Introduction  In this part of the course we will cover:

- Why time is such an issue for distributed computing
- The problem of maintaining a global state
- Consequences of these two main ideas
- Methods to get around these problems
Global Notion of Time

- Einstein showed that the speed of light is constant for all observers regardless of their own velocity.
- He (and others) have shown that this forced several other (sometimes counter-intuitive) properties including:
  1. length contraction
  2. time dilation
  3. relativity of simultaneity
    - Contradicting the classical notion that the duration of the time interval between two events is equal for all observers.
    - It is impossible to say whether two events occur at the same time, if those two events are separated by space.
    - A drum beat in Japan and a car crash in Brazil.
    - However, if the two events are causally connected — if A causes B — the RoS preserves the causal order.
Global Notion of Time

However, if the two events are causally connected — if A causes B — the relativity of simultaneity preserves the causal order.

In this case, the flash of light happens before the light reaches either end of the carriage for all observers.
Global Notion of Time

In Our World

- We operate as if this were not true, that is, as if there were some global notion of time
- People may tell you that this is because:
- On the scale of the differences in our frames of references, the effect of relativity is negligible
Global Notion of Time

In Our World

- We operate as if this were not true, that is, as if there were some global notion of time
- People may tell you that this is because:
- On the scale of the differences in our frames of references, the effect of relativity is negligible
- It’s true that on our scales the effects of relativity are negligible
- But that’s not really why we operate as if there was a global notion of time
- Even if our theoretical clocks are well synchronised, or mechanical ones are not
- We just accept this inherent inaccuracy build that into our (social) protocols
Physical Clocks

- Computer clocks tend to rely on the oscillations occurring in a crystal.
- The difference between the instantaneous readings of two separate clocks is termed their “skew.”
- The “drift” between any two clocks is the difference in the rates at which they are progressing. The rate of change of the skew.
- The drift rate of a given clock is the drift from a nominal “perfect” clock, for quartz crystal clocks this is about $10^{-6}$.
- Meaning it will drift from a perfect clock by about 1 second every 1 million seconds — 11 and a half days.
Global Notion of Time

Coordinated Universal Time and French

- The most accurate clocks are based on atomic oscillators
- Atomic clocks are used as the basis for the international standard International Atomic Time
- Abbreviated to TAI from the French Temps Atomique International
- Since 1967 a standard second is defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of Caesium-133 ($Cs^{133}$).
- Time was originally bound to astronomical time, but astronomical and atomic time tend to get out of step
- Coordinated Universal Time — basically the same as TAI but with leap seconds inserted
- Abbreviated to UTC again from the French Temps Universel Coordonné
Correctness of Clocks

- What does it mean for a clock to be correct?
- The operating system reads the node’s hardware clock value, \( H(t) \), scales it and adds an offset so as to produce a software clock \( C(t) = \alpha H(t) + \beta \) which measures real, physical time \( t \)
- Suppose we have two real times \( t \) and \( t' \) such that \( t < t' \)
- A physical clock, \( H \), is correct with respect to a given bound ‘\( p \)’ if:
  \[(1 - p)(t' - t) \leq H(t') - H(t) \leq (1 + p)(t' - t)\]
- \((t' - t)\) — The true length of the interval
  - The measured length of the interval
  - The smallest acceptable length of the interval
  - The largest acceptable length of the interval
Global Notion of Time

Correctness of Clocks

\[(1 - p)(t' - t) \leq H(t') - H(t) \leq (1 + p)(t' - t)\]

- An important feature of this definition is that it is monotonic
- Meaning that:
- If \( t < t' \) then \( H(t) < H(t') \)
- Assuming that \( t < t' \) with respect to the precision of the hardware clock
Monotonicity

- What happens when a clock is determined to be running fast?
- We could just set the clock back:
- but that would break monotonicity
- Instead, we retain monotonicity:
  - \( C_i(t) = \alpha H(t) + \beta \)
  - decreasing \( \beta \) such that \( C_i(t) \leq C_i(t') \) for all \( t < t' \)
External vs Internal Synchronisation

- Intuitively, multiple clocks may be synchronised with respect to each other, or with respect to an external source.
- Formally, for a synchronisation bound $D > 0$ and external source $S$:
  - **Internal Syncronisation**
    - $|C_i(t) - C_j(t)| < D$
    - No two clocks disagree by $D$ or more
  - **External Syncronisation**
    - $|C_i(t) - S(t)| < D$
    - No clock disagrees with external source $S$ by $D$ or more

- Internally synchronised clocks may not be very accurate at all with respect to some external source.
- Clocks which are externally synchronised to a bound of $D$ though are automatically internally synchronised to a bound of $2 \times D$. 
Synchronising Clocks

