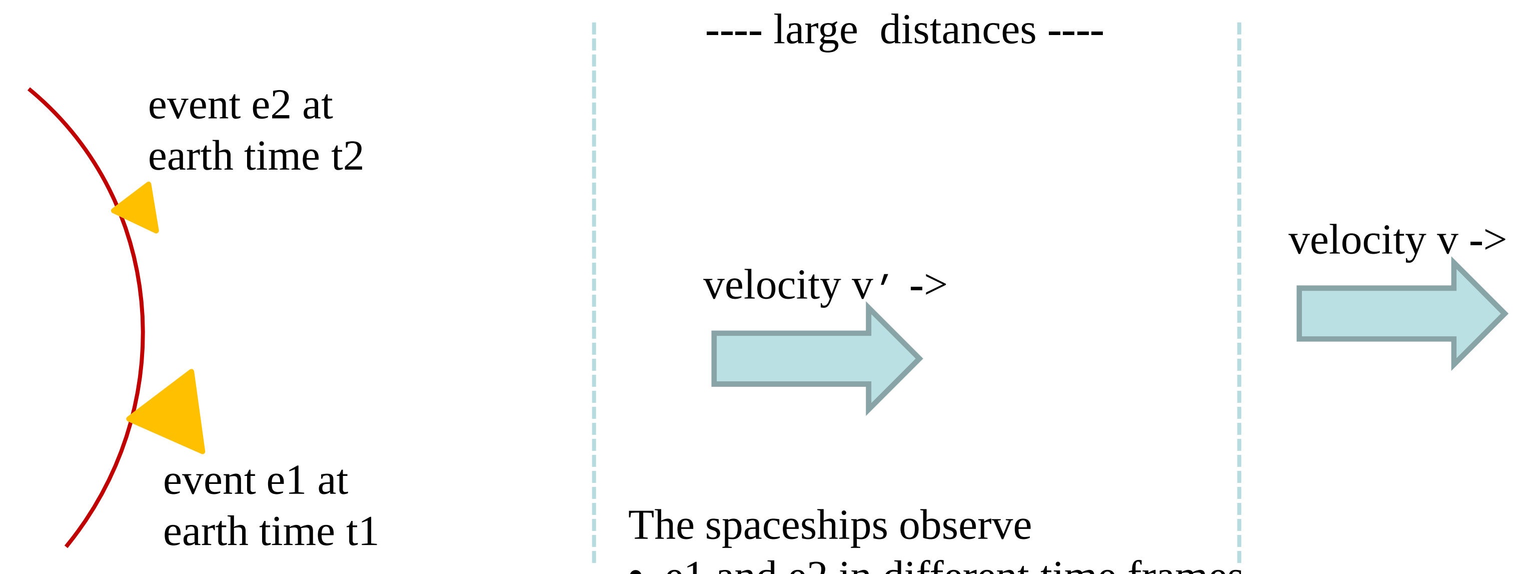
Time in Distributed Systems

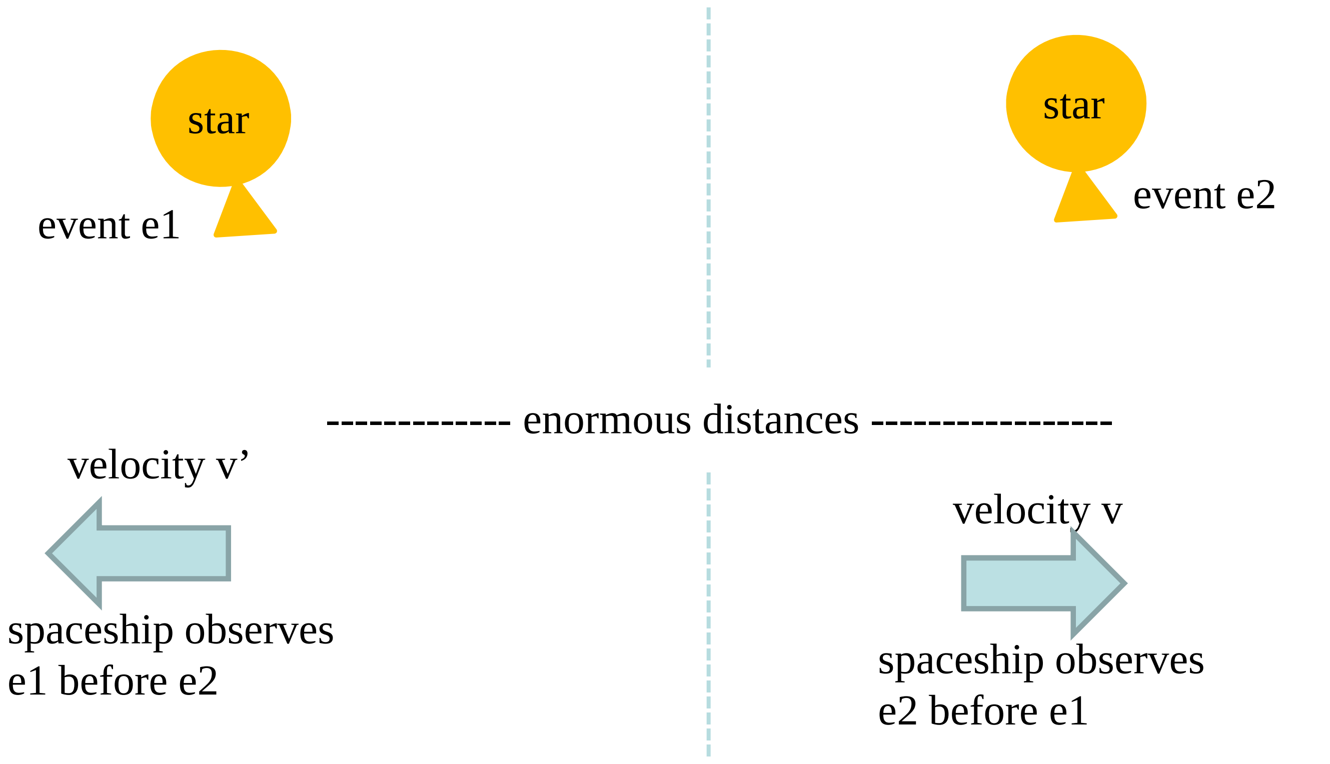
There is no common universal time (Einstein) but the speed of light is constant for all observers irrespective of their velocity



The spaceships observe

- e1 and e2 in different time frames
- different values for time elapsed between e1 and e2
- e1 and e2 (from the same source) in the same order.

Event ordering in space


star
event e1
star
event e2
------------- enormous distances -----------------
velocity v'
spaceship observes e1 before e2
velocity v
spaceship observes e2 before e1

Time in Distributed Systems

Assume our distributed system is earth-based

Earth time is defined w.r.t. the earth’s rotation

- solar year is constant
- solar day is lengthening (earth slowing)

From 1948, earth time has been based on atomically-defined caesium clocks
(atomic second = solar second)

There are now about 50 such clocks, average value = TIA (International Atomic Time)

BIH (Intn’l Bureau de l’Heure) announces leap seconds to keep in phase with the sun
---- about 30 so far, most recently Jan 1999, Jan 2006

Time in Distributed Systems - UTC

UTC (Universal Coordinated Time) is corrected TIA

UTC services are offered by radio stations and satellites
– receivers are available commercially
Accuracy varies with weather conditions
– stated bounds are 1ms – 10ms
radio 10ms
UK: Rugby since 1927, Anthorn Cumbria 2007,
US: Fort Collins Colorado
satellite: GOES 0.5ms, GPS 1ms

UTC signals take time to propagate – UTC can't be known exactly
For a given receiver we can estimate a time interval during which an event has happened w.r.t. UTC, see later "interval timestamps"

Timers in computers

Based on frequency of oscillation of a quartz crystal

Each computer has a timer that interrupts periodically
Clock drift: in practice, the number of interrupts per hour varies slightly
in the fabricated devices, also with temperature, so clocks may drift,
typically $1/10^6$ (1 sec in 11.6 days)

Timers can be set from transmitted UTC
We have already seen that we cannot know accurately the time at which an event occurs,
but can only specify an interval
We now have to increase that interval to allow for clock drift
as well as other sources of inaccuracy

Note that computer systems tag events with timestamps, usually a local clock reading.
Preferably, interval timestamps should be used.

