University of Freiburg, Germany
Department of Computer Science

Distributed Systems

Chapter 3 Time and Global States

Christian Schindelhauer

12. May 2014

2.2: Logical Time

Why?

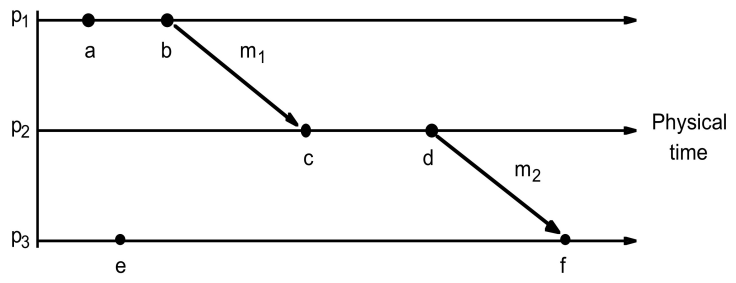
- Getting physical clocks absolutely synchronized is not possible.
- Thus it is not always possible to determine the order of two events.
- For such cases logical time can be used as a solution.
    - If two events happen in the same process they are ordered as observed.
    - If two processes interchange messages, then the sending event is always considered to be before the receiving event.

Lamport's happened-before relation (causal ordering)

- If two events a, b happen in the same process p_i they are ordered as observed and we write $a \longrightarrow_i b$.
  Moreover, this implies $a \longrightarrow b$ systemwide.
- If two processes interchange messages, then the sending event a is always considered to be before the receiving event b, thus $a \longrightarrow b$.
- Whenever $a \longrightarrow b$ and $b \longrightarrow c$, then also $a \longrightarrow c$.

Events not being ordered by $\longrightarrow$ are called concurrent.

Example



from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

We conclude $a \to b, b \to c, c \to d, d \to f, a \to f$, however not $a \to e$; a, e are concurrent.

Algorithm of Leslie Lamport

- Let $L_i(e)$ denote the time stamp of event e at process P_i.
- When a new event a occurs in process P_i:

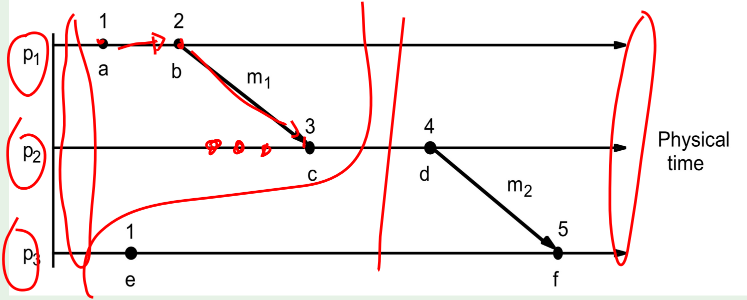
$$L_i := L_i + 1$$

- Each message m sent from P_i to P_j is piggybacked by the timestamp $L_i(a)$ of the send-event a.
- When (m, t_a) is received by P_j, P_j adjusts its logical clock L_j to the logical clock of P_j.

$$L_j := \max\{L_j, t_a\}$$

and increments L_j for the received message event.

Three clocks with application of Lamport's algorithm.



from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

Totally ordered logical clocks

- Extend the Lamport clock for each process P_i:
- Clock values must be systemwide unique
  - for this the clock value L_i is referred to with the process id i, i.e. (L_i, i)
  - all distinct clocks L_i can be unified into a system clock L.
- Define the total ordering

$$(T_i, i) < (T_j, j) \quad :\Longleftrightarrow \quad \begin{cases} i < j & \text{if } T_i = T_j \\ T_i < T_j & \text{else} \end{cases}$$

- So, we translate a partial ordering into a total ordering
- However from the total ordering $L(a) < L(b)$ one cannot conclude $a \rightarrow b$.

