

Department of Computer Science

PURDUE
UNIVERSITY

CS505: Distributed Systems

Lecture 5: Time in Distributed Systems

Overview

- ▶ **Time and Synchronization**
- ▶ **Logical Clocks**
- ▶ **Vector Clocks**

Distributed Systems

- ▶ **Asynchronous systems: no bounds**
 - On communication delays
 - On relative processing speeds

- ▶ **But sometimes we need information on what happened when**
 - Consistent view of distributed application

Examples

- ▶ **Distributed database transaction journaling and logging**
- ▶ **Stock market buy and sell orders**
- ▶ **Secure document timestamps (with cryptographic certification)**
- ▶ **Aviation traffic control and position reporting**
- ▶ **...**

Sensor Networks

- ▶ **Communication is expensive (even if a node does not have any data to receive, just listening consumes power)**
- ▶ **Power is limited**
- ▶ **Synchronization is important because**
 - Nodes can sleep and save battery
 - Communication may be avoided

Ordering of Events

- ▶ **Order of events, particularly *causality* helps in reasoning or analyzing a system**
- ▶ **Single process**
 - Each event has a timestamp, causality relation between events is given by time
- ▶ **Distributed processes**
 - Many events generated at different processes, concurrent, related?
- ▶ **Time is essential for ordering events in a distributed system**
 - Physical time: local clock, global clock
 - Logical time: partial ordering, total ordering

Using Real Clocks

▶ Global clock

- Processes have access to a central global clock, each event will carry a timestamp

▶ Local clock

- Each process has its own clock
 - What if the clocks are not synchronized
 - What if events happened at the same time?

Synchronizing Physical Clocks

▶ External synchronization

- Consider the source S and the synchronization bound $B > 0$, then none of the clocks drift with more than B from S , at any time

▶ Internal synchronization

- Consider the synchronization bound $B > 0$, then at any time, the difference between any two clocks is within B

Algorithms for Clock Synchronization

Cristian's algorithm

- ▶ Uses a time server to synchronize clocks
- ▶ Clients ask the time server for time and adjust their clock based on response
 - *RTT* estimated by the client: $T_{send} - T_{receive}$
 - $T_{client} = T_{server} + (RTT / 2)$

Algorithms for Clock Synchronization

Berkeley algorithm

- ▶ Uses elected master to synchronize
- ▶ The master obtains the local time from all machines, adjusts times received for *RTT* & latency, averages times, and tells each machine how to adjust
- ▶ In some systems multiple time servers are used
- ▶ Time is more accurate, but still drifts