Synchronising in a synchronous system

- Imagine trying to synchronise watches using text messaging
- Except that you have bounds for how long a text message will take
- How would you do this?
  1. Mario sends the time \( t \) on his watch to Luigi in a message \( m \)
  2. Luigi should set his watch to \( t + T_{\text{trans}} \) where \( T_{\text{trans}} \) is the time taken to transmit and receive the message \( m \)
  3. Unfortunately \( T_{\text{trans}} \) is only bound, it is not known
  4. We do know that \( \min \leq T_{\text{trans}} \leq \max \)
  5. We can therefore acheive a bound of \( u = \max - \min \) if the Luigi sets his watch to \( t + \min \) or \( t + \max \)
  6. We can do a bit better an achieve a bound of \( u = \frac{\max - \min}{2} \) if Luigi sets his watch to \( t + \frac{\max + \min}{2} \)
  7. More generally if there are \( N \) clocks (Mario, Luigi, Peach, Toad, ...) we can achieve a bound of \((\max - \min)(1 - \frac{1}{N})\)
  8. Or more simply we make Mario an external source and the bound is then \( \max - \min \) (or \( 2 \times \frac{\max - \min}{2} \))
Synchronising Clocks

Cristian’s Method

- The previous method does not work where we have no upper bound on message delivery time, i.e. in an asynchronous system.
- Cristian’s method is a method to synchronise clocks to an external source.
- This could be used to provide external or internal synchronisation as before, depending on whether the source is itself externally synchronised or not.
- The key idea is that while we might not have an upper bound on how long a single message takes, we can have an upper bound on how long a round-trip took.
- However it requires that the round-trip time is sufficiently short as compared to the required accuracy.
Cristian’s Method

- Luigi sends Mario (our source/server) a message $m_r$ requesting the current time, and records the time $T_{sent}$ at which $m_r$ was sent according to Luigi’s current clock.
- Upon receiving Luigi’s request message $m_r$ Mario responds with the current time according to his clock in the message $m_t$.
- When Luigi receives Mario’s time $t$ in message $m_t$, at time $T_{rec}$ according to his own clock the round trip took
  \[ T_{round} = T_{rec} - T_{sent} \]
- Luigi then sets his clock to $t + \frac{T_{round}}{2}$
- Which assumes that the elapsed time was split evenly between the exchange of the two messages.
Synchronising Clocks

Cristian’s Method

▶ How accurate is this?
▶ We often don’t have accurate upper bounds for message delivery times but frequently we can at least guess conservative lower bounds
▶ Assume that messages take at least $min$ time to be delivered
▶ The earliest time at which Mario could have placed his time into the response message $m_t$ is $min$ after Luigi sent his request message $m_r$.
▶ The latest time at which Mario could have done this was $min$ before Luigi receives the response message $m_t$.
▶ The time on Mario’s watch when Luigi receives the response $m_t$ is:
    ▶ At least $t + min$
    ▶ At most $t + T_{round} − min$
    ▶ Hence the width is $T_{round} − (2 \times min)$
▶ The accuracy is therefore $\frac{T_{round}}{2} − min$
Synchronising Clocks

The Berkley Algorithm

- Like Cristian’s algorithm this provides either external synchronisation to a known server, or internal synchronisation via choosing one of the players to be the master.
- Unlike Cristian’s algorithm though, the master in this case does not wait for requests from the other clocks to be synchronised, rather it periodically polls the other clocks.
- The other’s then reply with a message containing their current time.
- The master, estimates the slaves current times using the round trip time in a similar way to Cristian’s algorithm.
- It then averages those clock readings together with its own to determine what should be the current time.
- It then replies to each of the other players with the amount by which they should adjust their clocks.
Synchronising Clocks

The Berkley Algorithm

▶ If a straight forward average is taken a faulty clock could shift this average by a large amount, and therefore a fault tolerant average is taken

▶ This is exactly as it sounds, it averages all the clocks that do not differ by a chosen maximum amount.
Network Time Protocol

Pairwise synchronisation

- Similar to Cristian’s method however:
- Four times are recorded as measured by the clock of the process at which the event occurs:
  1. $T_{i-3}$ — Time of sending of the request message $m_r$
  2. $T_{i-2}$ — Time of receiving of the request message $m_r$
  3. $T_{i-1}$ — Time of sending of the response message $m_t$
  4. $T_i$ — Time of receiving of the response message $m_t$
- So if Luigi is requesting the time from Mario, then $T_{i-3}$ and $T_i$ are recorded by Luigi and $T_{i-2}$ and $T_{i-1}$ are recorded by Mario
- Note that because Mario records the time at which the request message was received and the time at which the response message is sent, there can be a non-negligible delay between both
- In particular then messages may be dropped
Network Time Protocol

