# Time and Global States
# (CDK chapter 10)

**The fundamental objective of any distributed application or information system is for the separate components to co-operate and co-ordinate to do useful work and / or achieve some overall common system goal.**

*However to do this:-*

A knowledge of **global state** is required at an **instant of time** to make a decision or verify a property.

A global state at a given instant may need to satisfy an invariant, a rule that cannot/should not be broken.

Examples of global invariants:

- Absence of deadlocks
- Write access to a distributed database never granted to more than one process
- The sum of all account debits and ATM payments in an electronic cash system is zero
- Objects are only subject to garbage collection when no further reference to them exists

In order to reason about a distributed system, we need to be able to "know" the global state, unfortunately this varies between a difficult and impossible problem!

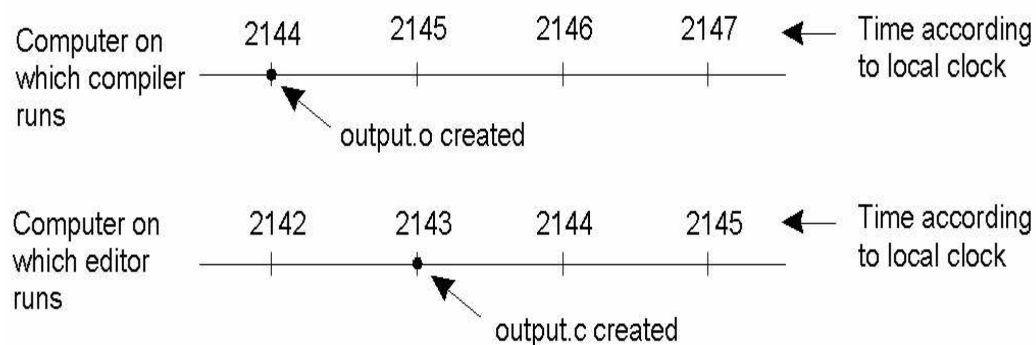
In general a global view of a distributed system is not possible because:

- **At an instant of time a node has only an approximate knowledge of the state of the other nodes**
- **The events that take place in a certain order at one site can be perceived in a different way at other sites.** ***(Theory of Special Relativity)***
- **However even with imperfect information it is possible to get a consistent view and apply global reasoning.**

The difficulty in constructing a global state of a distributed computation arises partly because there is no global real time available in the system. Time in a distributed system serves the following purposes:

- **It co-ordinates processes/processors so that they can perform an action at roughly the same time**
- **It facilitates reasoning about the global state of the system based only on local information**
- **It defines an ordering of events that preserve cause-effect relations.**

Consider the following:




It is always possible to determine the order of events in a centralized system because there is only one system clock. In a distributed system it requires the synchronization of all the local clocks.

# But what do we mean by *time* or *clocks*?

Time we'll revisit shortly. Clocks can be dealt with more shortly:

Physical Clocks

- are based on the regular variation of some physical phenomena. Those in computing are based on the oscilations of a stretched quartz crystal

Logical Clocks

- are monotonically increasing software counters whose values bears no relationship to any physical clock. It captures the *happened before* ordering which is important in many algorithms.

Time has had many different meanings (over time ☺ pardon the pun!)

We need to think what properties we want "time" to have:

Periodicity, Physical Relationships, Remote synchronization.

## *Astronomical Time*

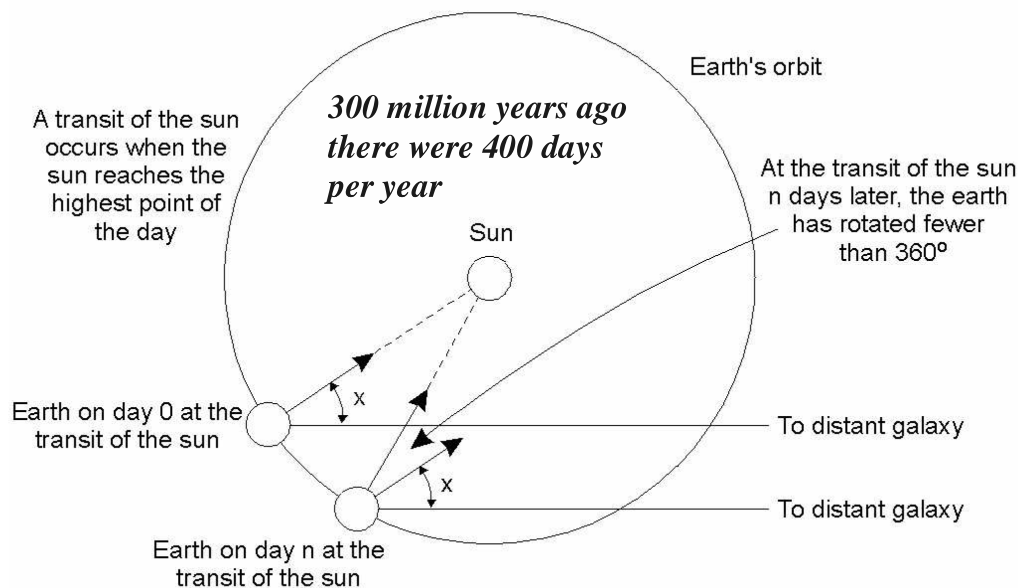
Is based on the sun rising in the east and setting in the west. A measurement is made of the suns highest apparent point in the sky called the transit of the sun which occurs about noon each day.

