CT420 REAL-TIME SYSTEMS

TIME SYNCHRONISATION IN DISTRIBUTED SYSTEMS

Dr. Michael Schukat

Ollscoil na Gaillimhe
University of Galway

Time in Distributed Systems

- A distributed system (DS) is a type of networked system where multiple computers (nodes) work together to perform a task
  - Such systems may or may not be connected to the Internet
- Time and time synchronisation are an important issues here
  - Think of error logs in distributed systems; how can error events recorded in different computers be correlated with each other, if there is no common time-base
- Problem:
  - GNSS-based time synchronisation may or may not be available, as GPS signals are absorbed or weakened by building structures
  - There is no other time reference such systems can rely on, as in such a distributed system there are just a series of imperfect computer clocks

Example: Airline Reservation System

- Assume an airline reservation system consisting of three servers A, B and C and some client computer that makes a booking
- Each server has its own local clock
- Server A receives a client request to purchase last ticket on flight ABC123
- Server A timestamps the purchase using its local clock reading (9h:15m:32.45s) and logs it. It replies "ok" to client
- That was the last seat. Server A sends message to Server B stating "flight full."
- B enters "Flight ABC123 full" + local its clock reading (9h:10m:10.11s) into its log
- At a later stage server C queries A's and B's logs. It reads that a client purchased a ticket after the flight became full

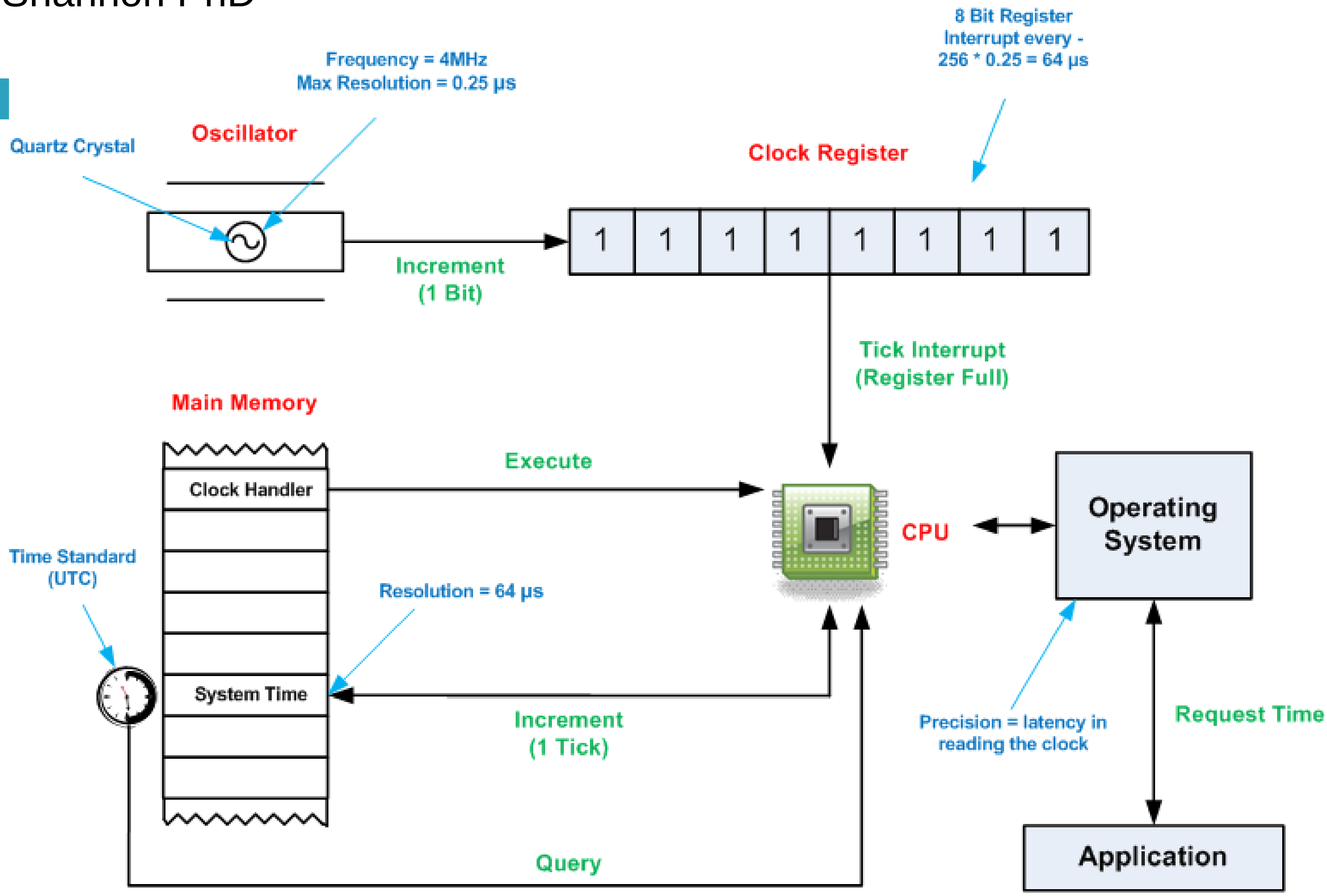
Recap: The Clock Synchronisation Problem