Does accurate time matter?

Important questions:
How accurate does time need to be?
How is time used in a distributed system?
What does "A happened before B" mean in a distributed system?

Sometimes we CAN'T SAY in which order two events occurred:

- if the events have point timestamps that differ by less than some value*
- if the events have interval timestamps, and the intervals overlap

*we prefer intervals because for point timestamps we need to know the characteristics of the originator in order to determine the tolerance

Use of time in distributed systems: examples 1

1. Any source of resource contention e.g. Airline booking

   POLICY: if the reservation requests of two transactions may each be satisfied separately but there are not enough seats left for both, then the transaction with the earlier timestamp wins

   Note that no causality is involved, the requests are independent.
   We don’t need accurate time but just an ordering convention so all agree who won.

   On a tie (equal timestamps) use an agreed tie-breaker
   e.g. IP address / processID

Use of time in distributed systems: examples 2

2 Programming environments e.g. UNIX ***make*** (compile and link)

Suppose a ***make*** involves many components that are edited on distributed computers.
Suppose a component is edited immediately after a ***make***,
but on a computer with a slow clock
so that the recorded time of the edit is before the recorded time of the ***make***.
On the next ***make*** this component is not recompiled.

This can be made unlikely to happen, if we ensure that clocks are initialised accurately
e.g. not from the operator's watch, but from a "time server" – see below.

This is an example of correctness depending on correct event ordering:
did the edit take place before or after the last ***make***?

of course – it's a bad idea to use a timestamp as a version number in a distributed system. ***make*** was designed for a centralised UNIX system.

Use of time in distributed systems: examples 3

3. Did a credit/debit transaction take place before or after midnight?
   This affects daily calculation of interest.

4. The value of shares at the time of buying/selling.

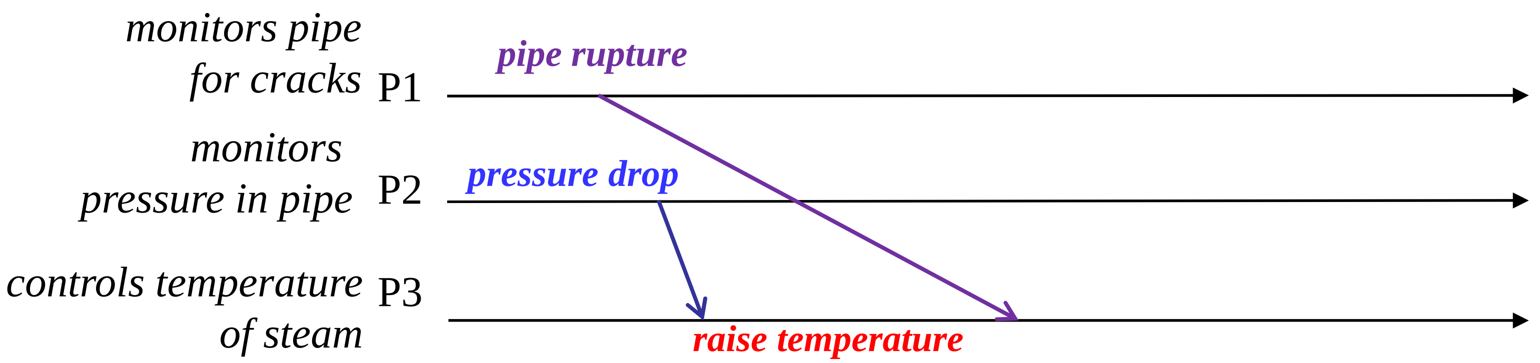
5. Insider dealing? Did X read Y before buying/selling?

Note that some of the above examples require only a means of agreement, so that all participants in the algorithm make the same decision.

Others require accurate time, or the order of events in the real world, when causality is at issue.

