The UNIVERSITY *of* EDINBURGH

SCHOOL *of* INFORMATICS

CS4/MSc

Distributed Systems

Björn Franke

bfranke@inf.ed.ac.uk

Room 2414

(Lecture 8: Physical and Logical Clocks, 16th October 2006)

Time and Coordination

- Both in real life and computer systems, we need to measure time to describe when and in what order events occur and how long they take. Clocks and timers are used for this purpose.
- In distributed systems, the coordination of many activities relies on the measurement of time and synchronisation. Consider, for example, maintaining the consistency of distributed data, checking the expiration time of a received message, triggering events such as file system backup at a certain point in the day.
- In order to measure when an event occurred, computers can synchronise their clocks with an external authoritative source of time (*external synchronisation*). In order to measure the interval between the events occurring at different computers *internal synchronisation* of local clocks can be sufficient.

Physical Clocks

A computer clock is an electronic device which counts oscillations of a crystal occurring at a well-defined frequency.

- Hardware activities are coordinated using physical clocks.
- Clocks can be programmed to generate *interrupts* (clock ticks) at certain intervals which can be used to coordinate software activities.
- A software clock is maintained by the operating system from the hardware clock $C(t) = \alpha H(t) + \beta$. The *clock resolution* is the period between updates of the clock value and should be less than the period between events.
- The difference between any two clocks is known as their *skew*.
- The frequency of oscillation depends on the properties of the crystal and the tension under which it oscillates which leads to clock *drifts*.

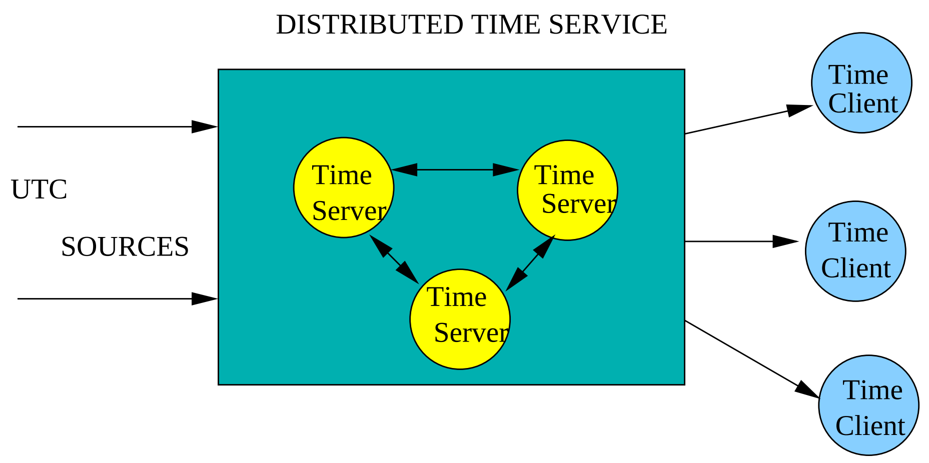
It is impossible to obtain an absolute global physical time.

Coordinated Universal Time

- Clocks which use atomic oscillators measure the time based on the transitions of the Caesium 133 atom and are more accurate than crystal-based clocks. The drift rate is approximately one part in 10^{13}.
- The time units we normally use are based on the astronomical time. However, the earth's rotation around its own axis is known to be slowing down making the days longer.
- *Coordinated Universal Time* (UTC) is an international standard based on atomic time, which introduces leap seconds to keep astronomical and atomic time in step.
- There are many UTC sources and the National Institute of Standards and Technology supports several UTC access methods with an accuracy of 10 milliseconds. Computers with receivers attached can synchronise their clocks with these timing signals.

A Distributed Time Service Architecture

The UTC sources include a dial-up modem service called the *Automated Computer Time Service*, a short-wave radio station WWV, Global Positioning System (GPS) satellites and Geostationary Environmental Satellites (GEOS).



Signal propagation delay and network communication delay are important factors to be considered when reporting UTC between clients and time servers and between the time servers themselves.

External and Internal Synchronisation

External Synchronisation A clock is considered to be externally synchronised if it has been synchronised with an authoritative external source of time (a UTC source). A system is externally synchronised if all clocks within it are externally synchronised.

Internal Synchronisation A system is internally synchronised if the clocks within it are synchronised with one another to a known degree of accuracy.

