Programming Distributed Systems

03 Time in Distributed Systems

Annette Bieniusa

FB Informatik
TU Kaiserslautern

Summer Term 2020

Coordination

- Need to manage the interactions and dependencies between interactions in distributed systems
- Data synchronization
- Process synchronization
  - Can be based on actual time or on relative order
  - Example: No simultaneous access to shared resource

Time in Distributed Systems

Bild von Gerd Altmann auf Pixabay

Example: Running `make` [5]

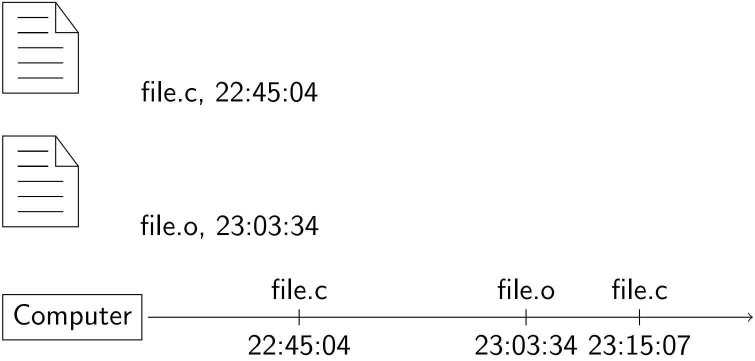
Timestamps of files used to check what needs to be recompiled

file.c, 22:45:04

file.o, 23:03:34

Example: Running `make` [5]

Here, compilation required:

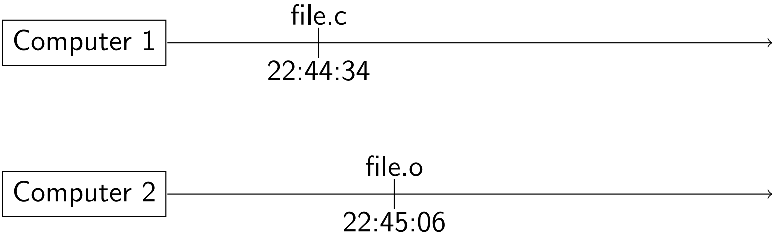


file.c, 22:45:04

file.o, 23:03:34

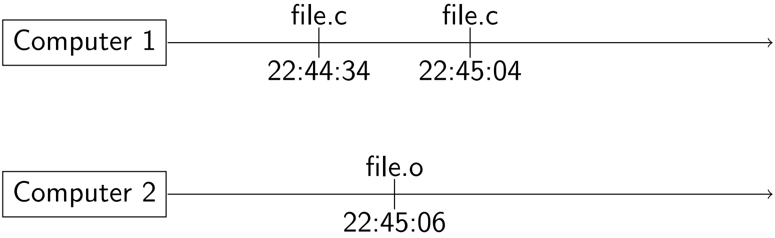
Example: Running `make` [5]

In a distributed file system where Computer 1 handles source files and Computer 2 handles object files:



Example: Running `make` [5]

In a distributed file system where Computer 1 handles source files and Computer 2 handles object files:



Goals of this Learning path

In this learning path, you will learn

- to name use cases for physical and logical clocks
- to describe the principle workings and challenges of constructing and synchronizing physical clocks
- to use Lamport timestamps and vector clocks to describe event relations
- to derive the construction of vector clocks from causal event histories
- to implement logical clocks in Erlang

Physical clocks

Timers based on quartz crystal oscillators

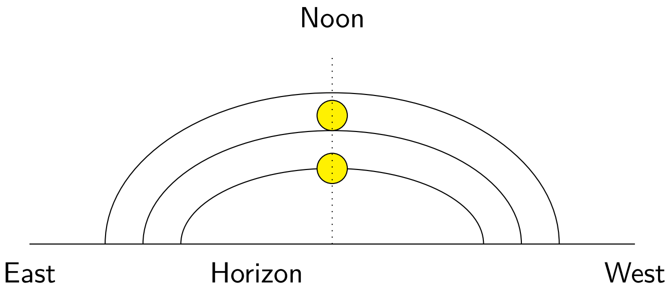
Wikipedia, Marcin Andrzejewski / CC BY-SA 3.0

- Computers use quartz crystals as timers
- Oscillates at specific frequency
- Used to update the system's software clock in CMOS RAM
- Consistent within one CPU