The interval between two consecutive transits is a *solar day* and a *solar second* is 1/86400 of a solar day.

Astronomers compute many transits of the sun and take the average before dividing by 86400 to get the *Mean Solar Second* on which *Greenwich Mean Time* is based.

## *But astronomical time drags, and is inconsistent.*

The earths rotation is gradually slowing:



## *International Atomic Time (TAI)*

With the invention of the atomic clock (1948) it was decided to use the Cesium 133 Atom on which to base time.

The second was defined in 1958 as the time it took Cesium 133 to make 9,192,631,770 transitions and in that year was the same time as a mean solar second.

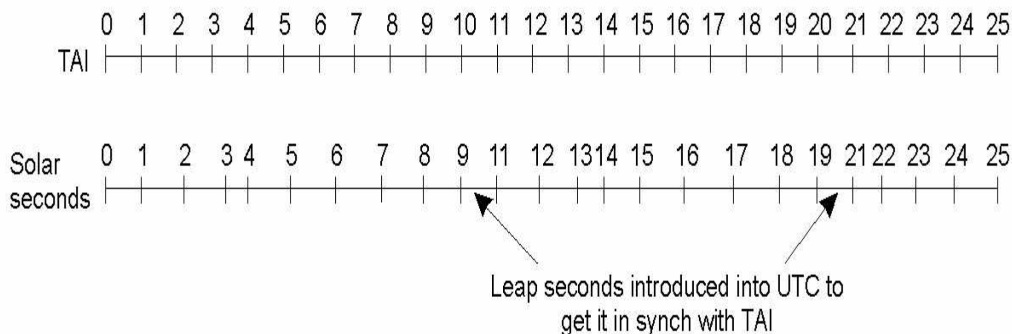
Around 50+ labs around the world have Cesium 133 clocks. Periodically BIH Paris average the clock ticks to produce International Atomic Time abbreviated to TAI.

However now 86400 TAI seconds is about 3 milliseconds less than the mean solar day

Using TAI time would mean that noon would eventually occur in the early hours of the morning.

## *Universal Coordinated Time (UTC)*

BIH Paris solve the problem by introducing leap seconds when necessary. This gives rise to a time system based on constant TAI seconds but keeps in phase with the motion of the sun.




This is called Universal Coordinated Time (UTC) which now replaces Greenwich mean Time which is astronomical time.

Pope Gregory tried a similar idea in February 1582 but got the granularity of the leap wrong when he decreed 10 days be deleted from the calendar in October. This event caused riots.

However global time presented problems for distributed systems even then:

```
Italy:           04 Oct 1582 was followed by 15 Oct 1582
France:          09 Dec 1582 was followed by 20 Dec 1582
UK/USA:          02 Sep 1752 was followed by 14 Sep 1752
Russia:          31 Jan 1918 was followed by 14 Feb 1918
China (unsure): 18 Dec 1928 was followed by 01 Jan 1929
```

So an event occurs on 05-Sep-1752?

## *Providing UTC*

Radio Services produce a pulse at the start of each UTC second accurate to around plus or minus 0.1-10.0 ms depending on the service & location:

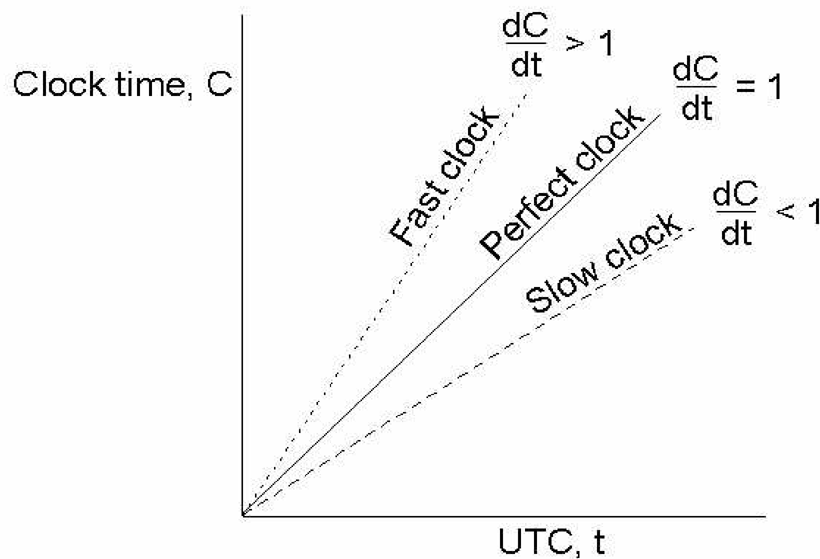
- NIST - short wave radio WWV Fort Collins Colorado
- MSF - Rugby Warwickshire

