University of Freiburg, Germany
Department of Computer Science

Distributed Systems

Chapter 3 Time and Global States

Christian Schindelhauer

12. May 2014

2: Time and Global States

How can distributed processes be coordinated and synchronized, e.g.

- when accessing shared resources,
- when determining the order of triggered events?

The importance of time

- Distributed systems do not have only one clock.
- Clocks on different machines are likely to differ.
- Physical versus logical time.

2.1: Physical Time

Example; distributed software development using UNIX `make`

- Computer sets its clock back after compiling a source file
- User edits the source file
- `make` assumes the source file has not been changed since compilation
- `make` will not recompile

TAI and UTC

- International Atomic Time TAI: mean number of ticks of caesium 133 clocks since midnight Jan. 1, 1958 divided by number of ticks per second 9,192,631,770.
- Problem: 86,400 TAI seconds (corresponding to a day) are today 3 msec less than a mean solar day (because solar days are getting longer because of tidal forces).
- Solution: whenever discrepancy between TAI and solar time grows to 800 msec a leap second is added to solar time.
- The corresponding time is called Universal Coordinated Time UTC.
- UTC is broadcast every second as a short pulse by the National Institute of Standard Time NIST. It is broadcast by GPS as well.

Time in distributed systems

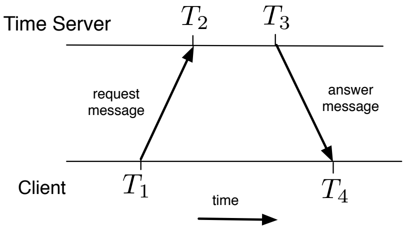
- Each computer p is equipped with a local clock C_p, which causes H interrupts per second. Given UTC time t, the clock value of p is given by $C_p(t)$.
- Let $C'_p(t) = \frac{dC_p}{dt}$
- Ideally, $C'_p(t) = 1$, real clocks have an error of about $\pm 10^{-5}$ (10 ppm)
- If there exists some constant ρ such that

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho,$$

  ρ is called the maximum drift rate.
- If synchronized Δt ago, two clocks may differ at most by $2\rho\Delta t$.
- To ensure synchronization within precision δ, then they need to be synchronized at least every $\frac{\delta}{2\rho}$ seconds.

Network Time Protocol NTP

- Assumption, one system C is connected to a UTC server. This system is called time-server.
- Each machine C , every $\frac{\delta}{2\rho}$ seconds, sends a time request to the time-server, which immediately responds with the current UTC.
- machine C sets its time to be T_3,
  - where T is the received time
  - RTT is the round trip time

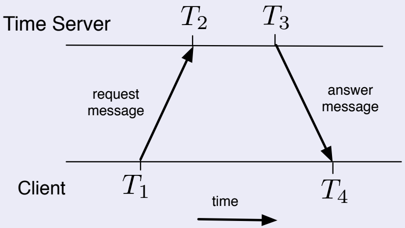


Problems and solutions

- Problem: time may run backwards!
- Solution: clocks converge to the correct time.
- Problem: Because of message delays, reported time will be outdated when received by a client.
- Solution: Try to find a good estimation for the delay.
- . . . (next slide)

Problems and solutions

- Problem: Because of message delays, reported time will be outdated when received by a client.
- Solution: Try to find a good estimation for the delay.
  - **Algorithm of Flaviu Cristian**
    Use $\frac{(T_4 - T_1)}{2}$ if no other information is available.
  - If interrupt handling time I is known, use $\frac{(T_4 - T_1 - I)}{2}$.
  - ... else ...



Problems and solutions

- Problem: Because of message delays, reported time will be outdated when received by a client.
- Solution: Try to find a good estimation for the delay
  NTP: Network Time Protocol
  - ... else ...
  - To adjust A to B, use piggybacking:
  - A sends a request to B timestamped with T_1.
  - B records the time of receipt T_2 (taken from its local clock) and returns a response timestamped with T_3 and piggybacking T_2.
  - A records the time of arrival T_4. The propagation time from A to B is assumed to be the same as from B to A, $T_2 - T_1 \approx T_4 - T_3$.
  - The offset θ of A relative to B can be estimated by A:

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

  - If $\theta < 0$, in principle, A has to set its clock backwards.
- Take the measures several times and compute the mean while ignoring outliers.