3rd

↓

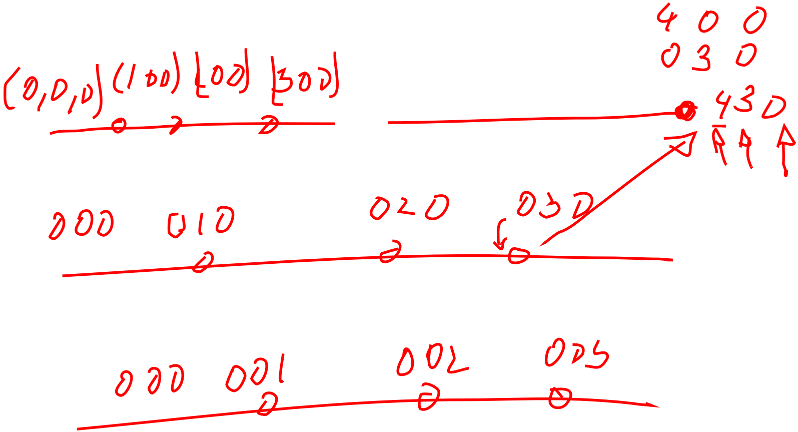
Mattern's Vector Clocks

- Vector clock for a system of n processes: array of n integers. (0,0,0,0)
- Each process P_i keeps its own vector clock V_i which is used to timestamp local events.
- Processes piggyback their own vector clock on messages they send.
- Update rules for vector clocks:

VC1: Initially, $V_i[j] := 0$ for $i, j \in \{1, \ldots, n\}$
VC2: P_i timestamps prior to each event: $V_i[i] := V_i[i] + 1$.
VC3: P_i sends the value $t = V_i$ with each message.
VC4: When P_i receives some message piggybacked with timestamp t, it sets

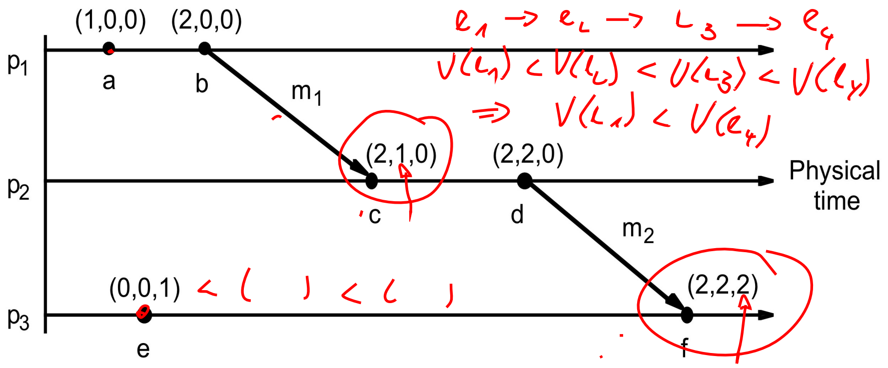
$$V_i[j] := max\{V_i[j], t[j]\} \quad \text{for } i = 1, 2, \ldots, n$$

- $V_i[i]$ is the number of events that P_i has timestamped.
- $V_i[j]$ for $i \neq j$ is the number of events that have occured at P_j to the knowledge of P_i.


(0,0,0)
(1 0 0)
(2 0 0)
(3 0 0)
4 0 0
0 3 0
4 3 0
0 0 0
0 1 0
0 2 0
0 3 0
0 0 0
0 0 1
0 0 2
0 0 3

Vector Clock Example

$c \parallel b$



$e_1 \to e_2 \to e_3 \to e_4$

$V(e_1) < V(e_2) < V(e_3) < V(e_4)$

$\Rightarrow V(e_1) < V(e_4)$

$< (\quad) < (\quad)$

from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

$(1\ 2\ 3)$

$(1\ 5\ 3)$

Comparing vector timestamps

- The clock vectors define a partial ordering
  - $V = V'$ *iff* $V[j] = V'[j]$ for all $j \in \{1, \ldots, n\}$
  - $V \leq V'$ *iff* $V[j] \leq V'[j]$ for all $j \in \{1, \ldots, n\}$
  - $V < V'$ *iff* $V \leq V' \wedge V \neq V'$.
- If for events a, b neither $V(a) \leq V(b)$ nor $V(a) \geq V(b)$ the events are called concurrent, i.e. $a || e$

