Chapter 7

Clocks

But always at my heels I hear
Times winged chariot hurrying near

Andrew Marvell

This thing all things devours: Birds, beasts, trees, flowers; Gnaws iron, bites steel; Grinds hard stones to meal; Slays king, ruins town, And beats high mountain down.

Time in (distributed) computing systems

Why?
To coordinate between processes.

Charging is elapsed time.

Access a resource … who asked first?
To come to a decision …. has everyone replied?

Time difference. No absolute measurement

To establish what happened and in what order - to establish correctness.

How?
Use the system clock – works if all processes are running on a single machine (may be multi-core so genuinely parallel execution)

On a distributed system we would like to find a single universal time source without making the system synchronous

Everyone needing to refer to central time.

Synchronising Clocks

To define state would like a global time.

A global time does not exist (Einstein).
There is no theoretical time which we can access.

Is this a problem?

Local (earth time). GR implies differences of 1ns/day/1000m in height.

Modern computers run at ~ 3-5 instructions/ns.

Light speed c=30cm/ns
signal speed along a wire is 1/3 to1/2 of c
Need to define the "wire" distance of n machines to within a few centimetres. Not practical.

Using GPS clocks – get a few metres – still not good enough

Synchronising Clocks

Propagation delay time

For the state of computer A to affect the state of computer B, the state of computer A must be communicated to computer B.

But you cannot be sure what the time separation is between two computers well enough to be useful, because of routers this time is not even constant
Even if a universal time exists – doesn't help.

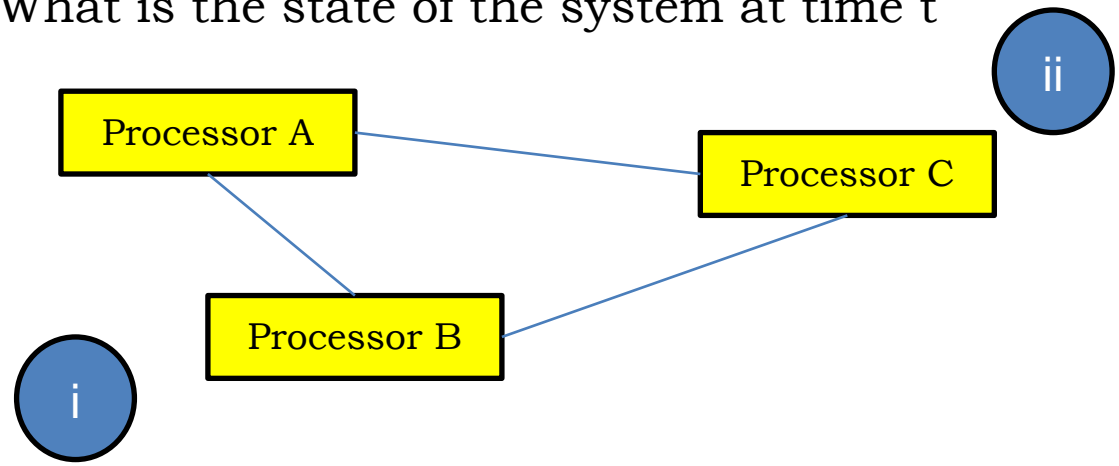
Result in one sub-job can only communicate with another sub job at a time given by the maximum speed of propagation of information.

Attempting to make everything work off a universal time and the problem of propagation delay still exits.

Independent of relativity, there is a maximum transmission speed for information in a number of computer systems.

Synchronising Clocks

What is the state of the system at time t

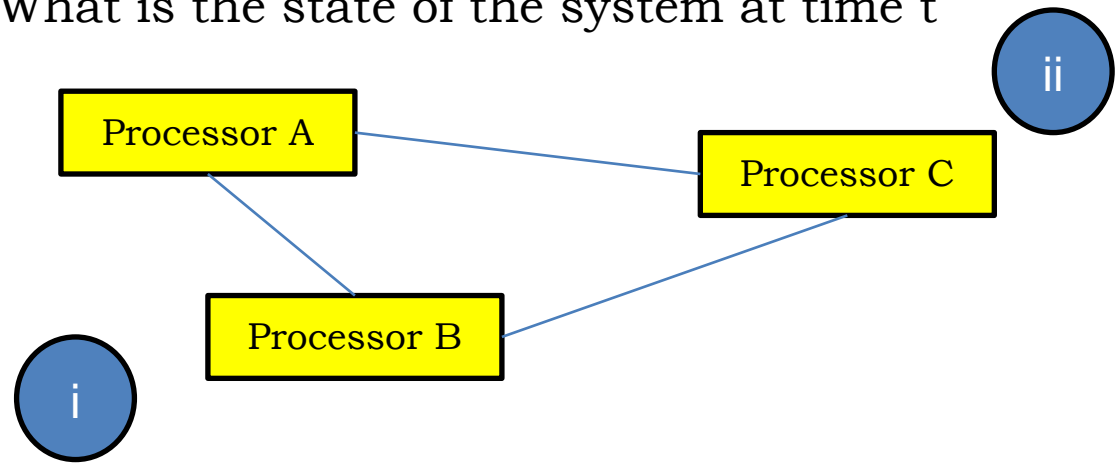


What do we mean by this?