- In distributed systems, all the different nodes are supposed to have the same notion of time, but quartz oscillators oscillate at slightly different frequencies
- Hence, clocks tick at different rates (→clock **skew**), resulting in an increasing gap in perceived time
- The difference between two clocks at a given point in time is called clock **offset**
- Clock synchronization aims to minimise clock skew (and subsequently) offset between two or more clocks

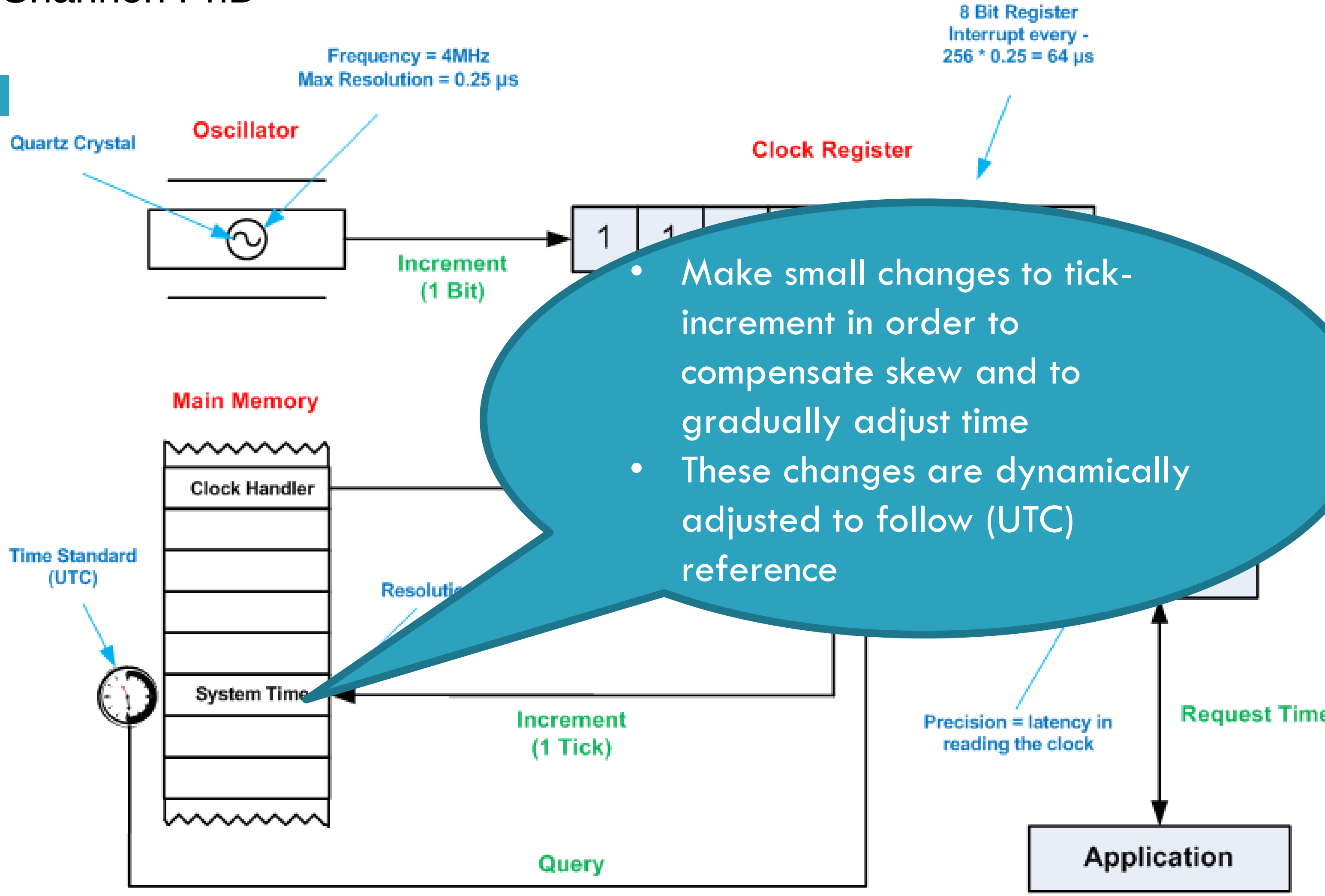
Dealing with Drifting Clocks

- A clock can show a positive of negative offset with regard to a reference clock (e.g. UTC)
  - Need to resynchronise clock periodically
- One can't just set clock to 'correct' time
  - Jumps (particularly backward!) can confuse software / operating systems
- Instead aim for gradual compensation by correcting the skew
  - If clock runs too fast, make it run slower until correct
  - If clock runs too slow, make it run faster until correct

J.Shannon PhD



J.Shannon PhD



- Make small changes to tick-increment in order to compensate skew and to gradually adjust time
- These changes are dynamically adjusted to follow (UTC) reference

Pseudo Code Clock Handler with Skew Compensation


```
// Global variable to store time
struct timespec Master_clock;
int Skew_comp;
…
#define CLOCK_TICK_INCREMENT            64000
#define ONE_SECOND_IN_NANO_SEC          1000000000
…
void init_Master_Clock() {
              Master_clock.tv_sec = 0;
              Master_clock.tv_nsec = 0;