Satellite Services are more accurate, typically 1 μs

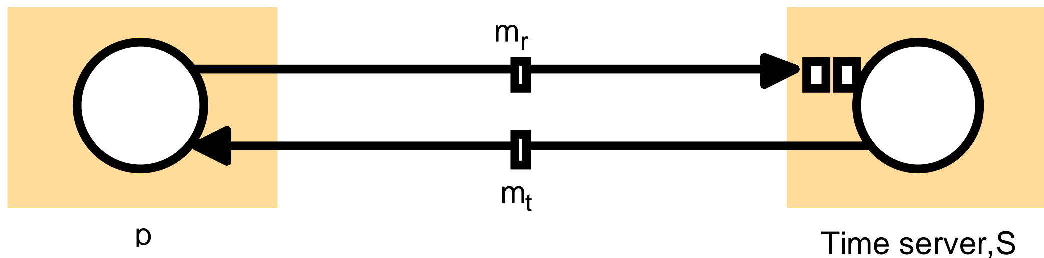
- GEOS
- GPS

Computers with receivers to these services can synchronize their clocks to these timing signals. However all other computers must keep synchronized with these.

But, non atomic clocks drift, either for electrical or physical properties.



## Simple synchronization



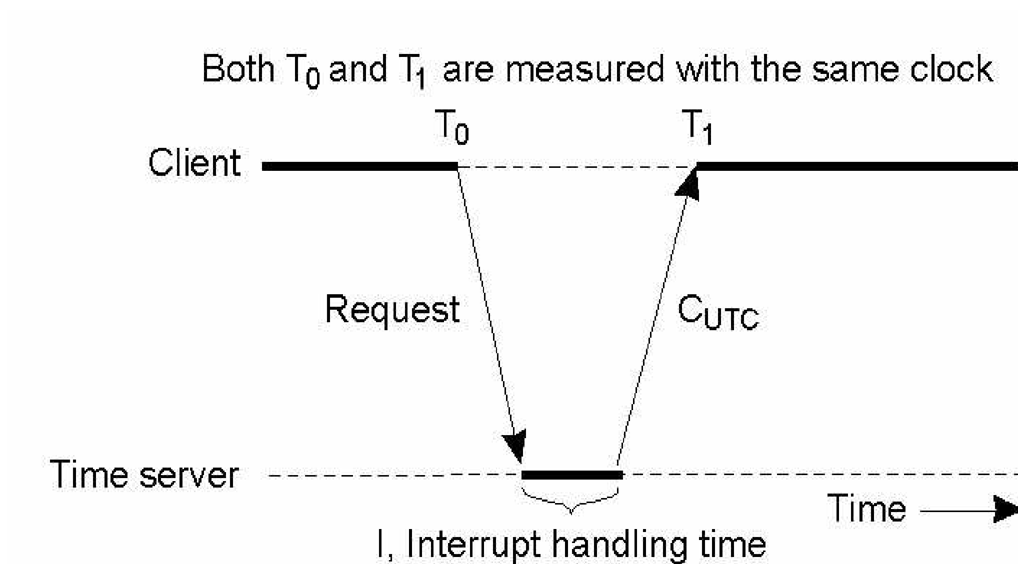

Time t is sent S -> p, p sets it clock to t+est. propagation delay.

propagation delay is variable!

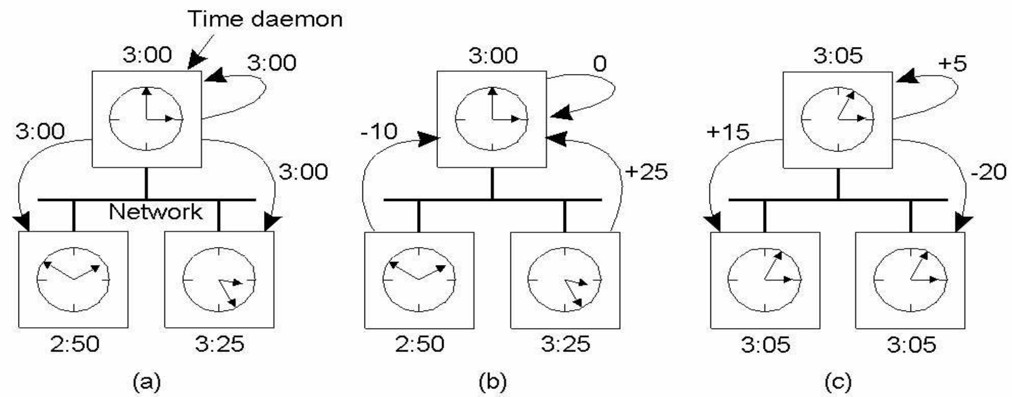
If max & min are known, simplest approach is use the mean.

The internet?