Physical causality in the environment

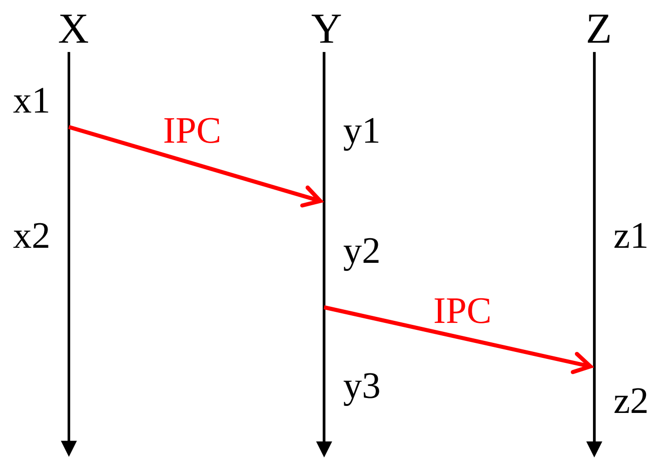
Causality may be absolute and physical – outside the scope of the message transport service



1. The pipe ruptures which causes a drop in pressure
2. P1 send a message to controller P3 to notify rupture
3. P2 sends a message to controller P3 to notify pressure drop
4. P3 receives P2's message (before P1's) and increases temperature
5. P3 receives P1s message .....
6. AUDIT may infer (wrongly) that temperature increase caused the pipe to rupture

The controller's algorithm must take delay and physical timestamps into account
AUDIT of system failure may have to report "***can't say***" for close timestamps

Event ordering in distributed systems



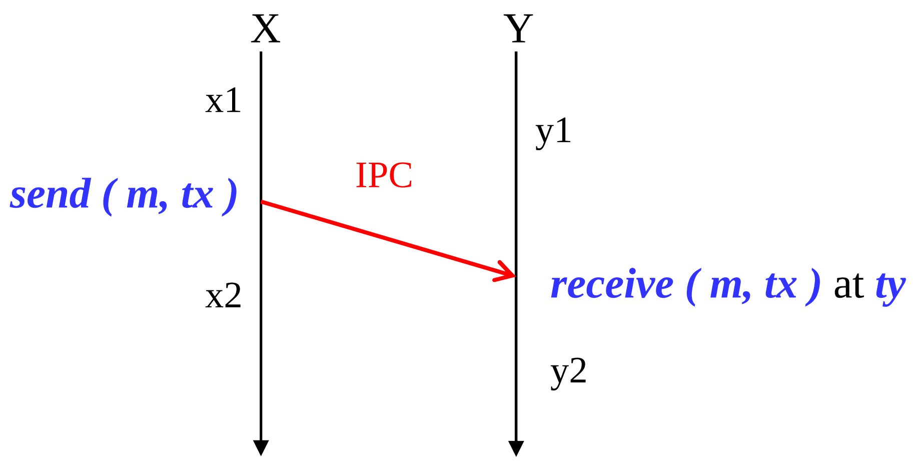
Define < to mean "happened before"
Events in a single system are assumed to be ordered

IPC: ***send*** is before ***receive,*** this is TRUE *whatever the local clocks of X, Y and Z indicate*
IPC imposes a partial order on events:

events in region x1 < events in regions y2 and y3
events in region x1 < events in region z2
events in regions y1 and y2 < events in region z2
for events in other regions we can't say what the order is
unless we know the precise accuracy of all physical clock values

Time

Local clocks must respect true event orderings



Note that X’s ***send*** caused Y’s receive

Suppose Y’s local clock reads ***ty*** on ***receive (m, tx)***

if ***ty > tx*** OK

if ***ty < = tx*** reset ***ty*** to ***tx*** plus one increment

This imposes logical time on the system

BUT system time adjusted in this way will drift ahead of UTC

- could use counters rather than timestamps if all we need is event ordering
- so-called “Lamport Time”

How can we generate timestamps that are reasonably close to UTC and preserve causal ordering?