Problems

- Oscillators get gradually out-of-sync
- **Clock skew**: difference in time values between different timers
- **Clock drift** at rate of $\approx 10^{-6} s/s$ or 31.5 s/year

Solar time as time reference

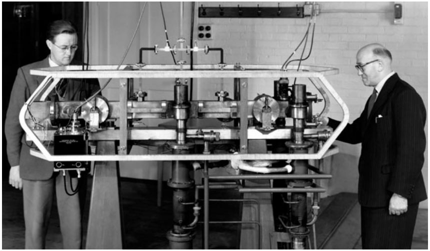


- Solar second is 1/86.400 of solar day
- *Problem:* Period of earth rotation is not stable

⇒ Our days are getting longer!

Atomic clocks

- 9.192.631.770 transitions of Cesium-133 atom corresponded to mean solar second in 1948
- Bureau International de l'Heure obtains averages from several atomic clocks to obtain the **International Atomic Time (TAI)**
- *Problem:* Diverges slowly from solar time
- **Universal Coordinated Time (UTC)** introduces **leap seconds**



National Physical Laboratory / Public domain World's first caesium-133 atomic clock

Definitions

- Let $C_p(t)$ be the time at processor p at time t.
- In a perfect world: $C_p(t) = t \quad \forall p, t$

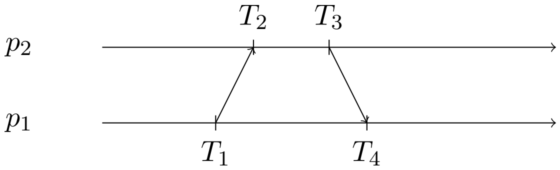
Accuracy

- $\forall t, p : \quad |C_p(t) - t| \leq \alpha$
- Achieved by *external synchronization* with a reference clock

Precision

- $\forall t, p, q : \quad |C_p(t) - C_q(t)| \leq \pi$
- Achieved by *internal synchronization* acroos all processors within a system

Network Time Protocol (NTP)



- Estimation of offset for process p_1:

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Clock adjustments in NTP

- What should p do if $\theta > 0$?
  - Push its own clock forward to adjust
- What should p do if $\theta < 0$?
  - Time should not go backwards!
  - Spread slowdown over time interval
- NTP used between pairs of servers
  - Adjust the one that is more accurate, i.e. closer to the reference clock in tree-like overlay

Google True Time Service [1]

- Offers service in Google's server infrastructure with guaranteed bounds
- `TT.now()` yields time value in interval $[T_{lwb}, T_{upb}]$ where $T_{upb} - T_{lwb} < 6ms$
- Requires dedicated infrastructure
  - Time masters with GPS receivers or atomic clocks placed in data centers
  - Detect and eliminate faulty time masters
  - Knowledge about speed of messages across data centers
- Used for Spanner, a globally distributed database with timestamped transactions

Conclusion

- Physical clocks are very useful for measuring durations in a single processor
- Clock drift must be controlled and adjusted to allow for comparing timestamps based on different physical clocks
- Protocols for clock synchronisation
  - NTP
  - Google True Time Service

Logical clocks

Motivation

- Relative order of events ⇒ Causal dependencies and relations
- Two prominent approaches: Lamport clocks and vector clocks

Happens-before relation (revisited)

- Three types of events in each process:
    - Send events
    - Receive events
    - Local / internal events

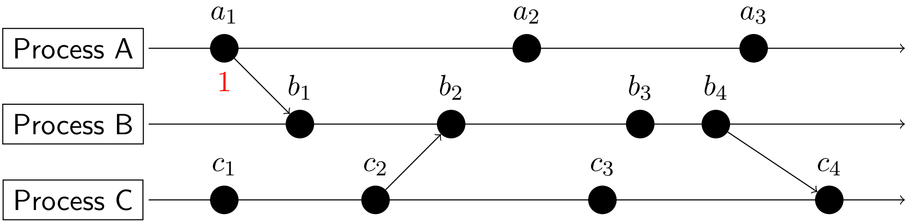
The happens-before relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3. If $a \rightarrow c$ and $c \rightarrow b$, then $a \rightarrow b$.

