Logical Time

For every minute spent in organizing, an hour is earned or a minute is lost.

Marco Aiello, Eirini Kaldeli

University of Groningen

Distributed Systems Course, 2009

Outline

1. **Capturing Causality**
   - Causal relation between events
   - Changing the order of events
2. **Assigning logical timestamps**
   - Lamport Timestamps
   - Vector Clocks
3. **Distributed snapshot**
   - Detecting a consistent global state
   - Cuts and consistent cuts
   - The Chandy-Lamport snapshot algorithm

Logical Time

No reference to global time:

- Local physical clocks cannot be perfectly synchronised.
- So, cannot appeal to physical time to order events in a total manner.
- However, what really interests us is an order that preserves causality, i.e. the relation between events that potentially influence each other.

➥ Assign logical timestamps to events, which are communicated through the standard message passing between the processors, and can be used to induce the causality relations between events.

Happens-Before relation

Definition

Event ϕ_i happens-before ϕ_j, denoted by $\phi_i \rightarrow \phi_j$ if either:

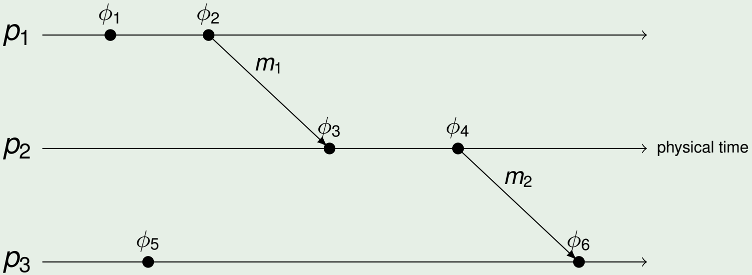
1. the two events occurred at the same process and ϕ_i precedes ϕ_j
2. ϕ_i is the event sending message m and ϕ_j is the event receiving m
3. there exists an event ϕ such that $\phi_i \rightarrow \phi$ and $\phi \rightarrow \phi_j$ (transitivity)

- The $\rightarrow$ relation is an irreflexive partial order.
- If $\phi_i \nrightarrow \phi_j$ and $\phi_j \nrightarrow \phi_i$, then ϕ_i and ϕ_j are *concurrent*: $\phi_i \| \phi_j$

Causal influence on a space-time diagram

Example

What is the happened-before relation between the events?



☞ What if we put ϕ_5 after ϕ_3 and before ϕ_6?

Causal Shuffle

Definition

Given a sequence of events $\sigma = \{\phi_1, ..., \phi_k\}$, a permutation π of σ is a causal shuffle of σ if:

1. the order of events occuring at individual processors remains unchanged, i.e. $\forall i, 1 \leq n, \sigma|_i = \pi|_i$, where $|_i$ refers to the events occuring in p_i.
2. if a message m is sent during p_i's event ϕ in σ, then in π, ϕ precedes the delivery of m.

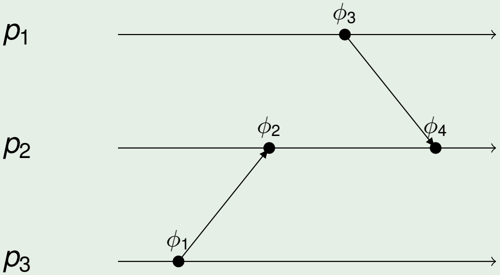
The resulting sequence π is indistinguishable to the processors.

Lemma

Any total ordering of the events in σ that is consistent with the $\rightarrow$ relation is a causal shuffle of σ.

Causal Shuffles of an example execution

Example

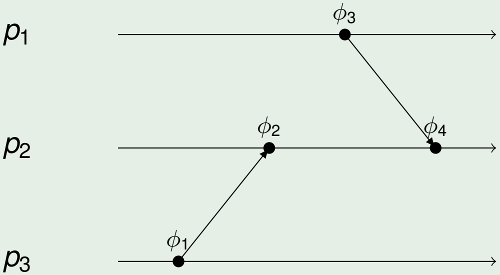


Which of the following permutations are causal shuffles?

