Distributed coordination

Distributed Coordination

- **Event Ordering**
- Mutual Exclusion
- Atomicity
- Concurrency Control
- Deadlock Handling
- Election Algorithms
- **Reaching Agreement**
  - **Fundamental problem that need to be solved under many different conditions**

➢ *Time and State in Distributed Systems*

- ➢ 1. Virtual Time in Distributed Systems
- ➢ 2. Lamport’s Logical Clocks
  - Solve the problem of Event ordering

Time in Distributed Systems

- Because each machine in a distributed system has its own clock there is no notion of *global physical time*

- The *n* oscillators on the *n* computers will run at slightly different rates (clock drift), causing the clocks gradually to get out of synchronization and give different values

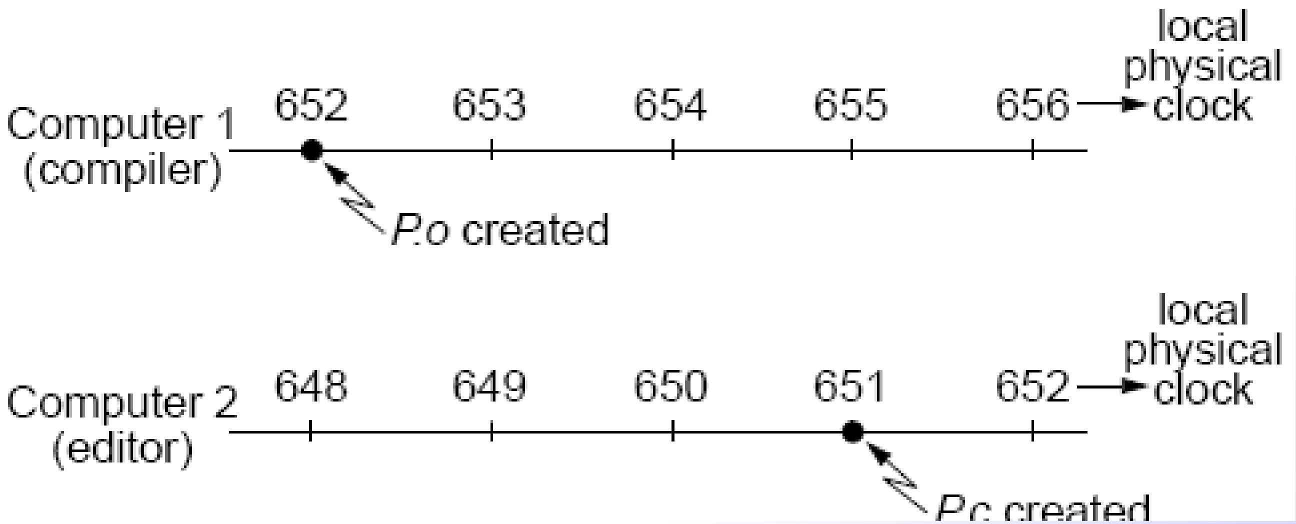
Problems:

- Time triggered activities: activities scheduled to occur at predefined moments in time. If such activities are to be coordinated over a distributed system we need a coherent notion of time.

- Maintaining the consistency of distributed data is often based on the *time* when a certain modification has been performed.

Time in Distributed Systems

- The *make*-program example
- • When the programmer has finished changing some source files he starts *make*; *make* examines the times at which all object and source files were last modified and decides which source files have to be (re)compiled.



- Although *P.c* is modified after *P.o* has been generated, because of the clock drift the time assigned to *P.c* is smaller. *P.c* will not be recompiled for the new version!

Time in Distributed Systems

Solutions:

- Synchronization of physical clocks
  - Computer clocks are synchronized with one another to an achievable, known, degree of accuracy ⇒ within the bounds of this accuracy we can coordinate activities on different computers using each computer's local clock.
- Physical clock synchronization is needed for distributed real-time cyber physiscal systems (we will see this later).

- Logical clocks
  - In many applications we are not interested in the physical time at which events occur; ONLY the *relative order* of events is important !
    (e.g. the *make*-program example)
  - In such situations we don't need synchronized physical clocks. Relative ordering can be based on a virtual notion of time - *logical time*.
  - Logical time is implemented using *logical clocks*.

