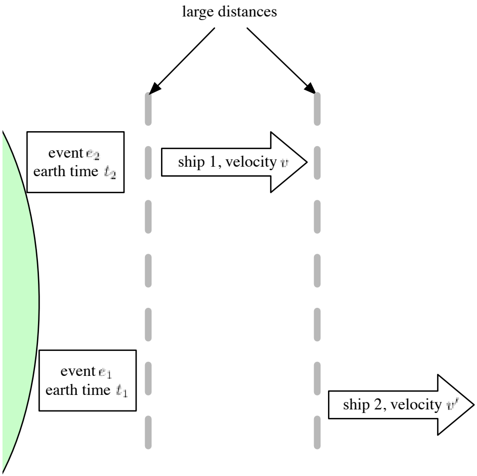
# DS 2009: time

David Evans
de239@cl.cam.ac.uk

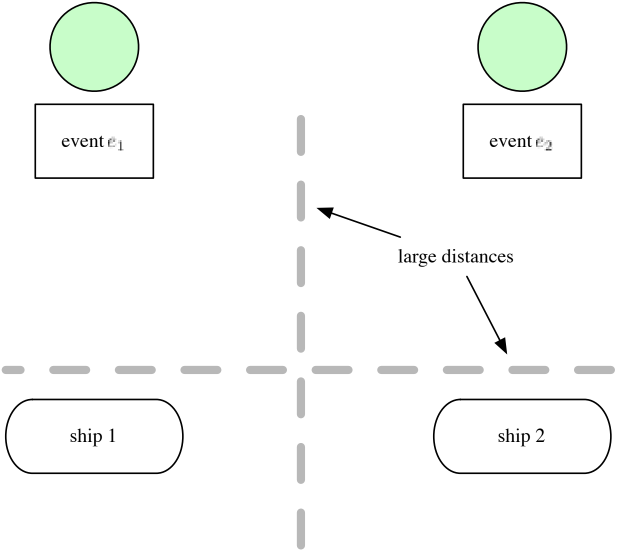
# Time in distributed systems

1. there is no common universal time
   - assume we don't need to worry about relativistic effects
2. time is still complicated
   - sunrise/sunset?
   - radioactive decay?
   - stars' positions?
   - seasons?
   - tides?
   - slowing of the planet's rotation?
3. UTC (Coordinated Universal Time) is in step with TAI but based on UT1
4. UTC services are offered by radio stations and satellites
5. RF signals take time to propagate—UTC can't be known exactly
6. For a given receiver we can estimate a time interval during which an event has happened w.r.t. UTC (see also page 14 and "interval timestamps")

# Timestamps can differ



# Order of observation can differ



# Timers in computers

- based on frequency of oscillation of a quartz crystal (usually)
- each computer has a timer which interrupts periodically
  - in practice, the number of interrupts per second varies slightly in the fabricated devices and with temperature, so clocks may drift (*clock skew*)
- timers can be set from transmitted UTC
- we cannot know the time at which an event occurs accurately, but have to increase the interval to allow for clock drift as well as other sources of inaccuracy
- important questions
  1. Do we need accurate time?
  2. how is time used in distributed systems?
  3. what does "A happened before B" mean in a distributed system?

# The problem with “happened before”

If two events have single-value timestamps which differ by less than some value we *can’t say* in which order the events occurred.
With interval timestamps, when intervals overlap, we *can’t say* in which order the events occurred

# Examples of the use of time

- resource contention, *e.g.*, airline booking

  **Policy** if the reservation requests for two transactions may each be satisfied separately but there are not enough seats left for both, then the transaction with the earliest timestamp wins

  Note that there is no causality, the requests are independent. We don't need fine-grained accuracy, we just need a timestamp ordering convention so all agree who won. On a tie (equal timestamps) use an agreed tie-breaker, *e.g.*, IP address/process ID

# Examples of the use of time

- programming environments, *e.g.*, UNIX make (compile and link) Suppose a make involves many components which are edited on distributed computer. A component is edited immediately after a make, but on a computer with a slow clock. The edited source is given a timestamp earlier than the make and the source will not be recompiled on the next make.
  - This can be made unlikely to happen, if we ensure that clocks are initialised reasonably accurately (*e.g.*, not from the operator's watch)
  - this is an example of correctness depending on correct event ordering: did the edit take place before or after the last make?

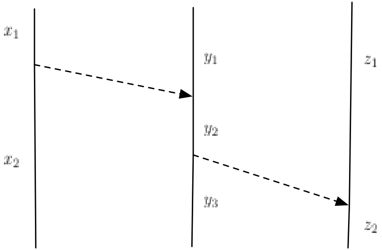
# Examples of the use of time

- did a credit/debit transaction take place before or after midnight? (This affects the calculation of interest.)
- the value of shares at the time of buying/selling
- insider dealing—did X read Y before buying/selling?