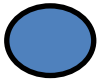
We can define an observer (machine or person), who has a clock and can describe the state of the system at any instant.
Observer ii knows the state of C at time t, but can only deduce the state of processor A sometime later when the information has reached C. But processor C wants to make a decision at time t.

Synchronising Clocks

What is the state of the system at time t

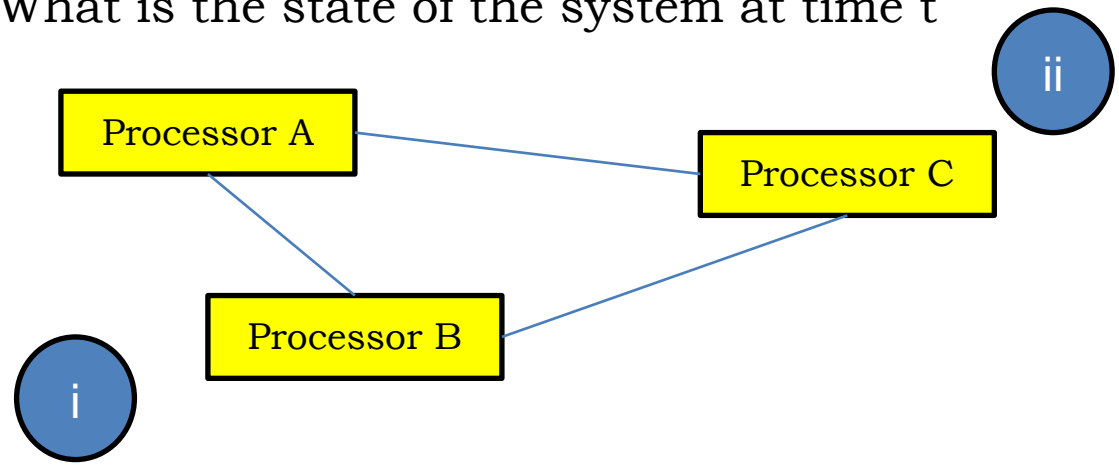


Observer i knows the state of A at time t, but can only deduce the state of processor C sometime later when the information has reached A. But processor A wants to make a decision at time t.

There is no special observer  who at time t knows the state of A, B and C, such that they can communicate to A,B and C the information necessary for them to make a decision which requires knowledge of the state of the other machines.

Synchronising Clocks

What is the state of the system at time t

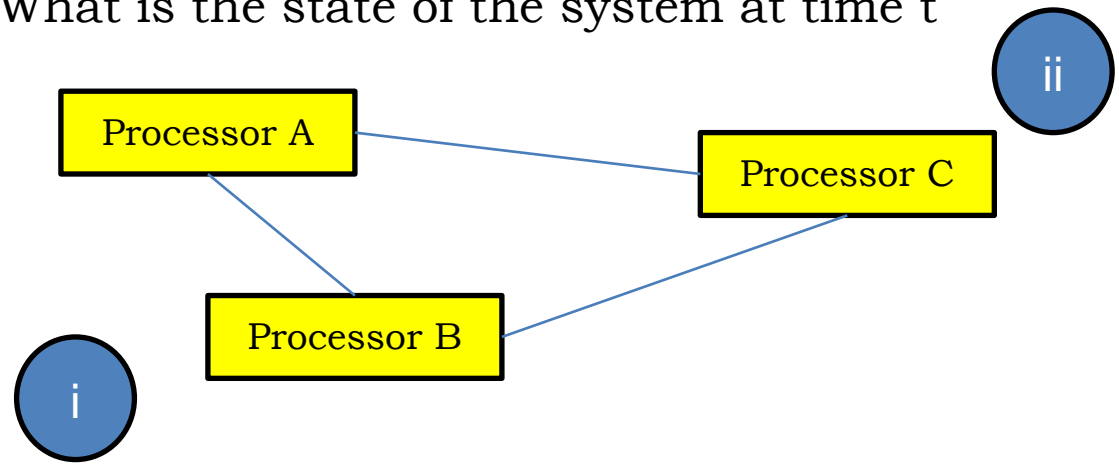


Even the time interval is not well defined. Signal paths and delays in routers/switches as typically many CPU cycles.

Apart from the fact the fact that a process which ran on A on one invocation may run on C next time; and for resilience may even migrate from one to the other.

Synchronising Clocks

What is the state of the system at time t



Look at a scene & you are looking back in time, by an amount which varies with distance.

As true looking at cars as stars.