Pairwise synchronisation

- If we assume that the true offset between the two clocks is $O_{true}$:
- And that the actual transmission times for the messages $m_r$ and $m_t$ are $t$ and $t'$ respectively then:
- $T_{i-2} = T_{i-3} + t + O_{true}$ and
- $T_i = T_{i-1} + t' - O_{true}$
- $T_{round} = (t + t') = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- $O_{guess} = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2}$
Pairwise synchronisation

- This is the non-trivial line:
  \[ O_{\text{guess}} = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2} \]
  \[ T_{i-2} - T_{i-3} = t + O_{\text{true}} \]
  \[ T_{i-1} - T_i = O_{\text{true}} - t' \]
  \[ = (t - t') + (2 \times O_{\text{true}}) \]

- \[ O_{\text{guess}} = \frac{t - t'}{2} + O_{\text{true}} \]
- \[ O_{\text{true}} = O_{\text{guess}} + \frac{(t - t')}{2} \]

Since we know that \( T_{\text{round}} > |t - t'| \):

- \[ O_{\text{guess}} - \frac{T_{\text{round}}}{2} \leq O_{\text{true}} \leq O_{\text{guess}} + \frac{T_{\text{round}}}{2} \]

- \( O_{\text{guess}} \) is the guess as to the offset

- \( T_{\text{round}} \) is the measure of how accurate it is which is essentially based on how long the messages were in transit
Network Time Protocol

- Network Time Protocol (actually abbreviated was NTP) is designed to allow clients to synchronise with UTC over the Internet.
- NTP is provided by a network of servers located across the Internet.
- Primary servers are connected directly to a time source such as a radio clock receiving UTC.
- Other servers are connected in a tree, with their *strata* determined by how many branches are between them and a primary server.
- Strata N servers synchronise with Strata N - 1 servers.
- Eventually a server is within a user’s workstation.
- Errors may be introduced at each level of synchronisation and they are cumulative, so the higher the strata number the less accurate is the server.
Network Time Protocol

Note: this picture does not show synchronisation between servers at the same strata, but this does occur
Synchronising Clocks

Network Time Protocol

1. Multicast mode
   ▶ Not considered very accurate
   ▶ Intended for use on a high-speed LAN
   ▶ Can be accurate enough nonetheless for some purposes

2. Procedure call mode
   ▶ Similar to Cristian’s method
   ▶ Servers respond to requests from higher-strata servers
   ▶ Who use round-trip times to calculate the current time to some degree of accuracy
   ▶ Used for example in network file servers which wish to keep as accurate as possible file access times

3. Symmetric mode
   ▶ Used where the highest accuracies are required
   ▶ In particular between servers nearest the primary sources, that is the lower strata servers
   ▶ Essentially similar to procedure-call mode except that the communicating servers retain timing information to improve their accuracy over time
Overview

- In all three modes messages are delivered using the standard UDP protocol
- Hence message delivery is unreliable
- At the higher strata servers can synchronise to high degree of accuracy over time
- But in general NTP is useful for synchronising accurately to UTC, whereby accurate is at the human level of accuracy
- Wall clocks, clocks at stations etc
- In summary: we can synchronise clocks to a bounded level of accuracy, but for many applications the bound is simply not tight enough
Logical Clocks

Asynchronous Orderings

► So we can achieve some measure of synchronisation between physical clocks located at different sites
► Ultimately though we will never be able to synchronise clocks to arbitrary precision
► For some applications low precision is enough, for others it is not.
► Where we cannot guarantee a high enough order of precision for synchronisation, we are forced to operate in the asynchronous world
► Despite this we can still provide a logical ordering on events, which may useful for certain applications
Logical Clocks

Logical Orderings

Logical orderings attempt to give an order to events similar to physical causal ordering of reality but applied to distributed processes.

Logical clocks are based on the simple principles:

- Any process can order the events which it observes/execute.
- Any message must be sent before it is received.
More formally we define the happened-before relation $\rightarrow$ by the three rules:

1. If $e_1$ and $e_2$ are two events that happen in a single process and $e_1$ proceeds $e_2$ then $e_1 \rightarrow e_2$
2. If $e_1$ is the sending of message $m$ and $e_2$ is the receiving of the same message $m$ then $e_1 \rightarrow e_2$
3. If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3
Logical Clocks