$\begin{pmatrix}0\\1\end{pmatrix} \,||\, \begin{pmatrix}1\\0\end{pmatrix}$

Comparing vector timestamps

$V(a)$	$V(b)$	Relation	
$(2,1,0)$	$(2,1,0)$	$V(a) = V(b)$	all entries are the same
$(1,2,3)$	$(2,3,4)$	$V(a) < V(b)$	all entries of V are prior to V'
$(1,2,3)$	$(3,2,1)$	$a \,||\, b$	two events are concurrent

Lamport Relationship and Vector Clocks

Theorem

For any two events e_j, e_i:

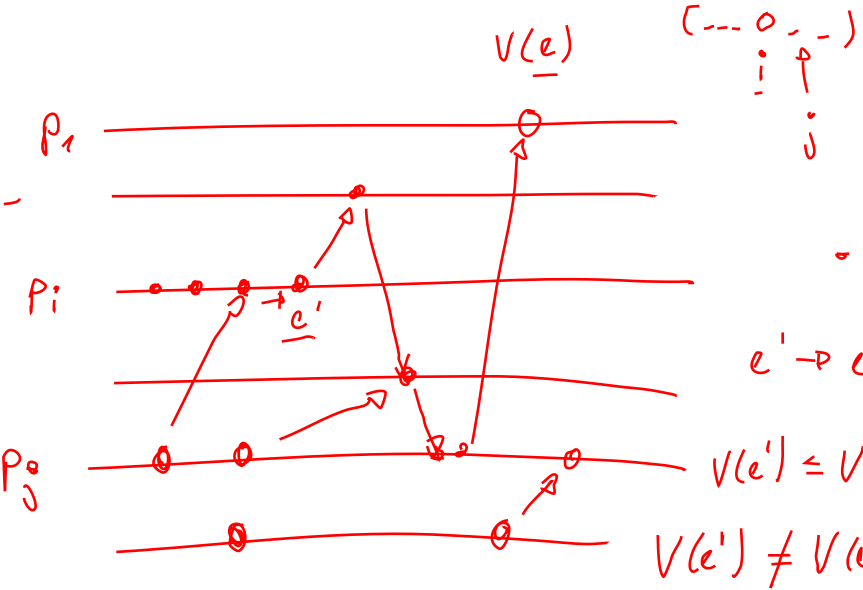
$$\Longrightarrow$$

$$\underline{e_j \to e_i} \iff \underline{V(e_j) < V(e_i)} \ .$$

$$\Longleftarrow$$

Proof sketch

- $\underline{e_j} \to e_i \implies V_j < V_i$.
  - If the events occur on the same process then $V_j < V_i$ follow directly. ✓
  - $e_j \to e_i$ implies a message is sent after e_j to the process with event e_i or two succeeding events of a process ✓
  - Since each entry of the receiving process is updated to at least the maximum of the entries of the sending processes, $V_j < V_i$
- $\underline{e_j} \to e_i \impliedby \underline{V_j <} V_i$.
  - If both events occur on the same process, $e_j \to e_j$ follows straightforward. ✓
  - An increase of the $\underline{i}$-th row can only be caused by a message path sent from the process of e_j to e_i
- complete proof is left as an exercise



$e' \to e$

$V(e') \leq V(e)$

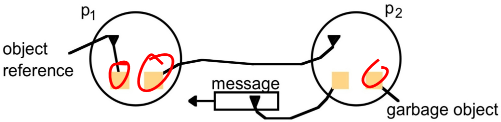
$V(e') \neq V(e)$

2.3. Global System States

Distributed Garbage Collection

- Non-referenced objects need to be erased
- p_2 has an object referenced in a message to p_1
- p_1 has an object referenced by p_2
- Neither one can be erased

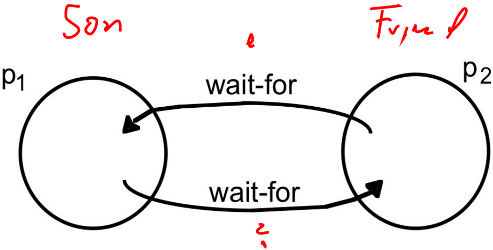
- How to determine a global state in the absence global time