[There is another problem with cars you reaction time is about 150ms. That means driving at 100 km/hour and you are making decisions based on relative position information which is about 10 metres wrong.]

Synchronising Clocks

The naïve idea of simultaneity only works because propagation delay time is normally rather small compared to the typical timescale of the system.

It is normally accepted that winning a war involves winning the last battle.

The war of 1812-1815 between Britain and USA.

The battle of New Orleans which was the last battle was won by the USA
The war was won by the British.

The peace treaty which was negotiated in London was signed “before” the battle of New Orleans had been fought.

Time in (distributed) computing systems

Causal Order

How could you ensure (in 1815) a battle did not occur after the treaty had been signed.

Lesson: any distributed system (in the absence of a universal clock) has trouble defining a time order.

Independence

When thinking of European History we use time to think about the relationship between events.

French Revolution introduced a new calendar. For discussions about the situation in France many historians choose to use the revolutionary calendar.

England went from the Julian to the Gregorian Calendar nearly 200 years after much of Europe (Sweden went a year later). Discussing English History during this period people tend to use the Julian Calendar. *Not when talking about foreign policy*

A “common” clock is only important when discussing interacting systems.

A **new** definition of a clock is going to be needed

We are going to introduce a rather different idea of what constitutes a clock.

But what you think about when you consider a clock is actually flexible and not necessarily exactly defined

Causal Order

A clock is something which increments its value as time passes.

A clock in a machine defines the time for that machine, and because a processor works on a clock tic, we can define a state of the system at a given time – by which we mean a given clock cycle. The clock tics *cause* the operations

A distributed system has a number of clocks which we need to "synchronise".

We do that by using the idea of causality.
An event *e* which can cause (or influence) another event *e'* must occur before it.
But in some cases we can neither say that
$e < e'$ nor $e' < e$

For a single machine we can unambiguously order all events.

Causal Order

In causal ordering some events can be provably ordered in time: $e < e'$.

Furthermore we assume that a message cannot be received before it is sent.
Thus the reception event *e'* must occur after the transmission event *e*.

But some events *e* & *e''* have no provable order.
We cannot say which came first.

Events in a distributed system are only partially ordered.

Each processor must maintain its own clock which it can use to provide a timestamp to relevant events.
.

Universal time

Events in the universe are only partially ordered.

Event 1 occurs and a flash of light departs in the direction of event 2.
Event 2 occurs and a flash of light departs in the direction of event 1

If event 2 happens before flash 1 arrives and event 1 happens before flash 2 arrives
There is **no** time order of the events

Since we can run a universe on that basis, it is perhaps not surprising that we can run a distributed system on such a basis.

Happens before

We can follow Leslie Lamport.
Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

and define the **happens before** relation $\Rightarrow_S$ on a schedule S. (S is the set of all events.

1.All pairs (*e, e'*) where *e* precedes *e'* in *S* and the events occur in the same process.
2.All pairs (*e, e'*) where *e* is a send event and *e'* is a receive event for the same message.
3.All pairs (*e, e'*) where there exists a third event *e''* where $e \Rightarrow_S e''$ and $e'' \Rightarrow_S e'$

A **causal shuffle** to *S'* preserves the happens before relation.
A causal shuffle on *S* will produce a similar schedule *S'*
Similar schedules are indistinguishable to all processes (participants).

A causal shuffle looks like a change of observer in relativity

Lamport's clock

We can follow Leslie Lamport and define a clock locally which allows us to define the *happens before* relation.

Every process maintains a local variable **clock**.

When a process executes an internal step or sends a message it sets **clock** ← **clock** + 1 and labels the step or the message with the new value of **clock**.

When a process receives a message with a time *t*, it sets its clock **clock** ← max(**clock**,t)
The time of receipt is set to this new time.

If we order the events locally by **clock** we get an order which is indistinguishable from the original execution.

Jumps in Lamport's clock

Because a process receiving a message does a
clock ← max(**clock**,t)
The clock may make a large jump – but if internally it is doing a regular task based on the value of the clock then the performance of that task may be compromised.

A solution from Neiger, Toueg and Welch.
Welch: Simulating synchronous processors. *Inf. Comput.*, 74(2):159–170, 1987.
Neiger & Toueg. Substituting for real time and common knowledge in asynchronous distributed systems. *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, PODC '87*, pages 281–293, New York, NY, USA, 1987. ACM.

clock is extended to ⟨clock, id, eventCount⟩ clock is incremented as before by the processor and eventCount is a count of send/receive and possibly local computational steps.

Neiger-Toueg-Welch clock

A message is received with a timestamp later than the local clock. Its delivery is delayed until the local clock exceeds the value of the clock in the message.

Of course if the clock on the receiving machine has a value significantly less than the message, then this might result in a lengthy delay before the message is “received”

This may slow down the response of the whole system.

However if we start the clock based on system time and the system time is set using an NTP protocol – then any delays should be minimal.