Distributed Systems 1: Synchronization

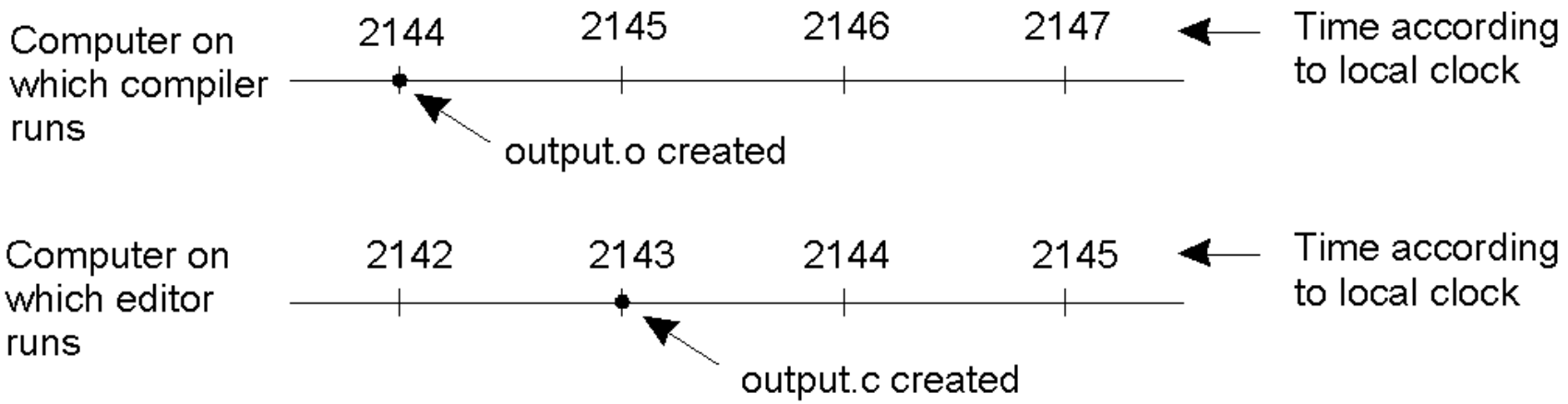
Gian Pietro Picco
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento, Italy
gianpietro.picco@unitn.it
http://disi.unitn.it/~picco

Synchronization in Distributed Systems

- The problem of synchronizing concurrent activities arises also in non-distributed systems
- However, distribution complicates matters:
  - Absence of a global physical clock
  - Absence of globally shared memory
  - Partial failures
- In these lectures, we study distributed algorithms for:
  - Synchronize (physical) clocks to approximate global time
  - Abstract time with logical clocks capturing (causal) ordering
  - Collect global state
  - Coordinate access to shared resources

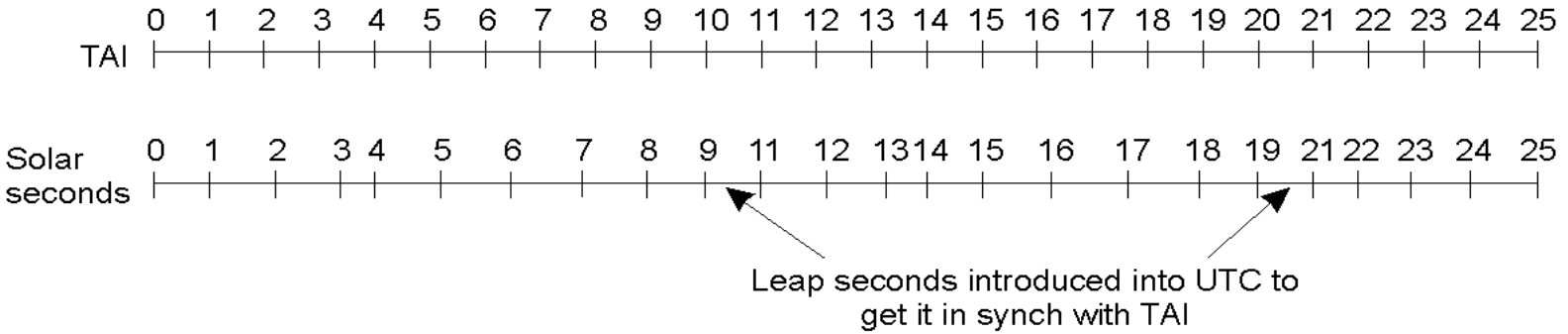
Time and Distributed Systems

- Time plays a fundamental role in many applications:
  - Execute a given action at a given time
  - Timestamping objects/data/messages enables reconstruction of event ordering
    - File versioning
    - Distributed debugging
    - Security algorithms
- Problem: ensure all machines “see” the same global time



