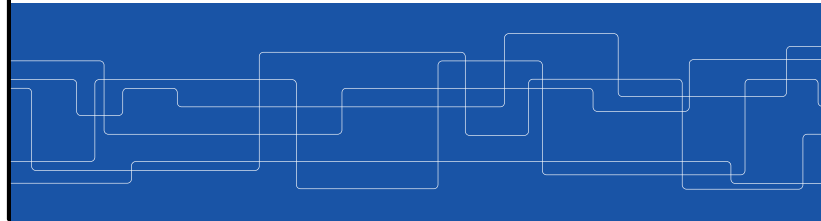





KTH ROYAL INSTITUTE
OF TECHNOLOGY

Time

Vladimir Vlassov and Johan Montelius





Time

Why is time important?

ID2201 DISTRIBUTED SYSTEMS / TIME 2

The importance of accurate timekeeping for distributed systems is rather apparent. **First**, time is a quantity we often want to measure accurately, e.g., for monitoring or ordering: To know at what time of day a particular event occurred at a particular computer. For example, an eCommerce transaction involves events at a merchant's and bank's computers. It is essential, for auditing purposes, that those events are timestamped accurately. **Second**, algorithms that depend upon clock synchronization have been developed for several problems in distributed computing, e.g., maintaining the consistency of distributed data (the use of timestamps to serialize transactions) to assess freshness (staleness) of replicas or cached copies, order updates and eliminating the processing of duplicate updates, checking the authenticity of a request sent to a server, etc.



The clock is not enough

In an asynchronous system, clocks can not be trusted entirely.
Nodes will not be completely synchronized.

We still need to:

- talk about before and after
- order events
- agree on order



Logical time

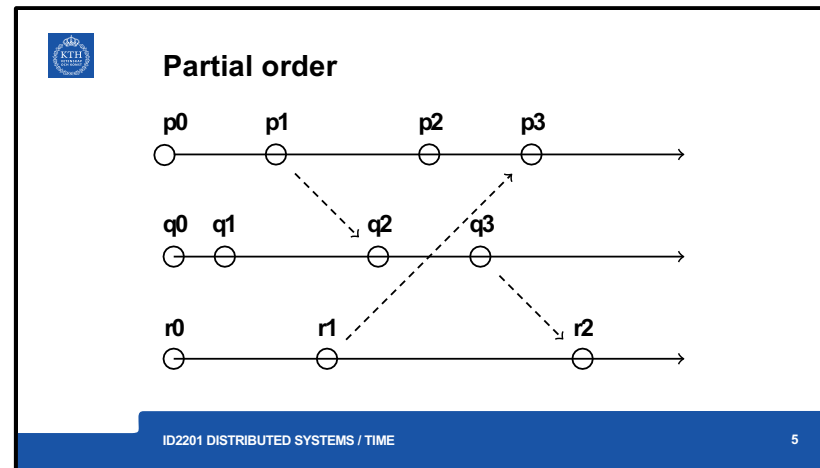
All events in one process are ordered.

The sending of a message occurs before the receiving of the message.

Events in a distributed system are partially ordered.

The order is called *happened before*.

Logical time gives us a tool to talk about ordering without having to synchronize clocks.



Sending All events in one process are ordered.

The sending of a message occurs before the receiving of the message.

Events in a distributed system are partially ordered.

The order is called *happened before*.

Logical time gives us a tool to talk about ordering without having to synchronize clocks.



Lamport clock

One counter per process:

- initially set to 0
- each process increments only its clock
- sent messages are tagged with a timestamp

Receiving a message:

- set the clock to the greatest of the internal clock and the time stamp of the message

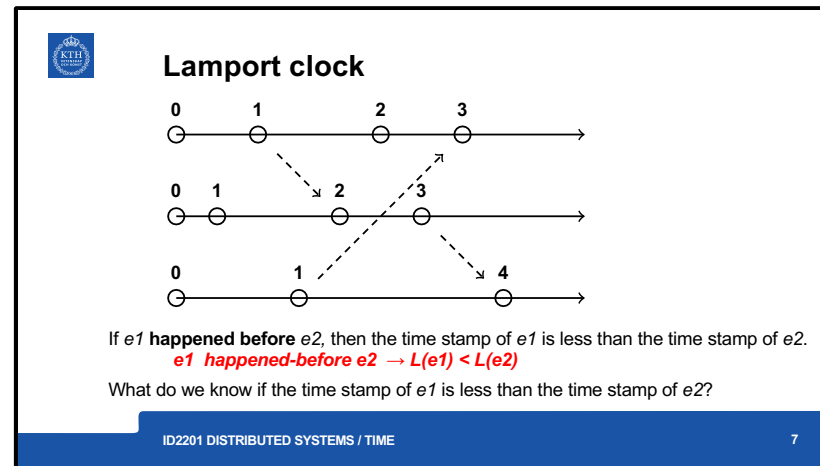
LC1: L_i is incremented before each event is issued at process p_i : $L_i := L_i + 1$.