- $\phi_1, \phi_3, \phi_4, \phi_2$

Causal Shuffles of an example execution

Example

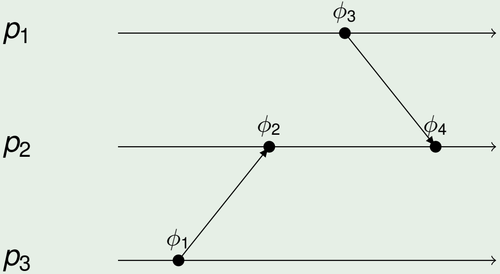


Which of the following permutations are causal shuffles?

- $\phi_1, \phi_3, \phi_4, \phi_2$ ✗

Causal Shuffles of an example execution

Example

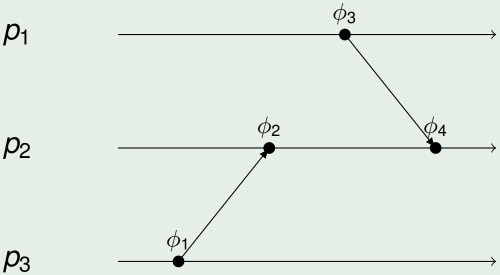


Which of the following permutations are causal shuffles?

- $\phi_1, \phi_3, \phi_4, \phi_2$ ✗
- $\phi_3, \phi_1, \phi_2, \phi_4$

Causal Shuffles of an example execution

Example



Which of the following permutations are causal shuffles?

- $\phi_1, \phi_3, \phi_4, \phi_2$ ✗
- $\phi_3, \phi_1, \phi_2, \phi_4$ ✓

Lamport Timestamps Definition

Want to mark events so that some information about causality is captured

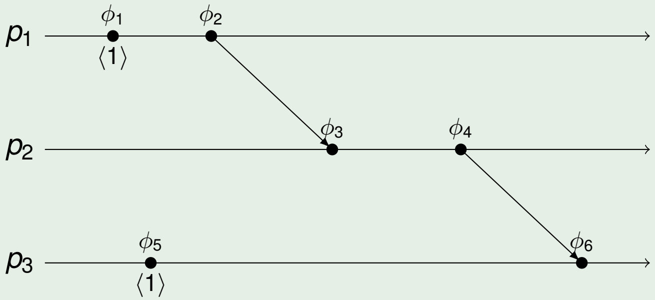
➥ Assign a Lamport Timestamp $LT(\phi)$ to each event ϕ:

- Each p_i keeps a local counter LT_i, which is initally set to 0.
- At each event ϕ in p_i,
  $LT_i = max\left\{LT_i, \max\{LT\langle msgs\ received\ upon\ \phi\rangle\}\right\} + 1$
- When p_i sends a message, it attaches the LT_i value to the message.

☞ For each p_i, LT_i is strictly increasing.

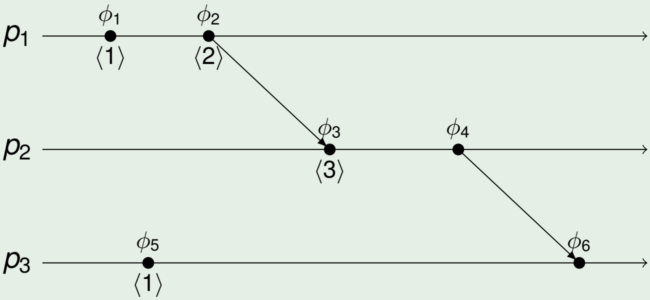
Lamport Timestamps for an example execution

Example



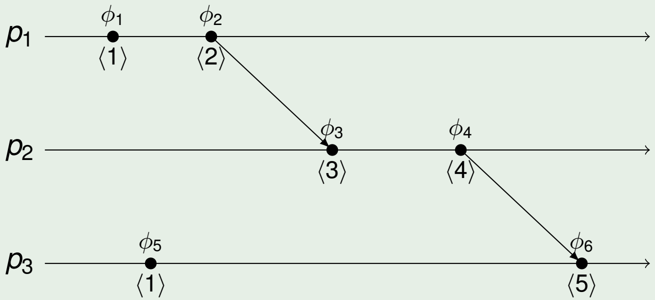
Lamport Timestamps for an example execution

