Time, Logical Time and Causality

Carlos Baquero
Distributed Systems Group
Universidade do Minho

MAPI 2007

Plan

We will try to cover a few of the many aspects of time and logical sequences of events in distributed systems:

- Time Synchronization
- Order Relations
- Logical Time and Causality
- Process Causality vs Data Causality

Plan

We will try to cover a few of the many aspects of time and logical sequences of events in distributed systems:

- Time Synchronization
- Order Relations
- Logical Time and Causality
- Process Causality vs Data Causality

Global Snapshots and Termination will only be covered in the next talk, so we will carefully avoid them.

Time Synchronization

Time, which time?

Non relativistic real time can be tracked by clocks. But clocks have drift. Where drift is the variation between a clock's time and a reference clock.

Time Synchronization

Time, which time?

Non relativistic real time can be tracked by clocks. But clocks have drift. Where drift is the variation between a clock's time and a reference clock.

- Quartz clocks drift at about 10^{-6} to 10^{-8} seconds per second.
- 10^{-6} amounts to about 1 second each 12 days. Not very good.

Time Synchronization

Time, which time?

Non relativistic real time can be tracked by clocks. But clocks have drift. Where drift is the variation between a clock's time and a reference clock.

- Quartz clocks drift at about 10^{-6} to 10^{-8} seconds per second.
- 10^{-6} amounts to about 1 second each 12 days. Not very good.
- Atomic clocks drift at about 10^{13} seconds per second.

Time Synchronization

Time, which time?

Non relativistic real time can be tracked by clocks. But clocks have drift. Where drift is the variation between a clock's time and a reference clock.

- Quartz clocks drift at about 10^{-6} to 10^{-8} seconds per second.
- 10^{-6} amounts to about 1 second each 12 days. Not very good.
- Atomic clocks drift at about 10^{13} seconds per second.
- Coordinated Universal Time (UTC) is a high-precision atomic time standard. It closely tracks Universal Time (UT), that maps earth rotation, by adding leap seconds when needed.

Time Synchronization

Synchronization

External Synchronization

Measures expected precision with reference to an authoritative time source.

For an envelope $D > 0$, a UTC source S and at any given instant t we need to have $|S(t) - C_i(t)| \leq D$.

Time Synchronization

Synchronization

External Synchronization

Measures expected precision with reference to an authoritative time source.
For an envelope $D > 0$, a UTC source S and at any given instant t we need to have $|S(t) - C_i(t)| \leq D$.

Internal Synchronization

Measures synchronization between two machines.
For an envelope $D > 0$, at any given instant t we need to have $|C_j(t) - C_i(t)| \leq D$.

Time Synchronization

Synchronization

External Synchronization

Measures expected precision with reference to an authoritative time source.
For an envelope $D > 0$, a UTC source S and at any given instant t we need to have $|S(t) - C_i(t)| \leq D$.

Internal Synchronization

Measures synchronization between two machines.
For an envelope $D > 0$, at any given instant t we need to have $|C_j(t) - C_i(t)| \leq D$.

A system with D external synchronization also depicts $2D$ internal synchronization.

Time Synchronization

Synchronous System

Consider the simple case of two node synchronization in a synchronous setting.

Time Synchronization

Synchronous System

Consider the simple case of two node synchronization in a synchronous setting.

- Node C asks node S the time. S replies with time t and node C knows the transit time t_d. C can set its time to $t + t_d$.

Time Synchronization

Synchronous System

Consider the simple case of two node synchronization in a synchronous setting.

- Node C asks node S the time. S replies with time t and node C knows the transit time t_d. C can set its time to $t + t_d$.
- Tipically t_d varies in a range, $t_m \leq t_d \leq t_M$. Leading to a variation range of $t_M - t_m$.

Time Synchronization
Synchronous System

Consider the simple case of two node synchronization in a synchronous setting.

- Node C asks node S the time. S replies with time t and node C knows the transit time t_d. C can set its time to $t + t_d$.
- Tipically t_d varies in a range, $t_m \leq t_d \leq t_M$. Leading to a variation range of $t_M - t_m$.
- If we set in C time to $t + \frac{T_M - t_m}{2}$ one can achieve synchronization within an envelope D of $\frac{T_M - t_m}{2}$.

Time Synchronization

Synchronous System

Consider the simple case of two node synchronization in a synchronous setting.

- Node C asks node S the time. S replies with time t and node C knows the transit time t_d. C can set its time to $t + t_d$.
- Tipically t_d varies in a range, $t_m \leq t_d \leq t_M$. Leading to a variation range of $t_M - t_m$.
- If we set in C time to $t + \frac{T_M - t_m}{2}$ one can achieve synchronization within an envelope D of $\frac{T_M - t_m}{2}$.

Asynchronous

In an asynchronous system t_d now varies in range $t_m \leq t_d \leq \infty$. Apparently, the envelope is now $D = \frac{\infty - t_m}{2} = \infty$. Not a very usefull bound, but its easy to do better.

Time Synchronization

Asynchronous System: Cristian's algorithm

Two node synchronization in an asynchronous setting.

- Node C memorizes time $t_i = t$ asks node S the time. S replies with time t_s and node C memorizes the reception time $t_f = t$.

Time Synchronization

Asynchronous System: Cristian's algorithm

Two node synchronization in an asynchronous setting.

- Node C memorizes time $t_i = t$ asks node S the time. S replies with time t_s and node C memorizes the reception time $t_f = t$.
- Node C calculates the roundtrip time as $t_r = t_f - t_i$.

Time Synchronization

Asynchronous System: Cristian's algorithm

Two node synchronization in an asynchronous setting.

- Node C memorizes time $t_i = t$ asks node S the time. S replies with time t_s and node C memorizes the reception time $t_f = t$.
- Node C calculates the roundtrip time as $t_r = t_f - t_i$.
- C can set the time to $t_s + \frac{t_r}{2}$ and expect to have a synchronization of $D = \frac{t_r}{2}$.

Time Synchronization

Asynchronous System: Cristian's algorithm

Two node synchronization in an asynchronous setting.

- Node C memorizes time $t_i = t$ asks node S the time. S replies with time t_s and node C memorizes the reception time $t_f = t$.
- Node C calculates the roundtrip time as $t_r = t_f - t_i$.
- C can set the time to $t_s + \frac{t_r}{2}$ and expect to have a synchronization of $D = \frac{t_r}{2}$.
- t_r can be made smaller if we adjust for a lower bound b on message transmition time. $t_r = t_f - (t_i + b)$.

Time Synchronization

Asynchronous System: Cristian's algorithm

Two node synchronization in an asynchronous setting.

- Node C memorizes time $t_i = t$ asks node S the time. S replies with time t_s and node C memorizes the reception time $t_f = t$.
- Node C calculates the roundtrip time as $t_r = t_f - t_i$.
- C can set the time to $t_s + \frac{t_r}{2}$ and expect to have a synchronization of $D = \frac{t_r}{2}$.
- t_r can be made smaller if we adjust for a lower bound b on message transmition time. $t_r = t_f - (t_i + b)$.
- The algorithm can be repeated until we eventually observe a t_r that gives us a "tight enough" synchronization.

Time Synchronization

Summary

- Both in synchronous and asynchronous settings one can expect at most time synchronization in an envelope D.

Time Synchronization

Summary

- Both in synchronous and asynchronous settings one can expect at most time synchronization in an envelope D. Synchronization can be usefull to coordinate access to shared channels; either to avoid two senders at the same time or to make shure that sender and receiver are both awake.

Time Synchronization

Summary

- Both in synchronous and asynchronous settings one can expect at most time synchronization in an envelope D. Synchronization can be usefull to coordinate access to shared channels; either to avoid two senders at the same time or to make shure that sender and receiver are both awake.
- With enough timing resolution, tight envelopes, and slow computation steps (or slow processors) one could expect to tottaly order a distributed computation.

Time Synchronization

Summary

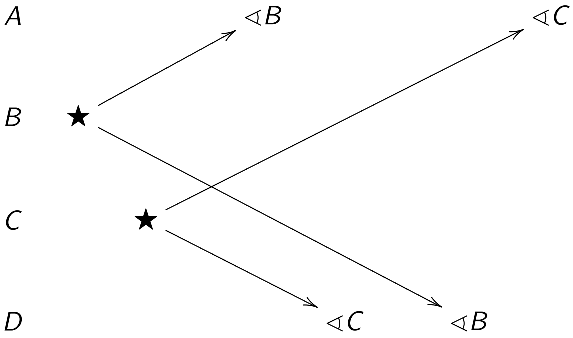
- Both in synchronous and asynchronous settings one can expect at most time synchronization in an envelope D. Synchronization can be usefull to coordinate access to shared channels; either to avoid two senders at the same time or to make shure that sender and receiver are both awake.
- With enough timing resolution, tight envelopes, and slow computation steps (or slow processors) one could expect to tottaly order a distributed computation.
  The resulting total order is not realistic and not always usefull, since it orders events that are in fact unrelated.

Time Synchronization
Summary

- Both in synchronous and asynchronous settings one can expect at most time synchronization in an envelope D. Synchronization can be usefull to coordinate access to shared channels; either to avoid two senders at the same time or to make shure that sender and receiver are both awake.
- With enough timing resolution, tight envelopes, and slow computation steps (or slow processors) one could expect to tottaly order a distributed computation.
  The resulting total order is not realistic and not always usefull, since it orders events that are in fact unrelated.
- Even on physical systems real time total ordering is not always consistent for diferent observers.

Time Synchronization

Ordering Explosions: Two independent ones

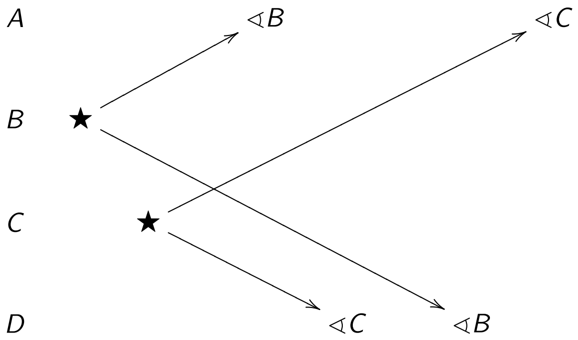


Observers

While A sees $\langle B, C \rangle$, D sees $\langle C, B \rangle$.

Time Synchronization

Ordering Explosions: Two independent ones



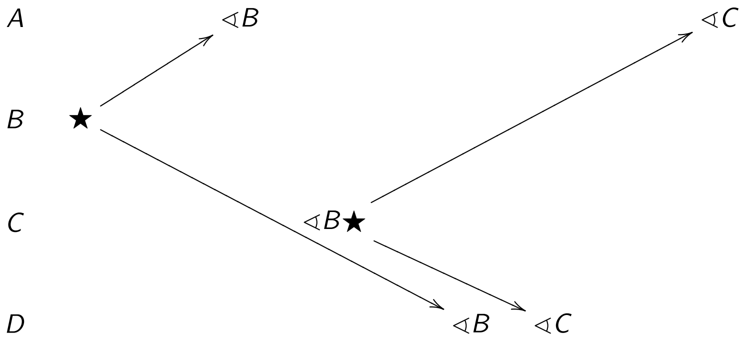
Observers

While A sees $\langle B, C \rangle$, D sees $\langle C, B \rangle$.

If we really need a total order (e.g. to make a replicated state machine) maybe we can give an arbitrary order to these events. As long as no one can contradict these decisions.

Time Synchronization

Ordering Explosions: One triggers the next

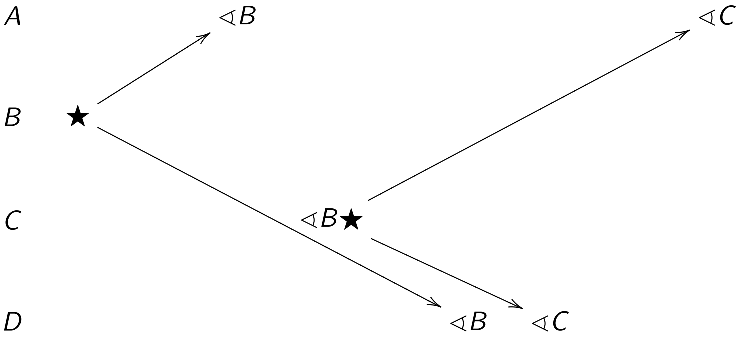


Observers

Now, both A and D see $\langle B, C\rangle$.

Time Synchronization

Ordering Explosions: One triggers the next



Observers

Now, both A and D see $\langle B, C \rangle$.

If message propagation speed is uniform, independent observers make consistent observations of events that might be causaly related. Otherwise the world would be much more confusing ...

Order Relations

Before digging deeper into order in distributed systems lets review some notions of order relations.

Order

- Concerns the comparison between pairs of objects.

Order Relations

Before digging deeper into order in distributed systems lets review some notions of order relations.

Order

- Concerns the comparison between pairs of objects.
- Is a binary relation on a set of objects. In order $\langle B, <_B \rangle$ we have $<_B \subseteq B \times B$.

Order Relations

Before digging deeper into order in distributed systems lets review some notions of order relations.

Order

- Concerns the comparison between pairs of objects.
- Is a binary relation on a set of objects. In order $\langle B, <_B \rangle$ we have $<_B \subseteq B \times B$.
- Order is transitive. $a < b \wedge b < c \Rightarrow a < c$.

Order Relations

Before digging deeper into order in distributed systems lets review some notions of order relations.

Order

- Concerns the comparison between pairs of objects.
- Is a binary relation on a set of objects. In order $\langle B, <_B \rangle$ we have $<_B \subseteq B \times B$.
- Order is transitive. $a < b \land b < c \Rightarrow a < c$.
- Order is antisymmetric. $a < b \Rightarrow b \not< a$.

Order Relations

Before digging deeper into order in distributed systems lets review some notions of order relations.

Order

- Concerns the comparison between pairs of objects.
- Is a binary relation on a set of objects. In order $\langle B, <_B \rangle$ we have $<_B \subseteq B \times B$.
- Order is transitive. $a < b \wedge b < c \Rightarrow a < c$.
- Order is antisymmetric. $a < b \Rightarrow b \not< a$.

If we miss *antisymmetry* we only have a **preorder**.

Order Relations

Before digging deeper into order in distributed systems lets review some notions of order relations.

Order

- Concerns the comparison between pairs of objects.
- Is a binary relation on a set of objects. In order $\langle B, <_B \rangle$ we have $<_B \subseteq B \times B$.
- Order is transitive. $a < b \wedge b < c \Rightarrow a < c$.
- Order is antisymmetric. $a < b \Rightarrow b \not< a$.

If we miss *antisymmetry* we only have a **preorder**.
Orders can be strict $<$ or non-strict $\leq$.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

reflexivity $x \leq x$.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

reflexivity $x \leq x$.

antisymmetry $x \leq y$ and $y \leq x$ imply $x = y$.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

- reflexivity $x \leq x$.
- antisymmetry $x \leq y$ and $y \leq x$ imply $x = y$.
- transitivity $x \leq y$ and $y \leq z$ imply $x \leq z$.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

reflexivity $x \leq x$.

antisymmetry $x \leq y$ and $y \leq x$ imply $x = y$.

transitivity $x \leq y$ and $y \leq z$ imply $x \leq z$.

In a **preorder** we can have $x \neq y$ and $x \leq y \wedge y \leq x$.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

reflexivity $x \leq x$.

antisymmetry $x \leq y$ and $y \leq x$ imply $x = y$.

transitivity $x \leq y$ and $y \leq z$ imply $x \leq z$.

In a **preorder** we can have $x \neq y$ and $x \leq y \wedge y \leq x$.
One also writes $x \parallel y$ to mean $x \not\leq y \wedge y \not\leq x$.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

reflexivity $x \leq x$.

antisymmetry $x \leq y$ and $y \leq x$ imply $x = y$.

transitivity $x \leq y$ and $y \leq z$ imply $x \leq z$.

In a **preorder** we can have $x \neq y$ and $x \leq y \wedge y \leq x$.
One also writes $x \parallel y$ to mean $x \not\leq y \wedge y \not\leq x$.

Chains and antichains

- If for all $x, y \in B$ either $x \leq y$ or $y \leq x$ we have a **chain**. Also known as **total order**, where all elements are comparable.

Order Relations

Non-strict order (or non strict partial order)

Let B be a set and $\leq$ a binary relation on B such that, for all $x, y, z \in B$:

reflexivity $x \leq x$.

antisymmetry $x \leq y$ and $y \leq x$ imply $x = y$.

transitivity $x \leq y$ and $y \leq z$ imply $x \leq z$.

In a **preorder** we can have $x \neq y$ and $x \leq y \wedge y \leq x$.
One also writes $x \parallel y$ to mean $x \not\leq y \wedge y \not\leq x$.

Chains and antichains

- If for all $x, y \in B$ either $x \leq y$ or $y \leq x$ we have a **chain**. Also known as **total order**, where all elements are comparable.
- We have an **antichain** if $x \leq y$ iff $x = y$.

Order Relations

Examples

Sets

A set X can be ordered by set inclusion, yielding $\langle X, \subseteq \rangle$.

Order Relations
Examples

Sets

A set X can be ordered by set inclusion, yielding $\langle X, \subseteq \rangle$.
The powerset $\mathcal{P}(X)$, consisting of all subsets of X, is ordered by set inclusion.

Order Relations
Examples

Sets

A set X can be ordered by set inclusion, yielding $\langle X, \subseteq \rangle$.
The powerset $\mathcal{P}(X)$, consisting of all subsets of X, is ordered by set inclusion.
Q: Does $\subseteq$ form a total order on $\mathcal{P}(X)$?

Order Relations

Examples

Sets

A set X can be ordered by set inclusion, yielding $\langle X, \subseteq \rangle$.
The powerset $\mathcal{P}(X)$, consisting of all subsets of X, is ordered by set inclusion.
Q: Does $\subseteq$ form a total order on $\mathcal{P}(X)$?
A: No, by counter example: $\{a, x, f\} \parallel \{x, b\}$.

Order Relations

Examples

Sets

A set X can be ordered by set inclusion, yielding $\langle X, \subseteq \rangle$.
The powerset $\mathcal{P}(X)$, consisting of all subsets of X, is ordered by set inclusion.
Q: Does $\subseteq$ form a total order on $\mathcal{P}(X)$?
A: No, by counter example: $\{a, x, f\} \parallel \{x, b\}$.

Binary sequences

Exhibit a **prefix** ordering. Let $\mathbf{2}^*$ be the set of all finite binary strings, including $\langle\rangle$. For $x, y \in \mathbf{2}^*$ we have $x \leq y$ iff x is a finite initial substring of v. E.g. $0100 < 010011$, $010 \parallel 100$.

Order Relations

Examples

Coordinatewise (pointwise) order

Let $P_1, \ldots, P_n$ be ordered sets. The cartesian product $P_1 \times \cdots \times P_n$ can define a ordered set by pointwise order:

$$(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n) \Leftrightarrow (\forall i) x_i \leq y_i \text{ in } P_i.$$

Order Relations

Examples

Coordinatewise (pointwise) order

Let $P_1, \ldots, P_n$ be ordered sets. The cartesian product $P_1 \times \cdots \times P_n$ can define a ordered set by pointwise order:

$$(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n) \Leftrightarrow (\forall i) x_i \leq y_i \text{ in } P_i.$$

Lexicographic order

Let A, B be two ordered sets. The product $A \times B$ can have a **lexicographic order** defined by
$(x_1, x_2) \leq (y_1, y_2)$ if $x_1 < y_1$ or ($x_1 = y_1$ and $x_2 \leq y_2$).
By iteration a lexicographic order can be defined on any finite product.

Order Relations

Relations among orders

Order isomorphism

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order isomorphism** is a surjective (onto) total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ iff $u \leq_S v$.

Order Relations

Relations among orders

Order isomorphism

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order isomorphism** is a surjective (onto) total function $h : S \rightarrow T$ such that for all $u, v \in S$:

$h(u) \leq_T h(v)$ iff $u \leq_S v$.

We say that $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ are equivalente and that one **characterizes** the other and vice-versa.

Order Relations

Relations among orders

Order isomorphism

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order isomorphism** is a surjective (onto) total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ iff $u \leq_S v$.
We say that $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ are equivalente and that one **characterizes** the other and vice-versa.

A weaker form is

Order preserving

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order preserving** maping is a total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ if $u \leq_S v$.

Order Relations

Relations among orders

Order isomorphism

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order isomorphism** is a surjective (onto) total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ iff $u \leq_S v$.
We say that $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ are equivalente and that one **characterizes** the other and vice-versa.

A weaker form is

Order preserving

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order preserving** maping is a total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ if $u \leq_S v$.
We say that $\langle T, \leq_T \rangle$ is **consistent** with $\langle S, \leq_S \rangle$.

Order Relations

Relations among orders

Order isomorphism

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order isomorphism** is a surjective (onto) total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ iff $u \leq_S v$.
We say that $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ are equivalente and that one **characterizes** the other and vice-versa.

A weaker form is

Order preserving

Given two partially ordered sets $\langle S, \leq_S \rangle$ and $\langle T, \leq_T \rangle$ an **order preserving** maping is a total function $h : S \rightarrow T$ such that for all $u, v \in S$:
$h(u) \leq_T h(v)$ if $u \leq_S v$.
We say that $\langle T, \leq_T \rangle$ is **consistent** with $\langle S, \leq_S \rangle$.

For instance, we will see that real time total ordering is consistent with causality.

Logical Time and Causality

Model

- An asynchronous system with a collection of totally ordered processes $p_1, \ldots, p_n$.

Logical Time and Causality
Model

- An asynchronous system with a collection of totally ordered processes $p_1, \ldots, p_n$.
- Reliable channels, not necessarely FIFO.

Logical Time and Causality
Model

- An asynchronous system with a collection of totally ordered processes $p_1, \ldots, p_n$.
- Reliable channels, not necessarely FIFO.
- Sequential processes, performing internal events, send events and corresponding receive events.

Logical Time and Causality
Model

- An asynchronous system with a collection of totally ordered processes $p_1, \ldots, p_n$.
- Reliable channels, not necessarely FIFO.
- Sequential processes, performing internal events, send events and corresponding receive events.

In each process p_i during a computation a *local history* is formed by the (potentially infinite) sequence of events: $h_i = \langle e_i^1, e_i^2, \ldots \rangle$. As expected, time between events varies.

Logical Time and Causality
Model

- An asynchronous system with a collection of totally ordered processes $p_1, \ldots, p_n$.
- Reliable channels, not necessarely FIFO.
- Sequential processes, performing internal events, send events and corresponding receive events.

In each process p_i during a computation a *local history* is formed by the (potentially infinite) sequence of events: $h_i = \langle e_i^1, e_i^2, \ldots \rangle$. As expected, time between events varies.
h_i^k denotes an initial prefix of local history h_i containing the first k events.

Logical Time and Causality

Model

- An asynchronous system with a collection of totally ordered processes $p_1, \ldots, p_n$.
- Reliable channels, not necessarely FIFO.
- Sequential processes, performing internal events, send events and corresponding receive events.

In each process p_i during a computation a *local history* is formed by the (potentially infinite) sequence of events: $h_i = \langle e_i^1, e_i^2, \ldots \rangle$. As expected, time between events varies.
h_i^k denotes an initial prefix of local history h_i containing the first k events.
The *global history* of the computation is the set $H = h_1 \cup \ldots \cup h_n$.

Logical Time and Causality

Causality

We can now define a causality relation in distributed systems.

Causality

Let $\langle H, \rightarrow \rangle$ be a global history H ordered by the smalest transitive binary relation $\rightarrow$ such that:

Logical Time and Causality

Causality

We can now define a causality relation in distributed systems.

Causality

Let $\langle H, \rightarrow \rangle$ be a global history H ordered by the smalest transitive binary relation $\rightarrow$ such that:

- $e_i^a \rightarrow e_i^b$ if $e_i^a, e_i^b \in H$ and $a < b$.

Logical Time and Causality

Causality

We can now define a causality relation in distributed systems.

Causality

Let $\langle H, \rightarrow \rangle$ be a global history H ordered by the smalest transitive binary relation $\rightarrow$ such that:

- $e_i^a \rightarrow e_i^b$ if $e_i^a, e_i^b \in H$ and $a < b$.
- $e_i^s \rightarrow e_j^r$ if e_i^s is a send event and e_j^r the corresponding receive event.

Logical Time and Causality
Causality

We can now define a causality relation in distributed systems.

Causality

Let $\langle H, \rightarrow \rangle$ be a global history H ordered by the smalest transitive binary relation $\rightarrow$ such that:

- $e_i^a \rightarrow e_i^b$ if $e_i^a, e_i^b \in H$ and $a < b$.
- $e_i^s \rightarrow e_j^r$ if e_i^s is a send event and e_j^r the corresponding receive event.

If $a \rightarrow b$ then a may have influenced b. In general we have potential causality.

Logical Time and Causality

Causality

We can now define a causality relation in distributed systems.

Causality

Let $\langle H, \rightarrow \rangle$ be a global history H ordered by the smalest transitive binary relation $\rightarrow$ such that:

- $e_i^a \rightarrow e_i^b$ if $e_i^a, e_i^b \in H$ and $a < b$.
- $e_i^s \rightarrow e_j^r$ if e_i^s is a send event and e_j^r the corresponding receive event.

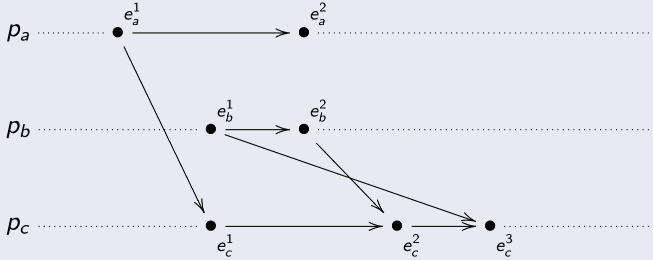
If $a \rightarrow b$ then a may have influenced b. In general we have potential causality.
On non trivial runs $\langle H, \rightarrow \rangle$ forms a partial order, and some events will be parallel $a \parallel b$ when neither $a \rightarrow b$ nor $b \rightarrow a$.

Logical Time and Causality

Preserving causal order

If we had a global time clock function $\mathcal{T} : H \rightarrow \mathbb{R}$ that would assign a real to each event. We would observe that the total order $\langle \mathcal{T}(H), < \rangle$ is consistent with $\langle H, \rightarrow \rangle$. Real time preserves the causal order.

Run

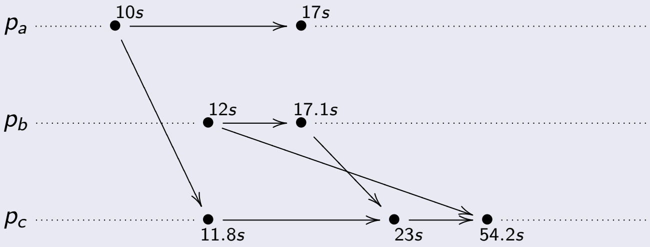


Logical Time and Causality

Preserving causal order

If we had a global time clock function $\mathcal{T} : H \rightarrow \mathbb{R}$ that would assign a real to each event. We would observe that the total order $\langle \mathcal{T}(H), < \rangle$ is consistent with $\langle H, \rightarrow \rangle$. Real time preserves the causal order.

Run with real time tags

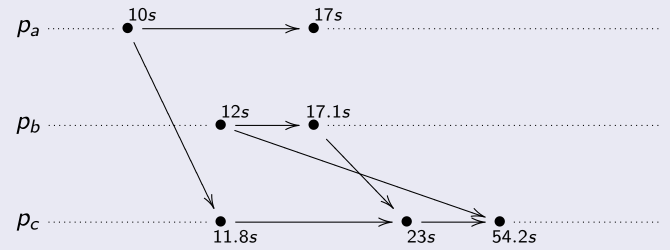


Logical Time and Causality

Preserving causal order

If we had a global time clock function $\mathcal{T} : H \rightarrow \mathbb{R}$ that would assign a real to each event. We would observe that the total order $\langle \mathcal{T}(H), < \rangle$ is consistent with $\langle H, \rightarrow \rangle$. Real time preserves the causal order.

Run with real time tags



Notice that while $11.8s < 12s$ the corresponding events are parallel $e_c^1 \parallel e_b^1$ in the causal order.

Logical Time and Causality
Clock condition

Being consistent with causality if often captured by a *clock condition*.

Logical Time and Causality

Clock condition

Being consistent with causality if often captured by a *clock condition*.

Clock Condition (Lamport 78)

A clock function $\mathcal{C} : H \rightarrow T$ and a ordered set $\langle T, < \rangle$ satisfies clock condition if:
For any events $a, b \in H$: if $a \rightarrow b$ then $\mathcal{C}(a) < \mathcal{C}(b)$.

Logical Time and Causality

Clock condition

Being consistent with causality if often captured by a *clock condition*.

Clock Condition (Lamport 78)

A clock function $\mathcal{C} : H \rightarrow T$ and a ordered set $\langle T, < \rangle$ satisfies clock condition if:
For any events $a, b \in H$: if $a \rightarrow b$ then $\mathcal{C}(a) < \mathcal{C}(b)$.

Notice that the timestamping function is necessarely one-to-one (injective) in order to satisfy the clock conditions and preserve the causal order.

Logical Time and Causality

Clock condition

Being consistent with causality if often captured by a *clock condition*.

Clock Condition (Lamport 78)

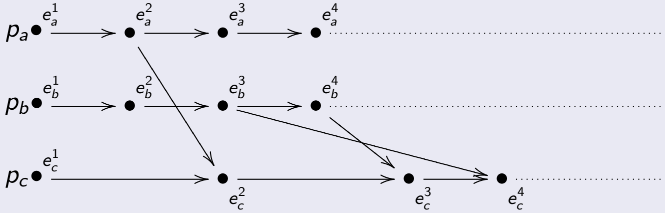
A clock function $\mathcal{C} : H \rightarrow T$ and a ordered set $\langle T, < \rangle$ satisfies clock condition if:
For any events $a, b \in H$: if $a \rightarrow b$ then $\mathcal{C}(a) < \mathcal{C}(b)$.

Notice that the timestamping function is necessarely one-to-one (injective) in order to satisfy the clock conditions and preserve the causal order.
Appart from real time there are other timestamping functions that satisfy this clock condition.

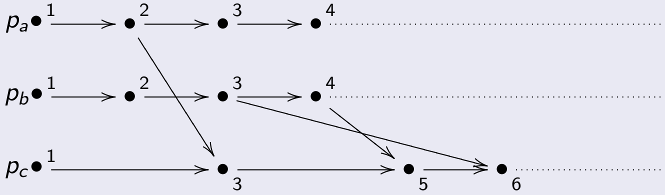
Logical Time and Causality

Lamport Time

Run



Run with timestamping consistent with causality



Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.
- On each internal event in p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$.

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.
- On each internal event in p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$.
- On a send event at p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$ and attach $\mathcal{L}_i$ to the message.

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.
- On each internal event in p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$.
- On a send event at p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$ and attach $\mathcal{L}_i$ to the message.
- On a receive event at p_i with $\mathcal{L}_x$ attached do $\mathcal{L}_i := max(\mathcal{L}_i, \mathcal{L}_x) + 1$.

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.
- On each internal event in p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$.
- On a send event at p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$ and attach $\mathcal{L}_i$ to the message.
- On a receive event at p_i with $\mathcal{L}_x$ attached do $\mathcal{L}_i := max(\mathcal{L}_i, \mathcal{L}_x) + 1$.

The value registred at $\mathcal{L}_i$ right after each event e_i^k is the one defining $\mathcal{L}(e_i^k)$.

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.
- On each internal event in p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$.
- On a send event at p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$ and attach $\mathcal{L}_i$ to the message.
- On a receive event at p_i with $\mathcal{L}_x$ attached do $\mathcal{L}_i := max(\mathcal{L}_i, \mathcal{L}_x) + 1$.

The value registred at $\mathcal{L}_i$ right after each event e_i^k is the one defining $\mathcal{L}(e_i^k)$. A positive integer could be used in place of 1.

Logical Time and Causality

Lamport Time

Lamport Time $\mathcal{L}$

We can assign integer valued timestamps by a function $\mathcal{L} : H \rightarrow \mathbb{N}$ constructed as follows, with local knowledge:

- Initially all processes p_i set $\mathcal{L}_i$ to 1.
- On each internal event in p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$.
- On a send event at p_i do $\mathcal{L}_i := \mathcal{L}_i + 1$ and attach $\mathcal{L}_i$ to the message.
- On a receive event at p_i with $\mathcal{L}_x$ attached do $\mathcal{L}_i := max(\mathcal{L}_i, \mathcal{L}_x) + 1$.

The value registred at $\mathcal{L}_i$ right after each event e_i^k is the one defining $\mathcal{L}(e_i^k)$. A positive integer could be used in place of 1.

Notice that while Lamport Time $\mathcal{L}$ and Real Time $\mathcal{L}$ are both consistent with causality $\langle H, \rightarrow \rangle$, the mutual relation between $\mathcal{L}$ and $\mathcal{T}$ is not tipically consistent in non trivial runs.

Logical Time and Causality

Lamport Time

We can further refine Lamport Time in order to obtain an injective function $\mathcal{L}^t$ that assigns a consistent total order to all events in H. It suffices to consider the lexicographic order on the pair formed by the Lamport Time and the process number.

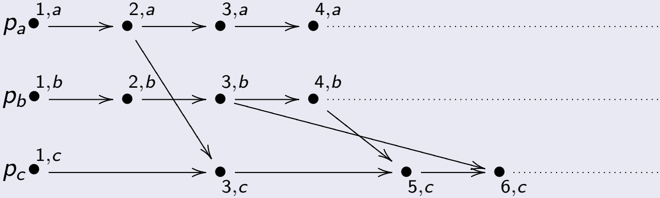
Logical Time and Causality

Lamport Time

We can further refine Lamport Time in order to obtain an injective function $\mathcal{L}^t$ that assigns a consistent total order to all events in H. It suffices to consider the lexicographic order on the pair formed by the Lamport Time and the process number.

Run with total order $\mathcal{L}^t$

Here, since processes have letters we assume the alphabetic order.



$(1, a)(1, b)(1, c)(2, a)(2, b)(3, a)(3, b)(3, c)(4, a)(4, b)(5, c)(6, c)$

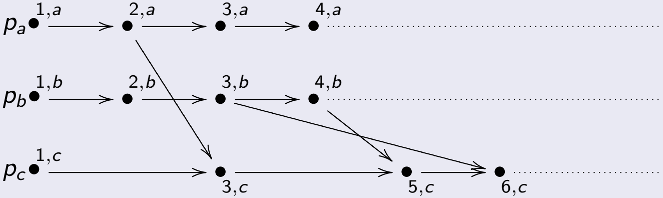
Logical Time and Causality

Lamport Time

We can further refine Lamport Time in order to obtain an injective function $\mathcal{L}^t$ that assigns a consistent total order to all events in H. It suffices to consider the lexicographic order on the pair formed by the Lamport Time and the process number.

Run with total order $\mathcal{L}^t$

Here, since processes have letters we assume the alphabetic order.



$(1, a)(1, b)(1, c)(2, a)(2, b)(3, a)(3, b)(3, c)(4, a)(4, b)(5, c)(6, c)$

This total order is usefull in many distributed algorithms (e.g. Lamport mutual exclusion algorithm), but it orders more events than causality. For other algorithms we need to capture causality precisely.

Logical Time and Causality
Characterizing causality

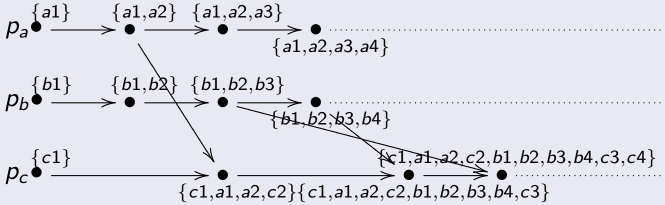
A simple timestamping mechanism that can characterize causality is to locally register the causal history $\mathcal{C} : H \rightarrow \mathcal{P}(H)$. This is done by collecting in a set each distinct event identifier.

Logical Time and Causality

Characterizing causality

A simple timestamping mechanism that can characterize causality is to locally register the causal history $\mathcal{C} : H \to \mathcal{P}(H)$. This is done by collecting in a set each distinct event identifier. Notice that each process has a unique number and can maintain a sequential counter for its events.

Run tagged with causal histories

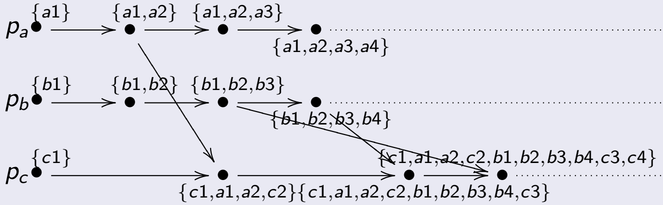


Logical Time and Causality

Characterizing causality

A simple timestamping mechanism that can characterize causality is to locally register the causal history $\mathcal{C} : H \to \mathcal{P}(H)$. This is done by collecting in a set each distinct event identifier. Notice that each process has a unique number and can maintain a sequential counter for its events.

Run tagged with causal histories



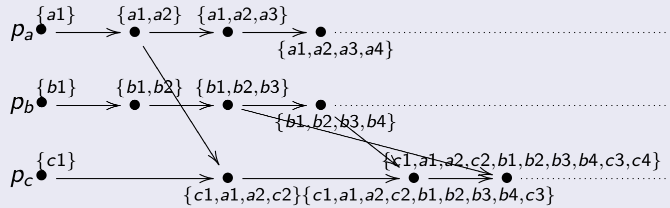
$$e_a^2 \to e_c^2 \Leftrightarrow \mathcal{C}(e_a^2) \subseteq \mathcal{C}(e_c^2) \Leftrightarrow \{a1, a2\} \subseteq \{c1, a1, a2, c2\}$$

Logical Time and Causality

Characterizing causality

A simple timestamping mechanism that can characterize causality is to locally register the causal history $\mathcal{C} : H \rightarrow \mathcal{P}(H)$. This is done by collecting in a set each distinct event identifier. Notice that each process has a unique number and can maintain a sequential counter for its events.

Run tagged with causal histories



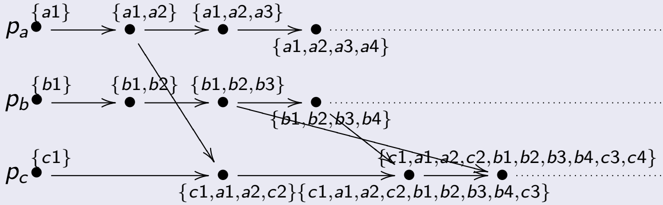
$$e_a^2 \rightarrow e_c^2 \Leftrightarrow \mathcal{C}(e_a^2) \subseteq \mathcal{C}(e_c^2) \Leftrightarrow \{a1, a2\} \subseteq \{c1, a1, a2, c2\}$$
$$e_a^3 \not\rightarrow e_b^3 \Leftrightarrow \mathcal{C}(e_a^3) \not\subseteq \mathcal{C}(e_b^3) \Leftrightarrow \{a1, a2, a3\} \not\subseteq \{b1, b2, b3\}$$

Logical Time and Causality

Characterizing causality

A simple timestamping mechanism that can characterize causality is to locally register the causal history $\mathcal{C} : H \rightarrow \mathcal{P}(H)$. This is done by collecting in a set each distinct event identifier. Notice that each process has a unique number and can maintain a sequential counter for its events.

Run tagged with causal histories

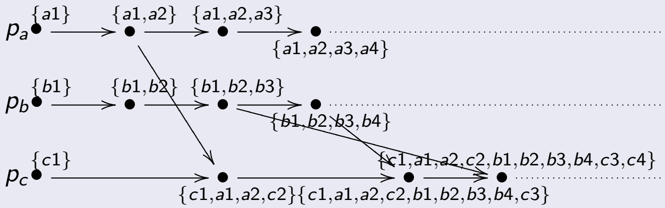


$$e_a^2 \rightarrow e_c^2 \Leftrightarrow \mathcal{C}(e_a^2) \subseteq \mathcal{C}(e_c^2) \Leftrightarrow \{a1, a2\} \subseteq \{c1, a1, a2, c2\}$$
$$e_a^3 \not\rightarrow e_b^3 \Leftrightarrow \mathcal{C}(e_a^3) \not\subseteq \mathcal{C}(e_b^3) \Leftrightarrow \{a1, a2, a3\} \not\subseteq \{b1, b2, b3\}$$
$$e_b^3 \not\rightarrow e_a^3 \Leftrightarrow \mathcal{C}(e_b^3) \not\subseteq \mathcal{C}(e_a^3) \Leftrightarrow \{b1, b2, b3\} \not\subseteq \{a1, a2, a3\}$$

Logical Time and Causality

Characterizing causality

Run tagged with causal histories

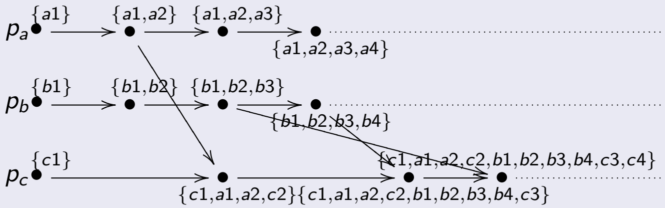


The problem of causal histories is their space complexity that grows linearly, $O(E)$, with the number of events E.

Logical Time and Causality

Characterizing causality

Run tagged with causal histories



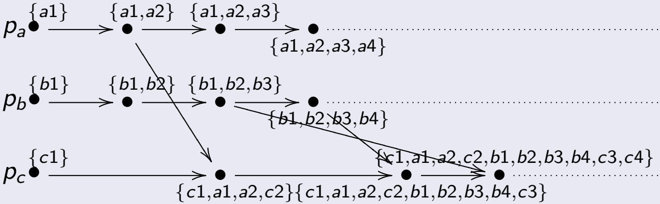
The problem of causal histories is their space complexity that grows linearly,$O(E)$, with the number of events E.

This can be solved by noticing that for all k and i and a causal history $\mathcal{C}_x$: if $e_i^k \in \mathcal{C}_x$ then $\{e_i^1, \ldots, e_i^{k-1}\} \subseteq \mathcal{C}_x$.

Logical Time and Causality

Characterizing causality

Run tagged with causal histories



The problem of causal histories is their space complexity that grows linearly, $O(E)$, with the number of events E.

This can be solved by noticing that for all k and i and a causal history $\mathcal{C}_x$: if $e_i^k \in \mathcal{C}_x$ then $\{e_i^1, \ldots, e_i^{k-1}\} \subseteq \mathcal{C}_x$.

Consequently one only needs to register the index of the last event from each process.

Logical Time and Causality

Vector Clocks

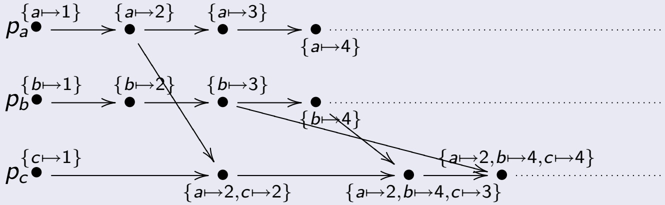
Vector clocks are compressed causal histories. $\mathcal{V} : H \longrightarrow \mathbb{N}^n$ where n is the number of processes. They can be represented in a vector or as mappings from process names to integers.

Logical Time and Causality

Vector Clocks

Vector clocks are compressed causal histories. $\mathcal{V} : H \longrightarrow \mathbb{N}^n$ where n is the number of processes. They can be represented in a vector or as mappings from process names to integers.

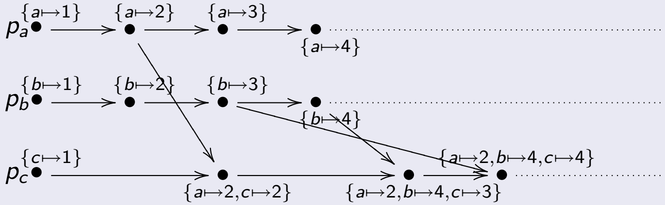
Run tagged with vector clocks



Logical Time and Causality
Vector Clocks

Vector clocks are compressed causal histories. $\mathcal{V} : H \longrightarrow \mathbb{N}^n$ where n is the number of processes. They can be represented in a vector or as mappings from process names to integers.

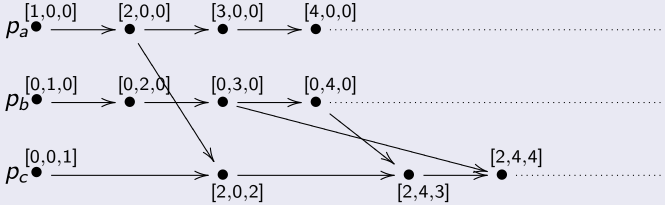
Run tagged with vector clocks



Vector clocks are used in many distributed algorithms. E.g. causal delivery of messages, an extension of FIFO delivery. They can be used as long as processes have unique ids. A total order on ids is only a convenience (trivially obtained from unique ids).

Logical Time and Causality
Vector Clocks

Run tagged with vector clocks

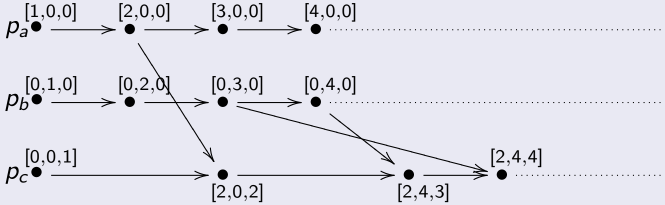


The cordinatewise (pointwise) order on version vectors characterizes causality. They define identical partial orders.
$[2, 0, 0] < [2, 0, 2]$ but $[0, 4, 0] \parallel [2, 0, 2]$

Logical Time and Causality

Vector Clocks

Run tagged with vector clocks



The cordinatewise (pointwise) order on version vectors characterizes causality. They define identical partial orders.
$[2, 0, 0] < [2, 0, 2]$ but $[0, 4, 0] \parallel [2, 0, 2]$
Complexity is $O(N \log E)$ and $\mathcal{V}$ is known to be the most concise timestamping mechanism for process causality tracking.

Process Causality vs Data Causality

- Causality is formed as relevant events are colected in a run.

Process Causality vs Data Causality

- Causality is formed as relevant events are colected in a run.
- In process causality the relevante events are *internal*, *send* and *receive* events.

Process Causality vs Data Causality

- Causality is formed as relevant events are colected in a run.
- In process causality the relevante events are *internal*, *send* and *receive* events.
- With causal histories a new event id is added in each case.

Process Causality vs Data Causality

- Causality is formed as relevant events are colected in a run.
- In process causality the relevante events are *internal*, *send* and *receive* events.
- With causal histories a new event id is added in each case.
- In data causality we are concerned with the ordering of replicas subject to optimistic operation.

Process Causality vs Data Causality

- Causality is formed as relevant events are colected in a run.
- In process causality the relevante events are *internal*, *send* and *receive* events.
- With causal histories a new event id is added in each case.
- In data causality we are concerned with the ordering of replicas subject to optimistic operation.
- The relevant events are *update* events on the replica state.

Process Causality vs Data Causality

- Causality is formed as relevant events are colected in a run.
- In process causality the relevante events are *internal*, *send* and *receive* events.
- With causal histories a new event id is added in each case.
- In data causality we are concerned with the ordering of replicas subject to optimistic operation.
- The relevant events are *update* events on the replica state.
- If each update event is distinguished, two replicas that know the same set of update events are equivalente.

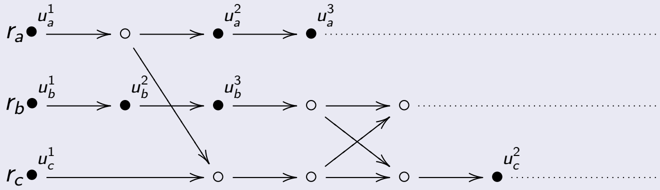
Data Causality

Data causality is at the core of version control systems, replicated file systems, partitioned operation and optimistic replication in general.

Data Causality

Data causality is at the core of version control systems, replicated file systems, partitioned operation and optimistic replication in general.

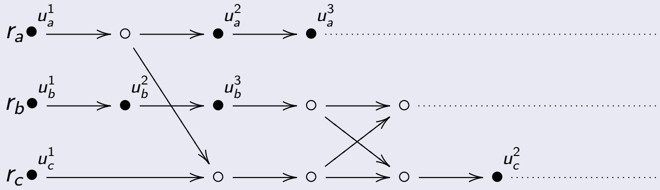
Run



Data Causality

Data causality is at the core of version control systems, replicated file systems, partitioned operation and optimistic replication in general.

Run

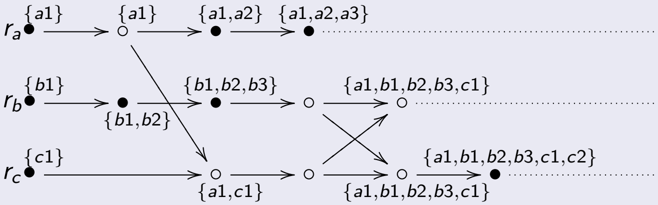


This run includes sending and receiving of messages but causality tracking can ignore these events.

Data Causality

Causal histories

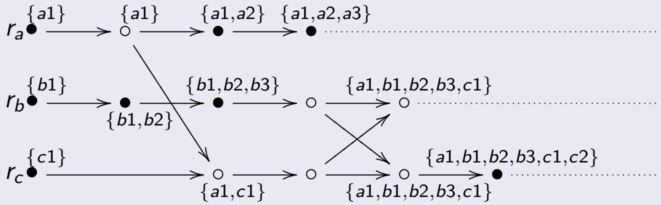
Run with causal histories



Data Causality

Causal histories

Run with causal histories

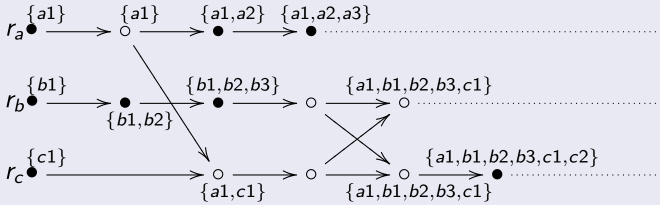


Unlike process causality, where all events depict different causal histories, here replicas can known the same set of events. In that case we say that replicas are equivalent.

Data Causality

Causal histories

Run with causal histories



Unlike process causality, where all events depict different causal histories, here replicas can known the same set of events. In that case we say that replicas are equivalent.

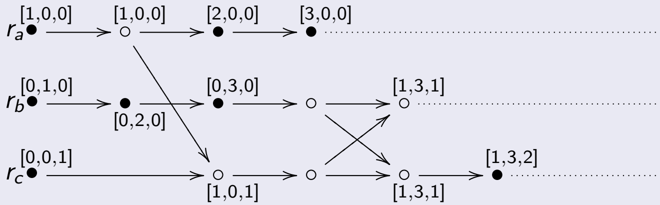
At the end of this run we have the following relations among replicas, as observed by set inclusion:

$r_a \parallel r_b$ and $r_a \parallel r_c$ and $r_b < r_c$.

Data Causality

Version vectors

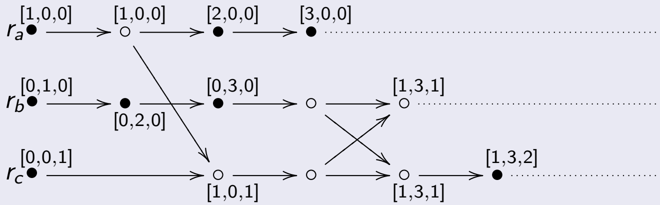
Run with version vectors



Data Causality

Version vectors

Run with version vectors

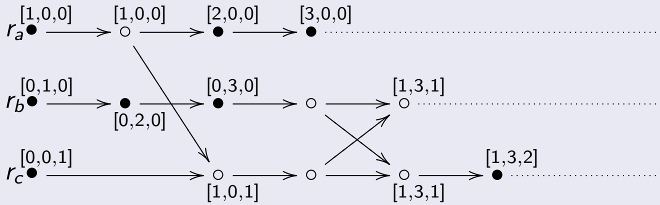


Both version vectors and causal histories characterize data causality.
Are version vectors the most concise representation of data causality?

Data Causality

Version vectors

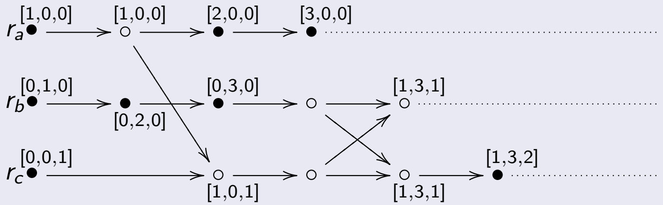
Run with version vectors



Both version vectors and causal histories characterize data causality. Are version vectors the most concise representation of data causality? In fact, no, altough it looks like.

Data Causality
Version vectors

Run with version vectors



Both version vectors and causal histories characterize data causality.
Are version vectors the most concise representation of data causality?
In fact, no, altough it looks like.
Altough we have a unbounded number of update events in data causality one is only concerned about the order among existing replicas. Those forming the frontier of the run.

Data Causality

Frontier configurations

With two replicas the following cases are possible:

- $r_a = r_b$
- $r_a < r_b$
- $r_a > r_b$
- $r_a \parallel r_b$

Data Causality

Frontier configurations

With two replicas the following cases are possible:

- $r_a = r_b$
- $r_a < r_b$
- $r_a > r_b$
- $r_a \parallel r_b$

With three replicas we have more cases, altough a finite number of them.

Data Causality
Frontier configurations

With two replicas the following cases are possible:

- $r_a = r_b$
- $r_a < r_b$
- $r_a > r_b$
- $r_a \parallel r_b$

With three replicas we have more cases, altough a finite number of them.
From "On the computer enumeration of finite topologies" they are found to be
$\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 29, 4 \mapsto 355, 5 \mapsto 6942, 6 \mapsto 209527, 7 \mapsto 9535241\}$

Data Causality

Frontier configurations

With two replicas the following cases are possible:

- $r_a = r_b$
- $r_a < r_b$
- $r_a > r_b$
- $r_a \parallel r_b$

With three replicas we have more cases, altough a finite number of them.

From ”On the computer enumeration of finite topologies” they are found to be

$\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 29, 4 \mapsto 355, 5 \mapsto 6942, 6 \mapsto 209527, 7 \mapsto 9535241\}$

Is there a local distributed algorithm that can characterize this partial order (possibly pre-order) with a bounded state?

Data Causality

Frontier configurations

With two replicas the following cases are possible:

- $r_a = r_b$
- $r_a < r_b$
- $r_a > r_b$
- $r_a \parallel r_b$

With three replicas we have more cases, altough a finite number of them.

From "On the computer enumeration of finite topologies" they are found to be

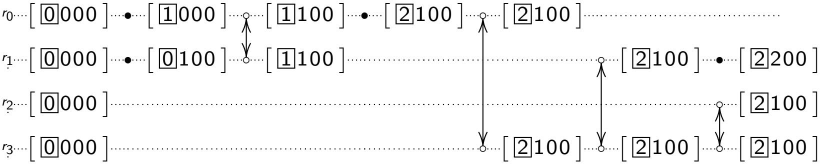
$\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 29, 4 \mapsto 355, 5 \mapsto 6942, 6 \mapsto 209527, 7 \mapsto 9535241\}$

Is there a local distributed algorithm that can characterize this partial order (possibly pre-order) with a bounded state?

This is possible with bounded version vectors.

Data Causality

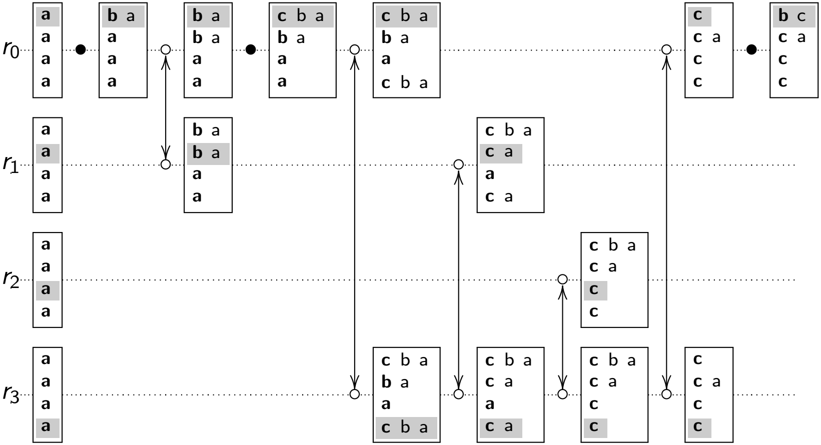
Pointwise order



Since version vectors are define with pointwise order it is enough do find a bounded replacement for each component that defines a total order in all frontiers.

Data Causality

Bounded version vectors



Data Causality

Bounded version vectors

Bounded version vectors can characterize data causality with a state that is independent on the number of updates. The required state is polynomial with respect to the number of replicas.

Let U be the number of updates, and N the number of replicas.

- Traditional version vectors have scale $O(N \log_2(U))$
- Bounded version vectors have scale $O(N^3 \log_2(N))$

Data Causality

Bounded version vectors

Bounded version vectors can characterize data causality with a state that is independent on the number of updates. The required state is polynomial with respect to the number of replicas.
Let U be the number of updates, and N the number of replicas.

- Traditional version vectors have scale $O(N \log_2(U))$
- Bounded version vectors have scale $O(N^3 \log_2(N))$

Consequently, the bounded approach can only be efficient for very small numbers of replicas or extremely high update rates.

Data Causality

Bounded version vectors

Bounded version vectors can characterize data causality with a state that is independent on the number of updates. The required state is polynomial with respect to the number of replicas.

Let U be the number of updates, and N the number of replicas.

- Traditional version vectors have scale $O(N \log_2(U))$
- Bounded version vectors have scale $O(N^3 \log_2(N))$

Consequently, the bounded approach can only be efficient for very small numbers of replicas or extremely high update rates.

In addition, synchronizations must be bidirectional.

Data Causality
Dynamic Number of Replicas

The previous mechanisms assumed a known number of replicas of global naming. This might not be possible in partitioned settings, just where optimistic replication is more needed.

Data Causality

Dynamic Number of Replicas

The previous mechanisms assumed a known number of replicas of global naming. This might not be possible in partitioned settings, just where optimistic replication is more needed.

Setting

- Replica forking, update and synchronization.

Data Causality

Dynamic Number of Replicas

The previous mechanisms assumed a known number of replicas of global naming. This might not be possible in partitioned settings, just where optimistic replication is more needed.

Setting

- Replica forking, update and synchronization.
- Variable number of replicas: variable-width frontier.

Data Causality

Dynamic Number of Replicas

The previous mechanisms assumed a known number of replicas of global naming. This might not be possible in partitioned settings, just where optimistic replication is more needed.

Setting

- Replica forking, update and synchronization.
- Variable number of replicas: variable-width frontier.
- Example: ad-hoc file copying and updating.

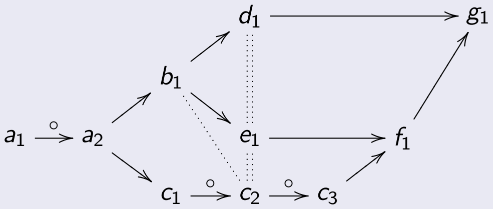
Data Causality

Dynamic Number of Replicas

The previous mechanisms assumed a known number of replicas of global naming. This might not be possible in partitioned settings, just where optimistic replication is more needed.

Setting

- Replica forking, update and synchronization.
- Variable number of replicas: variable-width frontier.
- Example: ad-hoc file copying and updating.



Data Causality
Version Stamps

Identity component

- Local management of the namespace.
- Distinguishes a replica from all coexisting ones.
- Available namespace from which other replicas can be generated.

Data Causality
Version Stamps

Identity component

- Local management of the namespace.
- Distinguishes a replica from all coexisting ones.
- Available namespace from which other replicas can be generated.

Update component

- Records when changes were applied.
- Identity-like value collected from ancestors.
- Global comparison of replicas.

Data Causality
Version Stamps

Identity component

- Local management of the namespace.
- Distinguishes a replica from all coexisting ones.
- Available namespace from which other replicas can be generated.

Update component

- Records when changes were applied.
- Identity-like value collected from ancestors.
- Global comparison of replicas.

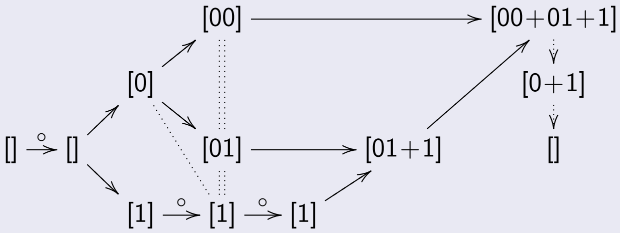
Other features

- Both components are a set of binary strings.
- No map from identifiers to counters is kept.
- No counters are used at all.
- Structure can grow and shrink.

Data Causality

Version Stamps

Identity component

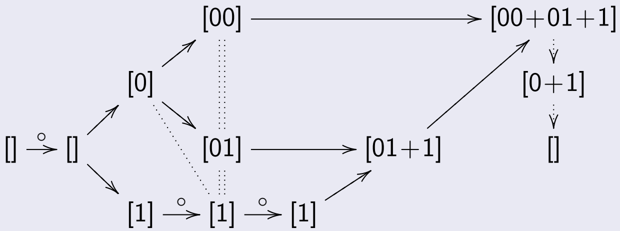


- An update causes no change on the id.

