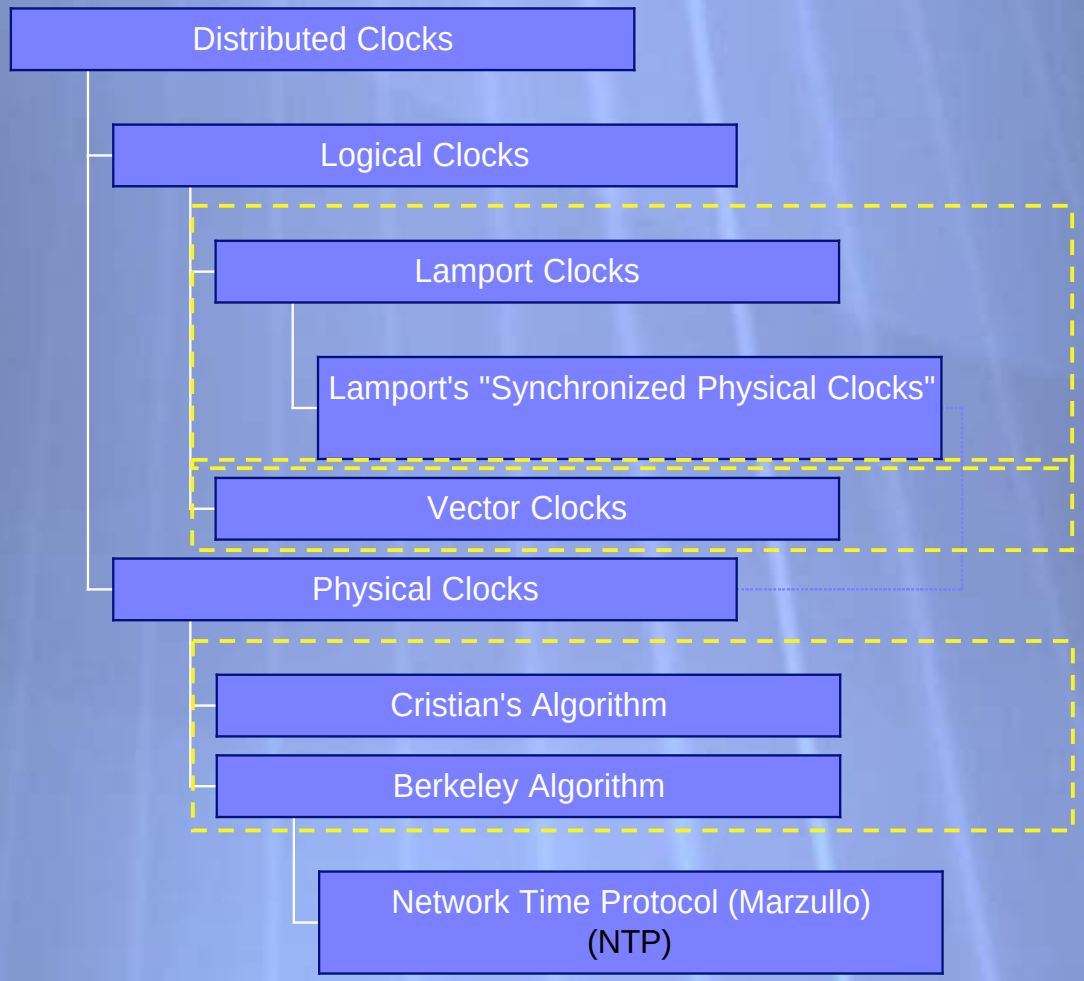
# The meaning of "time" in a distributed system

Jonathan S. Anderson
*andersoj@andersoj.org*

# Time and Clocks in Distributed Systems

Logical Hierarchy of Distributed Clocks




1978 (Lamport)

1989 (Mattern)

1989 (Cristian)

1989 (Gusella and Zatti)

1984 (Marzullo)

# Time in Distributed Systems: Why Worry?

- We think intuitively thatan event *a* happens before *b* if it happened "at an earlier time"
- This requires reference to physical clocks / physical perception of time
- Works most of the time, in our macroscopic world, but still breaks down…

# Time, Clocks, and the Ordering of Events in Distributed Systems

- Appeared in Communications of the ACM, July 1978
- ARPAnet just evolving; no established "internet"
- Addresses the logical clock problem primarily, but opens the door to…
  - Strongly consistent logical clocks
  - Physical clock synchronization

Operating Systems — R. Stockton Gaines, Editor

## Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport
Massachusetts Computer Associates, Inc.

**The concept of one event happening before another in a distributed system is examined, and is shown to define a partial ordering of the events. A distributed algorithm is given for synchronizing a system of logical clocks which can be used to totally order the events. The use of the total ordering is illustrated with a method for solving synchronization problems. The algorithm is then specialized for synchronizing physical clocks, and a bound is derived on how far out of synchrony the clocks can become.**

**Key Words and Phrases: distributed systems, computer networks, clock synchronization, multiprocess systems**
**CR Categories: 4.32, 5.29**

### Introduction

The concept of time is fundamental to our way of thinking. It is derived from the more basic concept of the order in which events occur. We say that something happened at 3:15 if it occurred *after* our clock read 3:15 and *before* it read 3:16. The concept of the temporal ordering of events pervades our thinking about systems. For example, in an airline reservation system we specify that a request for a reservation should be granted if it is made *before* the flight is filled. However, we will see that this concept must be carefully reexamined when considering events in a distributed system.



This work was supported by the Advanced Research Projects Agency of the Department of Defense and Rome Air Development Center. It was monitored by Rome Air Development Center under contract number F 30602-76-C-0094.

Author's address: Computer Science Laboratory, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.





A distributed system consists of a collection of distinct processes which are spatially separated, and which communicate with one another by exchanging messages. A network of interconnected computers, such as the ARPA net, is a distributed system. A single computer can also be viewed as a distributed system in which the central control unit, the memory units, and the input-output channels are separate processes. A system is distributed if the message transmission delay is not negligible compared to the time between events in a single process.

We will concern ourselves primarily with systems of spatially separated computers. However, many of our remarks will apply more generally. In particular, a multiprocessing system on a single computer involves problems similar to those of a distributed system because of the unpredictable order in which certain events can occur.

In a distributed system, it is sometimes impossible to say that one of two events occurred first. The relation "happened before" is therefore only a partial ordering of the events in the system. We have found that problems often arise because people are not fully aware of this fact and its implications.

In this paper, we discuss the partial ordering defined by the "happened before" relation, and give a distributed algorithm for extending it to a consistent total ordering of all the events. This algorithm can provide a useful mechanism for implementing a distributed system. We illustrate its use with a simple method for solving synchronization problems. Unexpected, anomalous behavior can occur if the ordering obtained by this algorithm differs from that perceived by the user. This can be avoided by introducing real, physical clocks. We describe a simple method for synchronizing these clocks, and derive an upper bound on how far out of synchrony they can drift.