Protocols for synchronizing physical clocks - 1

Cristian's algorithm 1989

- Assume one computer has a UTC receiver (call it a time server)
- Each computer polls the time server periodically
  (period depends on maximum clock drift and accuracy required).
- Server sends back its value of the time
- Client receives this value and may: use it as it is,
  add the known minimum network delay,
  add half the round trip time for this request/response

Client/receiver resets its clock from this value T:

if T > local time
use it to set the clock, or adjust the interrupt rate for a while to speed up the clock
e.g. 10ms -> 9ms

if T < local time
time cannot be put back or event ordering within the local system would be violated
so adjust the interrupt rate to slow down the clock e.g. 10ms –> 11ms

Protocols for synchronizing physical clocks - 2

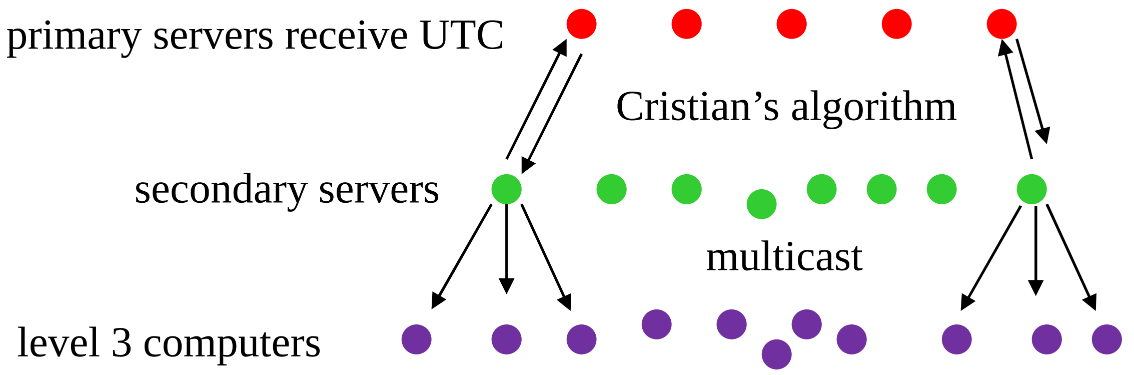
In the above, the time server is a single point of failure.

A number of time servers can be used to increase reliability
each computer multicasts to all time servers,
takes the average of the returned values then proceeds as above.

If there is no time server,
a nominated component can multicast to all, requesting their time
then multicast the average value to all (*Berkeley UNIX 1989*).

NTP Network Time Protocol

For the Internet as a hierarchy of computers:



- uses UDP
- allow for network delay and adjust clocks as described for Cristian's algorithm
- accurate to a few tens of milliseconds

Time servers also exist as web servers for explicit query from individual computers

Point timestamps and interval timestamps

For any computer we can *estimate* how long UTC takes to reach it, taking into account:
- atmospheric pressure
- network(s) transmission time
- software overhead e.g. in local OS

To tag a message with a timestamp:
The local clock reading could be used as a point timestamp and a tolerance could be estimated.
Note that this should be *source-specific* – hardly ever taken into account.

An interval timestamp, in which the UTC is estimated to lie, captures the uncertainty over measuring time, taking into account local conditions
i.e. interval width should be source-dependent.

Use of point and interval timestamps

If events are to be ordered,
Point timestamps closer than their associated tolerances and overlapping interval timestamps indicate that this cannot be done reliably.

The application may be told that a strong ordering is impossible (CAN’T SAY).
A weak ordering may be formed on the basis of e.g. the point timestamps taken literally, or the upper interval bounds, but it should be made explicit to the application that this is not correct/reliable
e.g. it cannot be used as an audit of the possibility or otherwise of cause and effect.