2.3. Global System States

Distributed Deadlock Detection

- occurs when processes wait for each other to send a message
- and the processes form a cycle

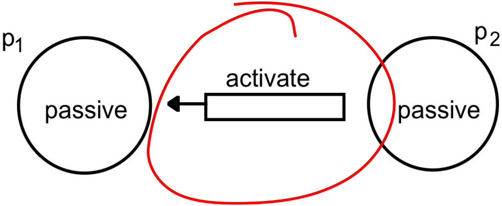


from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

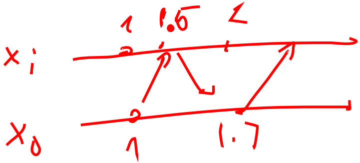
2.3. Global System States

Distributed Termination Detection

- How to detect that a distributed algorithm has terminated
- Assume p_1 and p_2 request values from the other
- If they wait for a value they are passive, otherwise active
- Assume both processes are passive. Can we conclude the system has terminated?
- No, since there might be an activating message on its way



2.3. Global System States



Distributed Debugging

- 0 Distributed systems are difficult to debug
- e.g. consider a program where each process has a changing variable x_i
- All variables are required to be in range $|x_i - x_j| \leq 1$.
- How to be sure that this will never be violated?

Cuts

- Consider system $\mathcal{P}$ of n processes p_i for $i = 1, \ldots, n$.
- The execution of a process is characterized by its history (of events e_i^t)

$$history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

- We denote a finite prefix

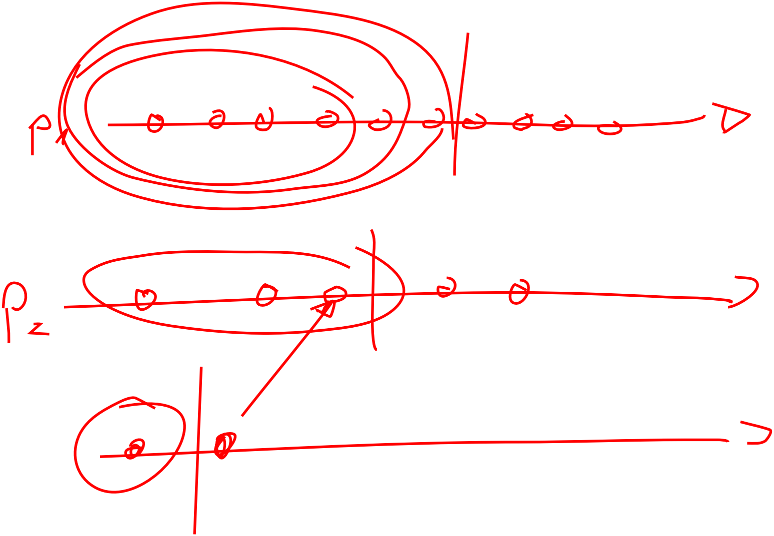
$$h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$$

- An event is either
    - an internal action or ✓
    - sending a message or ✓
    - receiving a message ✓
- Let s_i^k denote the state of process p_i immediately before event e_i^k.
- The global history H is