# Requirements of time are not the same

Some of these examples require only a means of agreement, so that all participants in the computer system make the same decision. Others require accurate time or the order of events in the real world when causality is at issue.

# Event ordering in DS




Define $<$ to mean "happened before"

Within the context of inter-process communication (IPC)

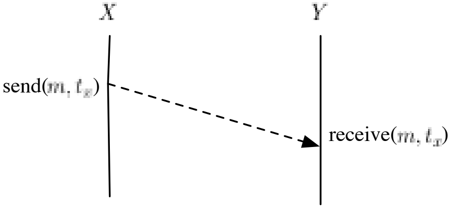
- events within a single process are ordered
- events in region $x_1$ $<$ events in regions $y_2$ and $y_3$
- events in region $x_1$ $<$ events in region $z_2$
- events in regions $y_1$ and $y_2$ $<$ events in region $z_2$
- for events in other regions we *can't say*, unless we know the precise accuracy of all physical clock values

# IPC defines a partial order

IPC defines a *partial ordering* on the events in the DS. This ordering is true whatever the local clocks of X, Y and Z indicate.

# Event ordering and clocks

It is easier if the values of the local clocks respect this event ordering. Suppose a message $m$ is timestamped $t_X$ by $X$ on sending:




($X$'s send *caused* $Y$'s receive.) Suppose $Y$'s local clock has reading $t_Y$ on receiving $m$ (remember that $Y$ also learns $t_X$). What if we do this:

**if** $t_Y > t_X$ **then**
  OK
**else**
  $t_Y \leftarrow t_X + 1$
**end if**

This imposes logical time on the system.

# A problem with logical time

System time adjusted in this way will drift ahead of UTC. We could use counters rather than timestamps if all we need is event ordering. So, can we generate timestamps that

- are reasonably close to UTC
- preserve causal ordering

# Synchronising physical clocks

Cristian's algorithm (1989)

- assume one machine has a UTC receiver (the "time server")
- each machine polls the time server periodically (period depends on maximum clock drift allowed and accuracy required)
- server responds to a poll with 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 time between this send and receive

# Synchronising physical clocks

Cristian's algorithm (cont'd)

Now consider resetting the receiver's local clock from this value; call it $t$

**if** $t \geq$ local time **then**
OK, use $t$ to set the clock
or adjust the interrupt rate for a while to speed up the clock (*e.g.*, 10ms to 9ms)
**else**
adjust the interrupt rate to slow down the clock (*e.g.*, 10ms to 11ms) (the clock can't be put back or event ordering within the local system would become incorrect!)
**end if**

# Synchronising physical clocks

Berkeley Unix (Gusella & Zatti, 1989)

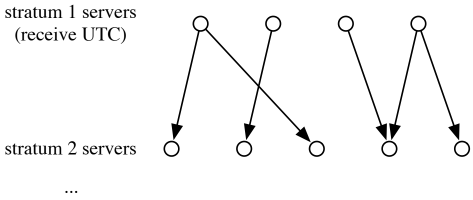
If no machines have receivers…

- a nominated "time-server" asks all machines for their times
- it computes the average value
- this is broadcast to all machines
- operator may set the time manually from time to time

# Synchronising physical clocks

NTP (Mills 1991, *etc.*)

Uses a hierarchy of machines (on the Internet, usually, but doesn't assume this)




- uses UDP
- allow for estimated network delay and adjust clocks as described above
- accurate to a few tens of ms

lots more info is available

# Interval timestamps

- for any computer we can estimate how long UTC takes to reach it, taking into account:
  - atmospheric propagation
  - network(s) transmission
  - software overhead (*e.g.*, local OS)
- instead of a single-valued timestamp, use an interval in which UTC is estimated to lie
- use these interval values for ordering events, for example when arriving messages need to be ordered
- sometimes we *can't say*—this is the nature of distributed systems (a weak ordering might be created, based on, *e.g.*, the upper interval bound)

# Composing events

- applications are often interested in patterns of events
  - fraud detection
  - fault detection
  - to control the volume of events propagated
- a composition service receives streams of events from distributed sources and creates a stream of composite events. Example with two event types, A, B

# Composing events

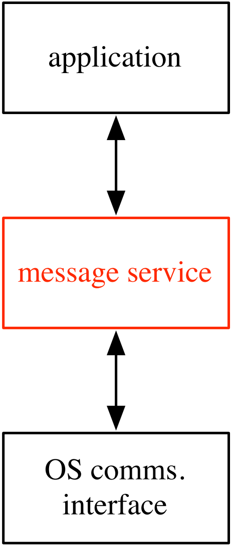
- possible composition operators: AND, OR, SEQ (before/after)? UNTIL?, AFTER?, NOT? . . .
- fundamental characteristics of DS make this tricky
  - are all sources of A and B and connections to them operational?
  - have all the As and Bs arrived? should we use a heartbeat protocol?
- what is the consumption policy for As and Bs? historical, most recent, . . . ?
- buffer size and garbage collection?