Lamport clocks

Idea: Associate time value $C(a)$ with event a such that

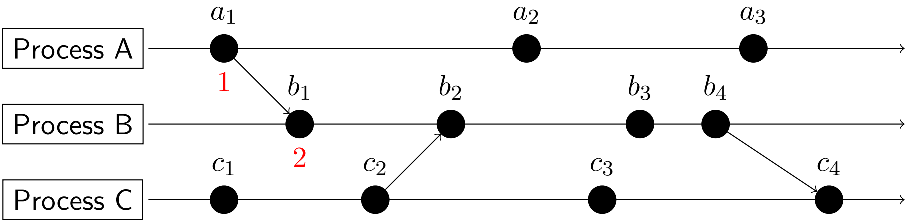
$$a \rightarrow b \quad \Rightarrow \quad C(a) < C(b)$$



Lamport clocks

Idea: Associate time value $C(a)$ with event a such that

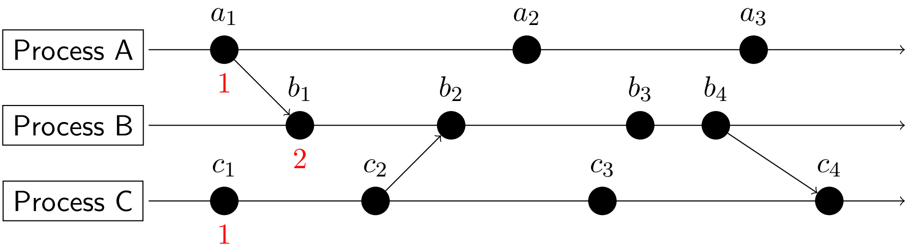
$$a \to b \quad \Rightarrow \quad C(a) < C(b)$$



Lamport clocks

Idea: Associate time value $C(a)$ with event a such that

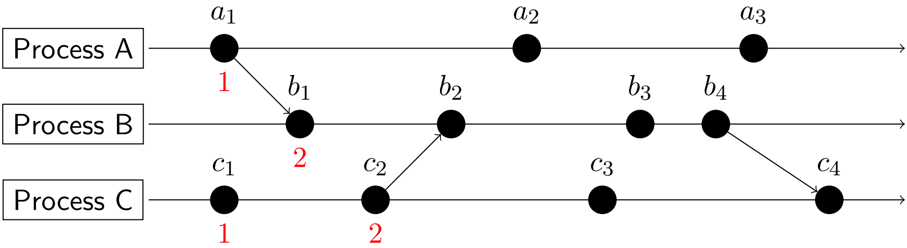
$$a \to b \quad \Rightarrow \quad C(a) < C(b)$$



Lamport clocks

Idea: Associate time value $C(a)$ with event a such that

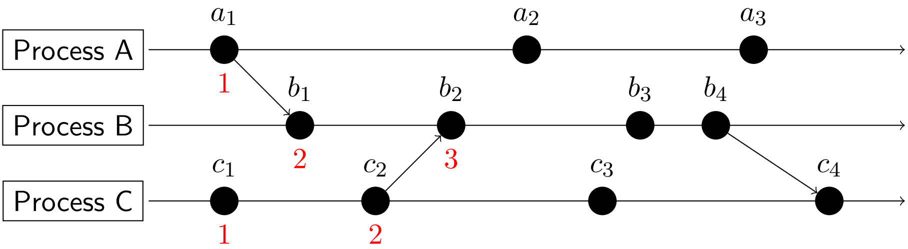
$$a \to b \quad \Rightarrow \quad C(a) < C(b)$$



Lamport clocks

Idea: Associate time value $C(a)$ with event a such that

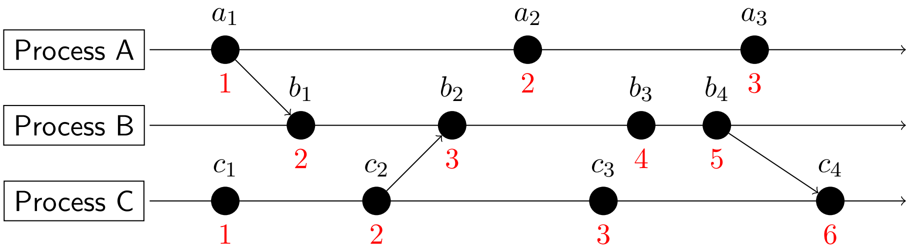
$$a \rightarrow b \quad \Rightarrow \quad C(a) < C(b)$$



Lamport clocks

Idea: Associate time value $C(a)$ with event a such that

$$a \rightarrow b \quad \Rightarrow \quad C(a) < C(b)$$



Lamport clocks[2]