Logical Ordering — A Logical Clock

- Lamport designed an algorithm whereby events in a logical order can be given a numerical value.
- This is a *logical clock*, similar to a program counter except that there is no backward jumping, and so it is monotonically increasing.
- Each process $P_i$ maintains its internal logical clock $L_i$.
- So in order to record the logical ordering of events, each process does the following:
  - $L_i$ is incremented immediately before each event is issued at $P_i$.
  - When the process $P_i$ sends a message $m$ it attaches the value of its logical clock $t = L_i(m)$.
  - Upon receiving a message $(m, t)$ process $P_j$ computes the new value of $L_j$ as $\max(L_j, t)$.
Logical Clocks

Properties

- Key point: using induction we can show that:
  - \( e_1 \rightarrow e_2 \) implies that \( L(e_1) < L(e_2) \)
- However, the converse is not true, that is:
  - \( L(e_1) < L(e_2) \) does not imply that \( e_1 \rightarrow e_2 \)
- It is easy to see why, consider two processes, \( P_1 \) and \( P_2 \) which each perform two steps prior to any communication.
- The two steps on the first process \( P_1 \) are concurrent with both of the two steps on process \( P_2 \).
- In particular \( P_1(e_2) \) is concurrent with \( P_2(e_1) \) but \( L(P_1(e_2)) = 2 \) and \( L(P_2(e_1)) = 1 \)
Logical Clocks

Lamport Clocks — No reverse implication

Here event $L(e) < L(b) < L(c) < L(d) < L(f)$
but only $e \rightarrow f$
e is concurrent with $b$, $c$ and $d$. 
Logical Clocks

Total Ordering

- Just as the happened-before relation is a partial ordering
- So to are the numerical Lamport stamps attached to each event
- That is, some events have the same number attached.
- However we can make it a total ordering by considering the process identifier at which the event took place
- In this case $L_i(e_1) < L_j(e_2)$ if either:
  1. $L_i(e_1) < L_j(e_2)$ OR
  2. $L_i(e_1) = L_j(e_2)$ AND $i < j$
- This has no physical meaning but can sometimes be useful
Vector Clocks augment Logical Clocks

- Vector clocks were developed (by Mattern and Fidge) to overcome the problem of the lack of a reversed implication.
- That is: \( L(e_1) < L(e_2) \) does not imply \( e_1 \rightarrow e_2 \).
- Each process keeps its own vector clock \( V_i \) (an array of Lamport clocks, one for every process).
- The vector clocks are updated according to the following rules:
  1. Initially \( V_i[j] = 0 \)
  2. As with Lamport clocks before each event at process \( P_i \) it updates its own Lamport clock within its own vector clock: \( V_i[i] = V_i[i] + 1 \)
  3. Every message \( P_i \) sends includes its entire vector clock \( t = V_i \)
  4. When \( P_i \) receives a timestamp \( V_x \) then it updates all of its vector clocks with: \( V_i[j] = \max(V_i[j], V_x[j]) \)
Vector Clocks

Vector Clocks augment Logical Clocks

- Vector clocks (or timestamps) are compared as follows:
  1. \( V_x = V_y \) iff \( V_x[i] = V_y[i] \) \( \forall i, 1 \ldots N \)
  2. \( V_x \leq V_y \) iff \( V_x[i] \leq V_y[i] \) \( \forall i, 1 \ldots N \)
  3. \( V_x < V_y \) iff \( V_x[i] < V_y[i] \) \( \forall i, 1 \ldots N \)

- As with logical clocks: \( e_1 \rightarrow e_2 \) implies \( V(e_1) < V(e_2) \)

- In contrast with logical clocks the reverse is also true: \( V(e_1) < V(e_2) \) implies \( e_1 \rightarrow e_2 \)
Vector Clocks augment Logical Clocks

- Of course vector clocks achieve this at the cost of larger time stamps attached to each message
- In particular the size of the timestamps grows proportionally with the number of communicating processes

Summary of Logical Clocks

- Since we cannot achieve arbitrary precision of synchronisation between remote clocks via message passing
- We are forced to accept that some events are concurrent, meaning that we have no way to determine which occurred first
- Despite this we can still achieve a logical ordering of events that is useful for many applications
Correctness of distributed systems frequently hinges upon satisfying some global system invariant.

Even for applications in which you do not expect your algorithm to be correct at all times, it may still be desirable that it is “good enough” at all times.