Examples: A has to be adjusted to B.

A sends a request to B timestamped with T_1. B records the time of receipt T_2 (taken from its local clock) and returns a response timestamped with T_3 and piggybacking T_2. A records the time of arrival T_4.
The offset θ of A relative to B can be estimated by A:

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

(a) No need for adaption detected.
$T_1 = 10, T_2 = 12, T_3 = 14, T_4 = 16 \Longrightarrow \theta = 0.$

(b) A has to slow down.
$T_1 = 10, T_2 = 12, T_3 = 14, T_4 = 18 \Longrightarrow \theta = -1.$

(c) A has to hurry up.
$T_1 = 10, T_2 = 12, T_3 = 14, T_4 = 14 \Longrightarrow \theta = 1.$

On scalability of NTP (roughly)

- NTP is an Internet standard (RFC 5905).
- NTP service is provided by a network of servers.
- Primary servers are directly connected to a UTC-source.
- Secondary servers synchronize themselves with primary servers.
- This approach is applied recursively leading to several layers.
- Server A adjusts itself to server B if B is assigned a lower layer than A.
- The whole network is reconfigurable and thus is able to react on errors.

2.2: Logical Time

Why?

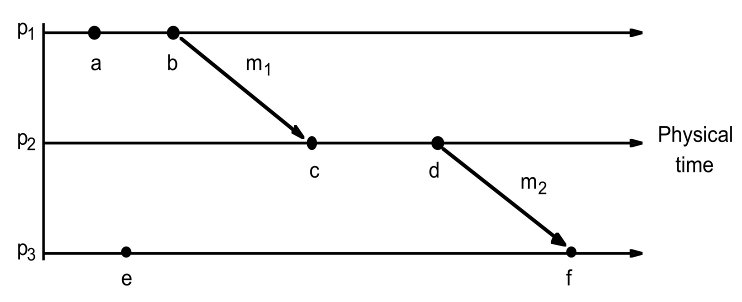
- Getting physical clocks absolutely synchronized is not possible.
- Thus it is not always possible to determine the order of two events.
- For such cases logical time can be used as a solution.
  - If two events happen in the same process they are ordered as observed.
  - If two processes interchange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

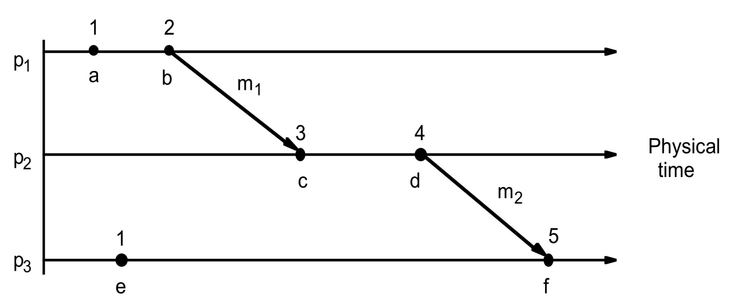
$$L_i := L_i + 1$$

- 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 $a \rightarrow b$.

Mattern's Vector Clocks

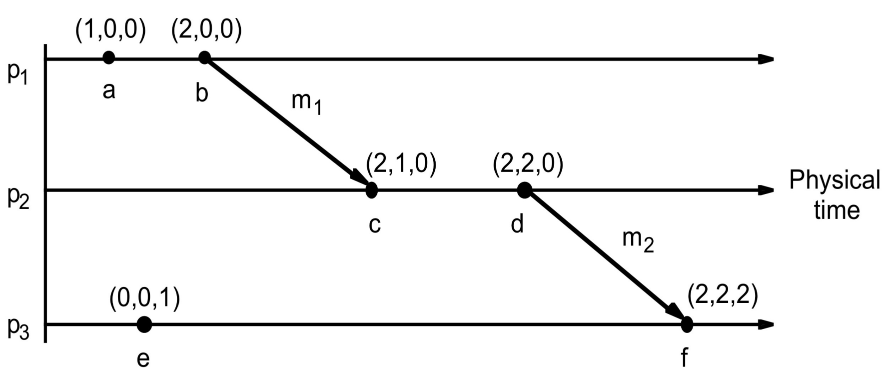
- Vector clock for a system of n processes: array of n integers.
- Each process P_i keeps its own vector clock V_i which is used to timestamp local events.
- Processes piggyback their own vector clock on messages they send.
- Update rules for vector clocks:

VC1: Initially, $V_i[j] := 0$ for $i, j \in \{1, \dots, n\}$
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

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

- $V_i[i]$ 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.

Vector Clock Example



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

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. $a||e$

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'
$(1,2,3)$	$(3,2,1)$	$a \mid\mid b$	two events are concurrent

Lamport Relationship and Vector Clocks

Theorem

For any two events e_j, e_i:

$$e_j \rightarrow e_i \iff V(e_j) < V(e_i)\ .$$

Proof sketch

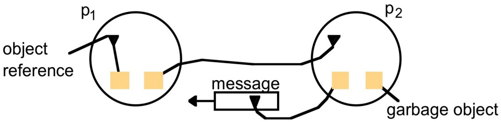
- $e_j \rightarrow e_i \implies V_j < V_i$.
  - If the events occur on the same process then $V_j < V_i$ follow directly.
  - $e_j \rightarrow e_i$ implies a message is sent after e_j to the process with event e_i or two succeeding events of a process
  - Since each entry of the receiving process is updated to at least the maximum of the entries of the sending processes, $V_j < V_i$
- $e_j \rightarrow e_i \impliedby V_j < V_i$.
  - If both events occur on the same process, $e_j \rightarrow e_j$ follows straightforward.
  - An increase of the 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

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

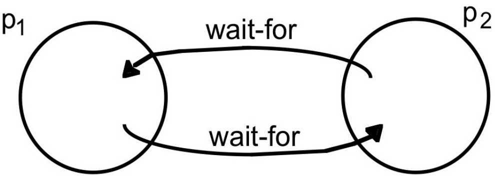
- How to determine a global state in the absence global time



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

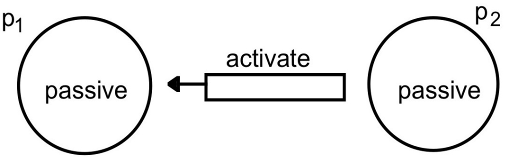


from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

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

- 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

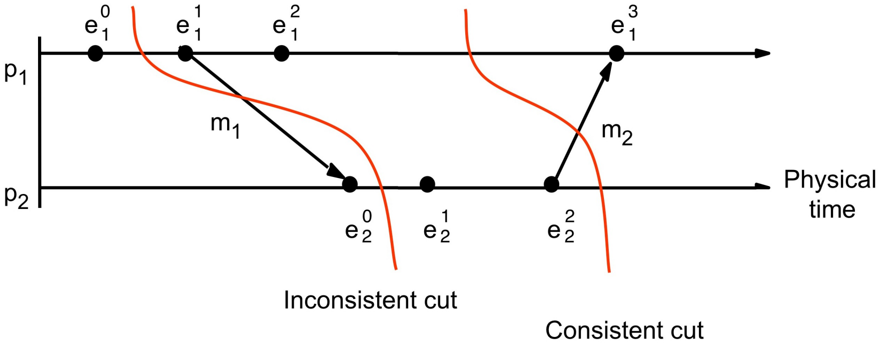
$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$$

Consistent Cuts

- A cut C is consistent if,

$$\text{For all events } e \in C: \quad f \rightarrow e \implies f \in C \quad .$$

- 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.

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).

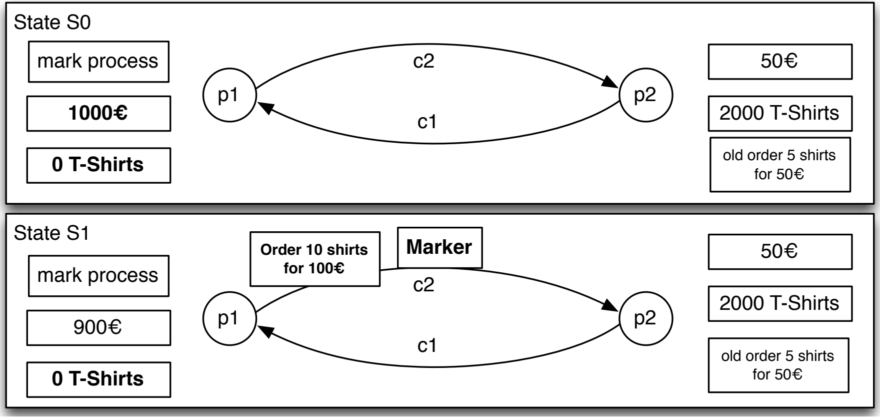
from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

General remarks

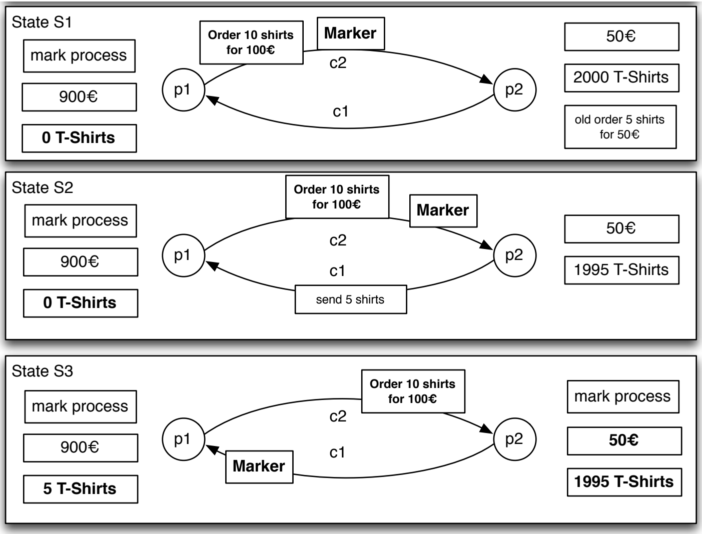
A snapshot consists of the state of a process and states of all incoming channels.

- 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.

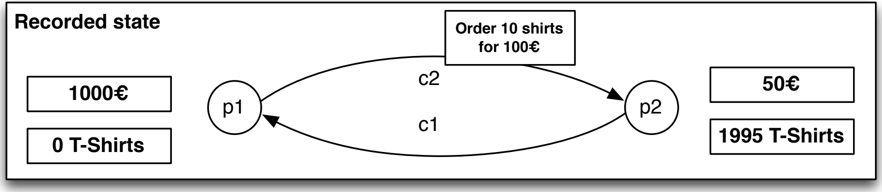
Distributed Snapshot of Chandy and Lamport



Distributed Snapshot of Chandy and Lamport



Distributed Snapshot of Chandy and Lamport



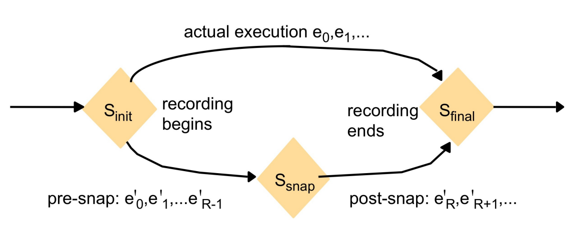
Termination of the snapshot algorithm

- 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
- $\Rightarrow$ only a finite number of messages need to be recorded

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 $e_i \longrightarrow e_j$ 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 operators

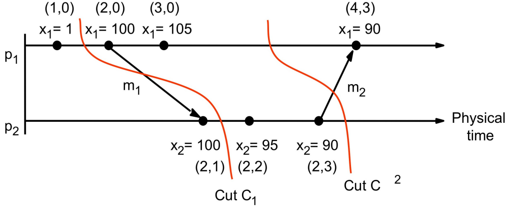
Consider all linearizations of H

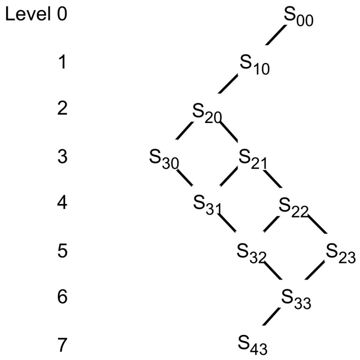
possible ϕ	There exists a consistent global state S through a linearization such that $\phi(S)$ is true.
definitely ϕ	For all linearizations a consistent global state will be passed such that $\phi(S)$ is true.

Relationship of Definitely and Possibly

1. $\forall S \in H : \phi(S) \quad \Longrightarrow \quad \textit{definitely } \phi$
2. $\forall S \in H : \phi(S) \quad \Longrightarrow \quad \textit{possible } \phi$
3. $\forall S \in H : \neg\phi(S) \quad \Longrightarrow \quad \neg\textit{definitely } \phi$
4. $\forall S \in H : \neg\phi(S) \quad \Longrightarrow \quad \neg\textit{possibly } \phi$
5. $\textit{definitely } \phi \quad \Longrightarrow \quad \textit{possibly } \phi$
6. $\neg\textit{possibly } \phi \quad \Longrightarrow \quad \textit{definitely } \neg\phi$
7. $\textit{definitely } \neg\phi \quad \not\Longrightarrow \quad \neg\textit{possibly } \phi$

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





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

Algorithm of Marzullo & Neiger

Collecting the states

- All initial states are sent to the monitor
- All state changes are sent to the monitor
- If only a predicate is monitored ϕ then only states are sent where ϕ changes
- With the states the corresponding vector clock is sent to the monitor
- The vector clocks will be used to establish the $\rightarrow$-relationship
- The monitor computes the DAG corresponding to the *happened-before*-relationship
- Arrange the graph in levels $L = 0, 1, \ldots$ such that no global state in level happened before a state in lower level.
- In Level 0 there is only the initial state.

1. *Evaluating possibly ϕ for global history H of N processes*

$L := 0;$

$States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \};$

while ($\phi(S) = False$ for all $S \in$ States)

$\quad L := L + 1;$

$\quad Reachable := \{ S' : S'$ reachable in H from some $S \in$ States $\wedge$ level$(S') = L \};$

$\quad States := Reachable$

end while

output "*possibly* ϕ";

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

2. *Evaluating definitely ϕ for global history H of N processes*

$L := 0;$

if $(\phi(s_1^0, s_2^0, \ldots, s_N^0))$ *then* $States := \{\}$ *else* $States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \};$

while $(States \neq \{\})$

$\quad L := L + 1;$

$\quad Reachable := \{ S' : S'$ reachable in H from some $S \in$ States $\wedge$ level$(S') = $ L$\};$

$\quad States := \{ S \in Reachable : \phi(S) = False \}$

end while

output "*definitely* ϕ";

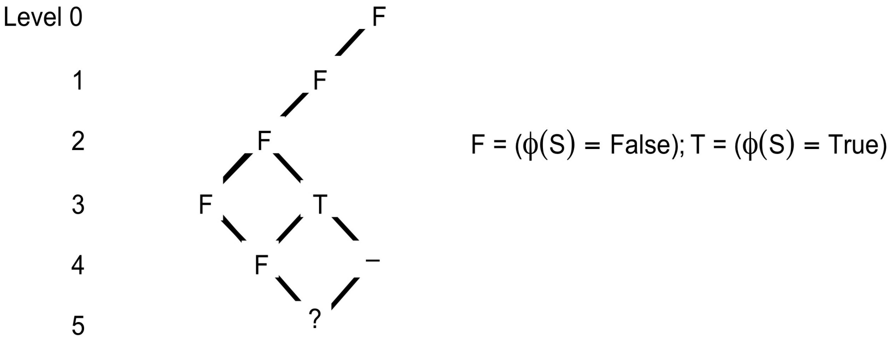
from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

Evaluating Definitely $\phi(S)$

Cost

Let n be the number of processes with k events each

- Time: $O(k^n)$
- Space: $O(kn)$.



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

End of Section 2