Data Causality

Version Stamps

Identity component

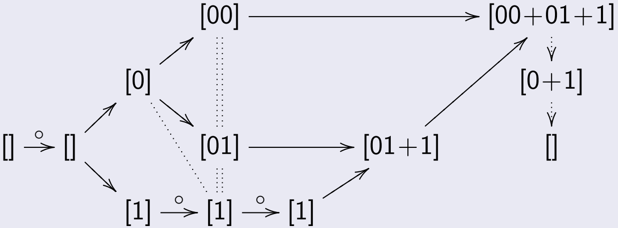


- An update causes no change on the id.
- Forks append either 0 or 1 to each string in the id.

Data Causality

Version Stamps

Identity component

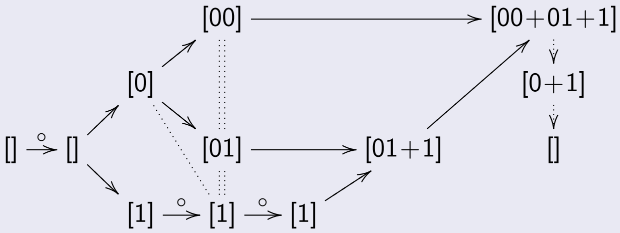


- An update causes no change on the id.
- Forks append either 0 or 1 to each string in the id.
- A join merges the sets of string.

Data Causality

Version Stamps

Identity component

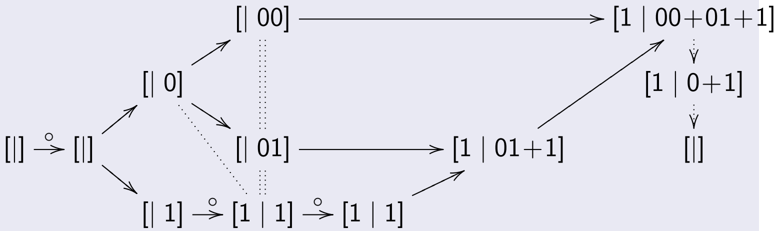


- An update causes no change on the id.
- Forks append either 0 or 1 to each string in the id.
- A join merges the sets of string.
- A possible simplification is attempted upon a join.

Data Causality

Version Stamps

Update component



[|] $\overset{\circ}{\to}$ [||]

[| 0]

[| 1] $\overset{\circ}{\to}$ [1 | 1] $\overset{\circ}{\to}$ [1 | 1]

[| 00]

[| 01]

[1 | 01+1]

[1 | 00+01+1]

[1 | 0+1]

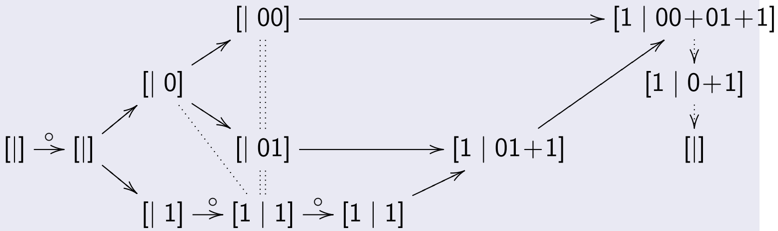
[|]

- Updates copy id into update.

Data Causality

Version Stamps

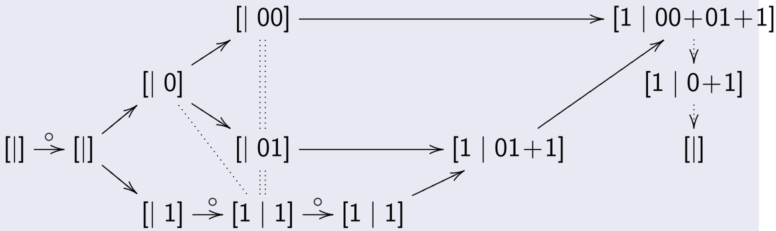
Update component



- Updates copy id into update.
- A fork causes no change in the update component.

Data Causality
Version Stamps

Update component

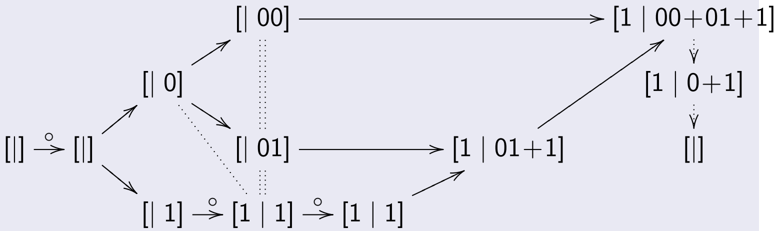


- Updates copy id into update.
- A fork causes no change in the update component.
- A join merges the update components.

Data Causality

Version Stamps

Update component

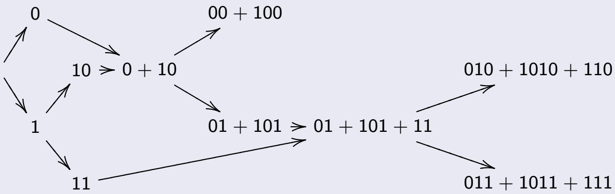


- Updates copy id into update.
- A fork causes no change in the update component.
- A join merges the update components.
- A simplification upon join also reflects in update.

Data Causality

Version Stamps: Pollution of the Namespace

Patologic run



- Pattern: join and fork again with alternating replicas.
- Leads to an overly refined namespace that cannot be simplified.
- This pattern can often occur in a real usage scenario.
- Copy of the identity to the update component aggravates the problem.
- Although correct practical application is severely compromised.

Logical Time and Causality

Function Graphs

Pointwise Order

- $[2, 5, 1] \leq [3, 6, 4]$
- $[2, 5, 1] \not\leq [3, 2, 5]$ and $[3, 2, 5] \not\leq [2, 5, 1]$ means $[2, 5, 1] \parallel [3, 2, 5]$

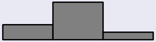
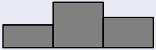
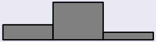
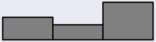
Logical Time and Causality

Function Graphs

Pointwise Order

- $[2, 5, 1] \leq [3, 6, 4]$
- $[2, 5, 1] \not\leq [3, 2, 5]$ and $[3, 2, 5] \not\leq [2, 5, 1]$ means $[2, 5, 1] \parallel [3, 2, 5]$

Vector Clocks as Function Graphs

- $[2, 5, 1] \leq [3, 6, 4] \equiv$  $\leq$ 
- $[2, 5, 1] \parallel [3, 2, 5] \equiv$  $\parallel$ 

Function graph containment characterizes causality. Vectors of integers can be re-interpreted as a way to encode the function.

Logical Time and Causality

Stamps

- In order to register new events (advance logical time) each active entity must know which position to update (vector index for that entity).

Logical Time and Causality
Stamps

- In order to register new events (advance logical time) each active entity must know which position to update (vector index for that entity).
- Stamps (logical clocks) are a pair (i, e), formed by an *id* and an *event* component that encodes causally known events.

Logical Time and Causality

Stamps

- In order to register new events (advance logical time) each active entity must know which position to update (vector index for that entity).
- Stamps (logical clocks) are a pair (i, e), formed by an *id* and an *event* component that encodes causally known events.

Vector Clock Stamp

Process P_b can hold a stamp $(2, [1, 2, 3])$ giving it access to the 2^{nd} index, and register an update deriving $(2, [1, 3, 3])$.

Logical Time and Causality

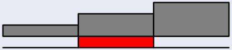
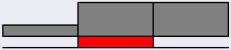
Stamps

- In order to register new events (advance logical time) each active entity must know which position to update (vector index for that entity).
- Stamps (logical clocks) are a pair (i, e), formed by an *id* and an *event* component that encodes causally known events.

Vector Clock Stamp

Process P_b can hold a stamp $(2, [1, 2, 3])$ giving it access to the 2^{nd} index, and register an update deriving $(2, [1, 3, 3])$.

Function Stamp

Stamp  can be updated to .

Logical Time and Causality

Global Invariants on IDs

Causality characterization condition

Each entity has a portion of its identity that is exclusive to it. This means each entity having an identity which maps to 1 some element which is mapped to 0 in all other entities.

$$\forall i.\ (i \cdot \bigsqcup_{i' \neq i} i') \neq i.$$

Entity events must use at least a part of the exclusive portion.

Logical Time and Causality

Global Invariants on IDs

Causality characterization condition

Each entity has a portion of its identity that is exclusive to it. This means each entity having an identity which maps to 1 some element which is mapped to 0 in all other entities.

$$\forall i.\ \ (i \cdot \bigsqcup_{i' \neq i} i') \neq i.$$

Entity events must use at least a part of the exclusive portion.

Disjoint condition

A less general but more practical condition is that all identities are kept disjoint. i.e. non-overlapping graphs for any pair of id functions.

$$\forall i_1 \neq i_2.\ \ i_1 \cdot i_2 = \mathbf{0}.$$

Any portion of the id can be used to register events.

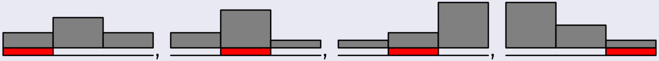
Logical Time and Causality

Plausible Clocks and Lamport Clocks

Plausible clocks [Torres-Rojas 99] and Lamport clocks [Lamport 78] do not meet the causality characterization condition. They are only consistent with causality.

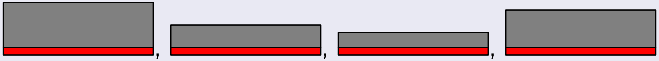
Plausible Clocks

Entities can share ids and update on the same position.



Lamport Clocks

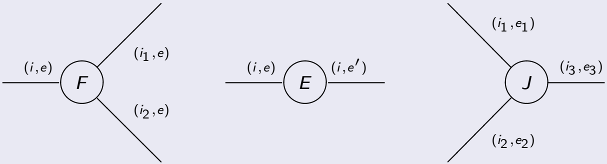
A single id position is used across all entities.



Fork-Event-Join Model

Causality tracking mechanisms (both static and dynamic) can be modeled by a set of core operations: *fork*; *event* and *join*, that act on stamps.