- Each process p keeps an event counter l_p, initially 0.
- When an event that occurs at p that is not a receipt of a message, l_p is incremented by 1:

$$l_p := l_p + 1$$

- The value of l_p during the execution (after incrementing l_p) of event a is denoted by $C(a)$ (the timestamp of event a).
- When a process sends a message, it adds a timestamp to the message with value of l_p at time of sending.
- When a process p receives a message m with timestamp l_m, p increments its timestamp to

$$l_p := max(l_p, l_m) + 1$$

Properties of Lamport clocks

- Not unique, but can be made unique by pairing with process id
- We can show: $a \rightarrow b \quad \Rightarrow \quad C(a) < C(b)$
  - Proof by induction over different cases of $a \rightarrow b$

1. a occurs just before b in same process : $C(b) = l_p + 1 > l_p = C(a)$
2. a is the send event for receiving event b : $C(b) = max(l_p, l_m) + 1 > l_p = C(a)$
3. There exists event c such that $a \rightarrow c$ and $a \rightarrow b$. By induction hypothesis, $C(a) < C(c)$ and $C(c) < C(b)$, hence $C(a) < C(b)$

Properties of Lamport clocks

- Not unique, but can be made unique by pairing with process id
- We can show: $a \to b \quad \Rightarrow \quad C(a) < C(b)$
    - Proof by induction over different cases of $a \to b$

1. a occurs just before b in same process : $C(b) = l_p + 1 > l_p = C(a)$
2. a is the send event for receiving event b : $C(b) = max(l_p, l_m) + 1 > l_p = C(a)$
3. There exists event c such that $a \to c$ and $a \to b$. By induction hypothesis, $C(a) < C(c)$ and $C(c) < C(b)$, hence $C(a) < C(b)$

- But:

$$C(a) < C(b) \quad \not\Rightarrow \quad a \to b$$

(see exercise)

Causality

- Fundamental to many problems occurring in distributed computing
- The happens-before relation of events is often also called *causality relation* [4].
- *Examples*: determining a consistent recovery point, detecting race conditions, exploitation of parallelism

An event a may causally affect another event b if and only if $a \rightarrow b$.

- The happens-before order $\rightarrow$ indicates only *potential* causal relationship.
- Tracking whether an event indeed is a cause of another event is much more involved and requires more complex dependency analyses.

Causal Histories[3]

- Let E_p denote the set of events occurring at process p and E the set of all executed events:

$$E = \bigcup_{p \in P} E_p$$

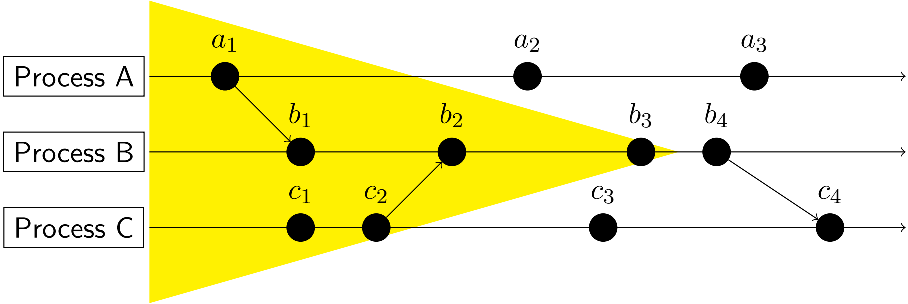
- The *causal history* of an event $b \in E$ is defined as

$$C(b) = \{a \in E \mid a \to b\} \cup \{b\}$$

- Note: Just a different representation of happens-before:

$$a \to b \quad \Leftrightarrow \quad a \neq b \wedge a \in C(b)$$

Example: Causal history of b_3



$$C(b_3) = \{a_1, b_1, b_2, b_3, c_1, c_2\}$$

Tracking causal histories with event sets

Each process p stores current causal history as set of events C_p.