$$H = h_1 \cup h_2 \cup \ldots \cup h_n$$

- A *cut* C of the system's execution is a set of prefaces

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

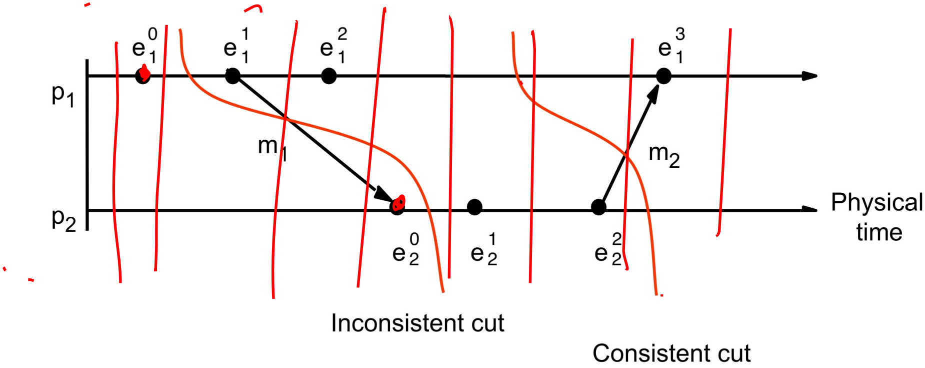

P1
P2

Consistent Cuts

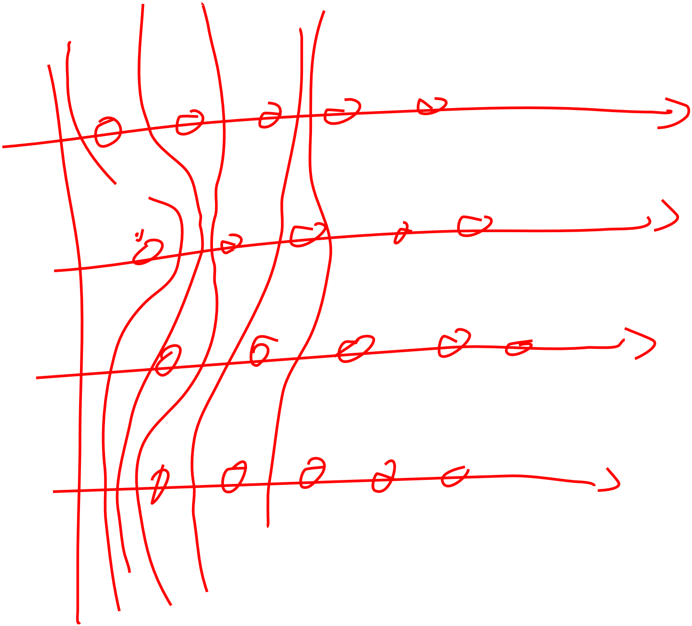
- A cut C is consistent if,

$$\text{For all events } e \in C: \quad \underline{f \to e} \quad \Longrightarrow \quad f \in C \quad .$$

- i.e. for each event it also contains all the events that happened-before the event.



from Distributed Systems – Concepts and Design, Coulouris, Dollimore, Kindberg

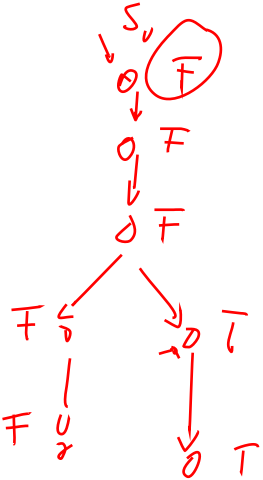


Global States

- A *consistent global state* corresponds to a consistent cut.
- A *run* is a total ordering of all events in a global history that is consistent with each local history's ordering ($\rightarrow_i$, for $i = 1, \ldots, n$).
- A *consistent run (linearization)* is an ordering of the events in the global history that is consistent with the happened-before-relation ($\rightarrow$) on H.
- Consistent runs pass only through consistent global states.

Global State Predicates, Stability, Safety and Liveness

- A *global state predicate* is a function that maps from the set of global states to $\{\texttt{true}, \texttt{false}\}$.
- *Stability* of a global state predicate: A global state predicate is *stable* if once it has reached `true` it remains in this state for all states reachable from this state.
- *Safety* is the assertion that an undesired state predicate evaluates to `false` to all states S reachable from the starting state S_0.
- *Liveness* is the assertion that a desired state predicate evaluates to `true` to all states S reachable from the starting state S_0.