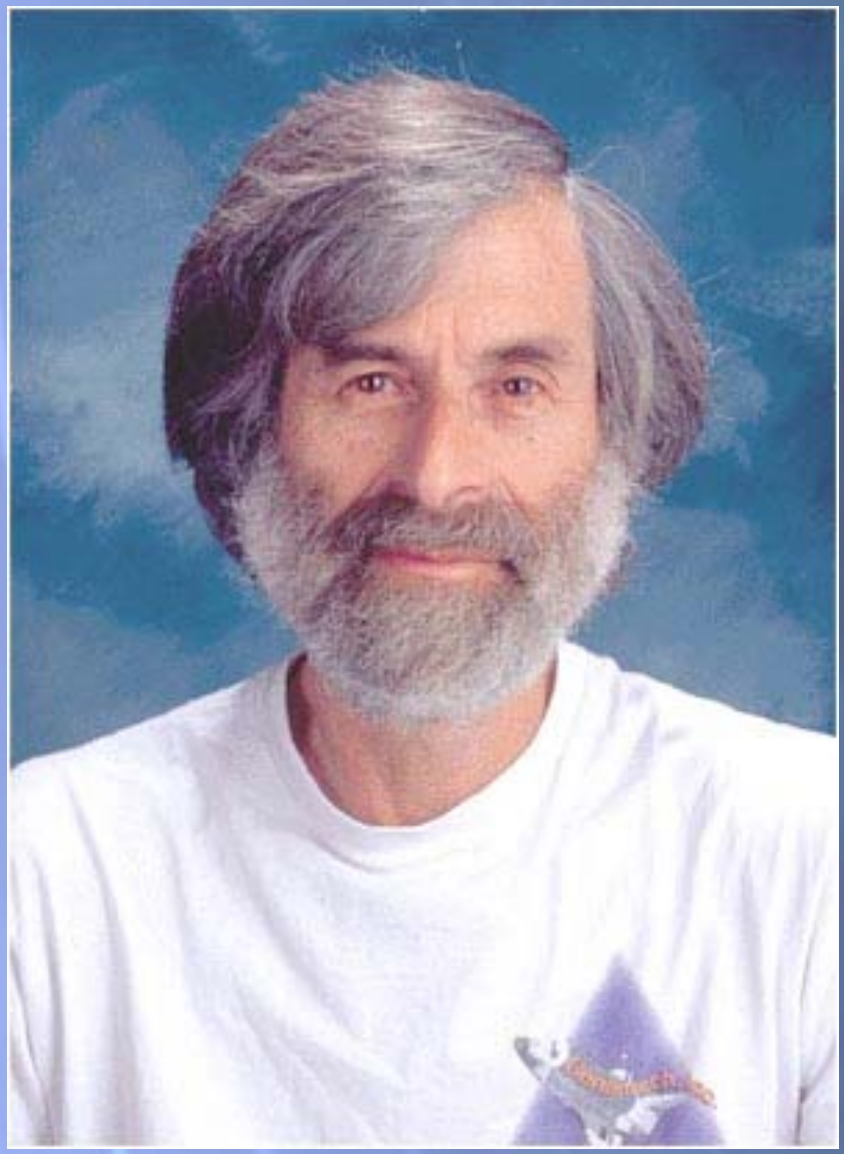
### The Partial Ordering

Most people would probably say that an event $a$ happened before an event $b$ if $a$ happened at an earlier time than $b$. They might justify this definition in terms of physical theories of time. However, if a system is to meet a specification correctly, then that specification must be given in terms of events observable within the system. If the specification is in terms of physical time, then the system must contain real clocks. Even if it does contain real clocks, there is still the problem that such clocks are not perfectly accurate and do not keep precise physical time. We will therefore define the "happened before" relation without using physical clocks.

We begin by defining our system more precisely. We assume that the system is composed of a collection of processes. Each process consists of a sequence of events. Depending upon the application, the execution of a subprogram on a computer could be one event, or the execution of a single machine instruction could be one

# The Author

- Les Lamport (1941-)
- MIT, SRI, DEC, and now Microsoft Research
- Influential in developing theory/algorithms for distributed systems [5]
  - Buridan's Ass
  - Paxos Consensus Algorithm
  - Temporal Logic of Actions (TLA)
- Also initial developer of LaTeX macros
- This paper was awarded the Edsger W. Dijkstra Prize in Distributed Computing (2000)



Leslie Lamport [3]

# Definitions

- “A system is *distributed* if the message transmission delay is not negligible compared to the time between events in a single process.”
  - Internet
  - Local-Area Network
  - A single computer of nontrivial complexity
- "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.” (Les Lamport [1])

# Contributions of this Paper

- Physical clocks (recourse to physical time) may be nontrivial or expensive to implement
- It is possible to implement "logical clocks" in distributed systems
- Logical clocks can be applied to a class of problems…
  - Including distributed mutual exclusion
- However, we may introduce physical clocks
  - Having done so, it is possible to synchronize such clocks
  - Lamport approach is costly

# Motivation for Logical Time

- Systems depend on many kinds of consistency
  - Temporal consistency is crucial to the way we reason about algorithm behavior
  - Also crucial to ideas of *causality*
- Physical clocks cannot be perfectly synchronized (though, this motivation is waning in importance)
- Simulations may require clocks which can stop, run backward, or have their rates modified

# The Partial Ordering

- Defining a formal definition of the "*happened-before*" relationship without reference to physical clocks
- A "process" is a sequence of events
  - Event may be a subprogram execution
  - Or a message send/receive step
  - Or execution of a single machine instruction
- Within a process, event *a* happens-before event *b* if and only if *a* occurs before *b* in the sequence (within a process, a priori total ordering of events)

# Partial Set Orderings (Posets)

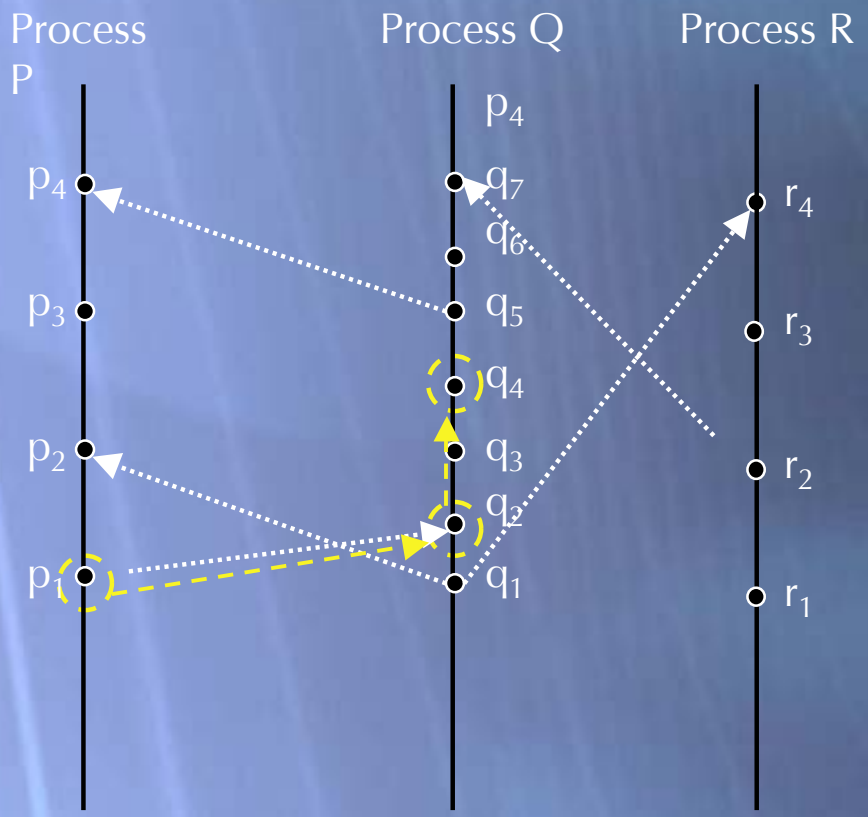
- An *irreflexive* or *strict partial order* $<$ on a set $S$ is a binary relation with:
  - If $a \in S$ then $\neg(a < a)$ or, colloquially $a \not< a$
  - If $a,b \in S$ and $a < b$, then $\neg(b < a)$ or, colloquially $b \not< a$
  - If $a,b,c \in S$ with $a < b$ and $b < c$, then $a < c$
- An irreflexive, asymmetric, and transitive binary relation
- A binary relation $R$ on a set $S$ is a subset $R \subseteq S \times S$. If $a,b \in S$, then $aRb \Leftrightarrow (a,b) \in R$. Or "$a$ is related through $R$ to $b$" if the tuple is in the subset $R$.