Example



Lamport Timestamps for an example execution

Example



☞ Note that $LT(\phi_5) < LT(\phi_4)$ but $\neg(\phi_5 \rightarrow \phi_4)$

Lamport Timestamps and Happens-Before Relation

Theorem (Weak consistency)

Let ϕ_1, ϕ_2 be two events in an execution. If $\phi_1 \rightarrow \phi_2$ then $LT(\phi_1) < LT(\phi_2)$.

Drawback of Lamport Timestamps

- If $LT(\phi_1) < LT(\phi_2)$ we can only tell that $\neg(\phi_2 \rightarrow \phi_1)$, but we don't know whether $\phi_1 \rightarrow \phi_2$ or $\phi_1 \parallel \phi_2$.
- The problem is that $<$ induces a total order over integers while $\rightarrow$ a partial one, so the non-causality relation is lost.

Capturing concurrency as well: vector timestamps

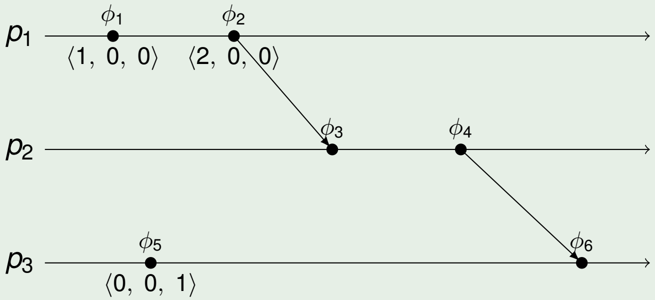
Choose logical timestamps from a non totally ordered domain
➥ vectors over integers

- Each p_i keeps a local vector of size n VC_i, whose entries $VC_i[j]$ are initially set to 0.
- At each event ϕ in p_i, $VC_i[i] = VC_i[i] + 1$ and for all $j \neq i$ $VC_i[j] =$ $\max\left\{VC_i[j],\ \max\{VC[j]\langle \textit{msgs received upon } \phi\rangle\}\right\}$
- VC_i is attached to every message sent by p_i.

☞ $VC_i[j]$ is an "estimate" maintained by p_i for $VC_j[j]$, i.e. the events having occurred in p_j so far.

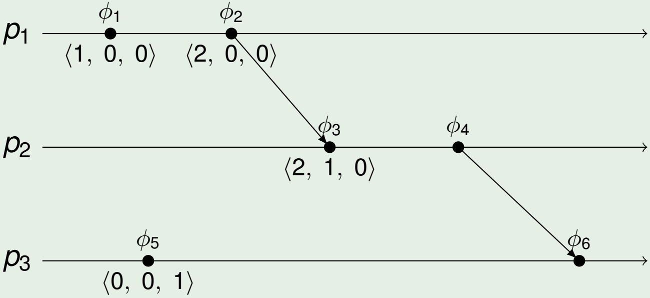
Vector Clocks in an example execution

Example



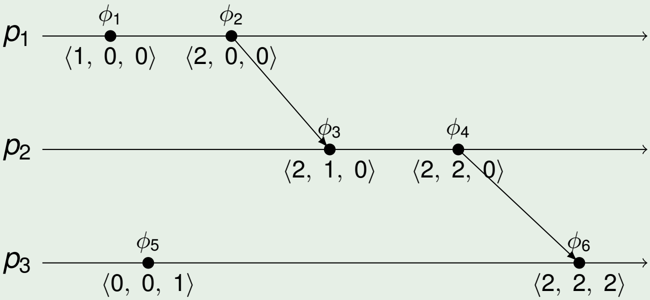
Vector Clocks in an example execution

Example



Vector Clocks in an example execution

Example



Vector clocks can indeed capture concurrency

Only p_i can increase $VC_i[i]$, so p_j's estimation about p_i's steps is less or equal than their actual number.