Stable ? ✓

Safety ? no

liveness ? no

How to detect and record a global state

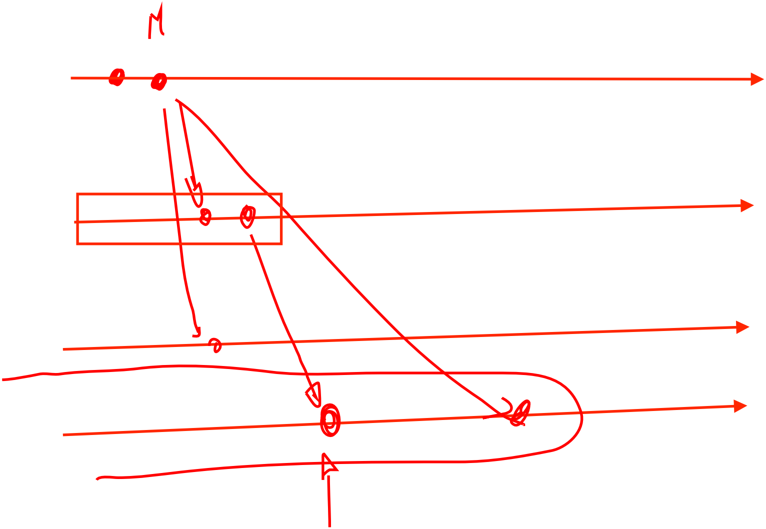
'Snapshot' algorithm of Chandy and Lamport

- Goal
  - record a set of events corresponding to a global state (consistent cut)
  - in a living system during run-time
  - without extra process
- Requirements
  - channels, processes do not fail. Communication is reliable
  - channels are uni-directional and have FIFO message delivery
  - graph of processes and channels is strongly connected
  - any process may initiate a snapshot
  - processes continue their execution (including messages)
- Notations
  - p_i's incoming channel: where all messages for p_i arrive
  - p_i's outgoing channel: where p_i sends all messages to other processes
  - Marker message: a special message distinct from every other message

Distributed Snapshot of Chandy and Lamport

```
Marker receiving rule for process $p_i$
  On $p_i$'s receipt of a marker message over channel $c$:
    if ($p_i$ has not yet recorded its state) it
        records its process state now;
        records the state of $c$ as the empty set;
        turns on recording of messages arriving over other incoming channels;
    else
        $p_i$ records the state of $c$ as the set of messages it has received over $c$
        since it saved its state.
    end if
Marker sending rule for process $p_i$
  After $p_i$ has recorded its state, for each outgoing channel $c$:
    $p_i$ sends one marker message over $c$
    (before it sends any other message over $c$).
```