Lamport's Logical Clocks

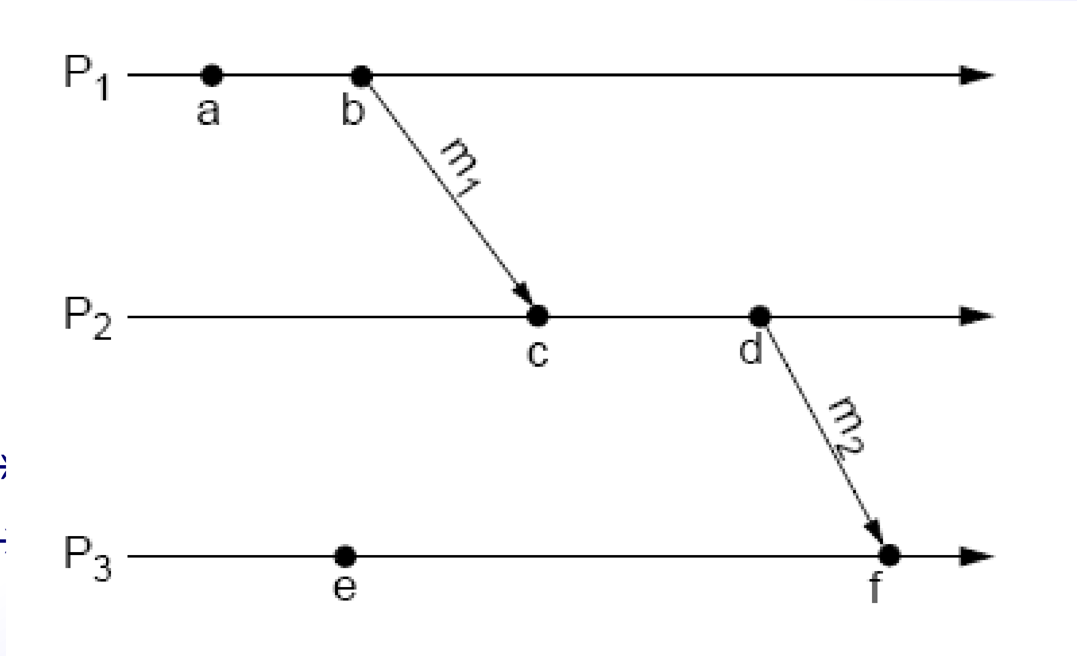
- - The order of events occurring at different processes is critical for many distributed applications.
  - (example: *P.o_created* and *P.c_created*)

- - Ordering can be based on two simple situations:
  - 1. If two events occurred in the same process then they occurred in the order observed following the respective process;
  - 2. Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving it.

- - Ordering by Lamport is based on the *happened before relation* (denoted by →):
  - • *a* → *b*, if *a* and *b* are events in the same process and *a* occurred before *b*;
  - • *a* → *b*, if *a* is the event of sending a message *m* in a process, and *b* is the event of the same message *m* being received by another process;
  - • If *a* → *b* and *b* → *c*, then *a* → *c* (a transitive relation).

Lamport's Logical Clocks

- • If $a \to b$, we say that event *a* causally affects event *b*. The two events are causally related.
- • There are events which are not related by the *happened-before* relation. If *both* $a \to e$ *and* $e \to a$ *are false*, then *a* and *e* are concurrent events; we write $a \parallel e$.



- *P1*, *P2*, *P3*: processes;
- *a*, *b*, *c*, *d*, *e*, *f*: events;
- $a \to b$, $c \to d$, $e \to f$, $b \to c$, $d \to$
- $a \to c$, $a \to d$, $a \to f$, $b \to d$, $b \to$
- $a \parallel e$, $c \parallel e$, ...

Lamport's Logical Clocks

- \- Using physical clocks, the happened before relation can not be captured. It is possible that $b \rightarrow c$ and at the same time $Tb > Tc$ (Tb is the physical time of b).
- \- *Logical clocks* can be used in order to capture the *happened-before* relation.
- • A logical clock is a monotonically increasing software counter.
- • There is a logical clock CPi at each process Pi in the system.
- • The value of the logical clock is used to assign *timestamps* to events. $CPi(a)$ is the timestamp of event a in process Pi.
- • There is no relationship between a logical clock and any physical clock.