Time

- Clocks vs. timers
- Time is a tricky issue per se:
  - Up to 1940, time is measured astronomically
    - 1 second = 1/86400th of a solar day
    - Earth is slowing down, making measures “inaccurate”
  - Since 1948, atomic clocks (International Atomic Time)
    - 1 second = 9,192,631,770 transitions of an atom of Cesium 133
    - Collected and averaged in Paris from 50 labs around the world
  - Skew between TAI and solar days accommodated by UTC (Universal Time Coordinates) when greater than 800ms
    - Greenwich Mean Time is only astronomical
    - About 30 leap seconds from 1958 to now
    - UTC disseminate via radio stations (DCF77 in Europe, WWV in US), GPS and GEOS satellite systems

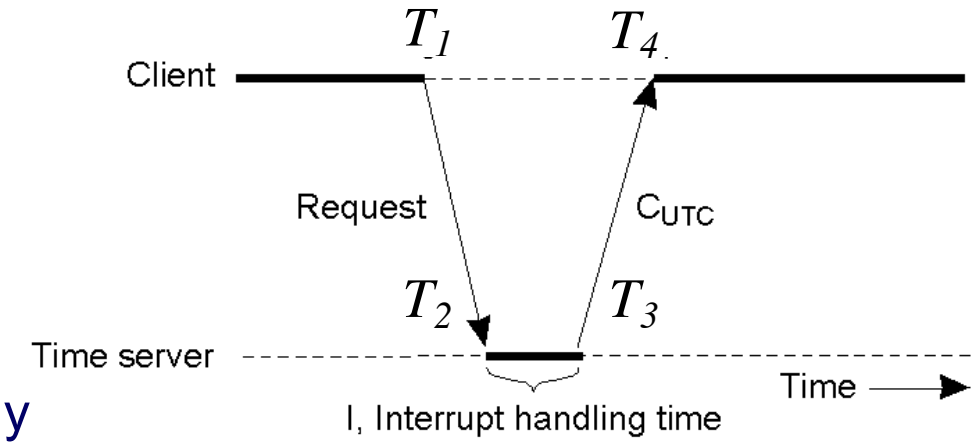


Synchronizing Physical Clocks

- To guarantee synchronization:
  - Maximum ***clock drift*** rate ρ is a constant of the timer
    - For ordinary quartz crystals, $\rho=10^{-6}$ s/s, i.e., 1s every 11.6 days
  - Maximum allowed ***clock skew*** δ is an engineering parameter
  - If two clocks are drifting in opposite directions, during a time interval Δt they accumulate a skew of $2\rho\Delta t$
    ➔ resynch needed at least every $\delta/2\rho$ seconds
- The problem is either:
  - Synchronize all clocks against a single one, usually the one with external, accurate time information (*accuracy*)
  - Synchronize all clocks among themselves (*agreement*)
- At least time monotonicity must be preserved
  - i.e., time cannot run backwards!
- Several protocols have been devised

Server-based Solutions

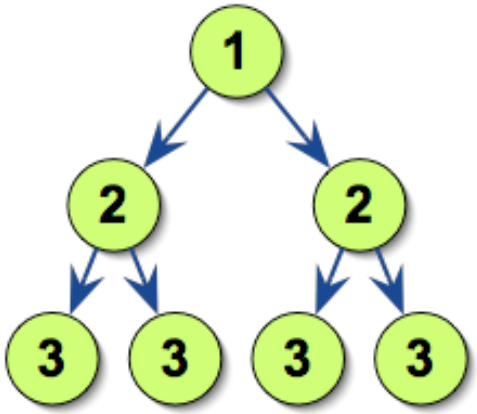
- Periodically, each client
  - sends a request to the time server, which replies with its clock
  - computes the offset θ, and
  - sets its local clock accordingly
- Assumption:
  - messages travel fast w.r.t. time accuracy
- Problems:
  - Major: time might run ***backwards*** on the client machine; therefore, introduce change gradually
    - e.g., advance clock 9ms instead of 10ms on each clock tick
  - Minor: it takes a non-zero amount of time to get the message to the server and back
    - measure round-trip time by encoding more timestamps (see below)
    - average over several measurements, …