from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

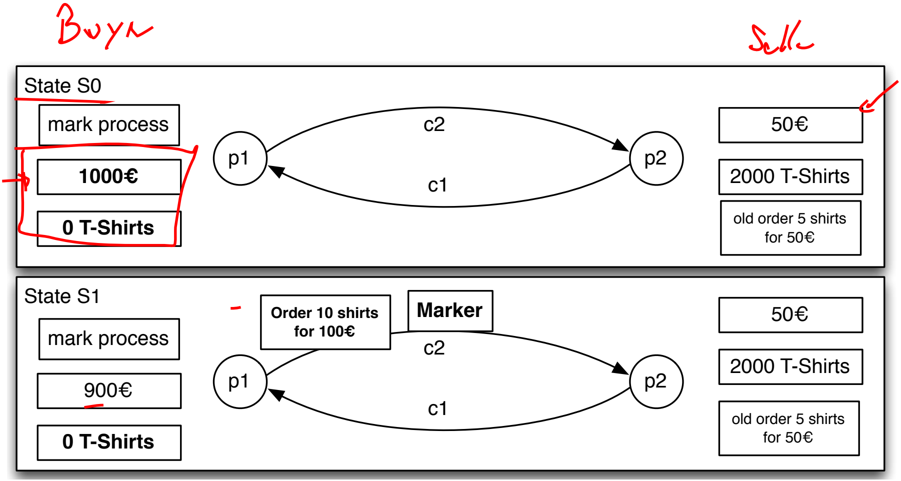

M

General remarks

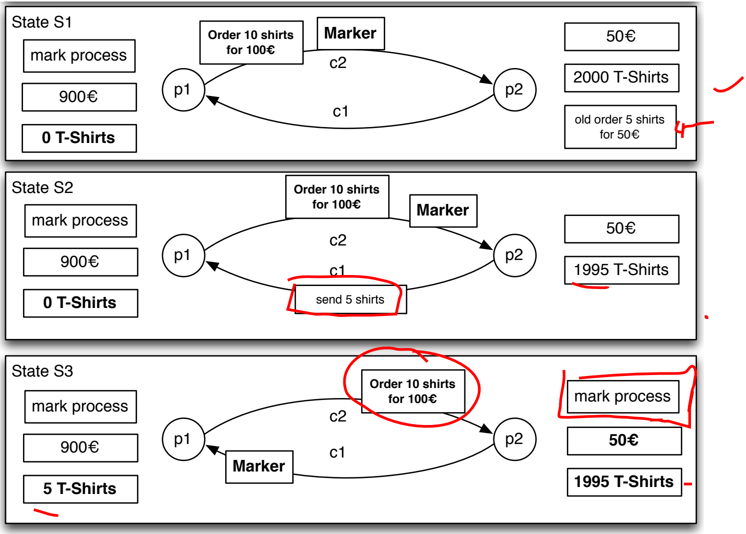
A snapshot consists of the state of a process and states of all incoming channels.

- Starting a snapshot:
  - Any process *P* can start a snapshot.
    1. Create a local snapshot of *P*'s state.
    2. Send marker message over all channels.
  - Upon receipt of a marker message, other processes participate in the snapshot.
- Collecting the snapshot:
  - Every process has created a local snapshot.
  - The local snapshot can be sent to a collector process.
- Terminating a snapshot:
  - If marker message has been received on all channels, then the snapshot terminates
  - Then the snapshot can be sent to a collector process.

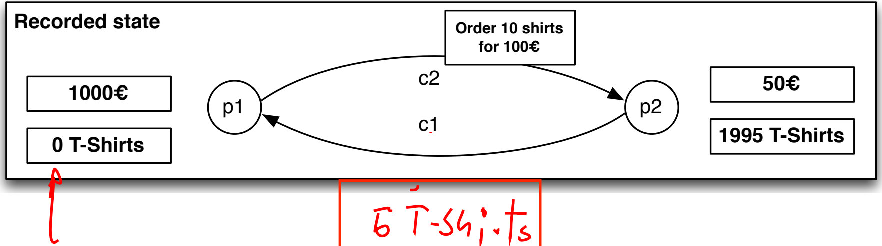
Distributed Snapshot of Chandy and Lamport



Distributed Snapshot of Chandy and Lamport



Distributed Snapshot of Chandy and Lamport

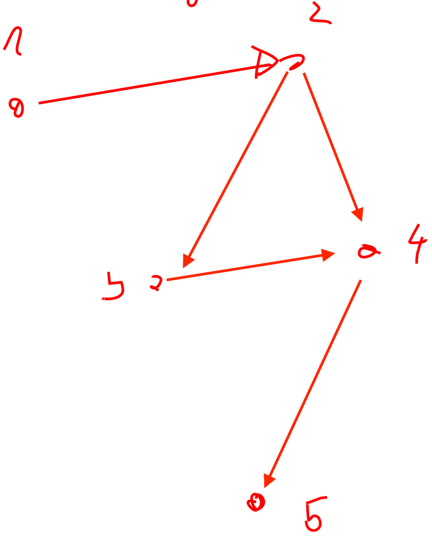


5 T-Shirts

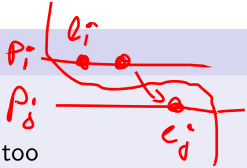
Termination of the snapshot algorithm

- If marker message has been received on all channels, then the snapshot terminates
- If the communication graph induced by the messages is strongly connected
- then the marker eventually reaches all nodes
- ⇒ only a finite number of messages need to be recorded

Strongly connected

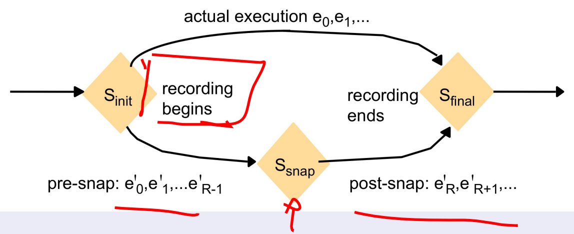


The snapshot algorithm selects a Consistent Cut



- Consider two events $e_i \rightarrow e_j$ on processes p_i and p_j
- If e_j is in the cut of the snapshot, then e_i should be, too
- If e_j occurred before p_j taking its snapshot, then e_i should have occurred before p_i has taking its snapshot
- If $p_i = p_j$ this is obvious.
- Now we consider $p_i \neq p_j$ and assume (*) that e_i is not in the cut and e_j is within the cut.
- Consider messages $m_1, m_2, \ldots m_h$ causing the *happened-before* relationship $e_i \rightarrow e_j$.
- So, m_1 must have sent after the snapshot, and m_2, and so forth. Each of this messages must have been sent after the marker message occurred on each channel (because of FIFO rules on the channel).
- Then, e_j cannot be in the cut. This contradicts (*) and proofs the claim.