## Cristian's Algorithm



# Berkeley algorithm




**The time daemon asks all the other machines for their clock values**

**The machines answer**

**The time daemon tells everyone how to adjust their clock (NOT the correct time!)**

# The (S)NTP Protocol

Currently the standard for Internet time

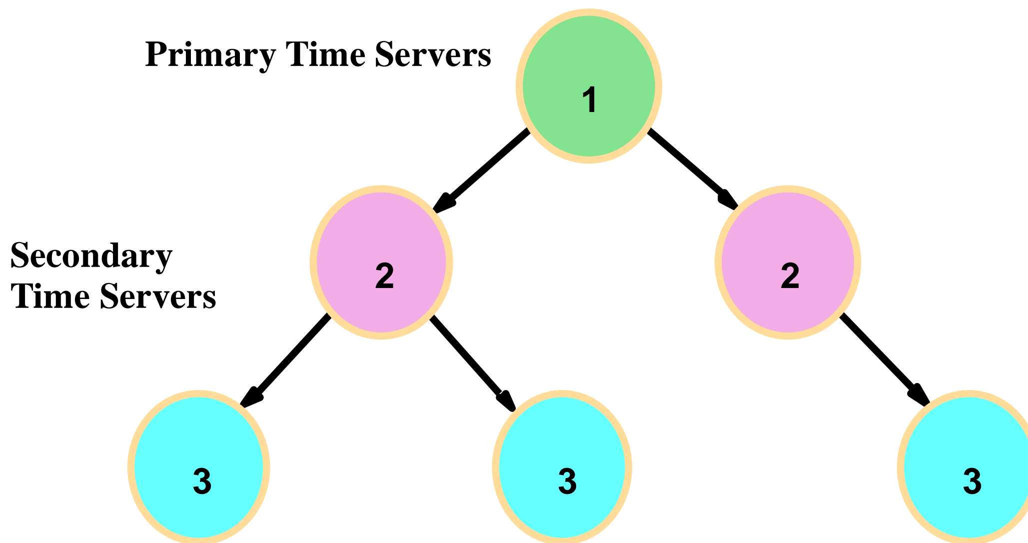
Multicast Lab exercise!

At the lowest level = Cristian's algorithm

Orientated to unreliable WAN's with a hierarchy of time servers

- Primary – have a time source
- Secondary – sync with Primaries
- Tertiary – sync with Secondaries

All the way down a synchronisation subnet.




Accurate to the order of 10's of ms for internet use.

Three modes:

- Multicast - (lab exercise) LAN only
- Procedure call - server replies to calls with its current time
- Symmetric – Timing data is exchanged and maintained on each server over time.

# Logical Time

So far we've discussed "real" time. Potentially a hard problem.

In reality, we may never be able to achieve the desired for accuracy.

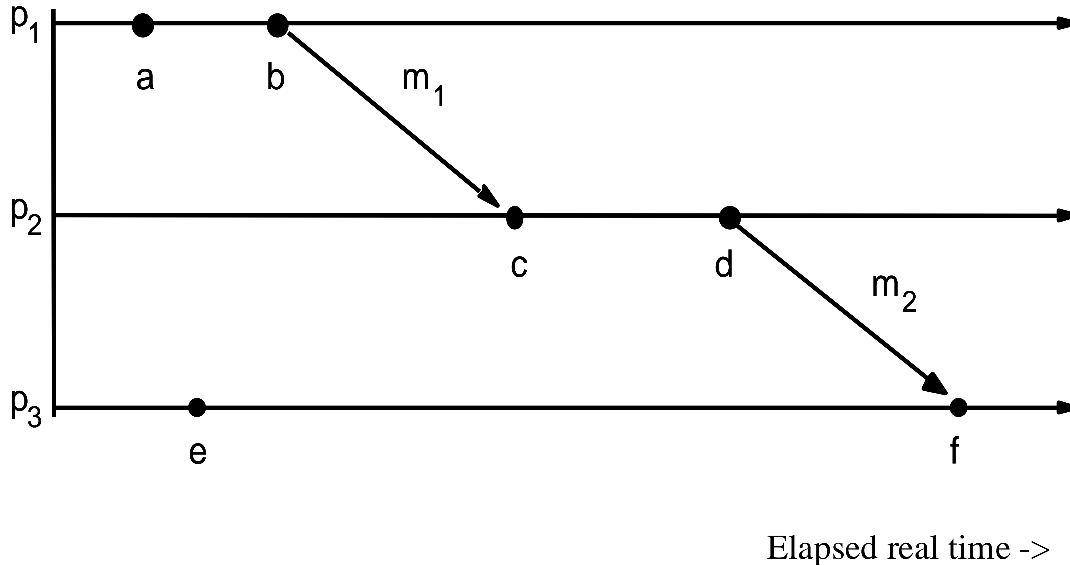
Many situations only require knowledge of sequences of events, event ordering.

Lamport defined these as the "***happened before***" or "***potential causal ordering***" relationship.