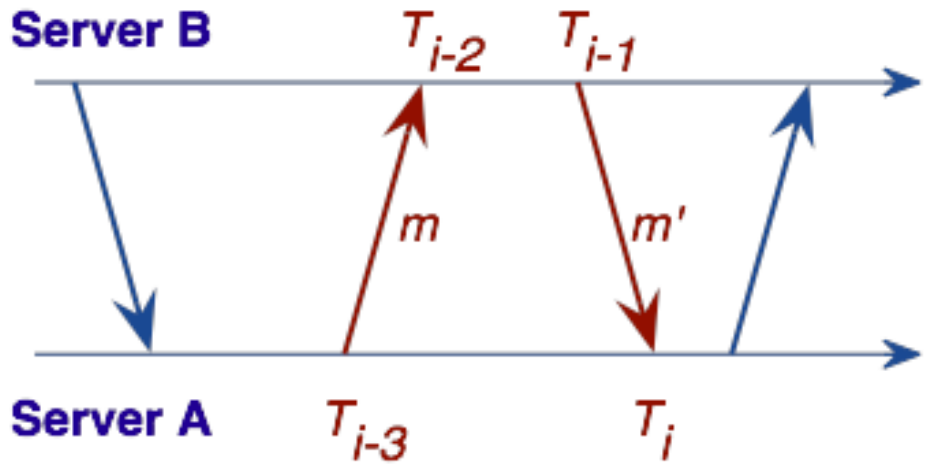
$$\theta = T_3 + \frac{(T_2 - T_1) + (T_4 - T_3)}{2} - T_4 = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Network Time Protocol (NTP)

- Designed for UTC synch over ***large-scale*** networks
  - Used in practice over the Internet, on top of UDP
  - Estimate of 10-20 million NTP clients and servers
  - Widely available on all OS
  - ~1ms over LANs, 1-50ms over the Internet
  - More info at **`www.ntp.org`**
- Hierarchical *synchronization subnet* organized in *strata*
  - Servers in stratum 1 are directly connected to a UTC source
  - Lower strata (higher levels) provide more accurate information
  - Connections and strata membership change over time
  - Servers in a given stratum synchronize among themselves
- Synchronization mechanisms
  - Multicast (over LAN)
  - Procedure-call mode (server-based)
  - Symmetric mode (for higher levels)



NTP: Symmetric Mode



- Similar to server-based, but the two nodes involved continuously exchange their roles, in a continuous message "ping-pong"
- The offset estimate is complemented by a delay estimate δ

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

- The pairs $<\theta, \delta>$ are stored and processed through a statistical filter, to obtain a measure of reliability
- Each node communicates with several others
  - Only the most reliable data is returned to applications
  - Allows to select the best primary data source

Some Observations

- In many applications it is sufficient to agree on a time, even if it is not accurate w.r.t. the absolute time
- What matters is often the ***ordering*** and ***causality*** relationships of events, rather than the timestamp itself
  - i.e., the fact that one event occurred after another, and that it may have been "caused" (or may contain information about) the previous event
- If two processes do not interact, it is not necessary that their clocks be synchronized

Modeling Distributed Executions

- A ***distributed algorithm*** is a collection of distributed automata, one per process
- The ***execution*** of a distributed algorithm (on a process) is a sequence of ***events*** executed by the process
- In distributed algorithms, relevant ***events*** are:
  - ***send*** event: the process sends a message to another process, *send(m, p)*
  - ***receive*** event: the process receives a message, *receive(m)*
  - ***local*** event: anything that changes the application state of the process
    - (e.g., setting a variable, writing a file)

Histories

- The ***local history*** of a process p_i is a (possibly infinite) sequence of events

$$h_i = e_i^0 e_i^1 e_i^2 \ldots e_i^{m_i}$$

- The ***partial history*** up to event e_i^k is denoted h_i^k and is the prefix of the first k events in h_i
- Histories are useful to formalize distributed algorithms…
- … but they do not specify any relative timing/ordering between events in different processes…
- ... that would help us determine whether they occurred one after the other, or concurrently