For example, our distributed algorithm maybe maintaining a record of all transactions:

- In this case it might be okay if some processes are behind other processes and thus do not know about the most recent transactions.
- But we would never want it to be the case that some process is in an inconsistent state, say applying a single transaction twice.
Global State

- Motivating examples:
  1. Distributed garbage collection
  2. Distributed deadlock detection
  3. Distributed termination detection
  4. Distributed debugging
Global State — Absence of a Global Time

You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time
- **Distributed Garbage Collection**
  - Agree a global time for each process to check whether a reference exists to a given object
  - This leaves the problem that a reference may be in transit between processes
  - But each process can say which references they have sent before the agreed time and compare that to the references received at the agreed time
Global State — Absence of a Global Time

You may have thought you got away from global time discussions

► Consider what happens to each of our distributed problems should we have a global time
► Distributed Deadlock Detection
  ► Somewhat depends upon the problem in question, however:
  ► At an agreed time all processes send to some master process the processes or resources for which they are waiting
  ► The master process then simply checks for a loop in the resulting graph
Global State — Absence of a Global Time

You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time

- Distributed Termination Detection
  - At an agreed time each process sends whether or not they have completed to a master process
  - Again this leaves the problem that a message may be in transit at that time
  - Again though, we should be able to work out which messages are still in transit
You may have thought you got away from global time discussions

- Consider what happens to each of our distributed problems should we have a global time

- Distributed Debugging
  - At each point in time we can reconstruct the global state
  - We can also record the entire history of events in the exact order in which they occurred.
  - Allowing us to replay them and inspect the global state to see where things have gone wrong as with traditional debugging
Global State — Consistent Cuts

- So, if we had synchronised clocks, we could agree on a time for each process to record its state.
- The combination of local states and the states of the communication channels would be an actual global state.
- Since we cannot do that, we attempt to find a “cut”.
- A cut is a partition of events into those occurring before the cut and those occurring after the cut.
- The goal is to assemble a meaningful global state from the local states of processes recorded at different times.
A consistent cut is one which does not violate the happens before relation →

If \( e_1 \rightarrow e_2 \) then either:

- both \( e_1 \) and \( e_2 \) are before the cut or
- both \( e_1 \) and \( e_2 \) are after the cut or
- \( e_1 \) is before the cut and \( e_2 \) is after the cut
- **but not**
- \( e_1 \) is after the cut and \( e_2 \) is before the cut
Global State — Consistent Cuts

Runs and Linearisations

- A *consistent global state* is one which corresponds to a consistent cut
- A “run” is a total ordering of all events in a global history which is consistent with the local history of each process
- A “linearisation” is a total ordering of all events in the global history which is consistent with the happens-before relation →
- So all linearisations are also runs
- Not all runs pass through consistent global states but all linearisations pass only through consistent global states
Global State — Safety and Liveness

- When we attempt to examine the global state, we are often concerned with whether or not a property holds.
- Some properties, \( B \), are properties we hope never hold and some properties, \( G \), are properties we hope always hold.
- **Safety** is the property that a bad property \( B \) does not hold for any reachable state.
- **Liveness** is the property that a good property \( G \) holds for all reachable states.
Some properties we wish to establish are *stable* properties.

Such properties may never become true, but once they do they remain true.

Our four example properties:

- **Garbage** is *stable*: once an object has no valid references (at a process or in transit) will never have any valid references.
- **Deadlock** is *stable*: once a set of processes are deadlocked they will always be deadlocked without external intervention.
- **Termination** is *stable*: once a set of processes have terminated they will remain terminated without external intervention.
- **Debugging** is not really a property but the properties we may look for whilst debugging are likely *non-stable*. 
The goal is to record a snapshot, or global state, of a set of processes. The algorithm is such that the combination of recorded states may never have occurred simultaneously. However, the computed global state is always a consistent one. The state is recorded locally at each process. The algorithm also does not address the issue of gathering the recorded global state. Though generally, the locally recorded state can then be sent to some pre-agreed master process.
Assumptions

▶ There is a path between any two pairs of processes, in both directions
▶ Any process may initiate a global snapshot at any time
▶ The processes may continue their execution and send/receive normal messages whilst the snapshot takes place
Global State — Chandy and Lamport

Assumptions

➤ There is a path between any two pairs of processes, in both directions
➤ Any process may initiate a global snapshot at any time
➤ The processes may continue their execution and send/receive normal messages whilst the snapshot takes place
➤ Neither channels nor processes fail
➤ Communication is reliable such that every message that is sent arrives at its destination exactly once
➤ Channels are unidirectional and provide FIFO-ordered message delivery.
Algorithm — Receiver
Receiving rule for process $p_i$:

1. On receipt of a Marker message over channel $c$:
2. if $p_i$ has not yet recorded state:
3. record process state now
4. record the state of $c$ as the empty set
5. turn on recording of messages arriving on all other channels
6. else
7. records the state of $c$ as the set of messages it has recorded since $p_i$ first recorded its state
Algorithm — Sender

Sending rule for process $p_i$:

1. After $p_i$ has recorded its state:
2. $p_i$ sends a marker message for each outgoing channel $c$
3. before it sends any other messages over $c$
Global State — Chandy and Lamport Example