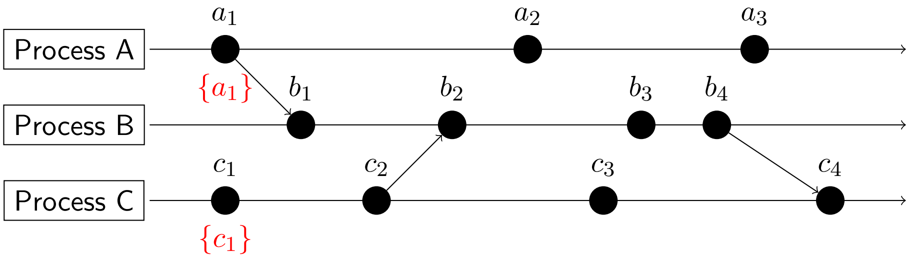
- Initially, $C_p := \emptyset$
- On each local event e at process p_i, the event is added to the set:

$$C_p := C_p \cup \{e\}$$

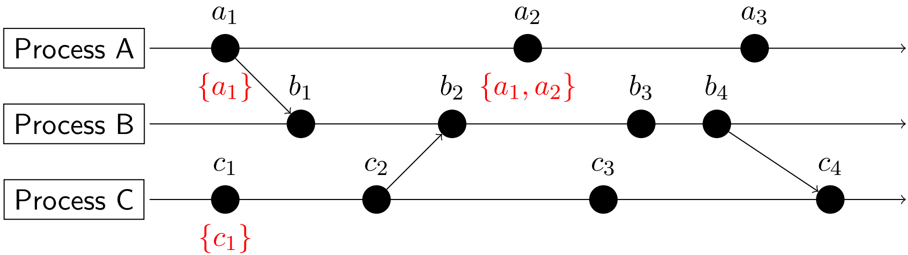
- On sending a message m, p updates C_p with a sending event e and attaches the updated C_p to m.
- On receiving message m with causal history $C(m)$, p updates with a receive event. Next, p adds the causal history from $C(m)$, yielding:

$$C_p := C_p \cup C(m)$$

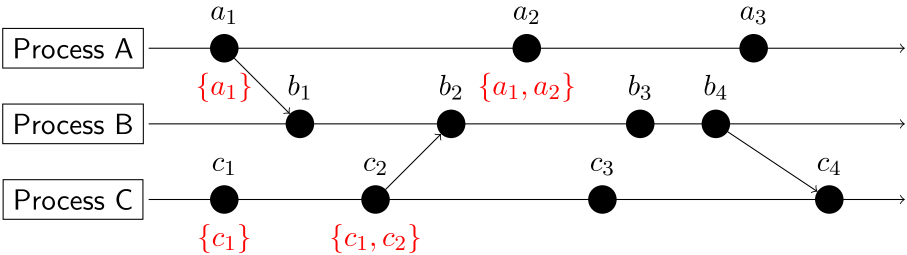
Example: Causal histories



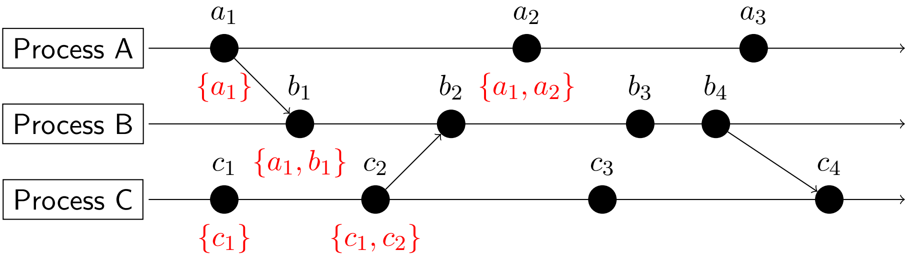
Example: Causal histories



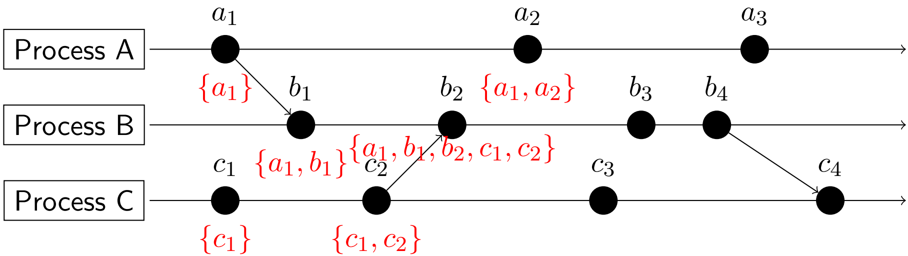
Example: Causal histories



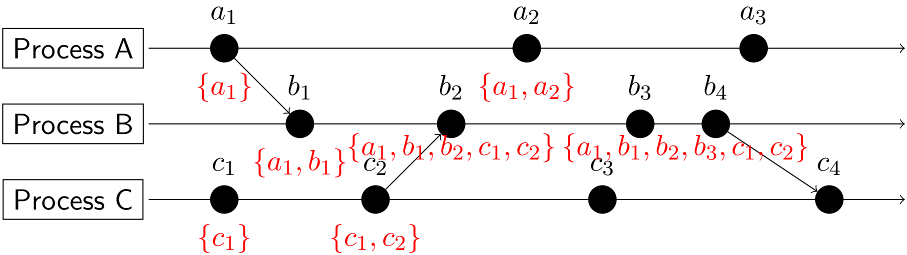
Example: Causal histories



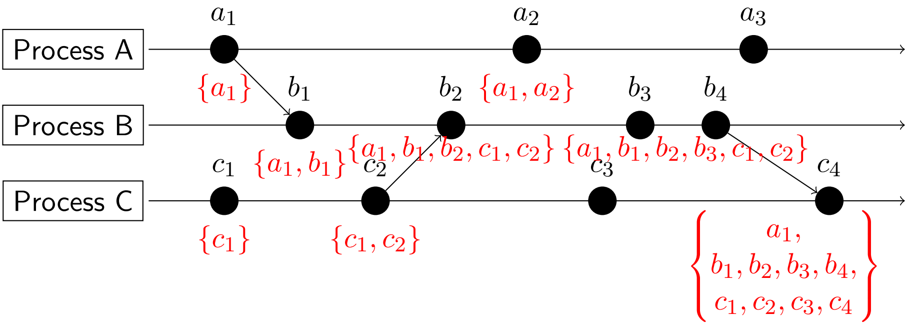
Example: Causal histories



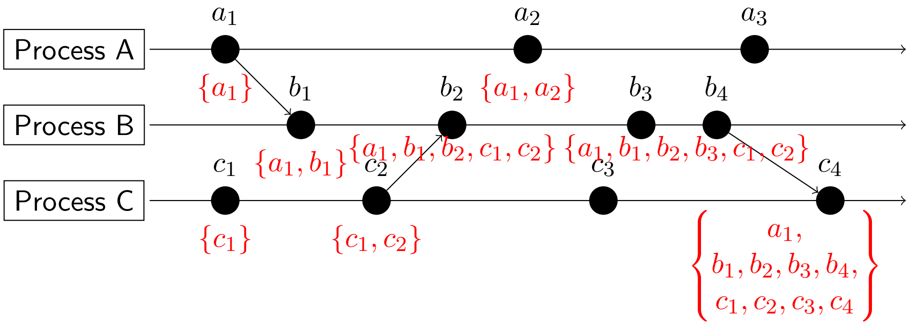
Example: Causal histories



Example: Causal histories

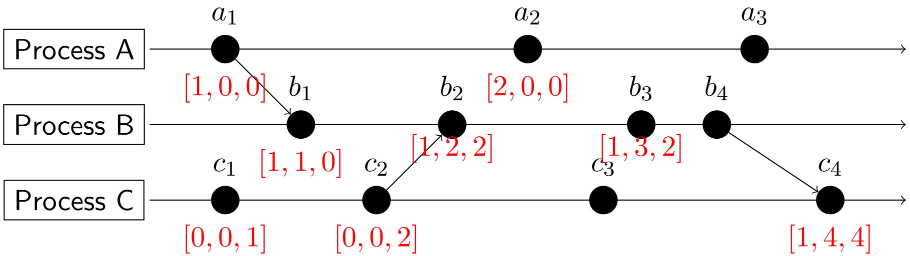


Example: Causal histories



Can we represent causal histories more efficiently?

Example: Efficient representation of causal histories



Efficient representation of causal histories

- Vector clock $V(e)$ as efficient representation of $C(e)$.
- Vector clock is a mapping from processes to natural numbers:
  - Example: $[p_1 \mapsto 3, p_2 \mapsto 4, p_3 \mapsto 1]$
  - If processes are numbered $1, \ldots, n$, this mapping can be represented as a vector, e.g. [3, 4, 1]
  - Intuitively: $p_1 \mapsto 3$ means "observed 3 events from process p_1''

Formal Construction

- Assume processes are numbered by $1, \ldots, n$
- Let $E_k = \{e_{k_1}, e_{k_2}, \ldots\}$ be the events of process k
  - Totally ordered: $e_{k_1} \rightarrow e_{k_2}, e_{k_2} \rightarrow e_{k_3}, \ldots$
- Let $C(e)[k] = C(e) \cap E_k$ denote the projection of $C(E)$ on process k.