LC2: (a) When a process p_i sends a message m , it piggybacks on m the value $t = L_i$.

(b) On receiving (m, t) , a process p_j computes $L_j := \max(L_j, t)$ and then applies LC1 before timestamping the event $receive(m)$.

Although we increment clocks by 1, we could have chosen any positive value. It can easily be shown, by induction on the length of any sequence of events relating two events e and e^* , that $e \rightarrow e^* \Rightarrow L(e) < L(e^*)$.

Note that the converse is not true. If $L(e) < L(e^*)$, then we cannot infer that $e \rightarrow e^*$.



LC1: Li is incremented before each event is issued at process pi : $Li := Li + 1$.

LC2: (a) When a process pi sends a message m , it piggybacks on m the value $t = Li$.

(b) On receiving (m, t) , a process pj computes $Lj := \max(Lj, t)$ and then applies LC1 before timestamping the event $receive(m)$.

Although we increment clocks by 1, we could have chosen any positive value.

It can easily be shown, by induction on the length of any sequence of events relating two events e and e^* , that $e \rightarrow e^* \Rightarrow L(e) < L(e^*)$.

Note that the converse is not true. If $L(e) < L(e^*)$, then we cannot infer that $e \rightarrow e^*$.

Some pairs of distinct events, generated by different processes, have numerically identical Lamport timestamps. However, we can create a total order on the set of events – that is, one for which all pairs of distinct events are ordered – by taking into account the identifiers of the processes at which events occur.



Let's play a game

DON'T VIOLATE THE "HAPPEND BEFORE" ORDER!



Can we do better

We should be able to timestamp events to capture the partial order.

We want to look at two timestamps and say:

If the time stamps are ordered, then the events are ordered

$T(e1) < T(e2) \rightarrow e1 \text{ happen-before } e2$



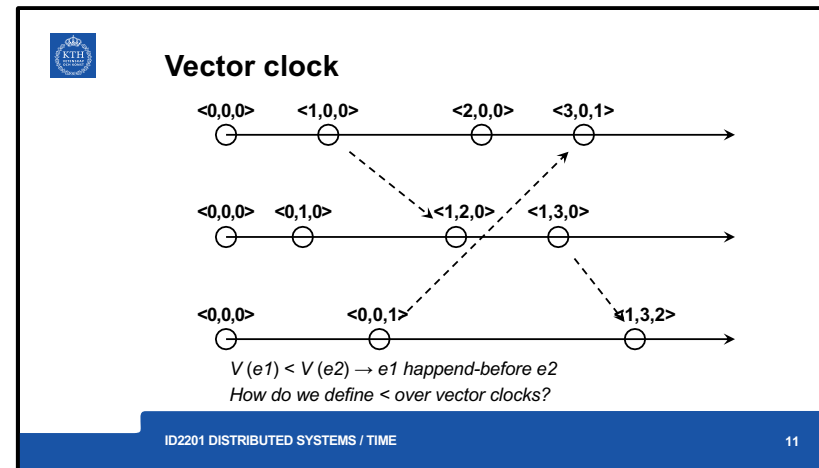
Vector clock

A **vector** with one counter per process:

- initially set to $\langle 0, \dots \rangle$
- each process increments only its index
- sent messages are tagged with a vector

Receiving a message:

- **merge** the internal clock and the time stamp of the message



We may compare vector timestamps as follows:

$V = V^*$ iff $V[j] = V^*[j]$ for $j = 1, 2, \dots, N$

$V \leq V^*$ iff $V[j] \leq V^*[j]$ for $j = 1, 2, \dots, N$

$V < V^*$ iff $V \leq V^*$ AND $V \neq V^*$



Compare vector timestamps

Vector timestamps can be compared as follows

- $V = V^*$ iff $V[j] = V^*[j]$ for $j = 1, 2, \dots, N$
- $V \leq V^*$ iff $V[j] \leq V^*[j]$ for $j = 1, 2, \dots, N$
- $V < V^*$ iff $V \leq V^* \wedge V \neq V^*$

$V(e1) < V(e2) \rightarrow e1$ happen-before $e2$

If neither $V(e1) \leq V(e2)$ nor $V(e2) \leq V(e1)$ then the $e1$ and $e2$ are *concurrent*.



Pros and cons

The partial order is complete; we can look at the time stamp and determine if two events are ordered.

The vectors will take up some space and could become a problem.

What should we do if more processes come and leave? There is no easy mechanism to add new clocks to the system.

Vector clocks could be overkill.



Summary

We have to use something else if we can not trust real clocks to be synchronized.

Logical time captures what we need:

- Lamport clock: sound
- Vector clock: complete

Implementation issues:

- do we have to timestamp everything
- how do we handle new processes