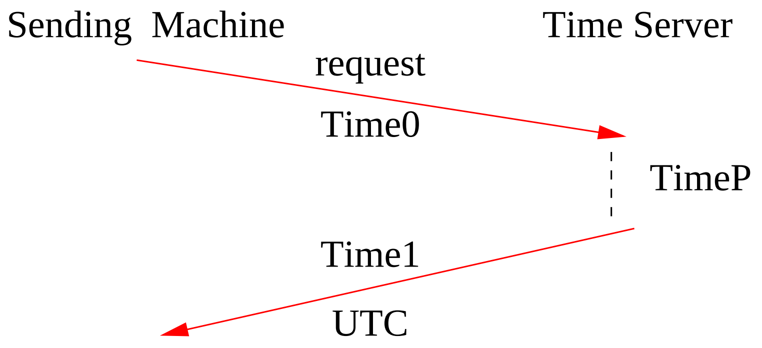
For an N processor system, with clocks $C_i,\ i = 1, \ldots N$, and external time source S, external synchronisation can be expressed as $|S(t) - C_i(t)| < D, \quad \text{for } i = 1, \ldots, N$ whereas internal synchronisation is expressed as $|C_i(t) - C_j(t)| < D, \quad \text{for } i, j = 1, \ldots, N$

An internally synchronised system is not necessarily externally synchronised, since the clocks may show collective drift. However an externally synchronised system will be internally synchronised.

External Synchronisation (Pull Model)

In this model, clients request the UTC from a time server. The time server is passive.

Cristian's algorithm



Assuming that the network traffic is symmetrical, the best estimate for one-way propagation time is $(Time1+Time0+TimeP)/2$. The accuracy of this estimate can be calculated if the minimum propagation time is known.

This is a suitable technique when the round-trip times between client and server are short compared with the desired accuracy.

Compensating for Clock Drift

After the client receives the UTC, it needs to adjust its local clock gradually. There are two possibilities:

- Local clock faster than UTC
  - time cannot go backward,
  - clock speed is slowed down by software. For example, interrupt routine adds only 9 milliseconds to the software clock if it was normally adding 10 milliseconds before.
- Local clock is slower than UTC
  - increase clock speed

Note that abrupt changes could cause problems. It is important to maintain *monotonicity*.

Internal Synchronisation (Push Model)

In this model, the time server is active. It polls every machine whose clocks need to be synchronised.

Berkeley algorithm

1. The time daemon of the time server announces to other machines its time and asks for their local times.
2. The machines respond to the time daemon and tell how far ahead or behind they are from the time of the daemon.
3. The time daemon computes the average and tells each machine how to adjust their clocks.

Consistency of Time Servers

Time servers are susceptible to inconsistencies in their UTCs due to communication delays. In order to reduce the discrepancy between their UTCs, time servers

- exchange their UTCs periodically by using pull or push models and collect UTCs from other servers; and
- by using a previously agreed decision criterion they adjust their own UTC. For example by taking the average of the UTCs they have collected.

In order to achieve higher accuracy, time servers may report an interval of time (UTC + indicator of inaccuracy) and the averaging procedure can be modified to take this into account.

Physical vs. Logical Clocks

- Physical clocks require a close approximation to the global real-time and are necessary to determine when an event occurs.
- Events need not always be scheduled or synchronised with respect to real-time. Sometimes, it is the order of events which matters.
- In a distributed system, we cannot rely on physical time to determine the order of events since we cannot synchronise physical clocks perfectly. Lamport introduced the concept of logical clock as a solution to this problem.
- Instead of aiming for absolute time ordering by means of physical clocks, processes can use logical clocks to agree on a relative ordering of events. Logical clocks are based on artificial time rather than real-time.