FEJ kernel



Function space based Clock Mechanisms

under the Disjoint Condition

- Core operations:

$$\text{fork} \quad \mathit{fork}((i,e)) \doteq ((i_1,e),(i_2,e))$$
$$\text{subject to } i_1 + i_2 = i \text{ and } i_1 \cdot i_2 = \mathbf{0}.$$

Function space based Clock Mechanisms

under the Disjoint Condition

- Core operations:

$$\begin{array}{rl} \text{fork} & \mathit{fork}((i,e)) \doteq ((i_1,e),(i_2,e)) \\ & \text{subject to } i_1+i_2=i \text{ and } i_1\cdot i_2=\mathbf{0}. \\ \text{event} & \mathit{event}((i,e)) \doteq (i, e+f\cdot i) \\ & \text{for any } f \text{ such that } f\cdot i>\mathbf{0}. \end{array}$$

Function space based Clock Mechanisms

under the Disjoint Condition

- Core operations:

$$\begin{array}{rl}
\text{fork} & \mathit{fork}((i,e)) \doteq ((i_1,e),(i_2,e)) \\
 & \quad \text{subject to } i_1 + i_2 = i \text{ and } i_1 \cdot i_2 = \mathbf{0}. \\
\text{event} & \mathit{event}((i,e)) \doteq (i, e + f \cdot i) \\
 & \quad \text{for any } f \text{ such that } f \cdot i > \mathbf{0}. \\
\text{join} & \sqcup((i_1,e_1),(i_2,e_2)) \doteq (i_1 + i_2, e_1 \sqcup e_2).
\end{array}$$

Function space based Clock Mechanisms

under the Disjoint Condition

- Core operations:

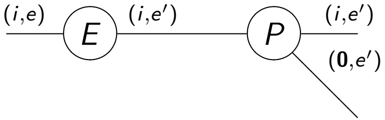
$$\begin{array}{rl} \text{fork} & \mathit{fork}((i,e)) \doteq ((i_1,e),(i_2,e)) \\ & \quad \text{subject to } i_1 + i_2 = i \text{ and } i_1 \cdot i_2 = \mathbf{0}. \\ \text{event} & \mathit{event}((i,e)) \doteq (i, e + f \cdot i) \\ & \quad \text{for any } f \text{ such that } f \cdot i > \mathbf{0}. \\ \text{join} & \sqcup((i_1,e_1),(i_2,e_2)) \doteq (i_1 + i_2, e_1 \sqcup e_2). \end{array}$$

- Peek, a special kind of fork is usefull to create imutable entities (messages or replicas):

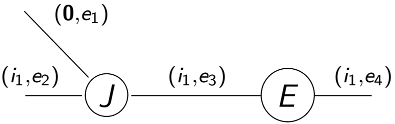
$$\begin{array}{rl} \text{peek} & \mathit{peek}((i,e)) \doteq ((\mathbf{0},e),(i,e)). \end{array}$$

FEJ
Send

Send: This operation is the atomic composition of *event* followed by *peek*. E.g. in vector clock systems, message sending is modeled by incrementing the local counter and then creating a new message.

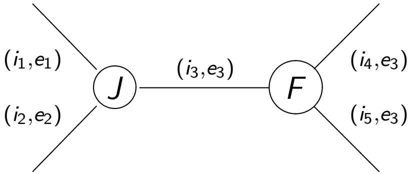


Receive: A *receive* is the atomic composition of *join* followed by *event*. E.g. in vector clocks taking the pointwise maximum is followed by an increment of the local counter.



Apendix
Sync

Sync: A *sync* is the atomic composition of *join* followed by *fork*. E.g. In version vector systems and in bounded version vectors it models the atomic synchronization of two replicas.



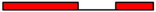
- ITC is a concrete mechanism that meets the FEJ specification.
- Allows decentralized creation and retirement of entities.
- The representation adapts automatically to the number of existing entities, growing or shrinking appropriately.
- ITC is based on functions over a continuous infinite domain ($\mathbb{R}$) with emphasis on the interval $[0, 1)$
- Functions are encoded as trees.

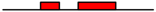
Interval Tree Clocks

Id Component

The id component is an *id tree* with the recursive form:

$$i \doteq 0 \mid 1 \mid (i_1, i_2).$$

$(1,(0,1)) \quad \sim$ 

$((0,(1,0)),(1,0)) \quad \sim$ 

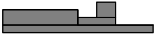
Interval Tree Clocks

Event Component

The event component is a binary *event tree* with non-negative integers in nodes:

$$e \doteq n \mid (n, e_1, e_2).$$

$$(1, 2, (0, (1, 0, 2), 0)) \quad \sim$$

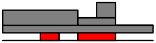


Interval Tree Clocks

ITC Stamps

A stamp in ITC is a pair (i, e).

$$(((0,(1,0)),(1,0)),(1,2,(0,(1,0,2),0))) \quad \sim$$

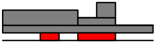


Interval Tree Clocks

ITC Stamps

A stamp in ITC is a pair (i, e).

$$(((0,(1,0)),(1,0)),(1,2,(0,(1,0,2),0))) \quad \sim$$

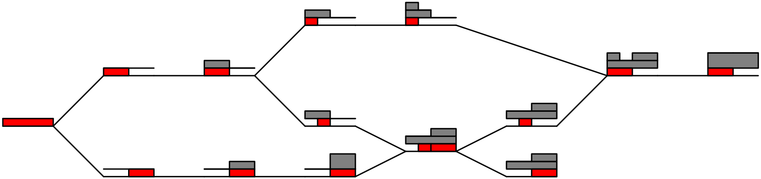


ITC makes use what we call the *seed* stamp, $(1, 0)$, from which we can fork as desired to obtain an initial configuration.

$$(1,0) \quad \sim$$

Interval Tree Clocks

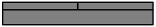
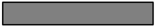
A run

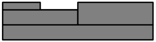
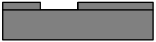


Interval Tree Clocks

Normal form

The event component can be normalized, preserving its interpretation as a function.

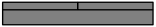
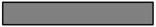
$(2, 1, 1) \sim$  $\equiv$  $\sim 3,$

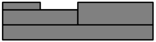
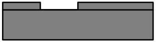
$(2, (2, 1, 0), 3) \sim$  $\equiv$  $\sim (4, (0, 1, 0), 1).$

Interval Tree Clocks

Normal form

The event component can be normalized, preserving its interpretation as a function.

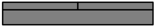
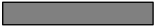
$(2,1,1) \sim$  $\equiv$  $\sim 3,$

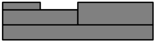
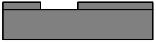
$(2,(2,1,0),3) \sim$  $\equiv$  $\sim (4,(0,1,0),1).$

- Normalization helps to keep a compact encoding.

Interval Tree Clocks
Normal form

The event component can be normalized, preserving its interpretation as a function.

$(2, 1, 1) \sim$  $\equiv$  $\sim 3,$

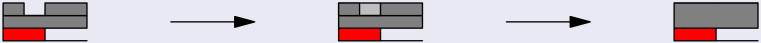
$(2, (2, 1, 0), 3) \sim$  $\equiv$  $\sim (4, (0, 1, 0), 1).$

- Normalization helps to keep a compact encoding.
- Counters flow from leaves to root, further helping encoding.

Interval Tree Clocks

Event operation

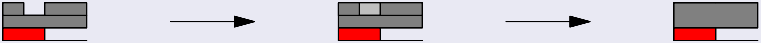
Fill



Interval Tree Clocks

Event operation

Fill

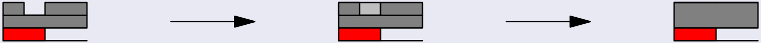


Fills can fill several areas and induce large simplifications.

Interval Tree Clocks

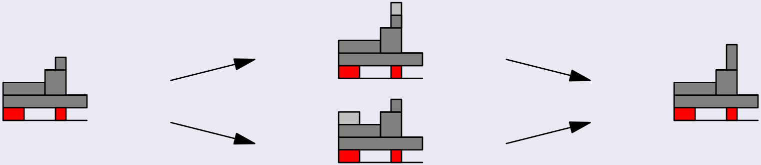
Event operation

Fill



Fills can fill several areas and induce large simplifications.

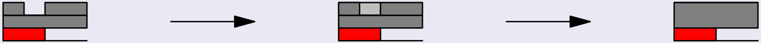
Grow



Interval Tree Clocks

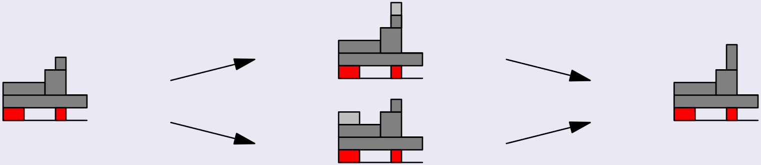
Event operation

Fill



Fills can fill several areas and induce large simplifications.

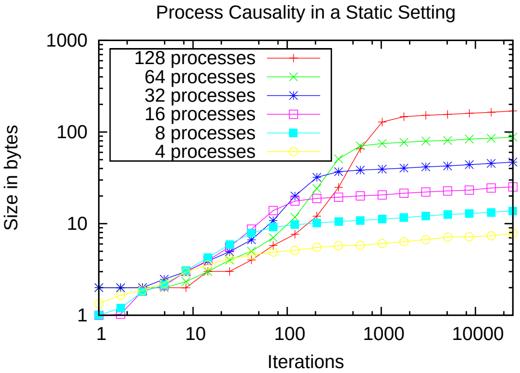
Grow



Fills are prefered to Grows, and alternative Grows are selected by evaluating its impact on encoding size.

Interval Tree Clocks

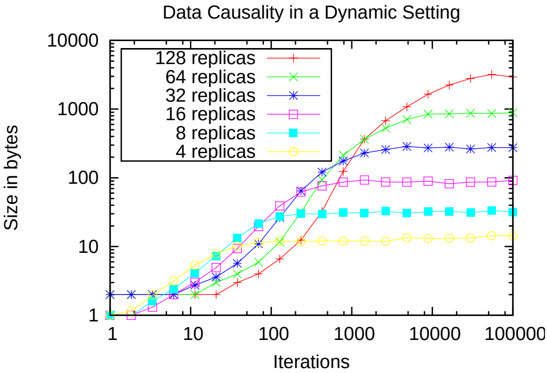
Fixed set of processes exchanging messages



For process causality in a static scenario, we operate on a fixed set of processes doing message exchanges (via peek and join) and recording internal events; here ids remain unchanged, since messages are anonymous.

Interval Tree Clocks

Data replicas under churn



Each iteration consists of forking, recording an event and joining two replicas, each performed on random replicas, leading to constantly evolving ids. This pattern maintains the number of existing replicas while exercising id management under churn.

Bibliography

- Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Özalp Babaoğlu, Keith Marzullo. 1993.
- Version Stamps: Decentralized Version vectors. Paulo Almeida, Carlos Baquero, Victor Fonte. ICDCS 2002.
- Bounded Version Vectors. Bacelar Almeida, Paulo Almeida, Carlos Baquero. DISC 2004.
- Interval Tree Clocks: A Logical Clock for Dynamic Systems. OPODIS 2008