We begin in this global state, where both channels are empty, the states of the processes are as shown, but we say nothing about what has gone before.
The left process decides to begin the snapshot algorithm and sends a Marker message over channel 1 to the left process. It then decides to send a request for 10 items at $10 each.
Meanwhile, the right process responds to an *earlier* request and sends 5 items to the left process over channel 2.
Finally the right process receives the Marker message, and in doing so records its state and sends the left process a Marker message over channel 2. When the left process receives this Marker message it records the state of channel two as containing the 5 items it has received since recording its own state.
The final recorded state is:

Left Process  | $1000, 0
Right Process | $50, 1995
Channel 1     | empty
Channel 2     | Five Items
Reachability

- The cut found by the Chandy and Lamport algorithm is always a consistent cut.
- This means that the global state which is characterised by the algorithm is a consistent global state.
- Though it may not be one that ever occurred.
- We can though define a reachability relation:
  - This is defined via the initial, observed and final global states when the algorithm is run.
  - Assume that the events globally occurred in an order $Sys = e_1, e_2 \ldots$
  - Let $S_{init}$ be the global state immediately before the algorithm commences and $S_{final}$ be the global state immediately after it terminates. Finally $S_{snap}$ is the recorded global state.
  - We can find a permutation of $Sys$ called $Sys'$ which:
    - contains all three states: $S_{init}, S_{snap}$ and $S_{final}$
    - Does not break the happens-before relationship on the events in $Sys$.
It may be that there are two events in Sys, $e_n$ and $e_{n+1}$ such that $e_n$ is a post-snap event and $e_{n+1}$ is a pre-snap event.

However we can swap the order of $e_n$ and $e_{n+1}$ since it cannot be that $e_n \rightarrow e_{n+1}$.

We continue to swap adjacent pairs of events until all pre-snap events are ordered before all post-snap events. This gives us the the linearisation $Sys'$.

The reachability property of the snapshot algorithm is useful for recording stable properties.

However any non-stable predicate which is True in the snapshot may or may not be true in any other state.

Since the snapshot may not have actually occurred.
Use Cases

- No work which depends upon the global state is done until the snapshot has been gathered.
- They are therefore useful for:
  1. Evaluating after the kind of change that happens infrequently.
  2. Stable changes, since the property that you detect to have been true “when” the snapshot was taken will still be true once the snapshot has been gathered.
  3. The kind of property that has a correct or an incorrect answer rather than a range of increasingly appropriate answers: Routing vs Garbage Collection.
  4. Properties that need not be detected and acted upon immediately, for example garbage collection.
Distributed Debugging

- Distributed debugging was the application of our four example applications that stood out for being concerned with unstable properties.
- This is a problem for our global snap-shot technique since its main usefulness is derived from our reachability relation which in turn means little for a non-stable property.
- Distributed debugging is in a sense a combination of logical/vector clocks and global snapshots.
Distributed Debugging

Example Non-Stable Condition

- Suppose we are implementing an online poker game
- There is a process representing each player and one representing the pot in the centre of the table
- Players can “send chips” to the pot, and once winners have been decided the pot may send chips back to some of the players.
- We wish to make sure that the total amount of chips in the game never exceeds the initial amount
- It may be less than the initial amount since some chips may be in transit between a player and the centre pot.
- But it cannot be more than the initial amount.
Distributed Debugging

- Suppose that we have a history $H$ of events $e_1, \ldots, e_n$
- $H(e_1, \ldots e_n)$ is therefore the true order of events as they actually occurred in our system
- Recall then that a run is any ordering of those events in which each event occurs exactly once
- But a linearisation is a consistent run
  - A consistent run is one in which the “happens-before” relation is satisfied for all pairs of events $e_i, e_j$
  - If $e_i \rightarrow e_j$ then any linearisation (or consistent run) will order $e_i$ before $e_j$.
  - Importantly then, all linearisations only pass through consistent states
Distributed Debugging

The possibly relation

- Any linearisation \( Lin \) of our history of events \( H \) must therefore pass through only consistent states.

- A property \( P \) that is true in any state through which \( Lin \) passes, was conceivably true at some global state through which \( H \) passed.

- If this is the case for some property \( p \) and some linearisation we say \( possibly(p) \).

- Note: suppose we had taken a global snapshot during the set of events \( H \) to determine if the property \( p \) was true and determined that it was: \( Snap(p) \) evaluates to true.

- This would imply that \( p \) was possible.