              Skew_comp = 0;
}
…
void change_skew_comp(int delta) { // delta can be positive of negative
              Skew_comp += delta;
}

__interrupt void clock_handler() {
             Master_clock.tv_nsec += CLOCK_TICK_INCREMENT + Skew_comp;
             while (Master_clock.tv_nsec  > ONE_SECOND_IN_NANO_SEC) {
                          Master_clock.tv_nsec  -= ONE_SECOND_IN_NANO_SEC;
                          Master_clock.tv_sec++;
             }
}
```

Time Synchronisation of DS – Some Examples

- Time synchronisation is crucial for many distributed systems
- Synchronisation needs of endpoints are application-specific
  - From nanoseconds to seconds
- As technology evolves, error margins tend to get smaller, and are easier to meet
  - E.g. Gigabit Ethernet
- This is turn makes systems far more vulnerable if synchronisation is interfered with

Example High Frequency Trading

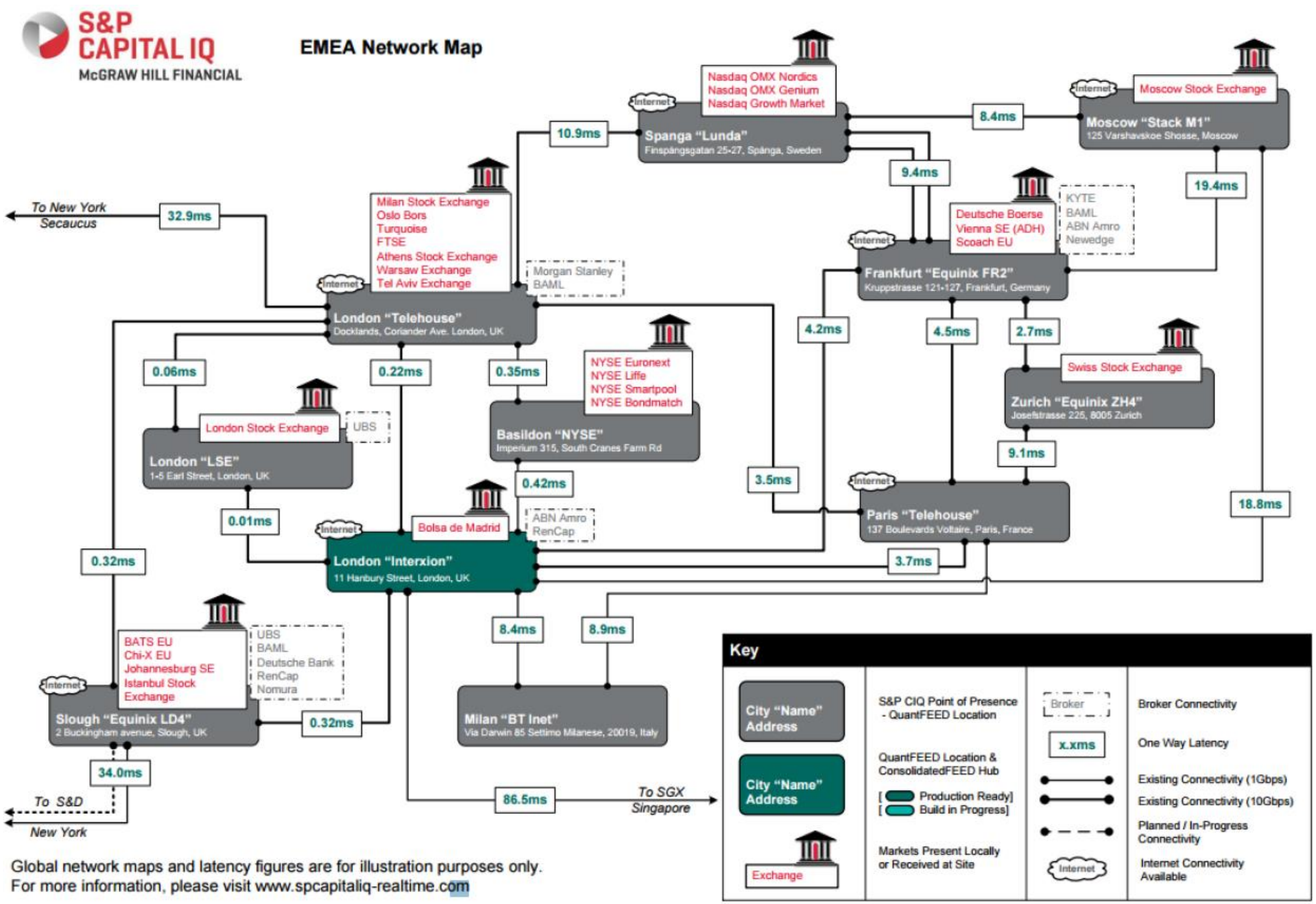
- High frequency trading (HFT) is an automated trading platform used by large investment banks
- It requires fast computers that run complex trading algorithms and fast network technology to trade large numbers of orders at extremely high speeds
  - https://www.youtube.com/watch?v=z4nCTdQIH8w
- Due to its speed it provides split second arbitrage opportunities for institutions to execute trades before the open market can
- Accurate time synchronisation ensures that orders are executed precisely at the intended time, avoiding discrepancies or delays that could impact trade outcomes

MiFID 2

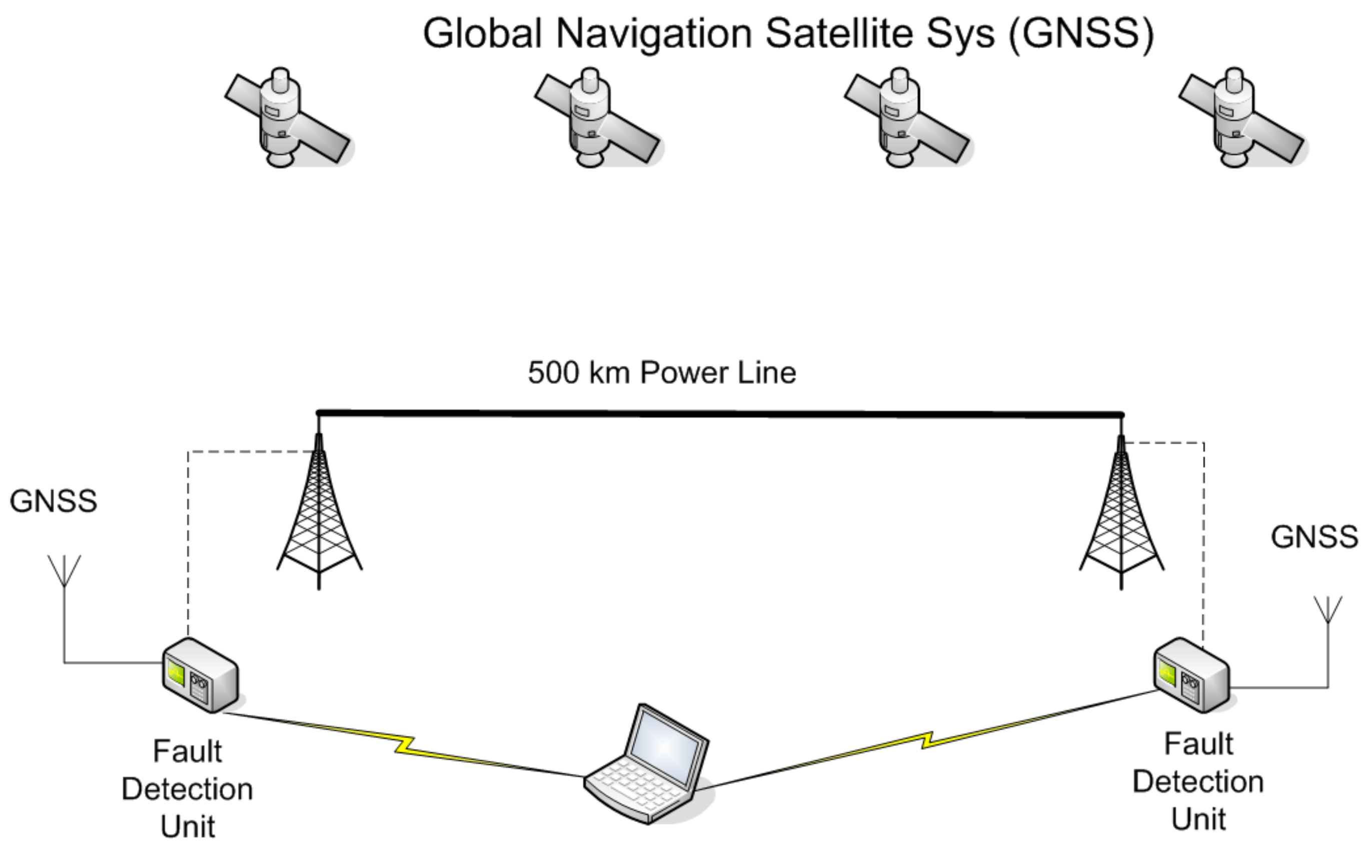
- Directive 2014/65/EU, commonly known as MiFID 2 (Markets in financial instruments directive 2), is a legal act of the EU
- It provides a legal framework for securities markets, investment intermediaries, and trading venues
- In particular, MiFID 2 introduced the requirement for trading venues, their members and participants **to synchronise the business clocks** used to record the date and time of reportable events to UTC

Gateway-to-gateway latency of trading system	*Maximum divergence from UTC*	*Granularity of time-stamp*
> 1 millisecond	1 millisecond	1 millisecond or better
=< 1 millisecond	100 microseconds	1 microsecond

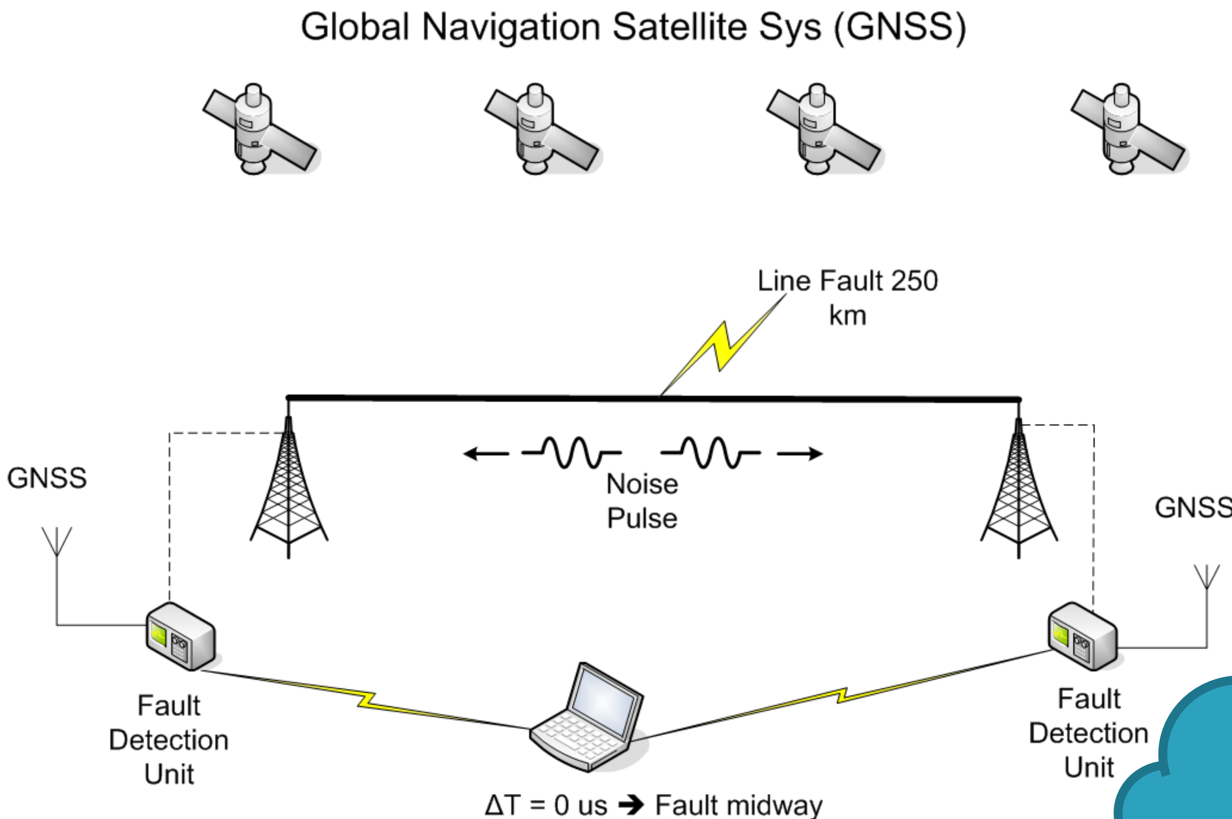
European Trading Platforms and Gateway Latencies (2015 Data)



Example: Energy Systems - Power Line Fault Detection



Example: Energy Systems - Power Line Fault Detection



Timestamp when noise pulse arrives

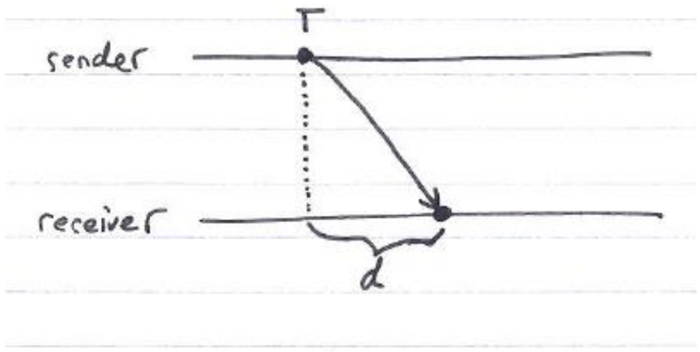
Speed of light c = 3 x 10^8 m/s ➔ 0.3 m / nanosecond
→ If the synchronisation error between the two fault detection units is one microsecond, the line fault can be narrowed down to a 300 m stretch of cable

Synchronising Distributed Systems

- Synchronisation can take place in different forms
  - Based on **physical** ("real") clocks - we look at them first
    - Absolute to each other by synchronising to accurate time source (e.g. UTC)
    - Absolute to each other by synchronising to locally agreed time (i.e. no link to global time reference)
    - Here the term *absolute* means that differences in timestamps are proper time intervals
  - Based on **logical** clocks (i.e. clocks are more like counters)
    - Timestamps may be ordered but with no notion of measurable time intervals
- In either way, the DS endpoints synchronise using a shared network
  - For physical clock synchronisation network latencies must be considered, as packets traverse from a sending node to a receiving node

Perfect Networks

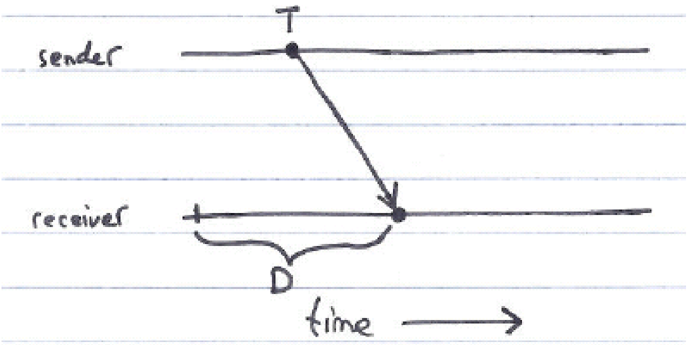
- Messages always arrive, with propagation delay exactly d



- Sender sends time T in a message
- Receiver sets clock to T + d
- Synchronisation is exact

Deterministic Networks

- Messages arrive with propagation delay d, with 0 < d <= D



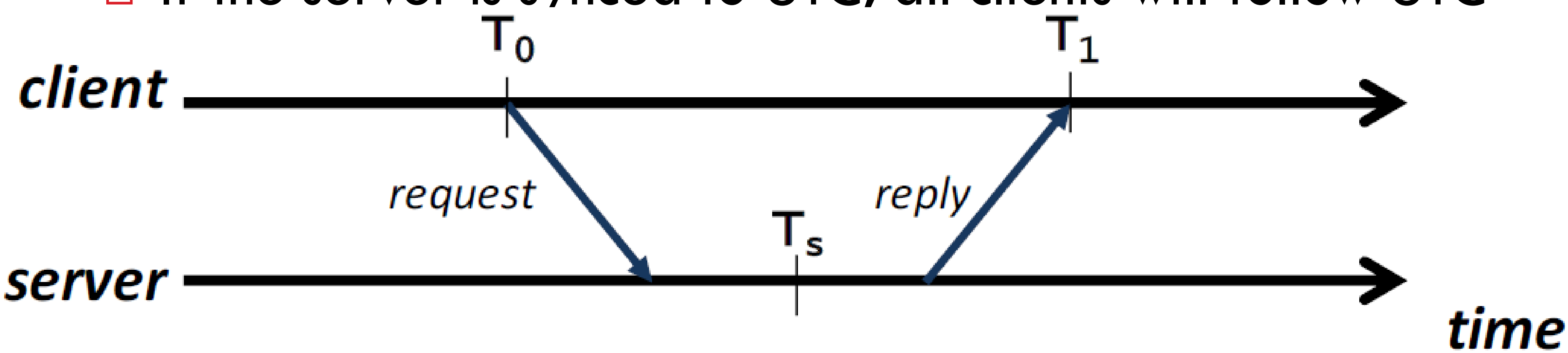
- Sender sends time T in a message
- Receiver sets clock to T + D /2
- Synchronisation error is at most D / 2
- **Deterministic communication** is the ability of a network to guarantee that a message will be transmitted in a specified, predictable period of time