$$C(e) = C(e)[1] \cup \cdots \cup C(e)[n]$$

- Now, if $e_{k_j} \in C(e)[k]$, then by definition it holds that $e_{k_1}, \ldots, e_{k_j} \in C(e)[k]$
- The set $C(e)[k]$ is thus sufficiently characterized by the largest index of its events, i.e. its cardinality!
- Summarize $C(e)$ by an n-dimensional vector $V(e)$ such that for $k = 1, \ldots, n$:

$$V(e)[k] = |C(e)[k]|$$

Note: Both representations are lattices

A lattice is a partially ordered set in which every two elements have a unique supremum and a unique infimum.

Operator	Causal history	Vector clock
$\bot$	$\emptyset$	$\lambda i.\, 0$
$A \leq B$	$A \subseteq B$	$\forall i.\ A[i] \leq B[i]$
$A \geq B$	$A \supseteq B$	$\forall i.\ A[i] \geq B[i]$
$A \sqcup B$	$A \cup B$	$\lambda i.\, max(A[i], B[i])$
$A \sqcap B$	$A \cap B$	$\lambda i.\, min(A[i], B[i])$

- $\bot$: bottom, or smallest element
- $A \sqcup B$: least upper bound, or join, or supremum
- $A \sqcap B$: greatest lower bound, or meet, or infimum

Tracking causal histories

Each process p_i stores current causal history as set of events C_i.

- Initially, $C_i := \emptyset$
- On each local event e at process p_i, the event is added to the set: $C_i := C_i \cup \{e\}$
- On sending a message m, p_i updates C_i as for a local event and attaches the new value of C_i to m.
- On receiving message m with causal history $C(m)$, p_i updates C_i as for a local event. Next, p_i adds the causal history from C(m):

$$C_i := C_i \cup C(m)$$

Tracking causal histories

Each process p_i stores current causal history as set of events C_i.

- Initially, $C_i := \bot$
- On each local event e at process p_i, the event is added to the set: $C_i := C_i \cup \{e\}$
- On sending a message m, p_i updates C_i as for a local event and attaches the new value of C_i to m.
- On receiving message m with causal history $C(m)$, p_i updates C_i as for a local event. Next, p_i adds the causal history from C(m):

$$C_i := C_i \sqcup C(m)$$

Vector time

Each process p_i stores current causal history as a vector clock V_i.

- Initially, $V_i[k] := \bot$
- On each local event, process p_i increments its own entry in V_i as follows: $V_i[i] := V_i[i] + 1$
- On sending a message m, p_i updates V_i as for a local event and attaches new value of V_i to m.
- On receiving message m with vector time $V(m)$, p_i increments its own entry as for a local event. Next, p_i updates its current V_i by joining $V(m)$ and V_i:

$$V_i := V_i[k] \sqcup V(m)$$

Relating vector times

Let u, v denote time vectors.

- $u \leq v$ iff $u[k] \leq u[k]$ for $k = 1, \ldots, n$
- $u < v$ iff $u \leq v$ and $u \neq v$
- $u \parallel v$ iff $u \nleq v$ and $v \nleq u$

For two events e and e', it holds that

$$e \to e' \quad \Leftrightarrow \quad V(e) < V(e')$$

- Proof: By construction.

Summary

- Causality important for many scenarios
- Vector clocks:
  - Efficient representation of causal histories / happens-before
  - How many events from which process?
- Causality not always sufficient

Further reading I

[1] James C. Corbett u. a. „Spanner: Google's Globally Distributed Database". In: *ACM Trans. Comput. Syst.* 31.3 (2013), 8:1–8:22. URL: https://dl.acm.org/citation.cfm?id=2491245.

[2] Leslie Lamport. „Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), S. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[3] Friedemann Mattern. „Virtual Time and Global States of Distributed Systems". In: *Parallel and Distributed Algorithms.* North-Holland, 1988, S. 215–226.

[4] Reinhard Schwarz und Friedemann Mattern. „Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail". In: *Distributed Computing* 7.3 (1994), S. 149–174. DOI: 10.1007/BF02277859. URL: https://doi.org/10.1007/BF02277859.

Further reading II

[5] Maarten van Steem und Andrew S. Tanenbaum. *Distributed Systems*. 2017. URL: distributed-systems.net.