# The Partial Ordering [continued]

- The relation $\rightarrow$ on the set of events in a system is the smallest relation satisfying:
  - If events *a* and *b* are in the same process, and *a* comes before *b*, then $a \rightarrow b$ (local total ordering)
  - If *a* is the "send message *m*" event from a process and *b* is the "receive message *m*" event in another process, then $a \rightarrow b$ (message causality)
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. (transitivity)
- Two distinct events *a* and *b* are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$ *(commonly denoted* $a \| b$*)*
- Note $a \not\rightarrow a$ for all events *a*
  (events cannot happen before themselves)
- This forms an irreflexive partial ordering [2]

# Space-Time Diagrams: *Happened-Before*

- Space-time diagrams
  - Horizontal axis is space
  - Vertical axis is time (later times upward)
  - Dots are events
  - Wavy lines are messages
- Happens-before relation: $a \rightarrow b$ means we can move from $a$ to $b$ by
  - Moving forward in time along process lines, or
  - Moving along message lines
- Example: $p_1 \rightarrow q_4$
  - Transitivity from $p_1 \rightarrow q_2 \wedge q_2 \rightarrow q_4$
- However: $p_1 \parallel r_1$
  - $p_1 \not\rightarrow r_1$ and $r_1 \not\rightarrow p_1$



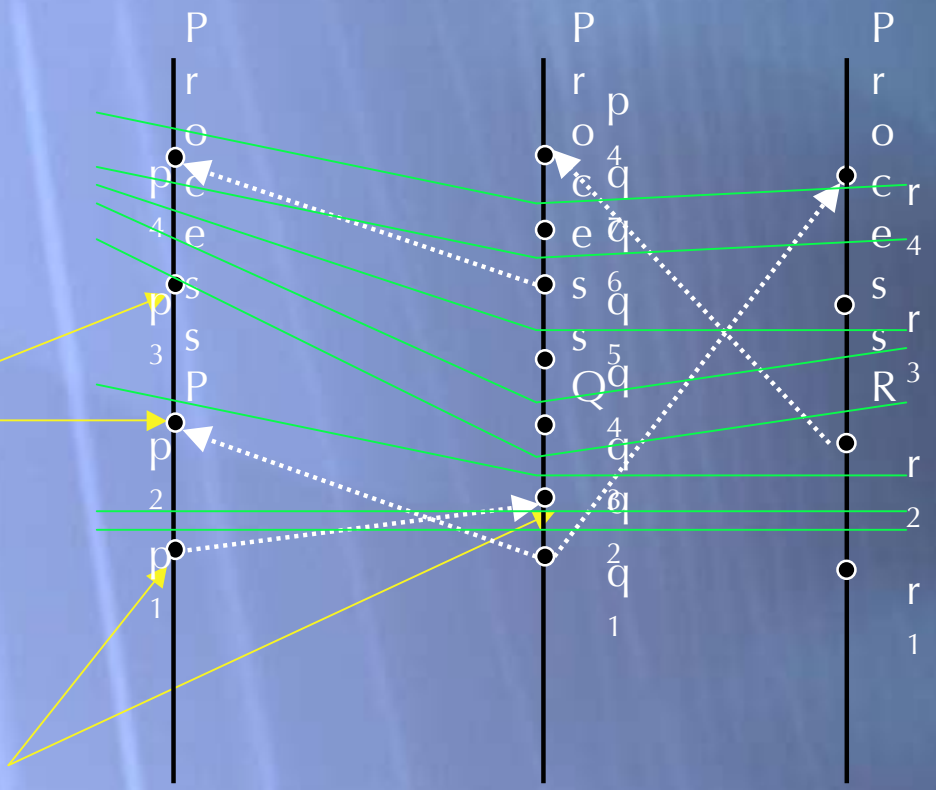

Space-Time Diagram (Figure 1 from [0])

# Introducing Logical Clocks

- Abstract clock: Simply associates a number with every event
  - For each process $P_i$ there is a clock $C_i$, assigns a number $C_i\langle a\rangle$ to every event $a$
  - Entire system of clocks is denoted $C$
  - $C\langle b\rangle = C_i\langle b\rangle$ if $b$ is an event in process $P_i$
  - No connection required to physical time; e.g., $C$ may be implemented simply as per-process counters

# Clock Correctness

- Clock Condition:
  For any events *a* and *b*,
  if $a \rightarrow b$ then $C\langle a\rangle < C\langle b\rangle$
- Corollaries
  - If *a*, *b* are events in process $P_i$ and *a* comes before *b*, then
    $C_i\langle a\rangle < C_i\langle b\rangle$
    *(> one tick between events)*
  - If *a* is the *send-message* event in process $P_i$ and *b* is the *receive-message* event in process $P_j$ and *a* comes before *b*,
    then $C_i\langle a\rangle < C_j\langle b\rangle$
    *(every message crosses tick)*




Space-Time Diagram
(Figure 2 from [0])

*Green Lines represent "ticks" of the logical clocks*

# Logical Clocks Recap

- Lamport clocks are consistent, but they do not capture causality:
  - Consistent: $a \rightarrow b \Rightarrow C(a) < C(b)$
  - But not: $C(a) < C(b) \Rightarrow a \rightarrow b$
    - (i.e., they are not strongly consistent)
- Two independent ways to extend them:
  - By creating total order (but not strongly consistent!)
    - $(C_i, P_m) < (C_k, P_n)$ iff $C_i < C_k \,||\, (C_i == C_k \,\&\&\, m < n)$
  - By creating a strongly consistent clock (but not a total order!)
    - Vector clocks



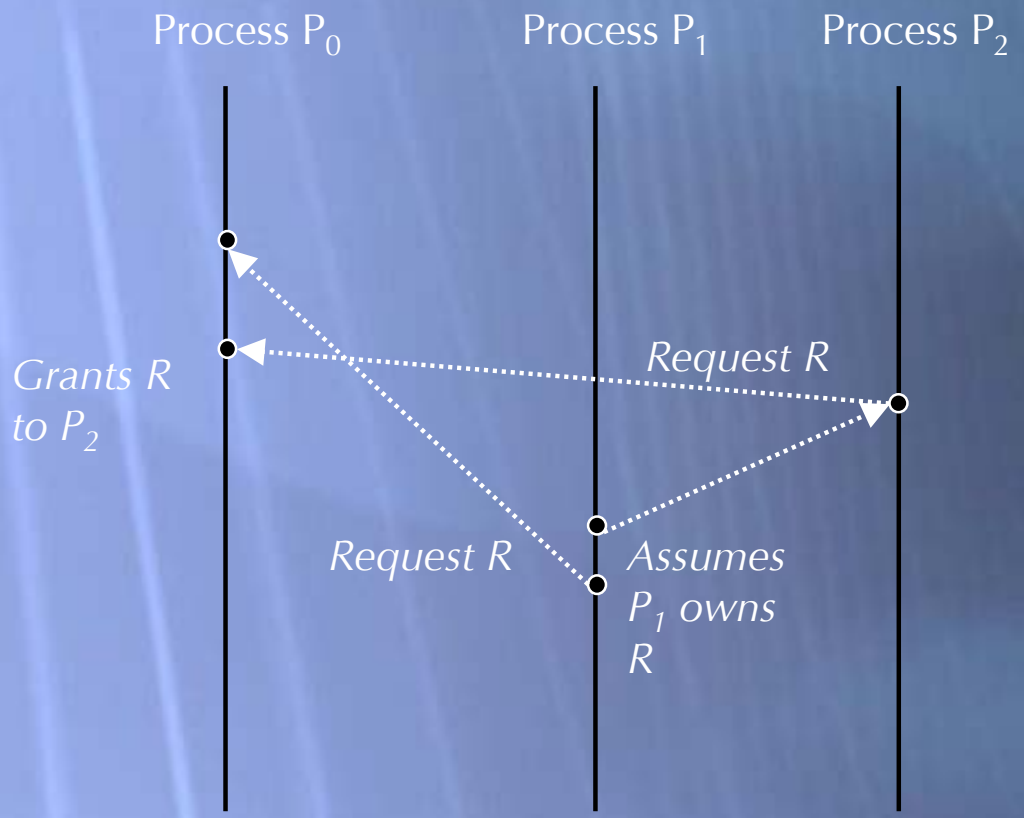
*This slide courtesy Prof. G. Back*

# Distributed Locking Algorithm

- A system composed of a fixed collection of processes Pi sharing a single resource R
- Satisfy the following:
  I. A process holding R must release it before it can be granted to another
  II. Requests for R must be granted in the order they were made
  III. If every process granted R eventually releases it, then every process requesting R eventually is granted it

# Centralized Distributed Locking

- One designated process $P_0$ manages the shared resource
- Ordering of events may leave $P_1$ confused due to message/processing delay



# Distributed Locking with Logical Clocks

- A few assumptions:
  - Resource begins as granted to $P_0$
  - Assume (process-pairwise) ordered message delivery
  - No message loss
  - Processes are well-connected
- To solve the problem
  - Each process $P_i$ maintains a private request queue
  - Initialized to *"T0:P0 requests resource R"*

# Implementing Distributed Mutual Exclusion

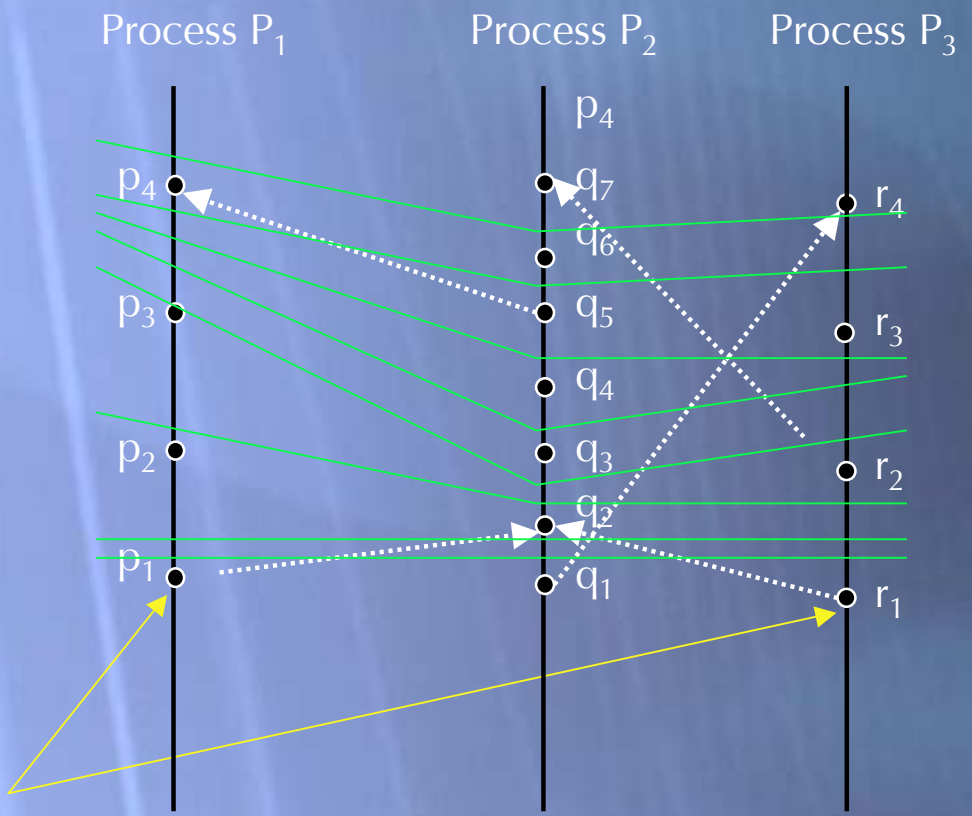
- Request_resource:
  - $P_i$ sends message "$T_m$:$P_i$ requests resource" to all processes and enqueues it
- Upon receiving request message, all processes $P_j$
  - Enqueue request
  - Send acknowledgement to $P_i$
- Release_resource on $P_i$:
  - Remove any enqueued "$T_m$:$P_i$ requests resource"
  - Send "$P_i$ releases resource" to all processes
- Upon receiving release message, all processes $P_j$
  - Remove any matching requests from queue

# Implementing Distributed Mutual Exclusion (cont'd)

- Grant $P_i$ resource when all the following are met:
  - There is a $T_m$:$P_i$ requests resource message in queue
  - $T_m$:$P_i$ message is ordered before all others by ->
  - $P_i$ has received a message from every other process timestamped after $T_m$
  - When identical timestamps arrive, the process ID "m" is used to provide ordering (arbitrary)

# Distributed Mutual Exclusion

- Note that if $P_1$ and $P_3$ each request the resource from $P_2$ at indistiguishable ticks
- Clock timestamps are identical… so
- Lamport algorithm imposes arbitrary ordering: 1 < 3, so $P_1$ gets the resource




Space-Time Diagram
(Figure 2 from [0])

*Green Lines represent "ticks" of the logical clocks*

# Distributed Mutual Exclusion

- Truly a distributed algorithm
- Frame solution as a state machine
- Not fault-tolerant (why?)
- Requires (imposes) a "total ordering" on the system
  - A total order $\leq$ on a set $S$ is a binary relation with:
    - If $a,b \in S$ with $a \leq b$ and $b \leq a$, then $a = b$
    - If $a,b,c \in S$ with $a < b$ and $b < c$, then $a < c$
    - If $a,b \in S$, $a \leq b$ or $b \leq a$
  - Antisymmetric, transitive, and "total" or "complete"

# Logical Clock Anomalies

- Total ordering ⇒ is consistent, but may not square with physical time
- Doesn't handle "out of band communication"
  - Lamport introduces synchronized physical clocks
- Two approaches to resolve:
  - Explicit timestamp manipulation (by application)
  - Build system which satisfies the Strong Clock condition:
    For any events *a*, *b*: if $a \rightarrow b$ then
    $C\langle a \rangle < C\langle b \rangle$

# Causality and Lamport Clocks

"Lamport timestamps assure us that if there is a causal relationship between two events, then the earlier event will have a smaller timestamp than the later event. Causality is achieved by successive events on one process or by the sending and receipt of messages on different processes."

*Paul Krzyzanowski, CS417: Lectures on Distributed Systems, Rutgers University, 2000.*
*http://www.cs.rutgers.edu/~pxk/rutgers/notes/content/06-clocks.pdf*

# Physical Clocks

- Let our universe of events be space-time events, with $\rightarrow$ be the partial ordering defined by special relativity
  - $a \rightarrow b$ iff $b$ is in the "light-cone" of $a$
  - More intuitively, if a "can be observed by" b
- Note that we can build real-world clocks such that this relationship satisfies the strong clock condition

# What to take away from this paper

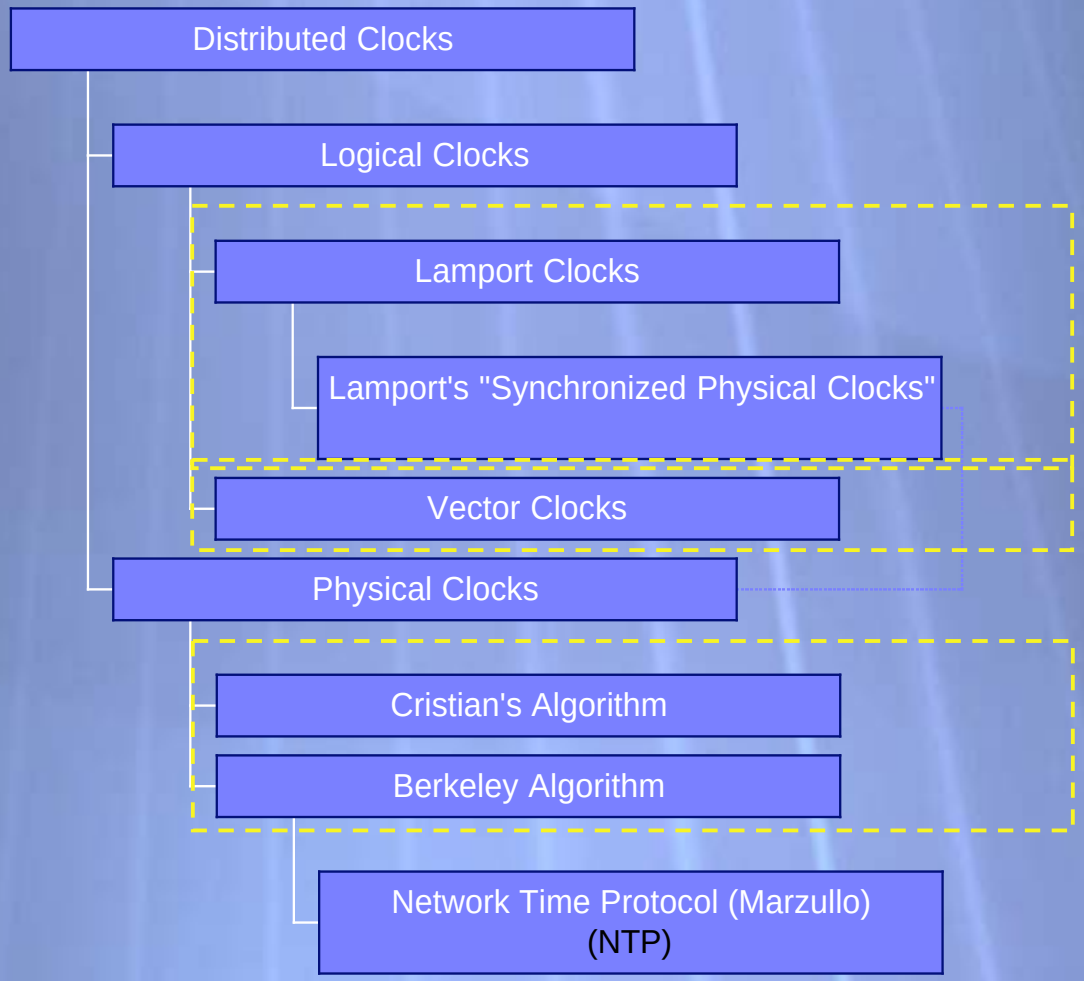
- In a distributed system (absent perfectly synchronized clocks) the concept of *happened-before* may be defined…
  - But it is fundamentally only a partial ordering
  - It can be turned into a total ordering, which is arbitrary (and thus introduces apparent inconsistencies when compared to wall-clock time)
- Logical clocks can be synchronized
  - And can be made to approximate physical clocks
  - But the synchrony is bounded
- With total ordering, it is possible to build a correct, distributed resource manager
  - Exclusion -- only one process holds resource R
  - Ordering -- resources granted in order
  - Progress -- if every resource is eventually released, every resource request is eventually granted
  - But: This breaks down in the presence of failures

# Impact of Lamport's Paper

- Total citations: 838 (ACM Portal)
- Represents the first formal steps towards development of distributed logical clocks, used in many application areas as well as formal/theoretical treatments in distributed systems
- Some aspects of this work proved less influential (e.g., distributed mutual exclusion approach; Lamport's extension to synchronized physical clocks)

# Time and Clocks in Distributed Systems

Logical Hierarchy of Distributed Clocks




1978 (Lamport)

1989 (Mattern)

1989 (Cristian)

1989 (Gusella and Zatti)

1984 (Marzullo)

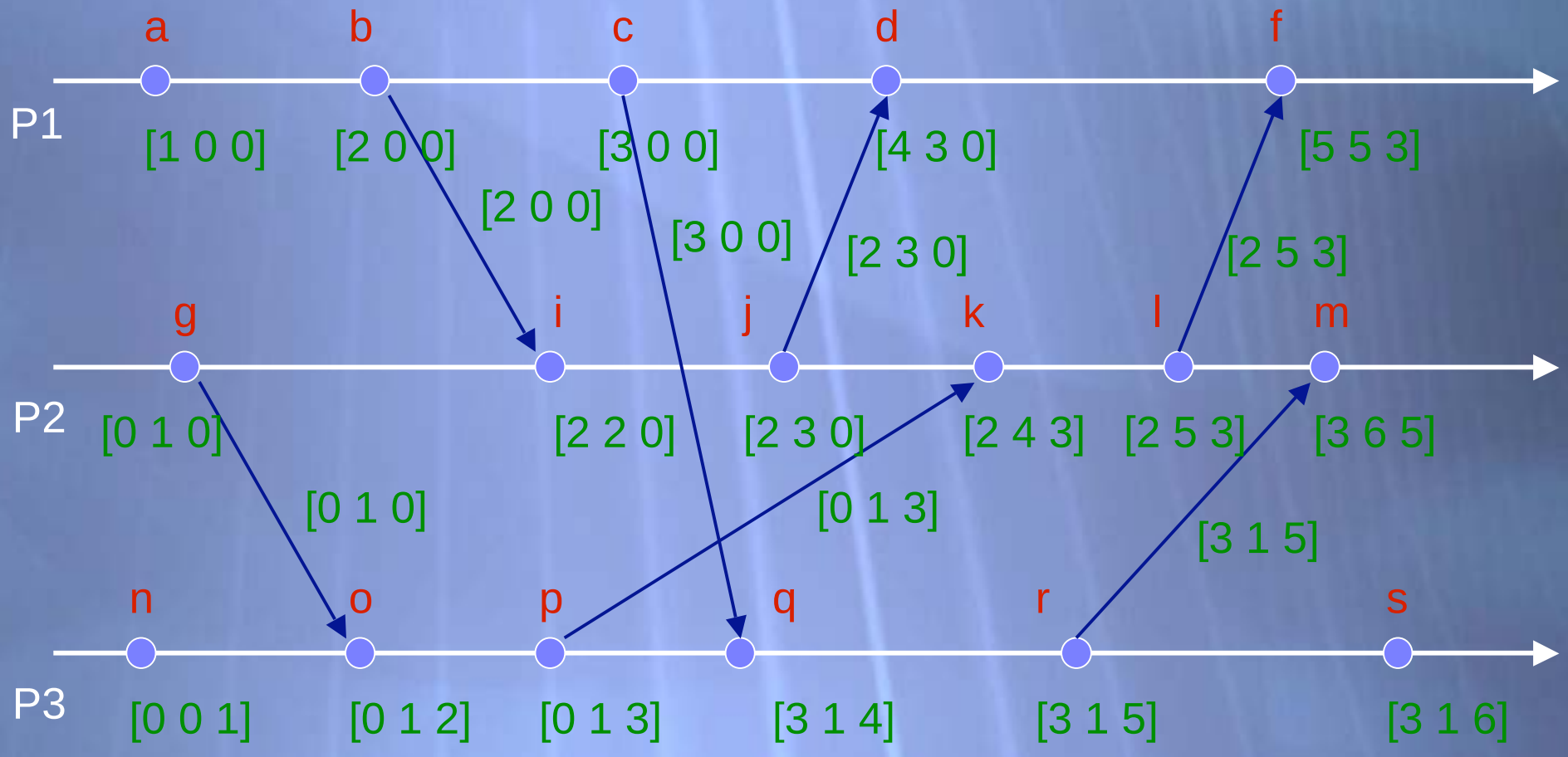
# Vector Clocks

- Vector timestamps:
  - Each node keeps track of logical time of other nodes (as far as it's seen messages from them) in $V_i[i]$
  - Send vector timestamp *vt* along with each message
  - Reconcile vectors timestamp with own vectors upon receipt using MAX(*vt*[k], $V_i$[k]) for all k
- Can implement "causal message delivery"



*This slide courtesy Prof. G. Back*

# Vector Clocks (1)




a → f

[1 0 0] < [5 5 3]

b → s

[2 0 0] < [3 1 6]

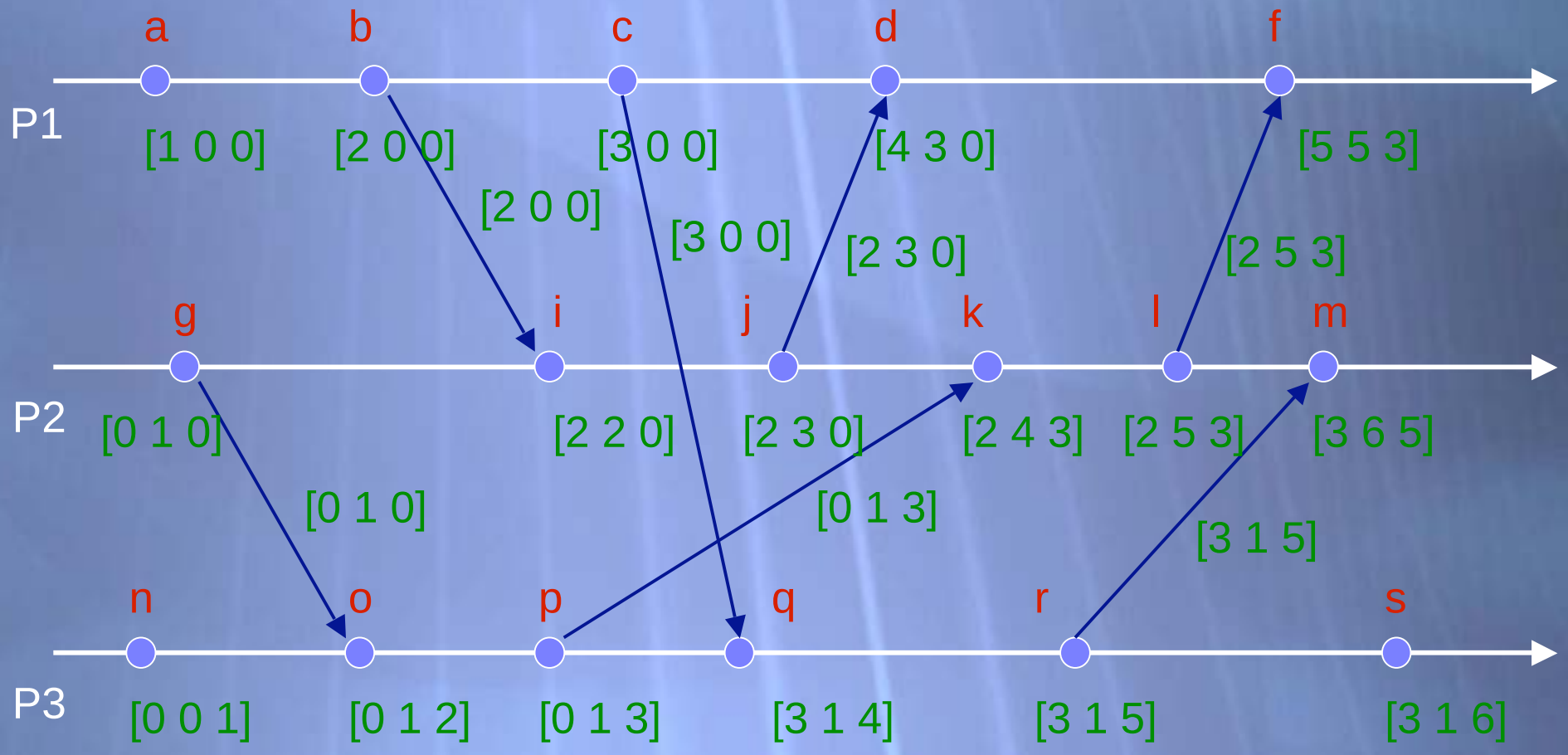
c → m

[3 0 0] < [3 6 5]



*This slide courtesy Prof. G. Back*

# Vector Clocks (2)




d || s
[4 3 0] < [3 1 6]

q || i
[3 1 4] < [2 2 0]

k || r
[2 4 3] < [3 1 5]



*This slide courtesy Prof. G. Back*

# Vector Clocks: Strong Consistency

- Definition:
  - $V(a) < V(b)$:
    $V(a) \leq V(b)$ and there exists an *i*: $V_i(a) < V_i(b)$
  - $V(a) \leq V(b)$: for all components *i*: $V_i(a) \leq V_i(b)$
- Strongly consistent:

$$a \rightarrow b \Leftrightarrow V(a) < V(b)$$

- Also:

$$a \,||\, b \Leftrightarrow V(a) \,||\, V(b)$$
$$\Leftrightarrow \neg\, (V(a) < V(b) \vee V(b) < V(a))$$



*This slide courtesy Prof. G. Back*

# Applications of Logical Clocks

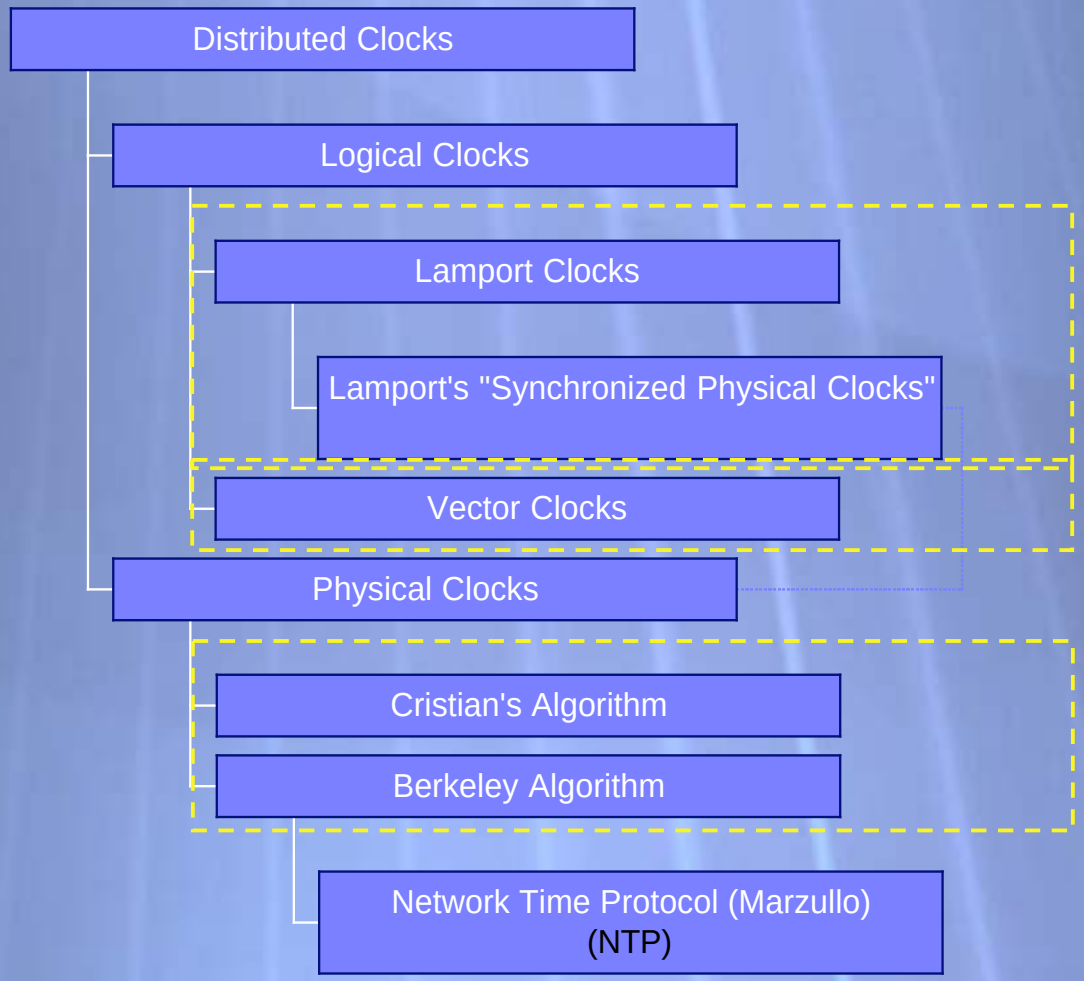
- Distributed mutual exclusion
  - Lamport's algorithm
- Totally ordered multicast
  - For updating replicas
- Causal message delivery
  - E.g., deliver message before its reply
  - message or application layer implementation
- Distributed Simulation



*This slide courtesy Prof. G. Back*

# Time and Clocks in Distributed Systems

Logical Hierarchy of Distributed Clocks




1978 (Lamport)