Proposition

For every p_i, $VC_i[j] \leq VC_j[j]$ for all i, j $1 \leq i, j \leq n$

- Vector clocks capture concurrency, i.e. it holds that $\phi_1 \| \phi_2$ iff $VC(\phi_1)$ and $VC(\phi_2)$ are incomparable.
- Recall that:
  - $V_1 \leq V_2$ iff for all $1 \leq i \leq n$ $V_1[i] \leq V_2[i]$.
  - $V_1 < V_2$ iff $V_1 \leq V_2 \wedge V_1 \neq V_2$. E.g. $(2,2,3) < (3,2,4)$
  - $V_1 \| V_2$ iff $\neg(V_1 \leq V_2) \wedge \neg(V_2 \leq V_1)$. E.g. $(3,2,4) \| (4,1,4)$

Theorem (Strong consistency)

$\phi_1 \rightarrow \phi_2$ iff $VC(\phi_1) < VC(\phi_2)$

Vector clocks strong consistency: proof

Proof.

$\Rightarrow$ If ϕ_1, ϕ_2 at same p_i trivial. If ϕ_1 at p_i sends message received by ϕ_2 at p_j, then $VC_j(\phi_2)[k] \geq VC_i(\phi_1)[k]$ for all $k \neq j$ and $VC_j(\phi_2)[j] = VC_j(\phi_1)[j] + 1$. Rest by transitivity of the $<$ relation of vectors.

$\Leftarrow$ If $\phi_2 \rightarrow \phi_1$, contradiction. If $\phi_1 \| \phi_2$ then $VC_j[i](\phi_2) < VC_i[i](\phi_1)$ since the only way that $VC_j[i](\phi_2) = VC_i[i](\phi_1)$ would be the existence of a sequence of events ϕ_i' s.t. $\phi_1 \rightarrow \phi_1' \ldots \phi_n' \rightarrow \phi_2$. Similarly $VC_i[j] < VC_j[j]$. Thus, $VC_i(\phi_1)$, $VC_j[\phi_2]$ would be incomparable.

$\square$

Vector Clock Size Lower Bound

Size of vector timestamps *n* is big, can we do better?

Theorem (Lower bound on the size of vector clocks)

If VC is a function that maps each event in an execution in a system of n processors to a vector in S^k, where S is any totally ordered set (e.g. $\Re$), in a manner that captures concurrency, then $k \geq n$.

☞ There are techniques for compressing the required data for maintaining vector clocks, however at the expense of additional processing required to reconstruct the complete vectors.

Recording a meaningful global state

- No omniscient observer to record the system's global state, i.e. the set of the processors' local states, as well as the state of each channel in which messages flow.
- Snapshot problem: compute a meaningful global state so that it looks to the processors as if the snapshot was taken at the same instance everywhere in the system.
- Processors have to compute an approximate snapshot of the global state that captures the notion of causality (every message that is recorded as received is also recorded as sent).
- How to find a global snapshot when processes cannot record their local states at precisely the same instant?

Some applications that need a snapshot record

- System recovery: global states (checkpoints) are saved periodically, so that the system can be restore to the last global state in case of a failure.
- Detection of stable properties, i.e. properties that once they become true at some state G, they stay true in every state H reachable from G. Deadlock, termination, loss of a token are some examples.
- Compute a global state G, if property A is true in G then done, otherwise repeat computation after some delay.
- Once A is found true in some past state, then it's also true in the current state.

Cuts

- A way to visualise global states on a space-time diagram, is to draw *cuts*.
- Slice the space-time digram vertically into past events (left side) and future events (right side).

Definition (Cut)

A cut of an execution is an n-vector $\vec{k} = \langle k_1, \ldots, k_n \rangle$ of positive integers, where k_i indicates the number of events taken by p_i.

- Given $\vec{k}$ one can construct the global state $S^k = (s_1, \ldots, s_n, c_1, \ldots, c_m)$, where s_i is the state of p_i immediately after its k_ith event, and c_i is the state of channel c_i immediately after the occurence of the events induced by $\vec{k}$.