# Ordering message delivery

Assumptions

1. messages are multicast to named process groups
2. reliable channels: a given message is delivered reliably to all members of the group (no lost messages)
3. FIFO from a given source to a given destination
4. processes don't crash (failure and restart not considered)-
5. processes behave as specified and send the same values to all processes (we are not considering Byzantine behaviour)

# Schematically




may be delivered message in a specified order

may buffer and reorder messages (as requested) before delivery

assume FIFO from each source at this level (done by lower levels)

# Order schemes

**no order** messages are delivered to the application process in the order received by the message service

**causal order** messages that are potentially causally related are delivered in causal order at all processes

**total order** every process receives all messages in the same order

# Aside: what are process groups

- membership management
  - create (name, group members, group member, . . . )
  - kill (name)
  - join (name, process)
  - leave (name, process)

# Aside: what are process groups

- membership management
  - create (name, group members, group member, . . . )
  - kill (name)
  - join (name, process)
  - leave (name, process)
- internal structure

**none** failure tolerant, complex protocols
**some** a single coordinator (and point of failure); simpler protocols

# Aside: what are process groups

- membership management
  - create (name, group members, group member, ... )
  - kill (name)
  - join (name, process)
  - leave (name, process)
- internal structure

**none** failure tolerant, complex protocols
**some** a single coordinator (and point of failure); simpler protocols

- closed or open

**closed** only members may send to the group name, *e.g.*, parallel, fault-tolerant algorithms
**open** a non-member can send to a group, e.g. to a set of servers

# Aside: what are process groups

- membership management
  - create (name, group members, group member, . . . )
  - kill (name)
  - join (name, process)
  - leave (name, process)
- internal structure

**none** failure tolerant, complex protocols
**some** a single coordinator (and point of failure); simpler protocols

- closed or open

**closed** only members may send to the group name, *e.g.*, parallel, fault-tolerant algorithms
**open** a non-member can send to a group, e.g. to a set of servers

- failures
  - a failed process leaves the group without executing leave

# Aside: what are process groups

- membership management
  - create (name, group members, group member, . . . )
  - kill (name)
  - join (name, process)
  - leave (name, process)
- internal structure

**none** failure tolerant, complex protocols

**some** a single coordinator (and point of failure); simpler protocols

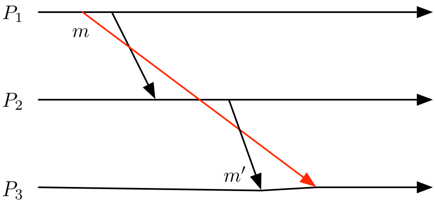
- closed or open

**closed** only members may send to the group name, *e.g.*, parallel, fault-tolerant algorithms

**open** a non-member can send to a group, e.g. to a set of servers

- failures
  - a failed process leaves the group without executing leave
- robustness
  - leave, join and failures happen during normal operation

# What is causal order?



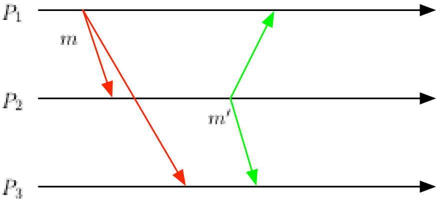

if causal delivery order is required, $m$ should be delivered before $m'$ at $P_3$

$$\text{send}_i\ m < \text{send}_j\ m' \Rightarrow \text{deliver}_k\ m < \text{deliver}_k\ m'$$

so $P_1$ sends $m$ before $P_2$ sends $m' \Rightarrow m$ should be delivered before $m'$ at $P_3$

# Causal order is feasible

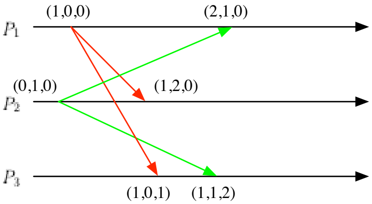
Suppose that $P_1$, $P_2$, and $P_3$ are in a process group and all messages are multicast to the group (all processes receive all messages)




In this case, the message delivery system can implement causal delivery order by using vector clocks. (Total order later.)

# Vector clocks: implementing causal order

A vector clock is maintained by the message service at each node for each process.