From Theory to Practice

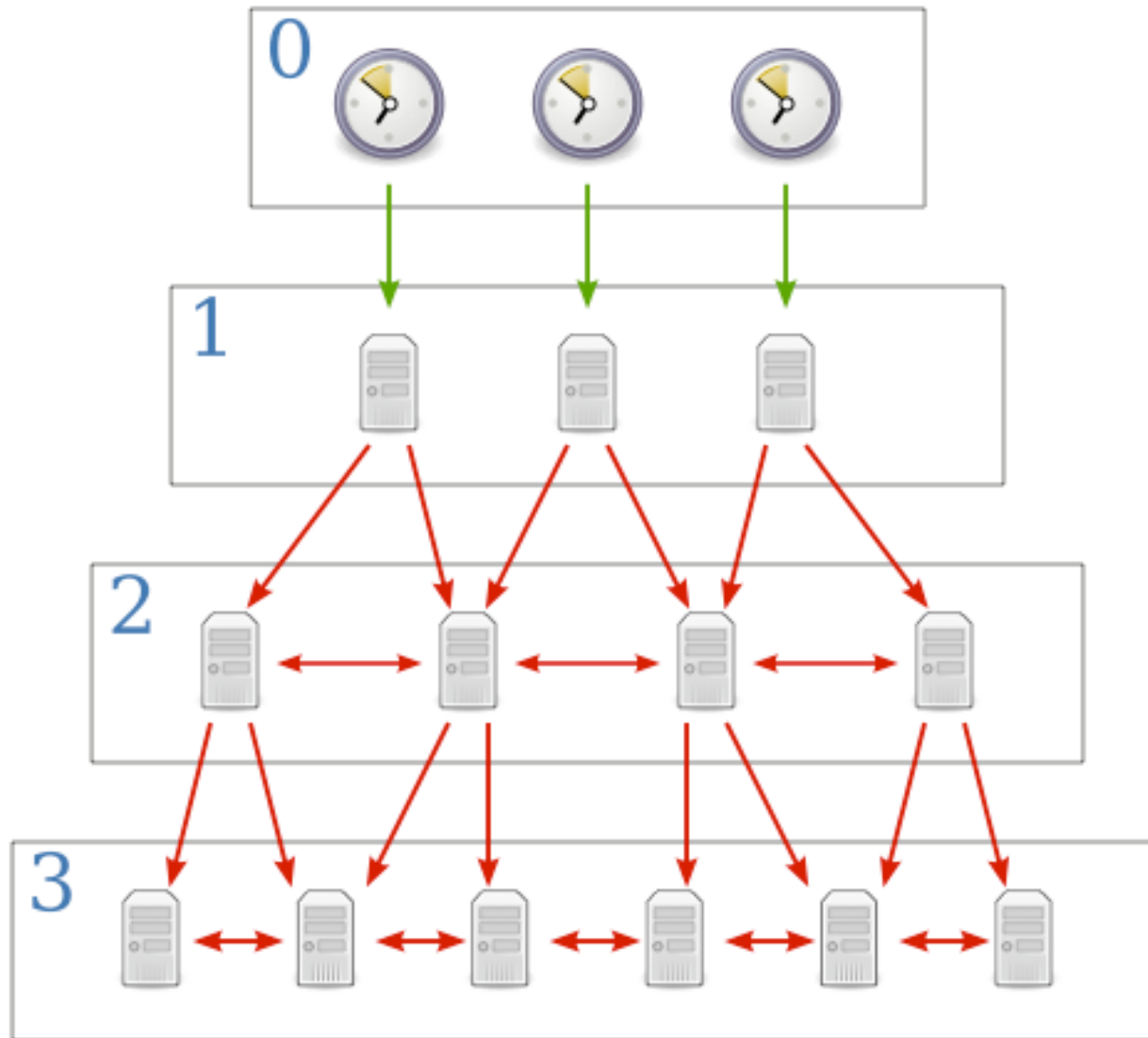
- ▶ **Synchronize computers across several networks?**
- ▶ **NTP (network time protocol) is a protocol designed to synchronize the clocks of computers over a network**
- ▶ **NTP version 3 is an internet draft standard, formalized in RFC 1305**
 - NTP version 4 is a significant revision of the NTP standard, current development version
- ▶ **Nominal accuracies of low tens of milliseconds on WANs, submilliseconds on LANs, and submicroseconds using a precision time source such as a cesium oscillator or GPS receiver**
- ▶ **Details**

www.eecis.udel.edu/~mills/database/brief/overview/overview.ppt

NTP “Capsule” Summary

- ▶ **Primary (stratum 1) servers synchronize to national time standards via radio, satellite and modem**
- ▶ **Secondary (stratum 2, ...) servers and clients synchronize to primary servers via hierarchical subnet**
- ▶ **Clients and servers operate in master/slave, symmetric and multicast modes with or w/o cryptographic authentication**
- ▶ **Reliability assured by redundant servers and diverse network paths**
- ▶ **Engineered algorithms reduce jitter, mitigate multiple sources and avoid improperly operating servers**
- ▶ **System clock is disciplined in time and frequency using an adaptive algorithm responsive to network time jitter and clock oscillator frequency wander**