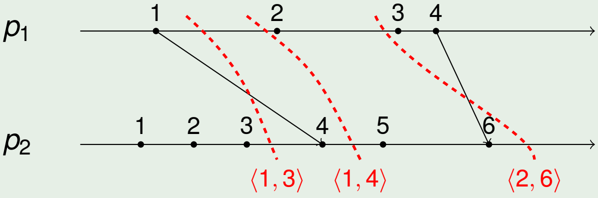
Consistent cuts

Definition (Consistent Cut)

A cut $\vec{k}$ is consistent if for all $1 \leq i, j \leq n$ the $(k_i + 1)$st event on p_i doesn't happen-before the k_jth event on p_j. I.e. for each event included in a concistent cut, all events that happened-before this event must also be included in it. A global state corresponding to a consistent cut is consistent.

Example

Some consistent and inconsistent cuts



Distributed snapshot algorithm: assumptions

- Look for an algorithm that can be initiated by one or more p_is that want to compute a consistent global snapshot without adding overhead to the normal execution.

Assumptions

1. No failures: all messages arrive intact and only once.
2. Communication channels are unidirectional and deliver messages in FIFO order (guarantees that the computed global state is consistent).
3. There is a path between any two processors, i.e. the graph of processors and communication channels is strongly connected (guarantees termination).
4. The snapshot algorithm doesn't interfere with the normal execution of the processes.

The Chandy-Lamport algorithm

(i) Each p_i that wants to initiate a snapshot records its local state s_i, sends a special marker message $\langle M \rangle$ to all outgoing channels and starts recording messages arriving over its incoming channels.

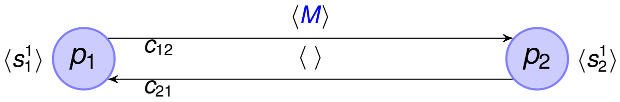
(ii) When a p_i receives a $\langle M \rangle$ over channel c and has not yet recorded its state, it:

- **a.** records its local state and the state of c as empty
- **b.** sends a $\langle M \rangle$ to all outgoing channels
- **c.** starts recording messages arriving over the other incoming channels

(iii) When a p_i receives an $\langle M \rangle$ over c and has already saved its state, it records the state of c as the set of messages recorded over c (channel states account for msgs that arrived after the receiver recorded its state and were sent before the sender recorded its own state)

The Chandy-Lamport algorithm in an example execution

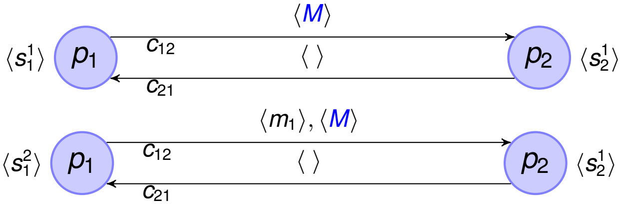
S_1
(p_1 records s_1^1,
then emits M)



The Chandy-Lamport algorithm in an example execution

S_1
(p_1 records s_1^1, then emits M)

S_2
(p_1 sends m_1)