- To capture the happened-before relation, logical clocks have to be implemented so that if $a \rightarrow b$, then $C(a) < C(b)$

Lamport's Logical Clocks - implementation

- Implementation of logical clocks is performed using the following rules for updating the clocks and transmitting their values in messages:

- [R1]: *CPi* is incremented before each event is issued at process *Pi*:

$$CPi := CPi + 1.$$

- [R2]:
- a) When *a* is the event of sending a message *m* from process *Pi*, then the timestamp *tm* = *CPi(a)* is included in *m* (*CPi(a)* is the logical clock value obtained after applying rule R1).
- b) On receiving message *m* by process *Pj*, its logical clock *CPj* is updated as follows: *CPj* := max(*CPj*, *tm*).
- c) The new value of *CPj* is used to timestamp the event of receiving message *m* by *Pj* (applying rule R1).

Lamport's Logical Clocks - implementation

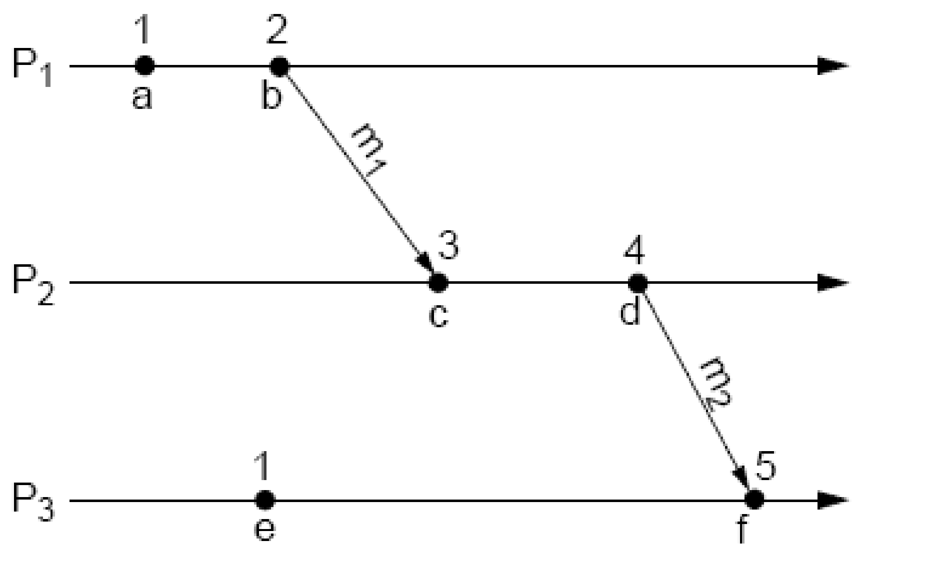
- • If *a* and *b* are events in the same process and *a* occurred before *b*, then $a \rightarrow b$, and (by R1) $C(a) < C(b)$.

- • If *a* is the event of sending a message *m* in a process, and *b* is the event of the same message *m* being received by another process, then $a \rightarrow b$, and (by R2) $C(a) < C(b)$.

- • If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$, and (by induction) $C(a) < C(c)$.

Lamport's Logical Clocks

- For the *make*-program example we suppose that a process running a compilation notifies, through a message, the process holding the source file about the event *P.o created* ⇒ a logical clock can be used to correctly timestamp the files.



Problems with Lamport's Logical Clocks

- Lamport's logical clocks impose only a partial order on the set of events; pairs of distinct events generated by *different* processes can have identical timestamp.

- For certain applications a total ordering is needed; they consider that no two events can occur at the same time.