**Lamport pointed out in 1979 that what usually matters is not that processes agree on exactly what time it is but rather agree on the order in which events occur. He defined the *happened before* relation denoted ->**

- ***a -> b*** **means event a happened before event b**
- **If a and b are events in the same process and a happened before b then a->b is true**
- **If a is the event of sending a message to another process and b is the event of receiving it then a->b**
- **If a->b and b->c then a->c**
- **if two events X and Y happen in different processes that do not exchange messages then**
    - **x -> y is not true & y-> x is not true**
    - **the events are said to be concurrent (x ||y)**

# An example (figure 10.5, p397 CDK)




Elapsed real time ->

**The following are true:-**

**a->b; b->c; c->d; d->f; a->f; a||e;**

## *How do the (Lamport logical) clocks tick?*

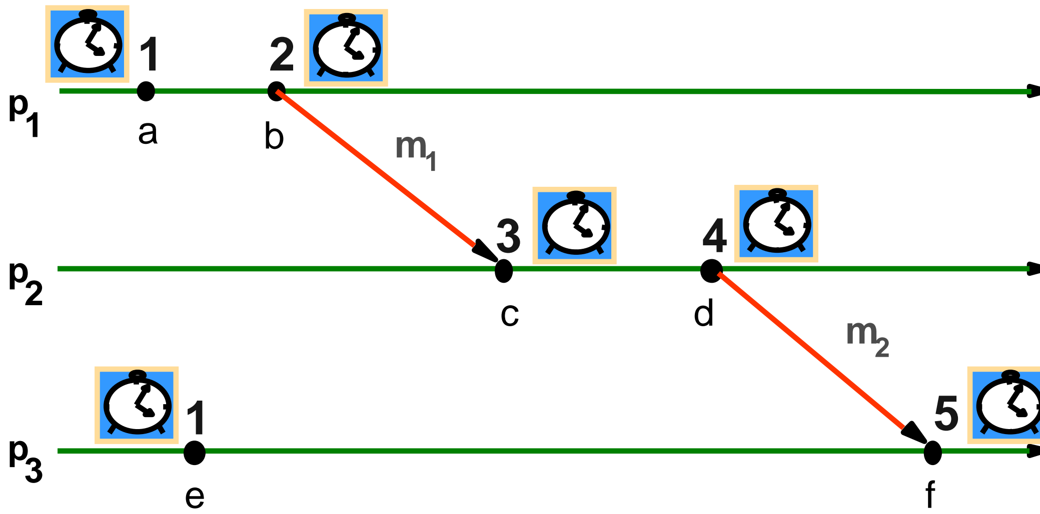
A software counter is assigned to an event such that the number can be thought of as the time that the event occurred. A clock $C_i$ for each process $P_i$ assigns a number $C_i(a)$ to an event a in that process. For the clock to be correct then the following conditions hold:-

- **For any event a, b, in process $P_i$ if a->b then $C_i(a) < C_i(b)$**
- **If a is the sending of a message by process Pi & b is the receipt of that message by process $P_j$ then $C_i(a) < C_j(b)$**

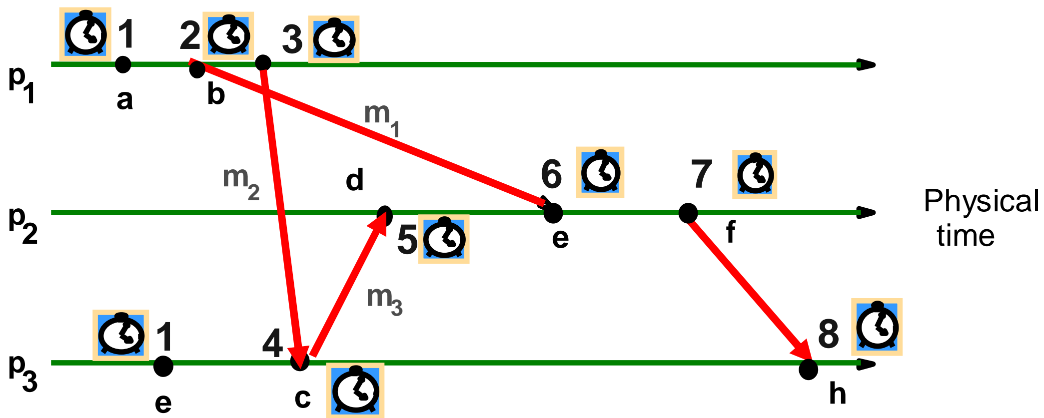
The following implementation rules must be obeyed

- ***Each process $P_i$ increments $C_i$ between any two successive events***
- ***If event a is the sending of a message m by $P_i$ then the message m contains the timestamp $T_m = C_i(a)$***
- ***When receiving a message m process Pj sets Cj to ( > or = to its current value and > Tm )***

## *Same example with logical clocks:*




However LLC's do not capture causality across processors. Consider:




Process $P_1$ sends message $m_1$ to $P_2$ & then $m_2$ to $P_3$ …
… but $P_2$ concludes that $m_3$ must have been sent before $m_1$

L(e) < L(e') does NOT mean e -> e', Lamport clocks do not capture causality!

# Vector Clocks (Mattern & Fidge)

The previous example illustrates FIFO message order delivery which only guarantees order for messages sent from the same process.