- However the reverse is not true, so:
  - \( Snap(p) \) \( \iff \) \( possibly(p) \)
  - \( possibly(p) \) \( \iff \) \( Snap(p) \)
Distributed Debugging

The definitely relation

- The sister relation to the possibly relation is the definitely relation

- This states that for any linearisation Lin of H, Lin must pass through some consistent global state S for which the candidate property is true

- Since H is a linearisation of itself, then the candidate property was certainly true at some point in the history of events.

More formally:

- The statement possibly(p) means that there is a consistent global state S through which at least one linearisation of H passes such that S(p) is true.

- The statement definitely(p) means that for all linearisations L of H, there is a consistent global state S through which L passes such that S(p) is True
Possibly vs Definitely

▶ You may think that the possibly relation is useless
▶ Since I knew before we started that some predicate was potentially true at some point.
▶ However, $\neg(possibly(p)) \implies definitely(\neg p)$
▶ But, from $definitely(\neg p)$ we cannot conclude $\neg(possibly(p))$.
▶ $definitely(\neg p)$ means that there is at least one state in all linearisations of $H$ such that $p$ is not true, but not all states.
▶ $\neg(possibly(p))$ however would require that $\neg(p)$ was true in all states in all linearisations
▶ Another way to put this is that $definitely(p)$ and $definitely(\neg p)$ may be true simultaneously but $possibly(p)$ and $\neg(possibly(p))$ cannot.
Distributed Debugging

Basic Outline

- The processes must all send messages recording their local state to a master process.
- The master process collates these and extracts the consistent global states.
- From this information the *possibly*(p) and *definitely*(p) relations may be computed.
Distributed Debugging

Collecting The Global States

- Each process sends their initial state to the master process in a state message and thereafter periodically send their local state.
- The preparing and sending of these state messages may delay the normal operation of the distributed system but does not otherwise affect it: so debugging may be turned on and off.
- “Periodically” is better defined in terms of the predicate for which we are debugging.
- So we do not send a state message to the master process other than, initially and whenever our local state changes.
- The local state need only change with respect to the predicate in question. We can concurrently check for separate predicates as well by marking our state messages appropriately.
- Additionally even if the local state changes we need only send a state message if that update could have altered the value of the predicate.
Distributed Debugging

State Message Stamps

- In order that the master process can assemble the set of consistent states from the set of state messages the individual processes send it..

- Each state message is stamped with the Vector clock value at the local process sending the state message: \( \{s_i, V(s_i)\} \)

- If \( S = \{s_1, \ldots s_n\} \) is a set of state messages received by the master process, and \( V(s_i) \) be the vector time stamp of the particular local state \( s_i \)

- Then it is known that \( S \) is a consistent global state \( \text{iff} \):

  \[ V_i[i] \geq V_j[i] \quad \forall i, j, 1, \ldots N \]
Assembled Consistent Global States

- $S$ is a consistent global state iff:
  - $V_i[i] \geq V_j[i] \quad \forall i, j, 1, ... N$
  - This says that the number of $p_i$’s events known at $p_j$ when it sent $s_j$ is no more than the number of events that had occurred at $p_i$ when it sent $s_i$.
  - In other words, if the state of one process depends upon another (according to happened-before ordering), then the global state also encompasses the state upon which it depends.
Imagine the simplest case of 2 communicating processes. A plausible global state is $S(s_0^x, s_1^y)$. The subscripts, 0 and 1, refer to the process index. The superscripts $x$ and $y$ refer to the number of events which have occurred at the particular process. The “level” of a given state is $x + y$, which is number of events which have occurred globally to give rise to the particular global state $S$. 
Assembling Consistent Global States
1. A state $S' = \{s_0^{x_0'}, \ldots, s_N^{x_N'}\}$ is reachable from a state $S = \{s_0^{x_0}, \ldots, s_N^{x_N}\}$

2. If
   - $S'$ is a consistent state
   - The level of $S'$ is 1 plus the level of $S$ and:
     - $x_{i'} = x_i$ or $x_{i'} = 1 + x_i \quad \forall 0 \leq i \leq N$
Evaluating Possibly

1. Level = 0
2. States = \{(s_0^0, \ldots s_N^0)\}
3. while (States is not empty)
   ▶ Level = Level + 1
   ▶ Reachable = \{\}
   ▶ for S’ where level(S’) = Level
      ▶ if S’ is reachable from some state in States
      ▶ then if p(S’) then output possibly(p) is True and quit
      ▶ else place S’ in Reachable
   ▶ States = Reachable
4. output possibly(p) is false
Evaluating Definitely

1. Level = 0
2. States = \{(s_0^0, \ldots s_N^0)\}
3. while (States is not empty)
   - Level = Level + 1
   - Reachable = {}
   - for S’ where level(S’) = Level
     - if S’ is reachable from some state in States
       - then if \neg(p(S’)) then place S’ in Reachable
   - States = Reachable
4. if Level is the maximum level recorded
5. then output definitely(p) is false
6. else output definitely(p) is true