- In order to enforce total ordering a ***global logical timestamp*** is introduced:
  - the global logical timestamp of an event *a* occurring at process *Pi*, with logical timestamp *CPi(a)*, is a pair (*CPi(a)*, *i*), where *i* is an identifier of process *Pi*;

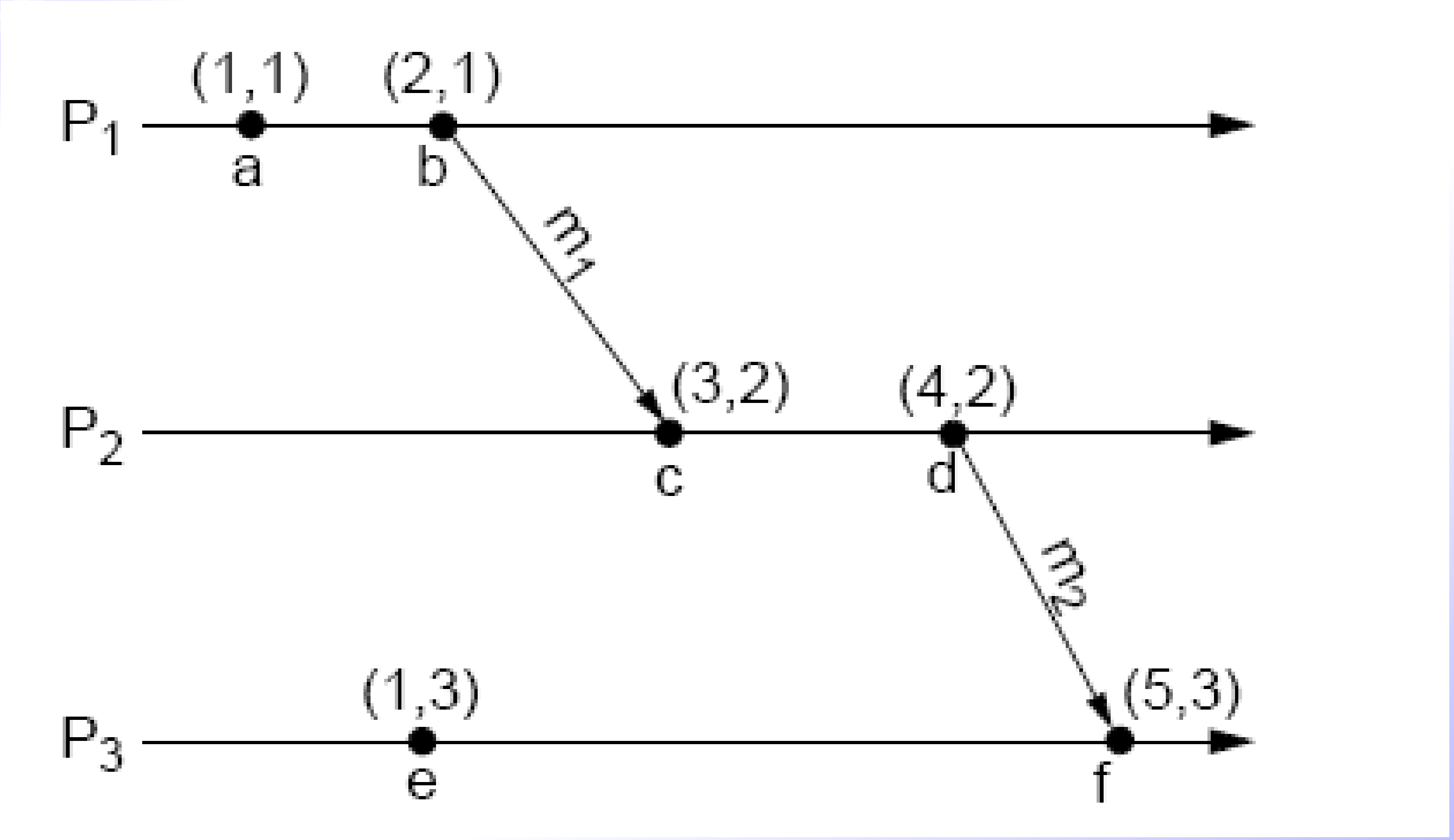
  we define

(*CPi(a)*, *i*) < (*CPj(b)*, *j*) if and only if

CPi(a) < *CPj(b)*, or *CPi(a)* = *CPj(b)* and *i* < *j*.