Happened-before Relation

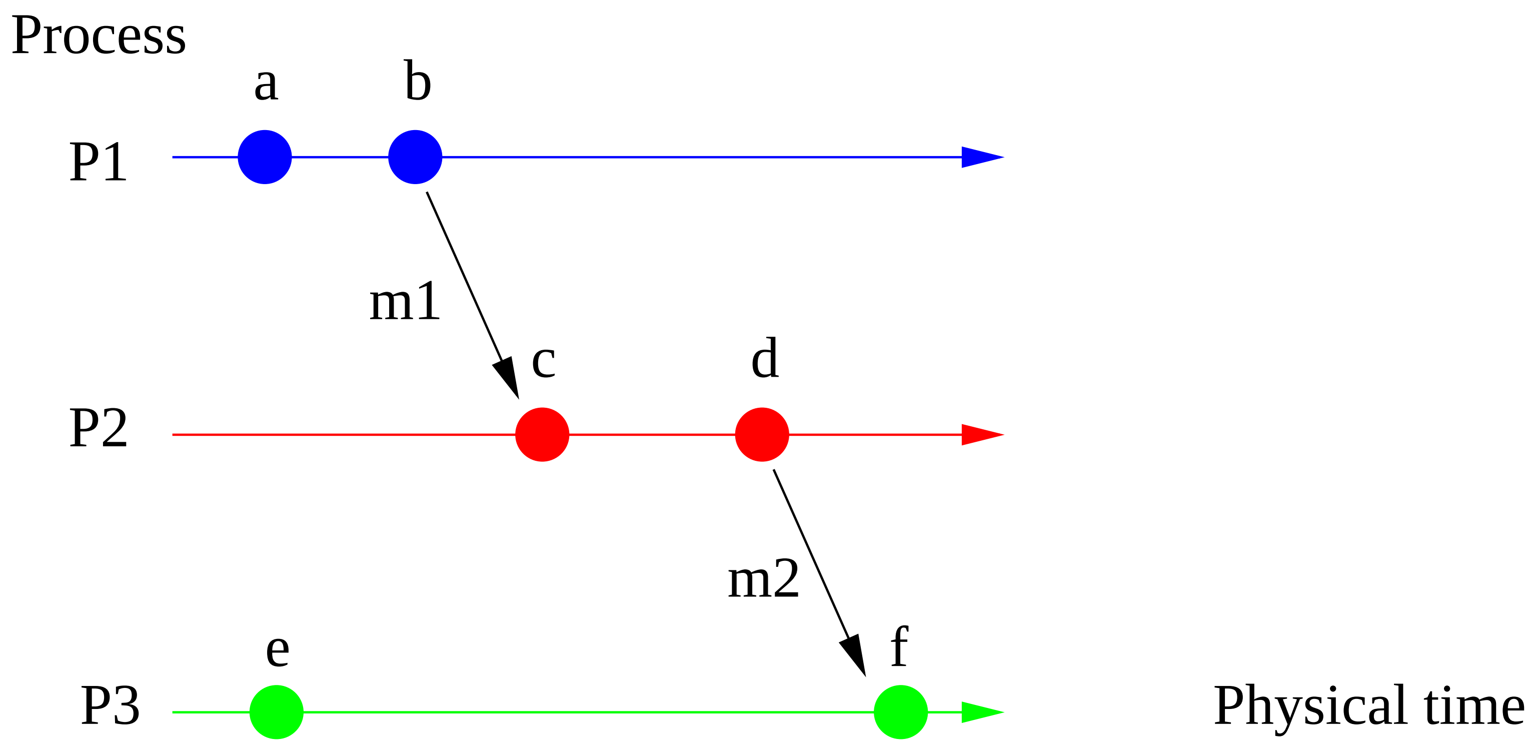
The **Happened-before** relation for distributed systems, also known as causal ordering or potential causal ordering relation, is a generalisation of the following points:

- If two events occur at the same process, then they occur in the order they were executed by the process.
- If a message-passing interaction occurs between two processes, the event of sending occurs before the event of receiving.

The happened-before relation is denoted by $\rightarrow$ where $e \rightarrow e'$ means event e happened before event e'. Note that $\rightarrow$ is a transitive relation: if $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$.

Note that this is a *partial* order, meaning that not all events will be related. Events that are not ordered by the relation are said to be *concurrent*, denoted by $e \parallel e'$.

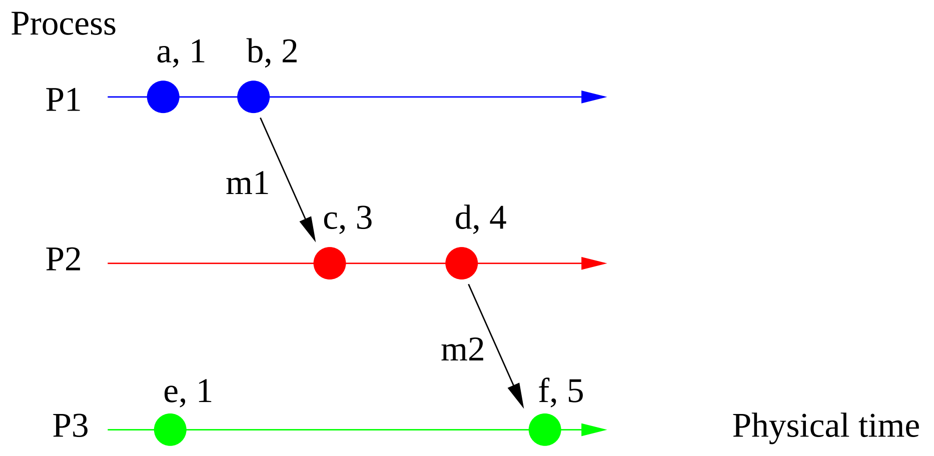
Example of the Happened-before relation



The following are examples of what we can infer from the figure:
$a \rightarrow b$, $c \rightarrow d$, $b \rightarrow c$, $d \rightarrow f$, $a \rightarrow f$. $a \parallel e$.