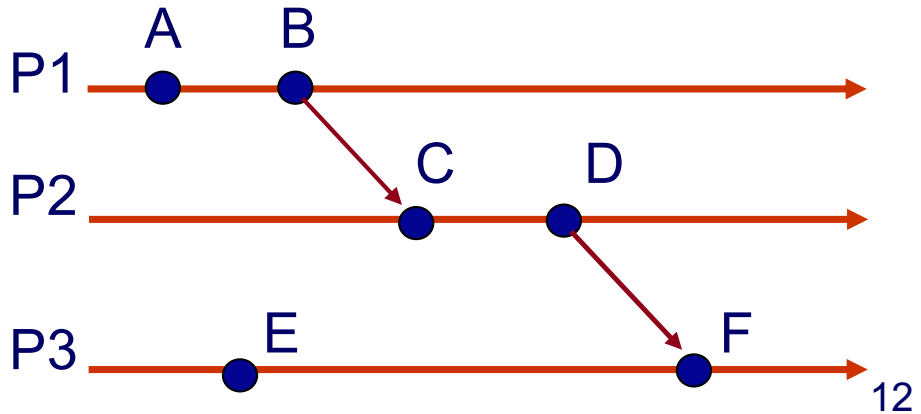
Happens-Before

Definition (Happen-before)

We say that an event e happens-before an event e', and write $e \rightarrow e'$, if one of the following three cases is true:

1. $\exists p_i \in \Pi : e = e_i^r, \quad e' = e_i^s, \quad r < s$
   (e and e' are executed by the same process, e before e')
2. $e = send(m, *) \wedge e' = receive(m)$
   (e is the send event of a message m and e' is the corresponding receive event)
3. $\exists e'' : e \rightarrow e'' \rightarrow e')$
   (in other words, $\rightarrow$ is transitive)

Two events e, e' for which happens-before does not hold $(e \not\rightarrow e' \wedge e' \not\rightarrow e)$ are called ***concurrent***, $e||e'$



What Does It Mean?

If $e \rightarrow e'$, this means that we can find a series of events $e^1 e^2 e^3 \ldots e^n$, where $e^1 = e$ and $e^n = e'$, such that for each pair of consecutive events e^i and e^{i+1}:

1. e^i and e^{i+1} are executed on the same process, in this order, or
2. $e^i = send(m, *)$ and $e^{i+1} = receive(m)$

- The happens-before relationship captures:
  - a flow of data between two events
  - the notion of (partial) ***causal ordering*** of events
- It is a key concept in concurrent systems
  - E.g., the multi-threaded memory model of Java is defined based on happens-before, where "events" are read/write of variables and acquisition/release of locks
    http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

Logical Clocks

- Enable coordination among processes without synchronization of physical clocks
  - which can be difficult and/or costly to achieve
- Every process has a logical clock
  - essentially a counter, which is incremented using a well-defined set of rules…
  - … and whose value has no relationship with a physical clock whatsoever
- Distributed algorithms exploit logical clocks to timestamp events
  - The causality relation between events is then inferred from their timestamps
- Different notions of logical clocks exist, with different rules and expressiveness

Logical Clocks: Definitions

Definition (Logical clock)

A logical clock LC is a function that maps an event e from the history H of a distributed system execution to an element of a time domain T:

$$LC : H \rightarrow T$$

This mapping enables timestamping of events, if the following holds…

Definition (Clock Consistency)

$$e \rightarrow e' \Rightarrow LC(e) < LC(e')$$

… which states that if two events happen one after the other, their timestamps respect time monotonicity

Definition (Strong Clock Consistency)

$$e \rightarrow e' \Leftrightarrow LC(e) < LC(e')$$

If this stronger property holds, we can also reconstruct the ordering of events given the clocks

Scalar Clocks

L. Lamport. "Time, clocks, and the ordering of events in a distributed system". *Communications of the ACM*, 21(7):558-565, July 1978.

- The question then becomes: how to assign logical clocks in a way that guarantees clock consistency?

Definition (Scalar logical clocks)

- A Lamport's scalar logical clock is a monotonically increasing software counter
- Each process p_i keeps its own logical clock LC_i
- The timestamp of event e executed by process p_i is denoted $LC_i(e)$
- Messages carry the timestamp of their *send* event
- Logical clocks are initialized to 0