Note: Should also check if it is true in the initial state
Evaluating Definitely

Recall:

Level 0

1

2

3

4

5

6

7
Evaluating Definitely

Level 0

1

2

3 False True

4 \(s_0^3, s_1^1\)

5 \(s_0^3, s_1^2\)

6 \(s_0^3, s_1^3\)

7 \(s_0^4, s_1^3\)
Evaluating Definitely

\[
\begin{align*}
\text{Level 0} & \\
1 & \quad \text{False} \\
2 & \quad \text{False} \\
3 & \quad \text{False} \quad \text{True} \\
4 & \quad \text{True} \\
5 & \\
6 & \\
7 & \\
\end{align*}
\]

Definitely(p) is True
Evaluating Possibly and Definitely

- Note that the number of states that must be evaluated is potentially huge
- In the worse case, there is no communication between processes, and the property is False for all states
- We must evaluate all permutations of states in which each local history is preserved
- This system therefore works better if there is a lot of communication and few local updates (which affect the predicate under investigation)
In a synchronous system

- We have so far considered debugging within an asynchronous system.
- Our notion of a consistent global state is one which could potentially have occurred.
- In a synchronous system, we have a little more information to make that judgement.
- Suppose each process has a clock internally synchronised with the each other to a bound of $D$.
- With each state message, each process additionally time stamps the message with their local time at which the state was observed.
- For a single process with two state messages $(s^x_i, V_i, t_i)$ and $(s^{x+1}_i, V'_i, t'_i)$ we know that the local state $s^x_i$ was valid between the time interval:
  
  $$t_i - D \text{ to } t'_i + D$$
Distributed Debugging

In a synchronous system

- Recall our condition for a consistent global state:
  \[ V_i[i] \geq V_j[i] \quad \forall i, j, 1, \ldots, N \]
- We can add to that:
  \[ t_i - D \leq t_j \leq t'_i + D \] and vice versa for all \( i, j \)
- Note, this makes use of the bounds imposed in a synchronous system but speaks nothing of the time taken for a message to be delivered.
- Therefore obtaining useful bounds is rather plausible.
- But if there is a lot of communication then we may not prune the number of states which must be checked.
Summary

- Each process sends to a monitor process state update messages whenever a significant event occurs.
- From this the monitor can build up a set of consistent global states which may have occurred in the true history of events.
- This can be used to evaluate whether some predicate was possibly true at some point, or definitely true at some point.
Summary

▶ We noted that even in the real world there is no global notion of time
▶ We extended this to computer systems noting that the clocks associated with separate machines are subject to differences between them known as the *skew* and the *drift*.
▶ We nevertheless described algorithms for attempting the synchronisation between remote computers
  ▶ Cristian’s method
  ▶ The Berkely Algorithm
  ▶ Pairwise synchronisation in NTP
▶ Despite these algorithms to synchronise clocks it is still impossible to determine for two arbitrary events which occurred before the other.
▶ We therefore looked at ways in which we can impose a meaningful order on remote events and this took us to logical orderings
Time and Global State

Summary

- Lamport and Vector clocks were introduced:
  - Lamport clocks are relatively lightweight provide us with the following \( e_1 \rightarrow e_2 \implies L(e_1) < L(e_2) \)
  - Vector clocks improve on this by additionally providing the reverse implication \( V(e_1) < V(e_2) \implies e_1 \rightarrow e_2 \)
  - Meaning we can entirely determine whether \( e_1 \rightarrow e_2 \) or \( e_2 \rightarrow e_1 \) or the two events are concurrent.
  - But do so at the cost of message length and scalability

- The concept of a true history of events as opposed to runs and linearisations was introduced

- We looked at Chandy and Lamport’s algorithm for recording a global snapshot of the system

- Crucially we defined a notion of reachability such that the snapshot algorithm could be usefully deployed in ascerting whether some stable property has become true.
Summary

- Finally the use of consistent cuts and linearisations was used in Marzullo and Neiger’s algorithm.
- Used in the debugging of distributed systems it allows us to ascertain whether some transient property was possibly true at some point or definitely true at some point.
- We compare these asynchronous techniques with the obvious synchronous techniques.
- We observe that while the synchronous techniques would be more accurate often, they will occasionally be wrong.
- The asynchronous techniques are frequently conservative in that they may be imprecise but never wrong.
- For example, two events may be deemed concurrent meaning that we do not know which occurred first, but we will never erroneously ascertain that $e_1$ occurred before $e_2$. 
Any Questions?