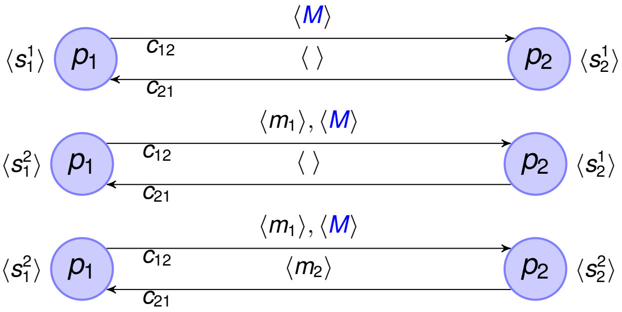


The Chandy-Lamport algorithm in an example execution

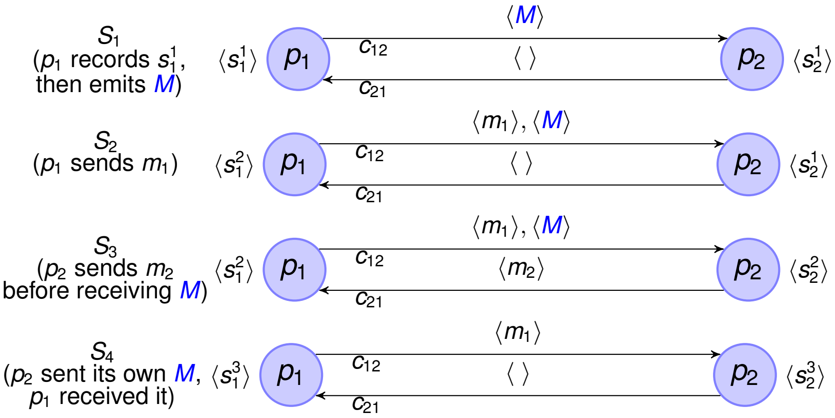
S_1
(p_1 records s_1^1, then emits M)

S_2
(p_1 sends m_1)

S_3
(p_2 sends m_2 before receiving M)



The Chandy-Lamport algorithm in an example execution



$$S^* : (s_1 = s_1^1, s_2 = s_2^2, c_{12} = \langle\,\rangle, c_{21} = \langle m_2 \rangle) \notin \{S_1, S_2, S_3, S_4\}$$

Chandy-Lamport algorithm: reachability of the recorded state

- The delivered global state S^* may differ from all actual global states through which the system passed.
- However, the system could have passed through S^* in some equivalent executions.

Theorem

Let S_i be the global state immediately before the first process recorded its state, and S_f the global state immediately after the last state-recording action. Let seq be the sequence of events that takes the system from S_i to S_f. Then there exists a sequence seq′ that is a causal shuffle of seq such that the recorded global state S^* is reachable from S_i and S_f is reachable from S_i.

Chandy-Lamport algorithm: stability properties

- If a stable property p is true in S^*, we can conclude that it is true in S_f (the converse doesn't hold).
- If a stable property is false in S^* then we can conclude it is false in S_i (the converse doesn't hold).
- E.g. to detect deadlocks, take a snapshot, then determine if there is a deadlock in the returned S^* (by performing cycle-detection e.g. through bridth-first search). If the snapshot is executed repeatedly, it is guaranteed to eventually detect a deadlock that occurs.

Chandy-Lamport algorithm: correctness

Theorem

The distributed snapshot algorithm delivers a consistent global state.

- Each p_i eventually records its local state: because of the connectivity of the graph, all p_is eventually receive a marker message.
- We will now prove that the computed global state satisfies the following 2 conditions:
  - C_1**:** Every message m_{ij} recorded as sent in the local state of p_i must be captured either in the state of channel c_{ij} it was sent over, or in the collected local state of the receiver p_j (conservation of messages).
  - $C2$**:** If an m_{ij} is not recorded as sent in the local state of p_i, then it must neither be present in the state of c_{ij}, nor in the collected local state of the receiver p_j (for every effect, its cause must be present).

Chandy-Lamport algorithm: correctness

- Proof of C_1: If a p_j receives a m_{ij} that precedes the marker $\langle M \rangle$ on channel c_{ij}, then if p_j has not taken a snapshot yet it includes m_{ij} in its recorded local state, otherwise it reports m_{ij} in the state of channel c_{ij}.
- Proof of C_2: If a m_{ij} is not included in the local state recorded by p_i, then it was sent after p_i had sent $\langle M \rangle$ over c_{ij}. Because channels are FIFO, p_j will receive $\langle M \rangle$ before m_{ij}, and thus it will report its local state before receiving m_{ij} and c_{ij}'s state as empty if this is the first marker it receives, or it will just stop recording c_{ij} again before receiving m_{ij}.

Chandy-Lamport algorithm: complexity

- Message complexity: $O(l)$, where l the number of links, plus the messages sent by the normal execution of the p_is.
- Time complexity: $O(d)$ where d is the diameter of the network.