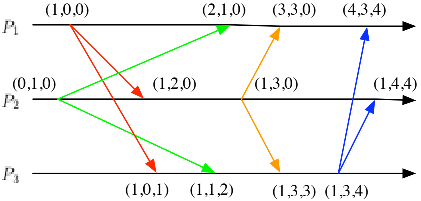



Properties

- fixed number of processes, $N$
- each process's message service keeps a vector of dimension $N$
- for each process, each entry records the most up-to-date value of a state counter, known to that process, for the process at that position

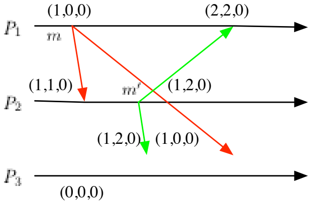
(set notation would be better for dynamic reconfiguration of groups—but vectors have stuck)

# Message service operation




- before send, increment local process's state-value in local vector
- on send, timestamp message with sending process's local vector
- on deliver, increment receiving process's state-value in its local vector and update the other fields of the vector by comparing its values with the incoming timestamp and recording the higher value in each field thus updating this process's knowledge of system state

# An example



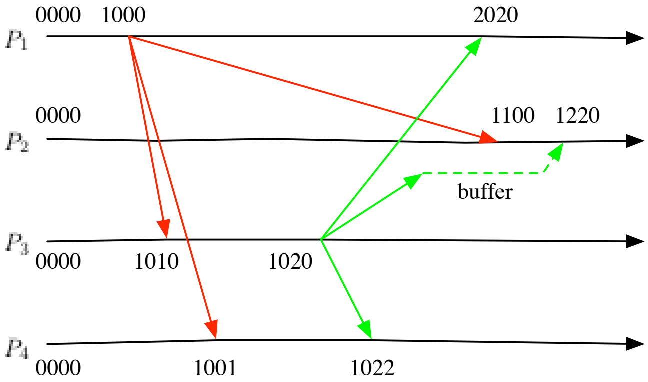

At $P_3$, local vector is (0,0,0). $m'$ arrives from $P_2$ with timestamp (1,2,0), meaning that $P_2$ received a communication from $P_1$ before sending $m'$.

Whole point: make it easy for a process to tell that it hasn't received some messages.

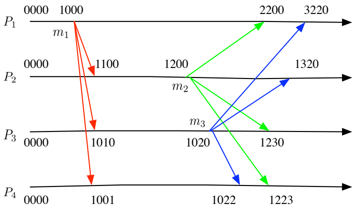
# More detail at $P_3$

| | receiver vector | sender | sender vector | decision | new receiver vector |
|---|---|---|---|---|---|
| | (0,0,0) | $P_2$ | (1,2,0) | hold in buffer | (0,0,0) |
| | (0,0.0) | $P_1$ | (1,0,0) | deliver | (1,0,1) |
| from buffer | (1,0,1) | $P_2$ | (1,2,0) | deliver | (1,2,2) |

# Another example



# Causal order is not total order!




$m_2$ and $m_3$ are not causally related

- $P_1$ receives $m_1, m_2, m_3$
- $P_2$ receives $m_1, m_2, m_3$
- $P_3$ receives $m_1, m_3, m_2$
- $P_4$ receives $m_1, m_3, m_2$

If application requires total order, this can be enforced using vector clocks with extension to include ACKs and delivery to self (see below). But the vectors can be a large overhead on message transmission.

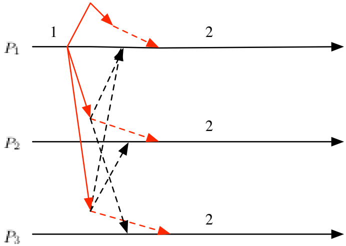
# Totally ordered multicast

Totally ordered multicast can be achieved using a single logical clock value as the timestamp

- sender multicasts to all including itself
- all acknowledge receipt as a multicast message
- message is delivered in timestamp order when all ACKs have been received

If the delivery system must support both, so that applications can choose, vector clocks can achieve both causal and total ordering.

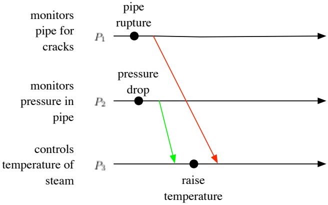
# Doing totally-ordered multicast




- all delivery systems collect messages, send ACKs and collect ACKs
- $P1$ increments its clock to 1 and multicasts a message with timestamp 1
- All delivery systems collect message, send ACK, and collect all ACKs. No contention $\Rightarrow$ deliver message and increment local clocks to 2
- if $P_2$ and $P_3$ both multicast messages with timestamp 3, use a tie-breaker to deliver $P_2$'s message before $P_3$'s

# Real-world causality

*e.g.*, monitoring and controlling a pipe along which steam is delivered under pressure




1. The pipe ruptures (which causes a drop in pressure)
2. $P_1$ sends message to controller $P_3$ to notify rupture
3. $P_2$ sends message to $P_3$ to notify pressure drop
4. $P_3$ receives $P_2$'s message before $P_1$'s and increases temperature of steam
5. $P_3$ then receives $P_1$'s message and infers (wrongly) that increasing the temperature has caused the pipe to rupture