Reachability of the snapshot algorithm selects a Consistent Cut



from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

- A snapshot characterizes events into two types
  1. pre-snap: An event happening before marking the corresponding process
  2. post-snap: An event happening after marking
- Note that pre-snap events can take place after post-snap events
- It is impossible that $e_i \rightarrow e_j$ if e_i is a post-snap event and e_j is a pre-snap event

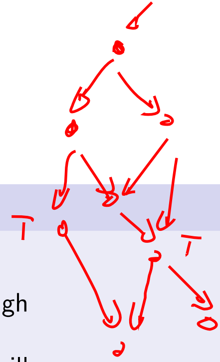
Distributed Debugging

Goal of algorithm of Marzullo and Neiger

- Testing properties post-hoc, e.g. safety conditions
- Capture traces rather than snapshots
- Gathered by a monitoring process (outside the system)
- How are process states collected
- How to extract consistent global states
- How to evaluate safety, stability and liveness conditions

Distributed Debugging

Temporal Logic



Temporal operators

Consider all linearizations of H

possible ϕ	There exists a consistent global state S through a linearization such that $\phi(S)$ is true.
definitely ϕ	For all linearizations a consistent global state will be passed such that $\phi(S)$ is true.

Relationship of Definitely and Possibly

✓ 1. $\forall S \in H : \phi(S) \implies \textit{definitely } \phi$

✓ 2. $\forall S \in H : \phi(S) \implies \textit{possible } \phi$

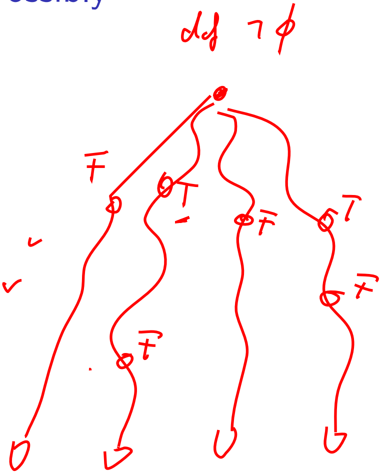
3. $\forall S \in H : \neg\phi(S) \implies \neg\textit{definitely } \phi$ ✓

4. $\forall S \in H : \neg\phi(S) \implies \neg\textit{possibly } \phi$ ✓

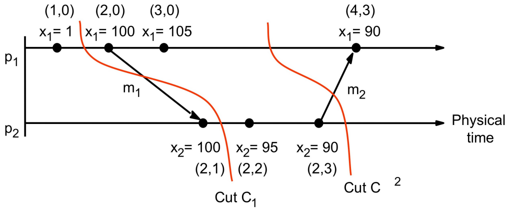
5. $\textit{definitely } \phi \implies \textit{possibly } \phi$

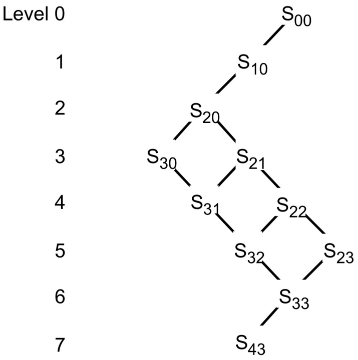
6. $\neg\textit{possibly } \phi \implies \textit{definitely } \neg\phi$

7. $\textit{definitely } \neg\phi \not\Longrightarrow \neg\textit{possibly } \phi$



Distributed Debugging: Definitely $|x_1 - x_2| \leq 50$





Sij = global state after *i* events at process 1 and *j* events at process 2