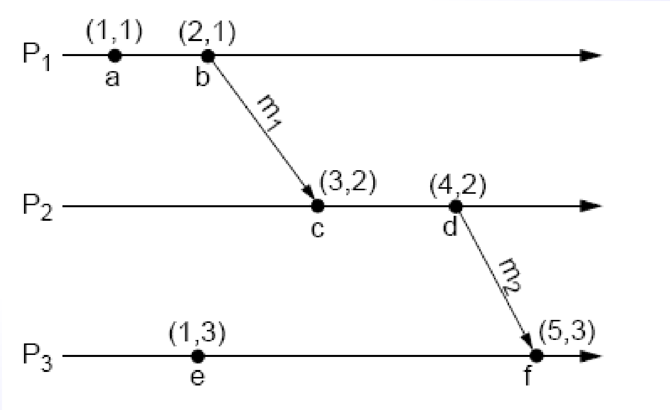
Global Logical Timestamps

- Example of timestamping with Global Logical Timestamps



Problems with Lamport's Logical Clocks

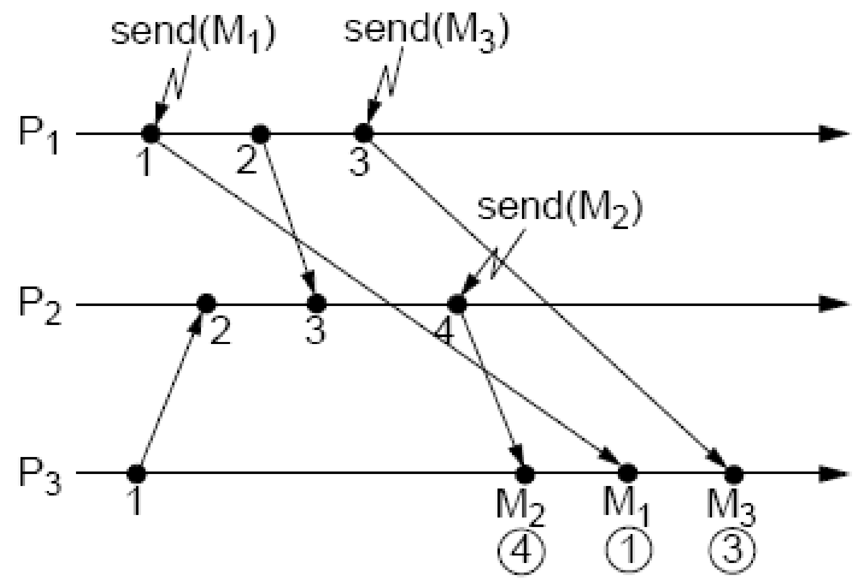
- Lamport's logical clocks are not powerful enough to perform a **causal ordering of events**.
  - if $a \rightarrow b$, then $C(a) < C(b)$
- However, the reverse is not always true (if the events occurred in different processes): if $C(a) < C(b)$, then $a \rightarrow b$ is not necessarily true. (it is only guaranteed that $b \rightarrow a$ is not true).



- $C(e) < C(b)$, however there is no causal relation from event *e* to event *b*.
- • By just looking at the timestamps of the events, we cannot say whether two events are causally related or not.

Problems with Lamport's Logical Clocks - qui

- We would like messages to be processed according to their causal order.
- Process *P3* receives messages *M1*, *M2*, and *M3*. *M1* → *M2*, *M1* → *M3*, *M3* || *M2*
- *M1* has to be processed before *M2* and *M3*. However *P3* has not to wait for *M3* in order to process it before *M2* (although *M3*'s *logical clock timestamp* is smaller than *M2*'s).



Vector Clocks

- - Vector clocks give the ability to decide whether two events are causally related or not by simply looking at their timestamp.
- • Each process *Pi* has a clock *Cv Pi*, which is an integer vector of length *n* (*n* is the number of processes).
- • The value of *Cv Pi* is used to assign timestamps to events in process *Pi*. *CvPi(a)* is the timestamp of event *a* in process *Pi*.
- • *CvPi*[*i*], the *i*th entry of *Cv Pi*,corresponds to *Pi*'s own logical time.
- • *CvPi*[*j*], $j \neq i$, is *Pi*'s "best guess" of the logical time at *Pj*.

- *CvPi*[*j*] indicates the (logical) time of occurrence of the last event at *Pj* which is in a *happened before* relation to the current event at *Pi*.