guarantees clock consistency by design

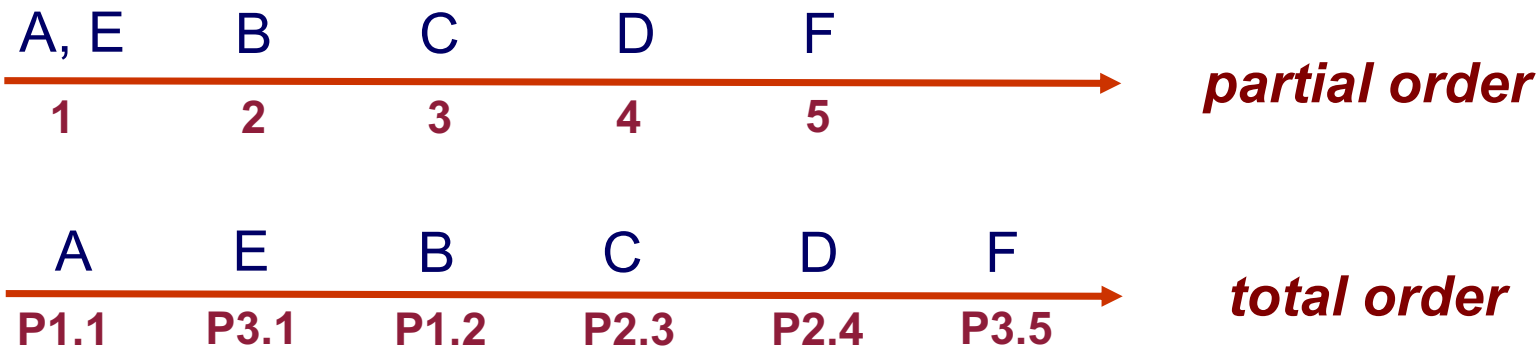
Update rule

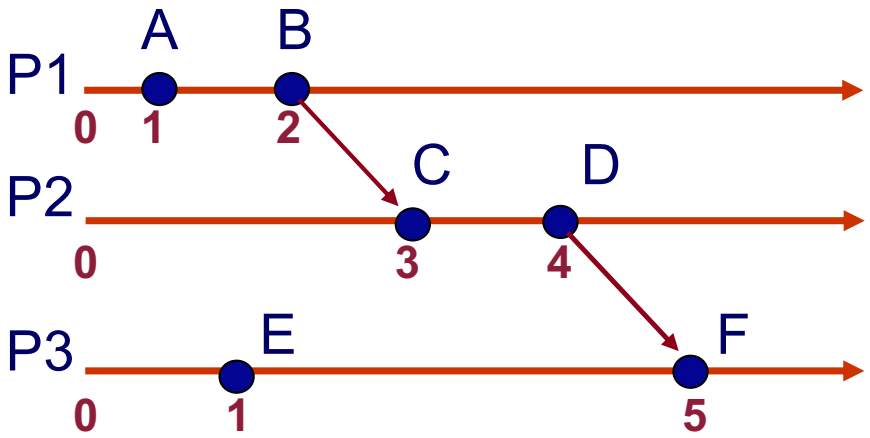
Whenever an event e is executed by process p_i, its local logical clock is updated as follows:

$$LC_i = \begin{cases} LC_i + 1 & \text{If } e_i \text{ is an internal or } send \text{ event} \\ \max\{LC_i, TS(m)\} + 1 & \text{If } e_i = receive(m) \end{cases}$$

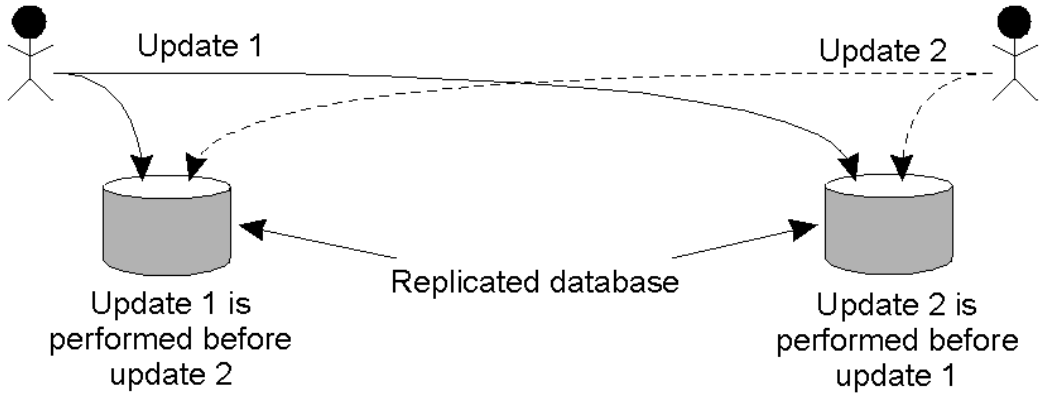
Partial vs. Total Order

- The previous strategy achieves only partial ordering
- ***Total ordering*** can be obtained trivially by attaching process IDs to clocks
  - Assuming an ordering (e.g., lexicographic) holds among the IDs





Example:Totally Ordered Multicast



- Updates in sequence:
  - Customer deposits $100
  - bank adds 1% interest
- Updates are propagated to all locations:
  - If updates arrive in the same order at each copy, consistent result:
  - If updates arrive in a different order, inconsistent result: