Dependable and Distributed Systems

Professor Matthew Leeke
School of Computer Science
University of Birmingham

Topic 8 - Clock Synchronisation, Logical Clocks and Vector Clocks

Synchronised Systems and Clocks

We have considered several algorithms under the synchronous network model

Most of the algorithms we have seen rely on execution in rounds or phases

- Notion of time
- In distributed systems, different processors can have different (and even divergent) notions of time

Fault tolerant clock synchronisation attempts to remedy the problems that arise

Role of Time

Used for ordering, sequencing and synchronising of events

- Captures elements of causality
- If event A happens before event B, we have $time(A) < time(B)$

Real time is an abstract, monotonically increasing function that marks the passage of time

How do we use time in a computer system?

- Times
- Local clocks

Why Is Time Important?

- Time stamping events
  - Mutual exclusion
  - Real time control systems need accurate time-stamps on sensor values
  - Multiprocessor task scheduling
- For synchronisation we need all parties to have a common understanding of how time passes during operation
  - Especially the case in the context of communication protocols

A Big Problem With Time

Clock never remain synchronised, even if all clocks are restarted at the same time

- Crystal clocks run a different rates
- Drift apart from real time

Clocks - Drift Rate, Skew and Synchronisation

Terminologies and Concepts

Local physical clock (PC) usually provides a time source

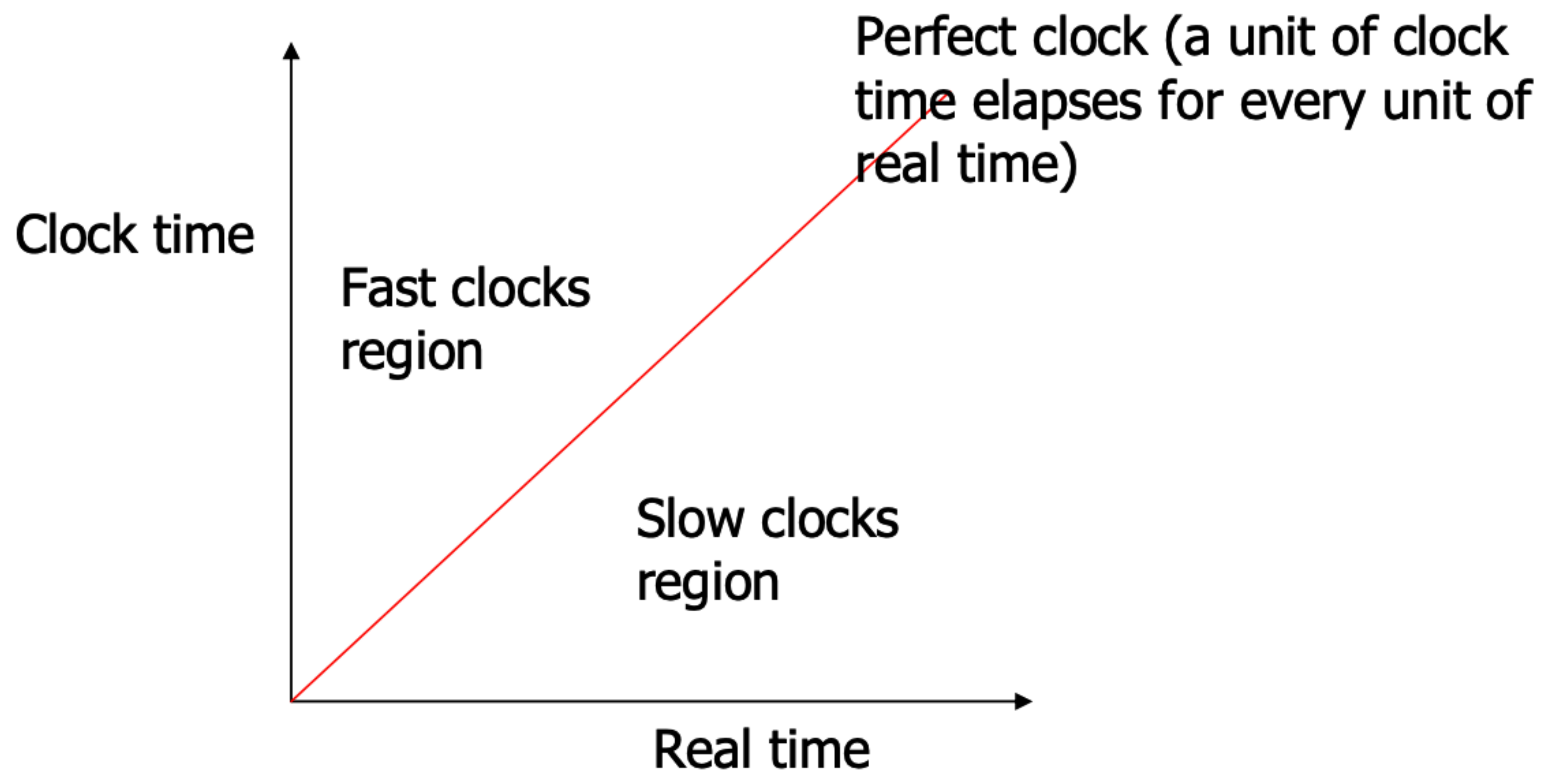
- Known as clock time
- Modelled as a monotonically increasing discrete function that maps real time (RT) into a local clock time (LT)
- $C : R \rightarrow L$
- Subject to a drift rate ρ
- Local clocks are used for local purposes, e.g., local timeouts, time-stamps, etc.

Terminologies and Concepts

We use $C(t) = T$, where t is real time and T is clock time

t is measured in Newtonian timeframe

$C_i(t)$ represents the clock time at node i


Perfect clock (a unit of clock time elapses for every unit of real time)
Clock time
Fast clocks region
Slow clocks region
Real time

Initial Clock Definitions

Clock C is non-faulty during real time interval $[s, t]$ if it is a monotonic, differentiable function over $[S, T]$ were $C(s) = S$ and $C(t) = T$ for all τ between S and T we have:

$$1 - \frac{\rho}{2} < \frac{dc(T)}{dT} < 1 + \frac{\rho}{2}$$

Again, ρ is known as the drift rate

It follows that:

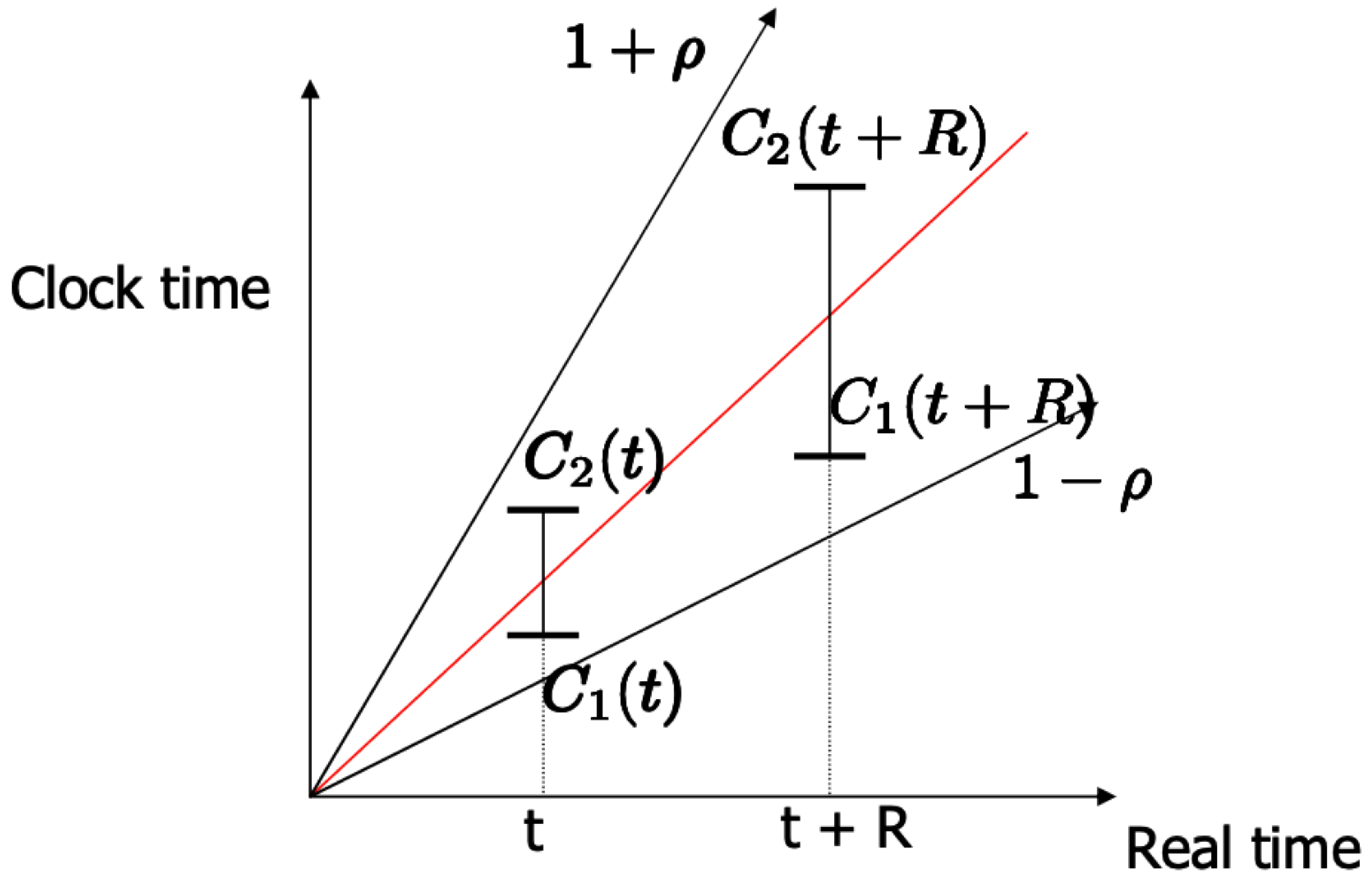
$$1 - \rho_2 < \frac{C(t_2) - C(t_1)}{t_2 - t_1} < 1 + \rho_2$$

Illustrating Drift Rate

Definition of non-faulty over a real time interval is essentially reasoning about the bounds of its discrepancy

Over a large enough interval, any drift rate will inevitably result in a faulty clock

What are our options?



Types of (Local) Clock

Hardware clocks

- Physical clock pulses that control circuitry timing within some tolerance

Logical clocks

- Values obtains from hardware clock - counters

These two notions coincide when we talk about clock synchronisation in terms of function and their properties

- The difference relates to the skew of synchronisation δ

Defining Forms Of Synchronisation

Two clocks $C1$ and $C2$ are said to be *δ-synchronised* at clock time t if and only if we have $abs((C1(t) - C2(t)) \leq \delta$

A set of clocks are *well-synchronised* if and only if any two non-faulty nodes are *δ-synchronised*

We have seen that any form of drift rate will mean that clocks will not remain *well-synchronised*

Defining Forms Of Synchronisation

What can we do to enforce *well-synchronised* clocks?

- Resynchronisation needed in a distributed system using a clock synchronisation algorithm
- Resynchronisation requires nodes to read the clocks of other nodes
- Mechanisms different according to clock implementation, particularly depending of whether clock have been implemented in hardware and / or software
- Skew can only ever be approximate due to delivery delays

Clock Synchronisation

Implements a virtual clock (VC) at each node

A virtual clock, in a way that is similar to physical clocks, can be modelled as a function that maps real time onto clock time

$$VC : R \rightarrow L$$

Clock synchronisation provides bound for (i) skew and (ii) drift rate

Clock Synchronisation

Two requirements

1. Virtual synchronisation or agreement

$$|VC_p t - VC_q t| \leq \hat{\delta} \text{ for all correct } p, q \text{ and } t \neq 0$$

2. Virtual rate or accuracy

$$0 < 1 - \hat{\rho} \leq \frac{VC_p(t+K) - VC_p(t)}{k} \leq 1 + \hat{\rho} \text{ for all } t \neq 0$$

This is where K is the interval between virtual clock ticks, $\hat{\delta}$ is how closely VCs are synchronised, and $\hat{\rho}$ is the drift rate of the VCs

Clock Synchronisation Notes

Agreement specifies that the skew between all non-faulty nodes is bounded

- Ensures that there is a sufficiently consistent view of time across all nodes in the system

Accuracy specifies that logical clocks keep up with real time

- Ensures view of time in a system is consistent with what is happening in the environment

Different implementation of synchronisation are possible

- Some approach use broadcast algorithms, whilst others are based on requests

Reliable Time Sources

Reliable Time Sources

If a reliable time source (RTS) is available, it is usually easy to satisfy synchronisation requirements

Quite simply, the RTS distributes the correct time to all nodes, subject to:

RTS Requirement 1 - Time must be distributed frequently enough to bound skew

RTS Requirement 2 - No node is required to implement too much adjustment in a singe resynchronisation action

How can we implement RTS?

Reliable Time Sources - Implementation

Two functions:

1. Generates events that causes nodes to resynchronise clocks

2. Provides a clock value to each correct node

Reliable Time Sources - Implementation

Formalising the requirements, we get:

$\tau_1, \tau_2 \ldots$ **such that** $t_1 = 0 \wedge (\forall j : r_{min} \leq \tau_{j+1} - \tau_j \leq r_{max})$

and

$t_p^1 = 0 \wedge (\forall j : 1 \leq j : 0 \leq t_p^j - \tau_i \leq \beta)$ **where** t_p^j **is real time at which node** p **detects the event produced at** τ_j

At t_p^j, node p obtains value V_j^p that can be uses to adjust VC_p to meet RTS Requirements 1 and 2

Reliable Time Sources - Implementation Notes

Requirements do not stipulate that each node receives the same value, only that nodes receive a value they can used to adjust so that they meet RTS Requirements 1 and 2

Can be implemented using single or multiple time sources

- Single source is not fault tolerant (single point of failure)
- Multiple sources use approximately synchronised clocks
    - RTS Requirement 1 met using individual node clock to trigger resynchronisation events
    - RTS Requirement 2 met using a fault tolerant average

An Algorithm With A Single RTS

Universal Coordinated Time (UTC)

WWV - A shortwave radio station in the USA that broadcasts UTC

Suitable for distributed systems where one node has WWV (it can be a server for time!)

Cristian's algorithm - Each client node periodically asks WWV sever for UTC. Server sends UTC to client. When client receives UTC, it sets its clock to UTC.

Why Doesn’t That Work?

Clocks can not go backwards or zoom forwards in time

Latencies are inevitable with requests and replies

The first problem may be addresses by introduction the correction “slowly”, hence trying not to violate RTS Requirement 1

The second problem may be addressed by endeavouring to calculate latencies

If we do these, local clocks would be set to a variant of UTC

The Berkeley Algorithm

Step 1. Time server polls each client periodically

Step 2. Server computes the average

Step 3. Server gives client an amount to “gradually” add or subtract from its clock

We will see what “gradually” means later

No WWV server is need but might we drift away from UTC? Endangering accuracy?

Implementing RTS1 and RTS2

To explore the implementation, think of resetting VC_p as starting another virtual clock that runs concurrently with the original

At real time 0, i.e., $t = 0$, node p uses VC_p^1

At real time t_p^{j+1}, node p starts VC_p^{j+1} when it detects the resynchronisation event produced at real time τ_{j+1}

$$t_p^j \leq t \leq t_p^{j+1} \implies VC_p(t) = VC_p^j(t)$$

VC_p implemented at node p using hardware clock C_p plus some adjustment value maintained by the clock synchronisation protocol

$$VC_p^j(t) \equiv C_p(t) + FIX_p^j(C_p(t))$$

What Is The FIX?

FIX is a function from clock time at node p to a correction for hardware clock C_p

Node p reads VC_p^j at real time t by reading C_p and then adding an appropriate adjustment value bases on the current value of FIX

For RTS Requirement 2 not to be violated, the value of FIX must change gradually as a function of time

FIX spreads any change in its correction to C_p over an adjustment interval (AI)

For FIX to work, the AI must be long enough to avoid violating accurate but not so long as to violate agreement

FIX To Satisfy RTS Requirements

Use a correction function that averages the values of the approximately synchronised clocks at correct nodes

$$V_p^{j+1} = CF(p, VC_1^j(t_p^{j+1}), \ldots, VC_N^j(t_p^{j+1}))$$

Correction function usually called convergence function because it brings clocks closer together / toward a common point over time

The difference between V_p^{j+1} and $C_p(t_p^{j+1})$ is the adjustment amount needed

Generic Clock Synchronisation Protocol

Generic Clock Synchronisation Protocol Anatomy

```
$j = 1$
$adj_p^0 = 0$
$adj_p^1 = 0$

$do\ forever$
    $detect\ event\ generated\ at\ \tau_{j+1}$
    $t_p^{j+1} = current\ real\ time$
    $adj_p^{j+1} = CF(p, VC_1^j(t_p^{j+1}), \ldots, VC_N^j(t_p^{j+1})) - C_p(t_p^{j+1})$
    $j = j + 1$
```

Generic Clock Synchronisation Protocol Anatomy

```
$j = 1$
$adj_p^0 = 0$
$adj_p^1 = 0$

do forever
    detect event generated at $\tau_{j+1}$ 1
    $t_p^{j+1}$ = current real time 2
    $adj_p^{j+1} = CF(p, VC_1^j(t_p^{j+1}), \ldots, VC_N^j(t_p^{j+1})) - C_p(t_p^{j+1})$ 3
    $j = j + 1$
```

1. Resynchronisation Event Detection

One way to implement “detect event” uses approximately synchronised clocks

For some predefined value R, each node p waits until VC_p^j reads jR before starting VC_p^{j+1}

Since τ_j is the earliest real time at some correct node’s VC with value jR

$$r_{min} = \frac{R}{1+\hat{\rho}}, \qquad r_{max} = \frac{R}{1-\hat{\rho}}, \qquad \beta = \frac{\hat{\delta}}{1-\hat{\rho}}$$

Another way to implement event detection is for each node to broadcast a message when its CV reaches some predefined value and to resent when such a message has been received by any correct node

2. Reading Current Real Time

Need to read N clock values simultaneously, which is not possible in a distributed system

A possible approach:

Each node locally implements approximations of VC at other nodes

Node p maintains a collection of table $\tau_p^j[1...N]$ that can be used to compute an approximation for $VC_q^j(t)$ and p approximates $VC_q^j(t)$ at real time t by computing $C_p(t) + \tau_p^j(t)$

Node p can approximate $VC_1(t_p^{j+1}), \ldots, VC_N(t_p^{j+1})$ by reading $C_p(t_p^{j+1})$ and using it with τ_p^j to compute N values

2. Reading Current Real Time - Constructing τ_p^j

Node p periodically communicates with other nodes

Denote min and max delay in sending, receiving and processing messages by Δmin and Δmax respectively

Node p can compute $\tau_p^j[q]$ by executing:

```
1. Send j-th clock time to node q
2. Receive $C$ from q timeout after $2\Delta max$
3. if (timeout) then $C = \infty$
4. t-now = current real time
5. $\tau_p^j[q] = C - (C_p(t\text{-}now) - \Delta_{min})$
```

3. Convergence Functions

It is a function of $N + 1$ arguments that satisfies three properties

1. Monotonicity

$$\forall j : 1 \leq j \leq N : x_j \leq y_j \text{ then } CF(p, x_1, \ldots, x_N) \leq CF(p, y_1, \ldots, y_N)$$

2. Translation Invariance

$$CF(p, x_1 + \sigma, \ldots, x_N + \sigma) = CF(p, x_1, \ldots, x_N) + \sigma$$

3. Accuracy Preservation

3. Convergence Functions

3. Accuracy Preservation

Let X_{OK} be a subset of $x_1, \ldots, x_N$ whose members are within δ of $N - k - 1$ of $x_1, \ldots, x_N$ then:

$$\forall p : x_p \in X_{OK} : |x_p - CF(p, x_1, \ldots, x_N)| \leq \alpha(\delta)$$

(Also $\forall j : 0 < j : |adj_p^{j+1} - adj_p^j| \leq \alpha(\delta)$)

Where α is the accuracy of the convergence function

3. Convergence Functions

Egocentric average - The average of all arguments that are no more than δ from x_p

Fast convergence algorithm - The average of all arguments that are within δ of at least $N - k$ other arguments

Fault-tolerant midpoint - The midpoint of the range spanned by the arguments after the k highest and k lowest have been discarded

Fault-tolerant average - The average of arguments after the k highest and k lowest values have been discarded

Logical Clocks

The Elephant In The Room

We have studied algorithms for synchronous distributed systems

Problem: Synchronous systems have poor coverage

Difficult to implement, e.g., we need to ensure that maximum assumed message delay is never exceeded

Asynchronous systems provide no assumptions on speeds / delays

Timestamping

Logical clocks

How do we timestamp events?

Vector clocks

How do we timestamp events?

Synchronous Distributed Systems

- Allow us to have synchronised clocks
  - Can determine the order of events
  - Important in applications such as mutual exclusion, broadcast, etc.
  - Helps in networking protocols that require timeouts
- Physical clocks
  - Quartz crystal clocks drift 1 microsecond / second
  - Measure global time but can be expensive to implement

Ordering In Asynchronous Systems

The idea:

- We don’t want to use physical clocks anymore
- Can we capture the ordering of events without having recourse to physical clock
- Leslie Lamport introduced the concept of logical clocks in 1978

For many applications, ordering is sufficient

- Order events
- Events can be message sends or receives, instructions, procedure calls, etc.

Events and Event Ordering

Within a single process or between processes on the same machine

- Order can be determined using physical clocks
- Timestamp each event and then compare to determine order

In a distributed system we can determine order iff all clocks are perfectly synchronised, i.e., all physical clocks read exactly the same time

- In general this can not be achieved
- What can we do instead?

Happened Before Relation

Define the happened before relation to overcome the problem

- Event A happened before event B
- Denoted by $A \to B$

It describes a causal order of events

- Events A causally affects event B

How do we define such a relation? How do we extend the relation to work for distributed systems?

Happened Before Relation

If A and B are events on the same process, and A occurred before B then $A \to B$

If A is the event of sending a message m in process p_k and B is the event of receiving m in process p_l then $A \to B$

The relation is transitive

If $A \to B$ and $B \to C$ then $A \to C$

Happened Before Relation

Two distinct events A and B are said to be concurrent if it is neither the case that $A \to B$ nor $B \to A$

- We denote A and B being concurrent by $A || B$
- This means that events A and B can not affect each other, since they occur at the "same time"

Only three possibilities exist for events A and B

- Events A and B must be $A \to B$, $B \to A$ or $A || B$

Happened Before Relation

- Good to heave a theoretical model for ordering
    - How do we actually implement such a thing?
- Use logical clocks
    - Introduced by Leslie Lamport
    - Idea is to simulate a clock using a counter

Happened Before Relation - Implementation

Each process K has access to a logical clock C_k

C_k can assign a value $C_k(A)$ to any event A occurring on process K

- $C_k(A)$ is called the timestamp of event A on process K
- $C(A)$ is the timestamp of A on whichever process is occurred
- Timestamp has no relation whatsoever with the physical time as we know it

How do we assign values to events? Randomly? What properties are required of logical clocks?

Happened Before Relation - Properties

Clock condition

If $A \rightarrow B$ then $C(A) < C(B)$, noting that we're not specifying a process

If event A happened before event B then we want the timestamp of A to be less than the timestamp of B

Simulate the notion of time

So does $C(A) < C(B)$ implies $A \rightarrow B$? Really?

Happened Before Relation - Properties

Correctness Conditions

Must be satisfied by logical clocks to satisfy clock condition

For any events A and B on K, $A \rightarrow B$, then $C_K(A) < C_K(B)$

If event A is send on K and B is receive on L then $C_K(A) < C_L(B)$

Implementation Correctness 1 (IC1) - Clock C_K must be incremented between any two successive event on process P_K: $C_K = C_K + d$ (usually $d = 1$)

Happened Before Relation - Properties

Implementation Correctness 2 (IC2) - If event A is send message M on P_K with timestamp $C_K(A)$ and event B is receive message M on P_L, we can calculate $C_L(B) = max(C_L, C_K + d)$

Clock initialised at zero

Most increments will be due to IC1 - less network-level events for most distributed algorithms?

Logical Clocks - Notes

Happened before relation defines an irreflexive partial order

Why? Can we extend this to be a total order?

We extend it by creating a relation that totally orders the processes

If A is an event on P_K and B is an event on P_L then $A \implies B$ (where $\implies$ is the total order) if and only if we have one of:

$C_K(A) < C_L(B)$

$C_K(A) < C_K(B)$ and $P_K << P_L$, where $<<$ denotes a relation that totally orders the processes

Logical Clocks - Limitations

The problem is that we can not determine whether two events are causally related from logical clock timestamps

$C(A) < C(B)$ does not imply $A \to B$

In a synchronous system with perfect synchronises clock, this works fine

How do we address the problem?

Use vector clocks

Vector Clocks

Vector Clocks

Assume a system contains N processes, where each process P_K has a clock C_K which is an integer vector of length N, i.e.,$C_K = (C_K[1], \ldots, C_K[N])$

$C_K(A)$ is the timestamp of event A at P_K

$C_K[k](A)$ is the entry of P_K in C_K (P_K logical time)

$C_K[j](A)$ is the entry of P_J in C_K ($K \neq J$)

Represents P_K's best guess of logical time at P_J

The time of the occurrence of the last event in P_J which happened before the current event on P_K (based on message received)

Vector Clocks - Implementation

Implementation that guarantees that logical clocks will satisfy correctness conditions

Implementation Correctness 1 (IC1) - Clock C_K must be incremented between many two successive events on process P_K : $C_K[K] = C_K[K] + d$

Implementation Correctness 2 (IC2) - If event A is the sending of message M on P_K with vector timestamp $C_K(A)$ and event B is receive message M on P_L with vector timestamp $C_L(B)$ we can calculate $C_L(B) = max(C_L[p], C_K[p] + d)$ for all p

Vector Clock - How Can We Compare?

Strong clock condition

$$\forall A, B : C_K(A) < C_L(B) \leftrightarrow A \rightarrow B$$

We have improved on logical clocks!

We define > as follow: $C_K(A) < C_L(B)$ if and only if $C_K(A) \neq C_L(B)$ and $\forall x, 1 \leq x \leq N : C_K(A)[x] \leq C_L(B)[x]$