Synchronisation in the Real World

- Most off-the-shelf networks are asynchronous
  - I.e., data is transmitted intermittently on a best effort basis
- They are designed for flexibility, not determinism
  - CSMA/CD contention mechanism isn't helpful either
- As a result, propagation delays are arbitrary and sometime even unsymmetric (i.e. upstream and downstream latencies are different)
- Therefore, synchronisation algorithms are needed to accommodate these limitations

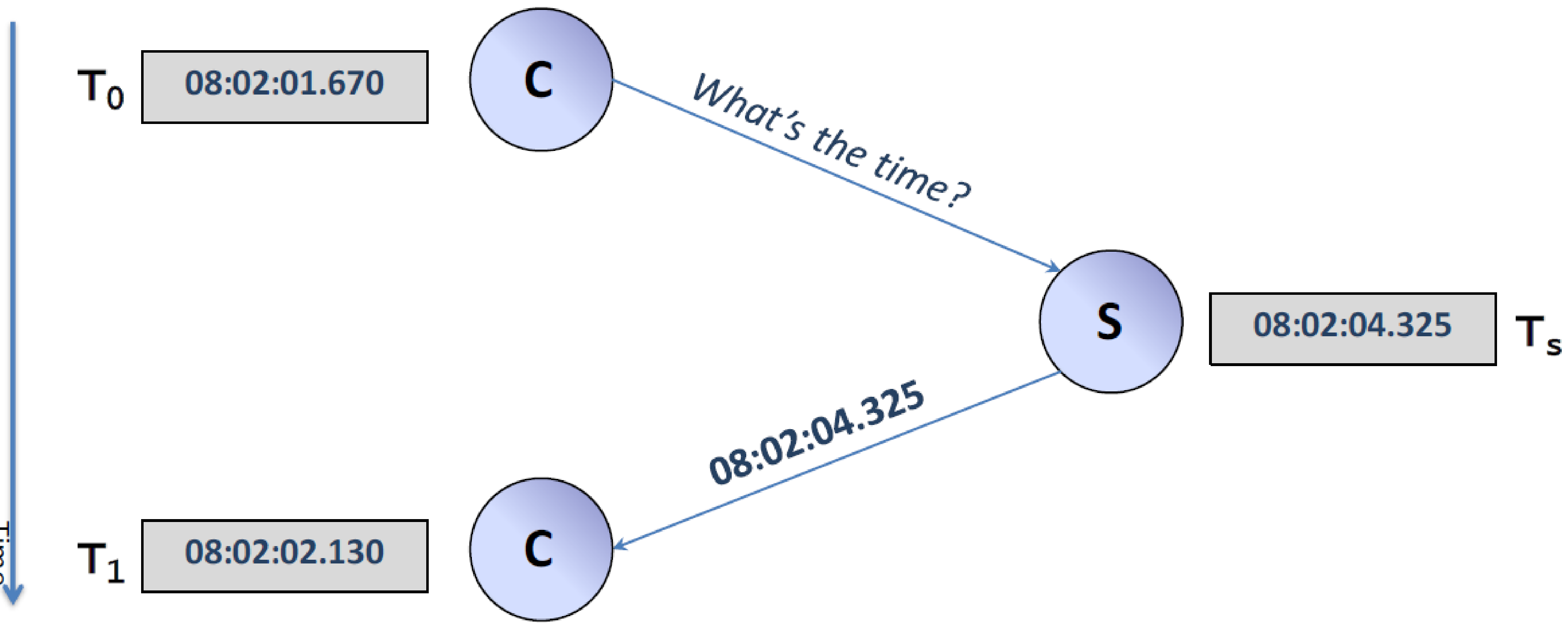
Cristian's Algorithm

- Attempt to compensate for symmetric network delays
  - Client remembers local time T_0 just before sending request
  - Server receives request, determines T_s and puts it into reply
  - When client receives reply, it notes local arrival time T_1
  - The correct time is then approximately $(T_s + (T_1 - T_0) / 2)$
- Algorithm assumes symmetric network latency
- If the server is synced to UTC, all clients will follow UTC