What is required is that the message order is extended to include all messages that are causally related known as *Causal Delivery*.

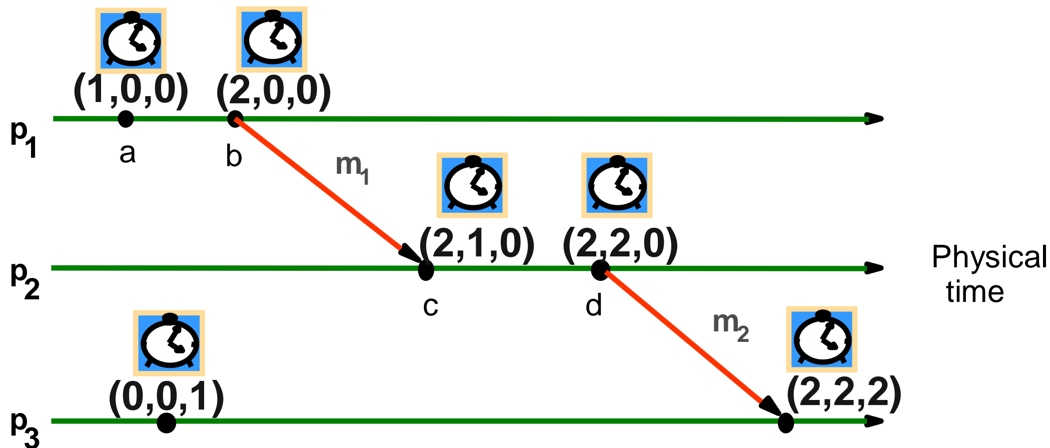
## *Causal Delivery Rule*

**for all messages m, m* from the sending processes $P_i$ and $P_j$ to the destination $P_k$ then if:-**

**$send_i$ (m ) -> $send_j$ (m* )**

**then**

**$deliver_k$ (m) -> $deliver_k$ (m*)**




**In this example each message contains a time stamp (TS) which is the updated vector clock of this send event.**

## *Vector Clock Update Rules*

**For any process Pi when**

- **an internal event or send event occurs then;**

**VC[ i ] = VC[ i ] + 1**

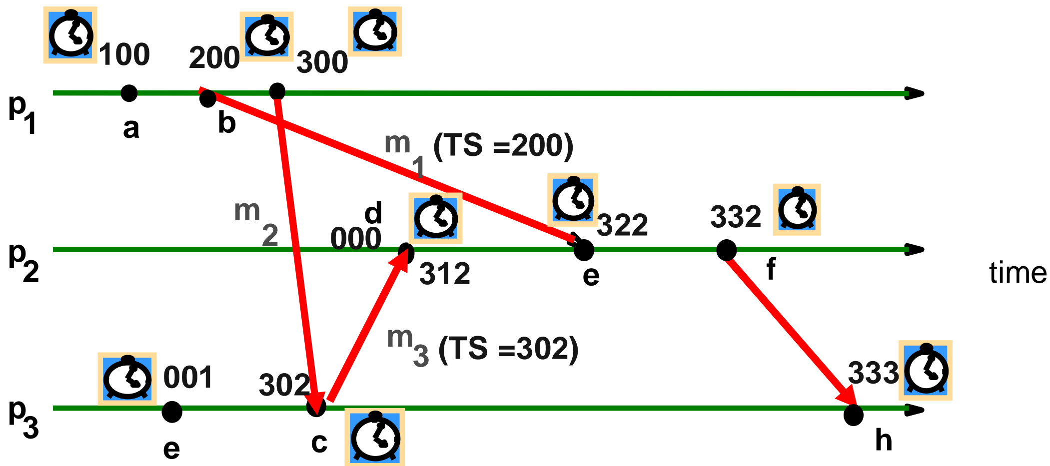
**If the event is the receiving of a messages m, then:-**

**VC[ ] = Max { VC[ ], TS( m ) }**

**and**

**VC[ i ] := VC[ i ] + 1**

## *Our previous "problem" example, this time with vector clocks.*




Now $P_2$ receives message $m_3$ with the time-stamp 302 & compares with its clock 000 so obviously it has missed events; so should it wait before delivering this message in case there are some more messages (like $m_1$ ) in transit sent at an earlier time. In a *totally reliable* network it can wait.

The vector time-stamp associated with vector clocks provides enough information for each process to decide whether there might exist a proceeding message.

If so it will buffer the current message and deliver in order later. However this is not a realistic solution.

It can only be done by

- multicasting to all processes within the closed group known as a causal broadcast [Birman-Schiper-Stephenson].

- a centralised monitor which is responsible for the delivery of all messages

However, a reliable [or atomic] causal broadcast would be ideal.

# Revisiting global state

A large class of distributed systems problems can be cast as a construction of consistent global states and the evaluation of properties over these states ( global predicate).

- deadlock detection
- system optimization
- distributed monitoring and control

Determining a condition such as deadlock amounts to evaluating a global state predicate. It requires evaluating from a set of global states of system processes whether a particular property is {True, False}. This is a *stable predicate* since once this state is achieved it stays in this state as compared with a *non-stable predicate*.