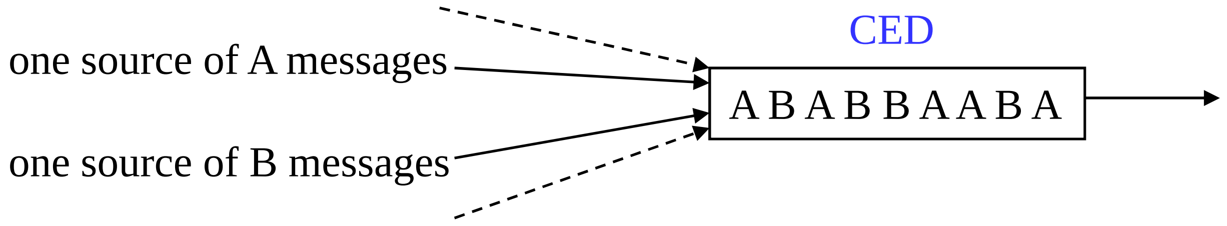
This is the nature of distributed systems – we have to live with it.
Ref: Fundamental Properties, introduction slide 13

Applications that abstract above distributed time should be aware that they are doing this
e.g. arrival time of a request at a server may be used to order requests.
Source timestamps may indicate a different order or may be indeterminate.
Database and stream processing applications tend to use arrival time at the server.

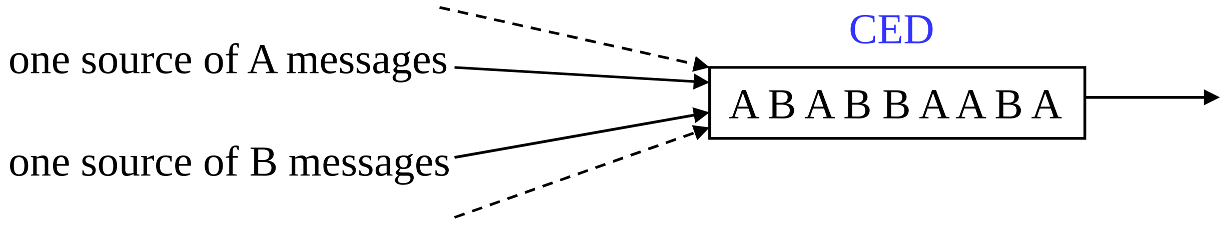
Composition of events (sent as messages)

Applications are often interested in patterns of events, perhaps discovered through data mining
- fraud detection
- fault detection
- raising alarms – medical, environmental, ...
- controlling the volume of events propagated, e.g. from sensors, from faulty components

A Composite Event Detector (CED) receives streams of events from distributed sources and notifies a stream of composite events. An example showing two event types A and B:



Composition of events – composition algebra



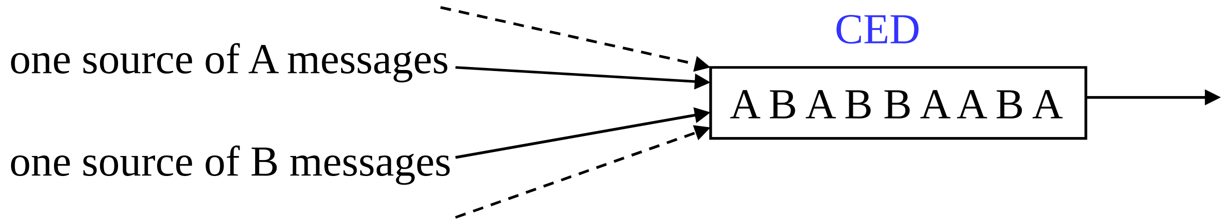
An event algebra defines composition operators: e.g.

AND, OR, SEQ (before/after),
UNTIL (stream with a terminator),
AFTER (stream with a starter),
NOT? (difficult to decide)

Recall fundamental uncertainty over time if event ordering (SEQ, AFTER, UNTIL) is offered.
perhaps offer choice to application of strong and weak ordering, or tag whether strong or weak

Timestamp of composite event?
– the interval spanning all component events (easy/natural with interval timestamps on events)
– or the timestamp of the latest component event? – when did the CE complete?

CED engineering issues



Engineering issues:

- are all the event sources registered with the CED, and the connections to them, operational?
  use a heartbeat protocol with each source
  should processing be delayed if lack of a heartbeat indicates an event may have been delayed ?
  the NOT operator makes this problem explicit

- buffer size and garbage collection?

- consumption policy (in this example, which As with which Bs?) *historical? most recent?*

A CED may take as input primitive and/or composite events
CED components (subtrees) may be distributed
e.g. placed close to event sources, optimising communication