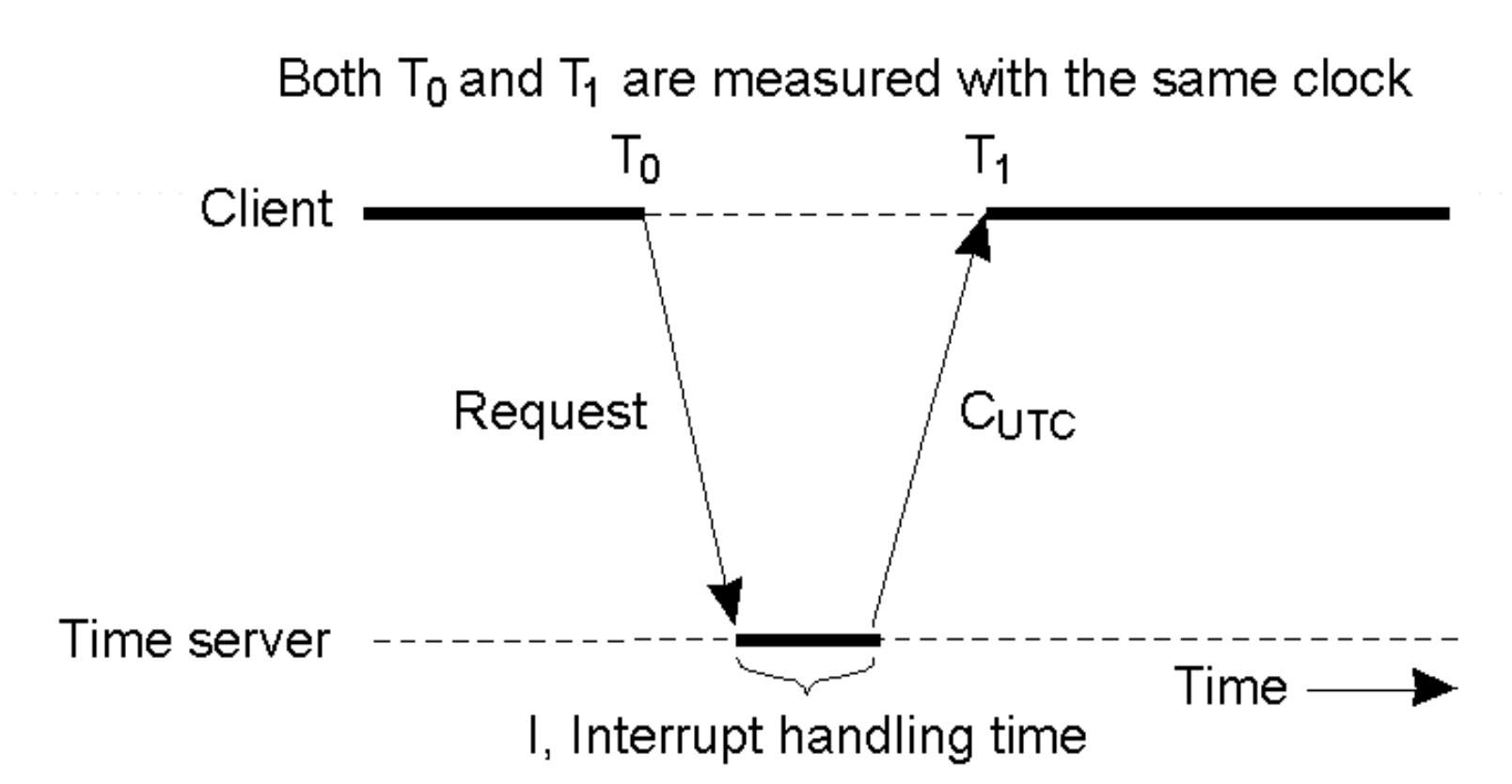
1989 (Mattern)

1989 (Cristian)

1989 (Gusella and Zatti)

1984 (Marzullo)

# Cristian's Algorithm

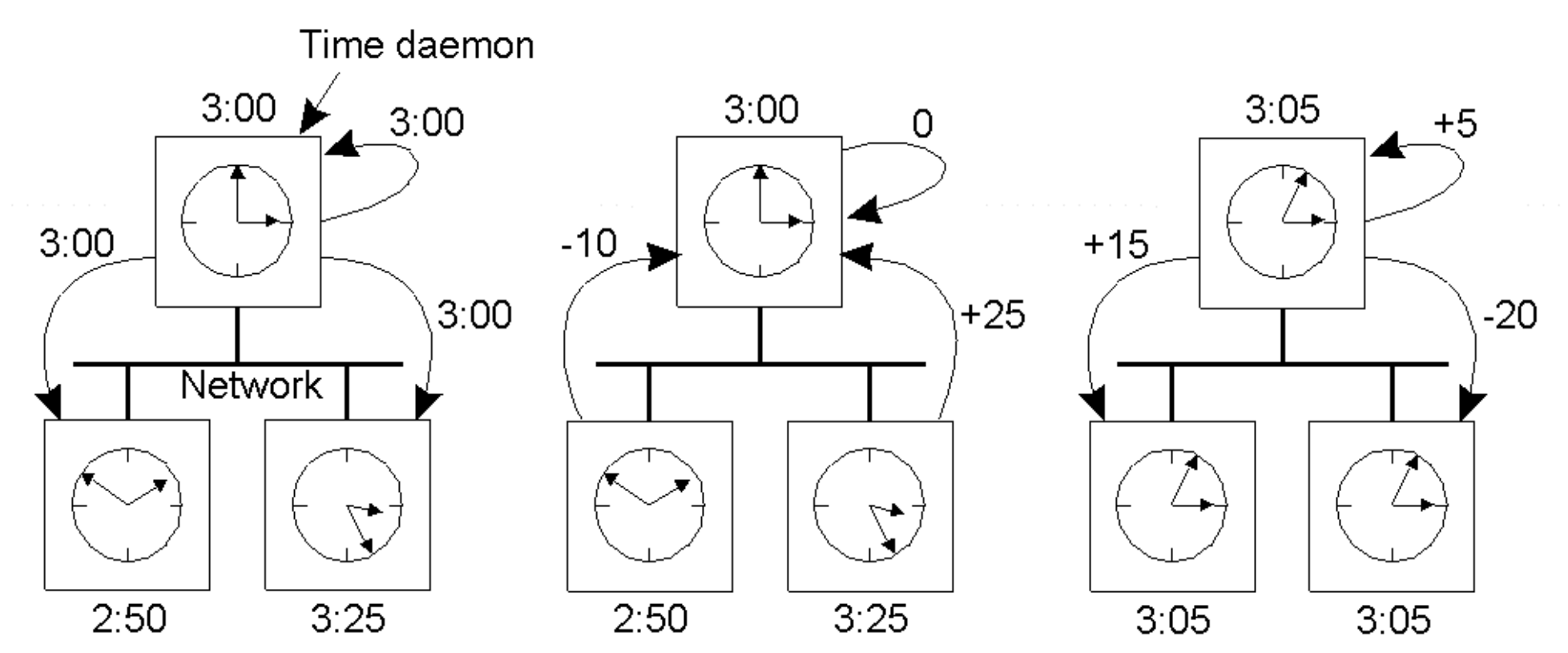



Central server keeps time, clients ask for time
Attempts to compensate for latency – component of modern NTP Protocol – accuracy 1-50ms



*This slide courtesy Prof. G. Back*

# Berkeley Algorithm




*timed* polls

nodes reply with delta

*timed* instructs nodes to adjust to 3:05

No time reference necessary



*This slide courtesy Prof. G. Back*

# Summary

- Lamport (Logical) Clocks
  - Weak consistency
  - Lightweight
  - Causal consistency
- Lamport's Extension to Physical Clocks
  - Very heavyweight (messaging)
  - Strong causal consistency
- Vector Clocks
  - Strong clock consistency (All events *happen-before*, *happen-after*, or are *concurrent*)
- Cristian's Algorithm
  - Assumes central, correct clock source
  - Compensates for latency
- Berkeley Algorithm
  - Seeks clock synchronization, not correctness
  - Also centralized -- relative skews for all clocks and mean computerd by central server

# References


[0] Leslie Lamport (1978). *Time, clocks, and the ordering of events in a distributed system*. Communications of the ACM 21 (7): 558-565.

[1] Lamport Email:
http://research.microsoft.com/users/lamport/pubs/distributed-system.txt

[2] Wolfram Mathworld:
http://mathworld.wolfram.com/PartialOrder.html

[3] Les Lamport photograph
http://en.wikipedia.org/wiki/Image:Leslie_Lamport.jpg

[4] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, Proceedings of Parallel and Distributed Algorithms. Elsevier Science Publishers, 1988.
http://citeseer.ist.psu.edu/mattern89virtual.html

[5] Les Lamport, Wikipedia
http://en.wikipedia.org/wiki/Leslie_Lamport