At an instant of time we wish to take a distributed snapshot of the system. In particular we wish the snapshot to reflect a consistent global state.

The essential problem is the absence of global time because as we have seen *perfectly* synchronized clocks is impossible.

## *Global history & Consistent cuts*

To determine global state each process Pi can record the events that took place and the corresponding states ( hi)

The global history can then be constructed as the union

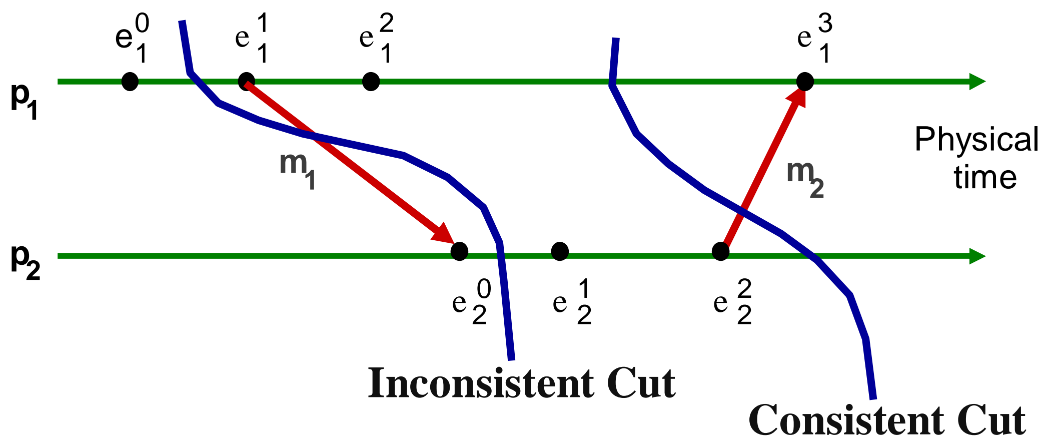
H = { h0 , h1, h2, h3, …….. hn }

We can also take any set of states from the individual processes and form a global state

S = { s0 , s1, s2, s3, …….. sn }

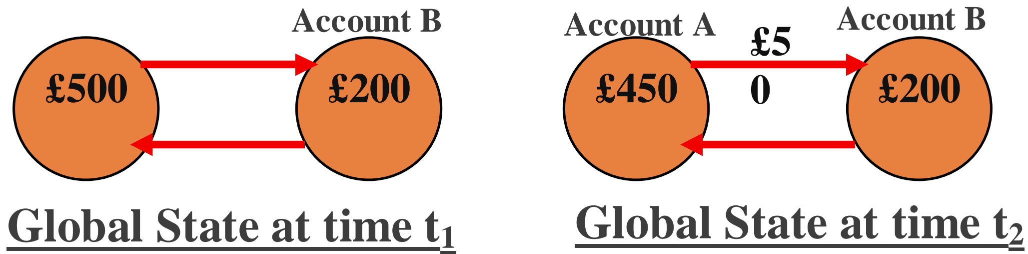
But is S meaningful in other words did the process states occur at the same time. The notion of a global state can be represented by what is called a cut

A cut represents the last event recorded for each process.




**A *cut C is consistent*, if for each event it contains, it also contains all the events that happened before that event**

***A consistent global state is one that corresponds to a consistent global cut.***




The global predicate:

total savings across all bank accounts = £700

…but if only the process states are recorded then at t2 it appears that £50 has been lost.

**Any algorithm to record global state must include the state of the communication channels.**

# Chandy & Lamport's Snapshot algorithm

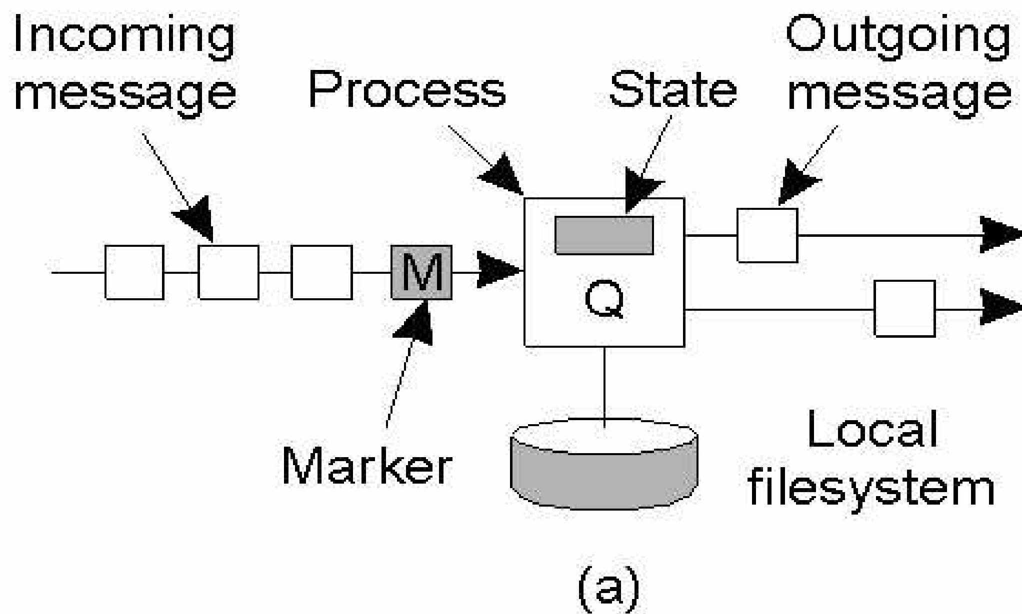
*For much, much more detail see sec. 10.5.3!*