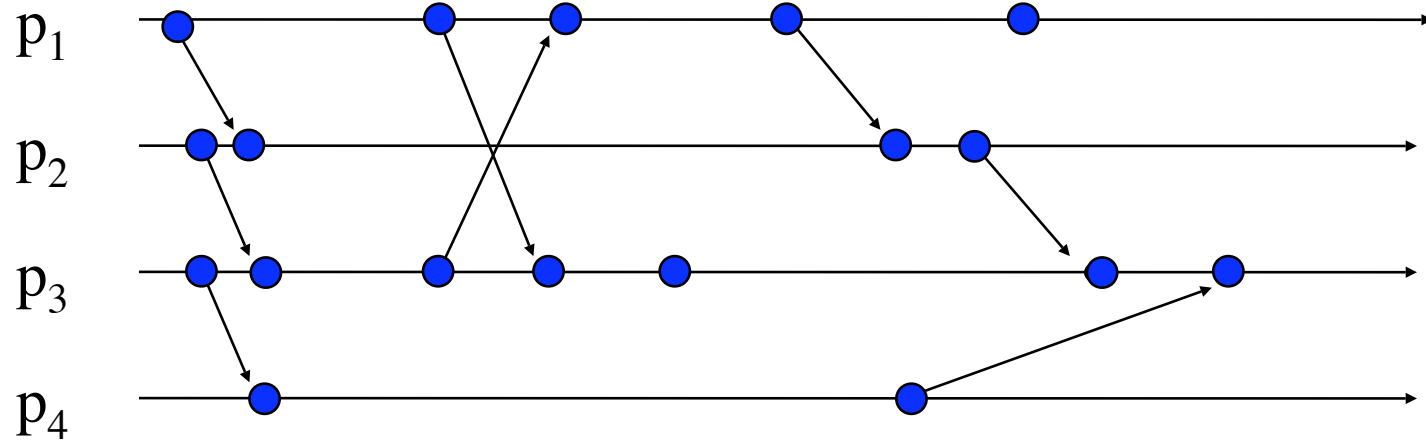
Overview



From Physical Clocks to Logical Clocks

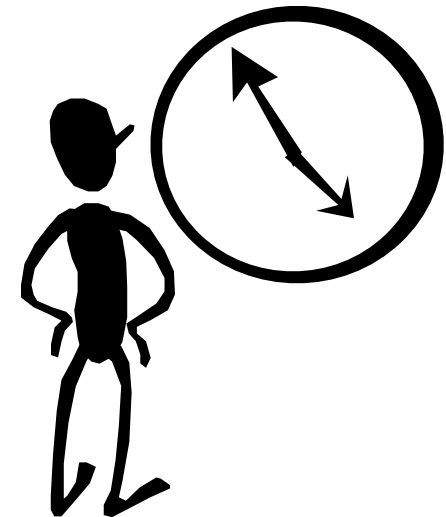
- ▶ **Synchronized clocks are great if we have them, but**
- ▶ **Do we need the real time anyway?**
- ▶ **In distributed systems we often only care about “relative time”**

“Happened Before”



$a \rightarrow b$ if either

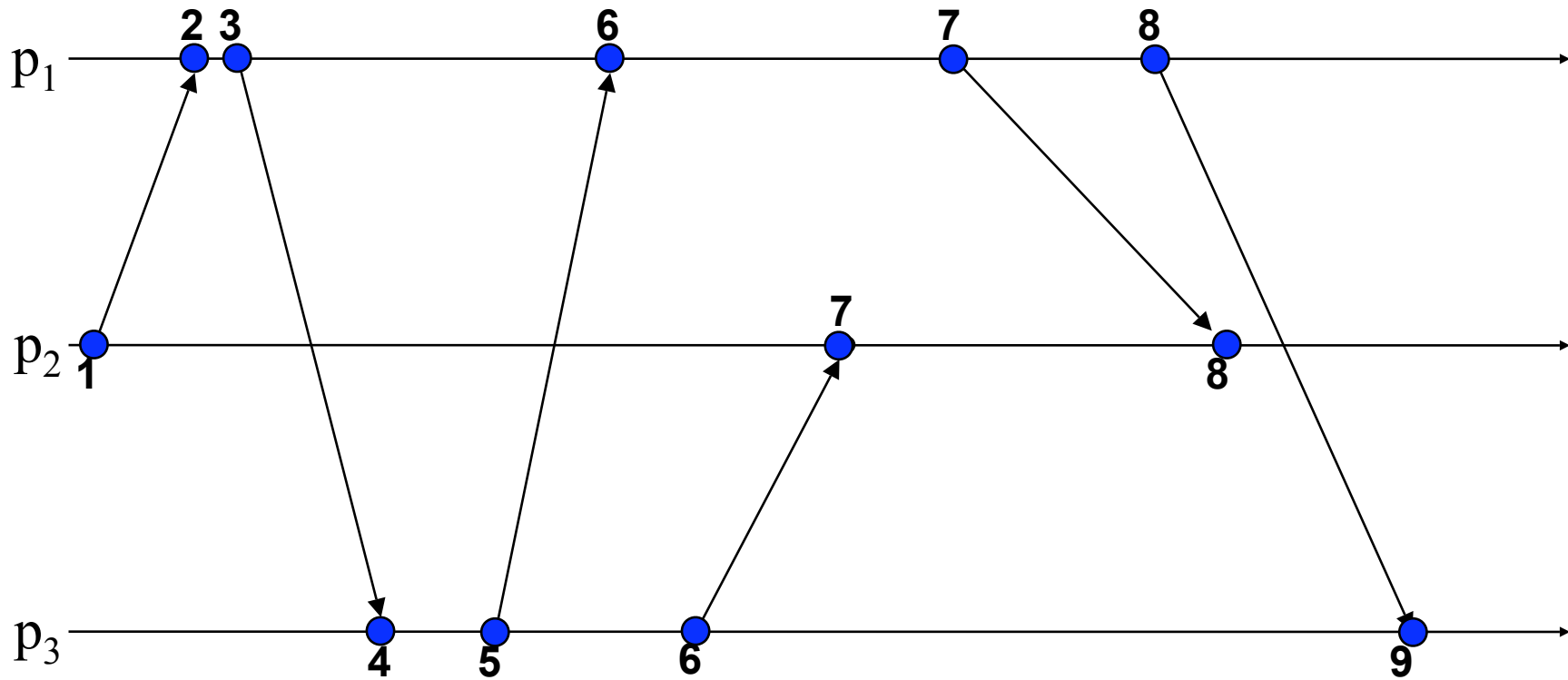
- ▶ events a and b take place at the same process and a occurs before b
- ▶ a is a send (m) event at p_i and b is receive (m) event at p_k , $p_i \neq p_k$
- ▶ exists $c : a \rightarrow c$ and $c \rightarrow b$



