestamps
  - Partial: timestamp of concurrent events not ordered

LOGICAL CLOCK INVARIANTS

Lamport clock

If a $\rightarrow_\beta$ b then t(a) < t(b) (1)

If t(a) < t(b) then **not b** $\rightarrow_\beta$ **a** (2)

Vector clock

If a $\rightarrow_\beta$ b then v(a) < v(b) (1)

If v(a) < v(b) then a $\rightarrow_\beta$ b (2)