**Chandy & Lamport (1985) describe a snapshot algorithm for determining global states of a distributed system.**

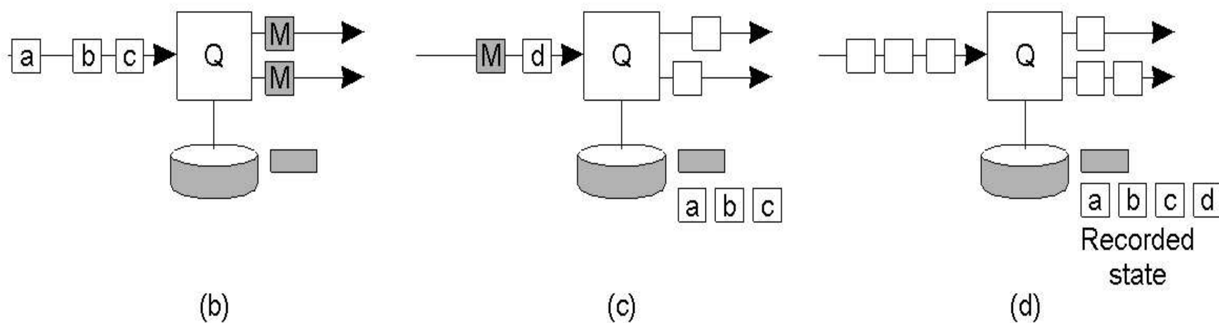
The goal of the algorithm is to record a set of process and channel states for a set of processes such that even though the combination of recorded states may never have occurred at the same time, the global state is consistent.

Assumptions

- channels & processes don't fail, communication is reliable
- channels provide FIFO ordered exactly once message delivery
- there is a connection between any two processes




Any process may initiate the algorithm. The initiating process, say P, starts by recording its own local state. It then sends a marker along each of its out-going channels indicating that the receiver should participate in recording global state. When a process Q receives a marker its action depends on whether it has stored its local state.




(b) Process Q receives a marker for the first time and records its local state and sends markers on all of its out-going channels
(c) Q then records all incoming messages
(d) When *Q* receives markers from each of its incoming channels it finishes recording the state of the incoming channel

At this point it's recorded state as well as the recorded state for each incoming channel can be collected and analyzed meanwhile the distributed system can run as normal

***Marker receiving rule for process pi***

On *pi*'s receipt of a *marker* message over channel *c*:

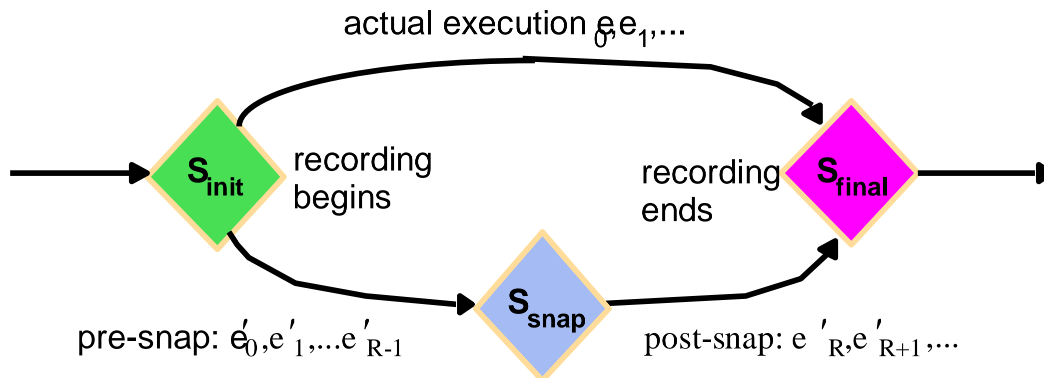
*if* (*pi* has not yet recorded its state) it
records its process state now;
records the state of *c* as the empty set;
turns on recording of messages arriving over other incoming channels;

*else*
*pi* records the state of *c* as the set of messages it has received over *c* since it saved its state.

*end if*

***Marker sending rule for process pi***

After *pi* has recorded its state, for each outgoing channel *c*:
*pi* sends one marker message over *c*
(before it sends any other message over *c*).




If a stable predicate is true in $S_{snap}$ then it is true in $S_{final}$ & $S_{init}$

However for an unstable predicate then if it is true in $S_{snap}$ then it may or may not be true in $S_{final}$