Assignments of Logical Clock Time

Lamport's logical clock is based on the idea of capturing the happened-before relation numerically.



1. L_i is incremented before each event is issued at process P_i
2. (a) When P_i sends a message m it piggybacks on m the value $t = L_i$.
   (b) On receiving (m, t) P_j computes $L_j := \max(L_j, t)$ and then increments L_j before timestamping the receive event.

Partial and Total Ordering

- The happened-before relation describes causality and imposes only a partial order on events. The events a and e in our example are not ordered; they are assigned identical logical timestamps. Note also that if event e and e' have timestamps $L(e) < L(e')$ it does not imply $e \to e'$.
- Total order of events can be obtained by imposing that all pairs of concurrent events are ordered: for all events a and b, $L(a) \neq L(b)$. This is achieved by making the timestamp a pair consisting of the local timestamp and the process number: $L(e) = (L_i(e), i)$, if e occurs on process i at local (logical) time $L_i(e)$.
- Now we can distinguish two events occurring at different processes. We define $(L_i, i) < (L_j, j)$ if $L_i < L_j$ or $L_i = L_j$ and $i < j$.
- The fact that $L(e) < L(e')$ does not lead to the conclusion that $e \to e'$ was regarded as a shortcoming by some and led to the development of *vector logical clocks*.

Vector Logical Clocks

- Vector logical clocks provide the means for deciding whether two events are causally ordered or concurrent.
- Each process P_i maintains a vector of logical clocks, with one entry for each process in the system. It uses this to timestamp each event e: $V_i(e) = [t_1, t_2, \ldots, L_i(e), \ldots t_N]$.
  - N : number of cooperating processes.
  - $L_i(e)$: logical clock time of event e at process P_i.
  - t_k : best estimate of the logical clock time for process P_k.
- As with Lamport timestamps, processes piggyback vector timestamps onto messages to each other in order to allow for clock updates.

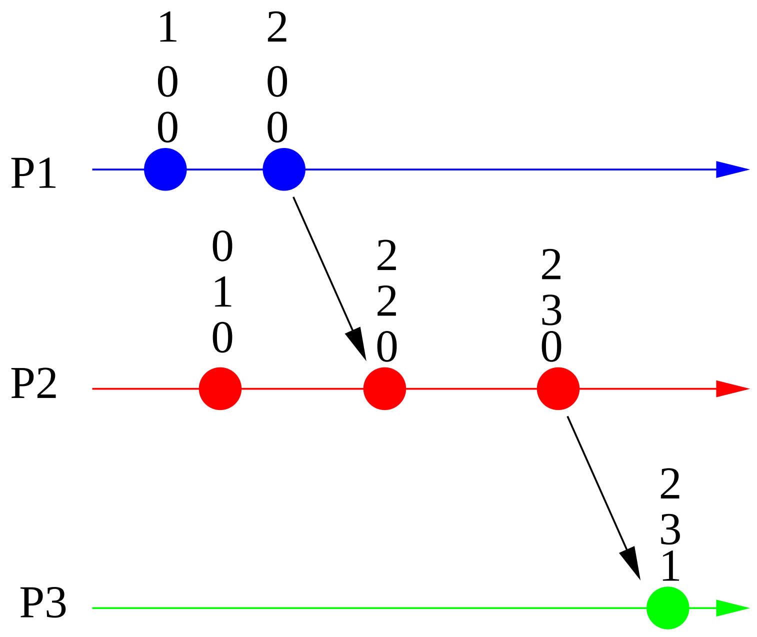
Intuition: $V_i[i]$ is P_i's local logical clock and $V_i[j]$ is P_i's latest knowledge of P_j's logical clock.

Implementing Vector Logical Clocks

1. Initialise all vectors to 0: $V_i[j] = 0$ for all $i, j = 1, \ldots, N$
2. Process P_i increments its local entry $V_i[i]$ just before executing an event: $V_i[i] := V_i[i] + 1$.
3. (a) When P_i send a message m it piggybacks $t = V_i$ (the whole vector) on to it.
   (b) On receiving (m, t) the process P_j updates its logical clock vector $V_j(k) := \max(V_j[k], t_j[k])$ for all $k = 1, \ldots, N$.
   (c) P_j then increments $V_j[j]$ before issuing the receive event.

For a vector clock V_i, $V_i[i]$ is the number of events that P_i has timestamped, and $V_i[j]$ is the number of events that have occurred at P_j that P_i has potentially been affected by.

Example



Given two events a and b with vector timestamps V_i and V_j we can determine the causal order between e and e' because

- $e \rightarrow e'$ if and only if $V_i < V_j$.
- e and e' are concurrent if and only if neither $(V_i < V_j)$ nor $(V_j < V_i)$.