Lamport (Logical) Clocks

- ▶ Each process maintains its own clock C_i (a counter)
- ▶ For any events a and b at process p_i
if $a \rightarrow b$ then $C_i(a) < C_i(b)$
- ▶ Implementation
 - Each process p_i increments C_i between any successive events
 - On send (m) event a , attach local clock T_m to message m
$$T_m = C_i(a)$$
 - On receive (m) process p_k sets C_k to
$$C_k = \max(C_k, T_m) + 1$$

Example



Total Order?

- ▶ Logical Clocks provide *partial* order
- ▶ Often total order is required
 - E.g., delivery of requests/messages by replicas
- ▶ TO can be obtained by “breaking ties”
 - E.g., use ordered process identifiers

If a is event in p_i and b is event in p_k then

$a \rightarrow b$ iff

$C_i(a) < C_k(b)$ or

$C_i(a) = C_k(b)$ and $p_i < p_k$

Reminder: Partial and Total Order

- ▶ **Definition:** A relation R over a set S is a *partial* order iff for each a , b , and c in S
 - $aRb \wedge bRa \Rightarrow a = b$ (antisymmetric)
 - $aRb \wedge bRc \Rightarrow aRc$ (transitive)
 - aRa (reflexive)
- ▶ **Definition:** A relation R over a set S is a *total* order if for each distinct a and b in S
 - R is antisymmetric
 - R is transitive
 - *either* aRb or bRa (totality)

Concurrent Events

▶ Concurrent events

a and b are concurrent if neither causally influences the other

▶ Logical clocks assign order to events that are causally independent, in other words events that are causally independent appear as if they happened in a certain order

- $a \rightarrow b$ implies $C(a) < C(b)$
- $C(a) < C(b)$ implies $\neg b \rightarrow a$
- $C(a) < C(b)$ does not imply $a \rightarrow b$!!!

▶ We need higher resilience

Vector Clocks

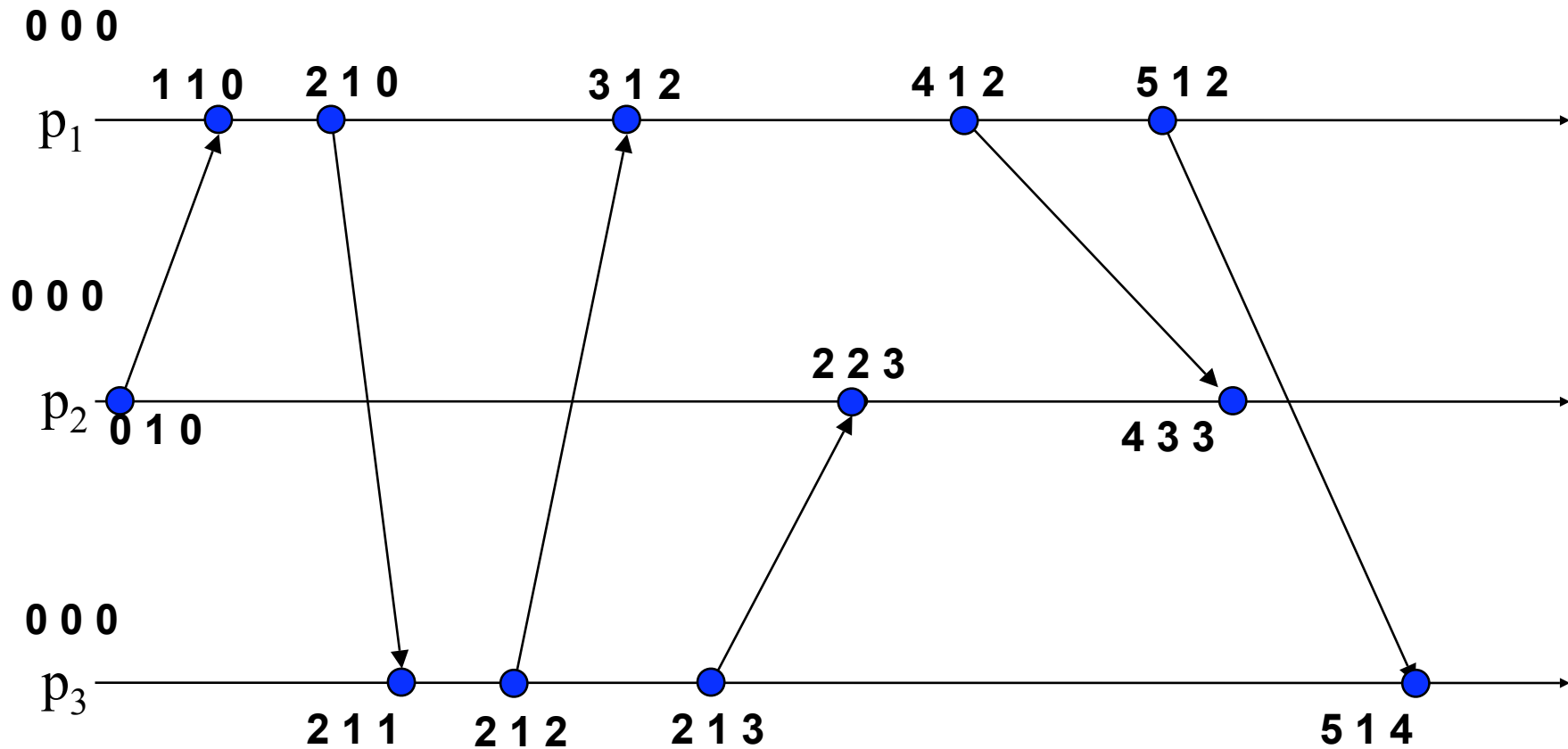
- ▶ Each process maintains *vector* V_i initially $[0, 0, \dots]$
- ▶ When p_i executes an event, it increments $V_i[i]$
- ▶ When p_i does `send(m)` to p_j , `m` piggybacks V_i
- ▶ When p_i does `receive(m)` from p_k

$$V_i[j] = V_i[j] + 1$$

$$V_i = \text{sup}(V_i, m.V), \quad \text{where } \text{sup}(V_1, V_2) = V \text{ s.t. } \forall j: 1 \leq j \leq n, \\ V[j] = \max(V_1[j], V_2[j])$$



Vector Clocks: Example



How to Order with Vector Clocks

► Given two events a and b , $a \rightarrow b$ iff

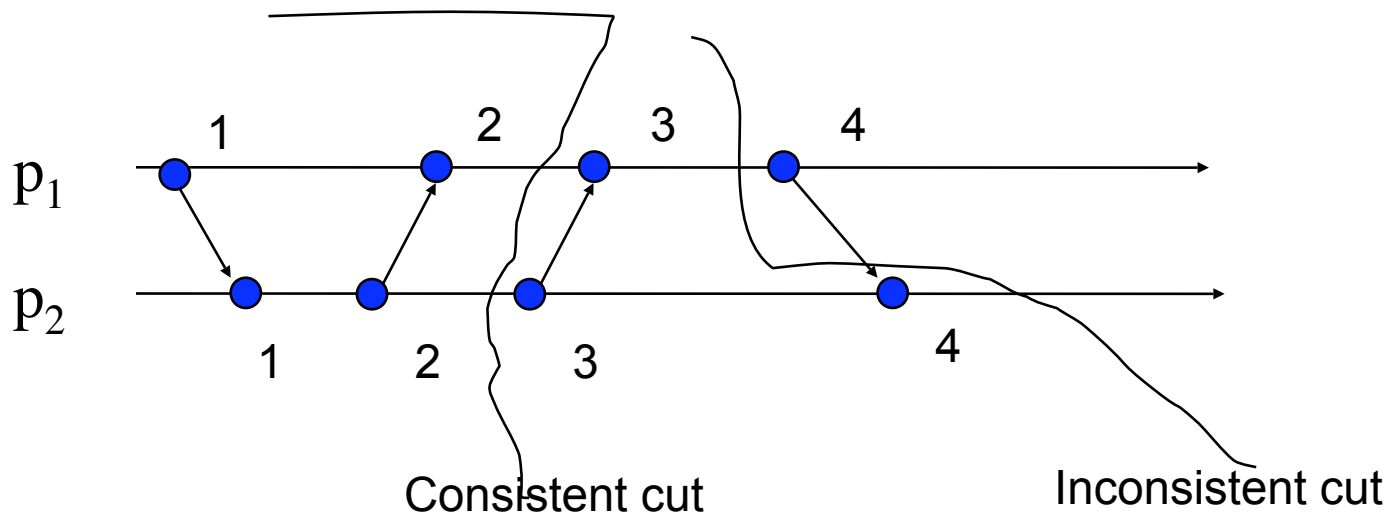
- b has a counter value for the process in which a occurred greater than or equal to the value of that process at event a inclusive, and
- a has a counter value for the process in which b occurred strictly less than the value of that process at event b inclusive

► Formally

- $a \rightarrow b \equiv \forall i: 1 \leq i \leq n: V(a)[i] \leq V(b)[i]$
 $\wedge \exists j: 1 \leq j \leq n: V(a)[j] < V(b)[j]$
- $b \parallel a \equiv \exists i: 1 \leq i \leq n: V(b)[i] < V(a)[i]$
 $\wedge \exists j: 1 \leq j \leq n: V(a)[j] < V(b)[j]$

Cuts

- ▶ There is no outside observer that can look at the system and detect problems, for example a deadlock
- ▶ Cut: n -vector (c_1, \dots, c_n) of positive integers
 - c_i represents cut event on p_i ; $c_i = V_i[\bar{l}]$ at that point
- ▶ Consistent cut: if for all i, j , $(c_i + 1)$ event at process p_i did not 'happen before' c_j event at p_j



Vector Clocks and Consistent Cuts

▶ A consistent cut K is a subset of events in E s.t.

– $\forall e \text{ in } K, e' \rightarrow e \Rightarrow e' \text{ in } K$

▶ c_i event of cut K on p_i (or last event on p_i in K)

– Global time of cut K is $T = \sup(V(c_1), \dots, V(c_n))$

▶ Theorem

K is consistent $\Leftrightarrow T = (V(c_1)[1], \dots, V(c_n)[n])$

Why?

▶ (\Rightarrow)

- For any time diagram with a consistent cut consisting of cut-events c_1, \dots, c_n , there is an equivalent time diagram where c_1, \dots, c_n occur simultaneously (in real time), i.e., where the cut line forms a straight vertical line
 - Any message has a delay
- At any point in (real) time, $V_i[l] = \max(V_k[l]) \forall k \neq i: 1 \leq k \leq n$

▶ (\Leftarrow , contradict.)

- If K is inconsistent then some p_i sends message m *after* c_i , received by p_k *before* c_k
- As a consequence $c_i[l] < m.V[l] \leq c_k[l]$

Constructing A Consistent Cut

- ▶ p_i ticks and fixes snapshot time s to $V_i + (0, \dots, 1, \dots, 0)$
 - Send s to all processes
 - Waits for acknowledgements from all processes
- ▶ p_i ticks again (setting V_i to s), takes a local snapshot, and broadcasts dummy message to all processes
 - This forces all processes to advance their clocks to a value $\geq s$
- ▶ Each process takes a local snapshot and sends it to p_i when its local clock becomes equal to s or jumps from some value smaller than s to a value larger than s

References

- ▶ ***Time, Clocks, and The Ordering of Events in Distributed Systems.*** L. Lamport, CACM 27(7): 558-565, 1978.
- ▶ ***Virtual Time and Global States of Distributed Systems,*** F. Mattern, WDPA'89.

Next Time

- ▶ ***(A Modular Approach to) Fault-tolerant Broadcasts and Related Problems.*** V. Hadzilacos and S. Toueg, **Distributed Systems**, 97-145, 1993.