Cristian's Algorithm: Example

- Round Trip Time (RTT) $T_1 - T_0$ = 460ms → one-way delay is ~ 230 ms
- Estimate correct time: 08:02:04.325 + 230 ms = 08:02:04.555
- Client C gradually adjusts local clock to gain 2.425 seconds (as seen before) – i.e. C's lock will be adjusted to tick slower or faster



Limitations of Cristian's Algorithm

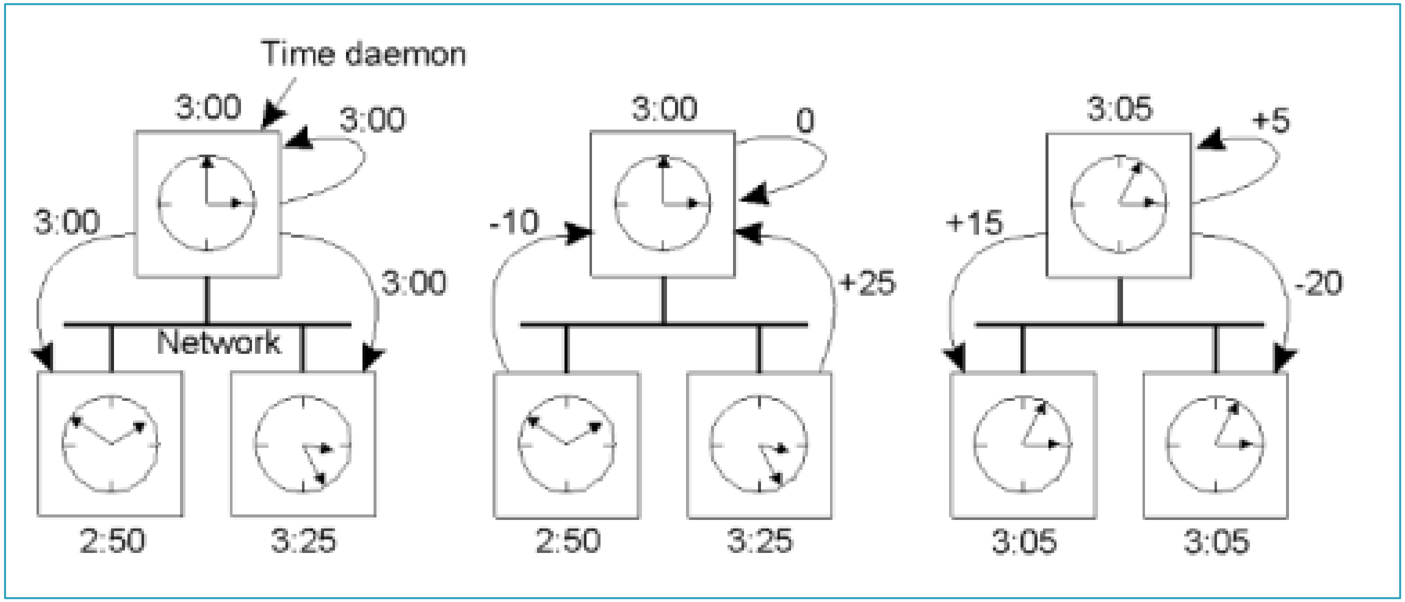
- The algorithm assumes
  - a symmetric network latency
  - timestamps can be taken as the packet hits the wire / arrives at the client
  - T_S is right in the middle of server process
    - E.g., consider the server process being pre-empted just before it sends the response back to the client; this will corrupt the synchronisation of the client

Berkeley Algorithm

- In this algorithm there is no accurate time server, instead a set of client clocks is synchronised to their average time
  - Assumption is that offsets / skews of all clocks follow some symmetric distribution (e.g. a normal distribution) with some clocks going faster and others slower, i.e. with a mean value close to 0
- One node is designated the master (or leader) M
- It periodically queries all other clients for their local time
- Each client returns a timestamp or their clock offset to the master
- Christian's algorithm is used to determine and compensate for RTTs, which can be different for each client (not shown in the following examples)
- Using these, the master computes average time (thereby ignoring outliers), calculates the difference to all timestamps it has received, and sends an adjustment to each client
  - Again, each computer gradually adjusts its local clock

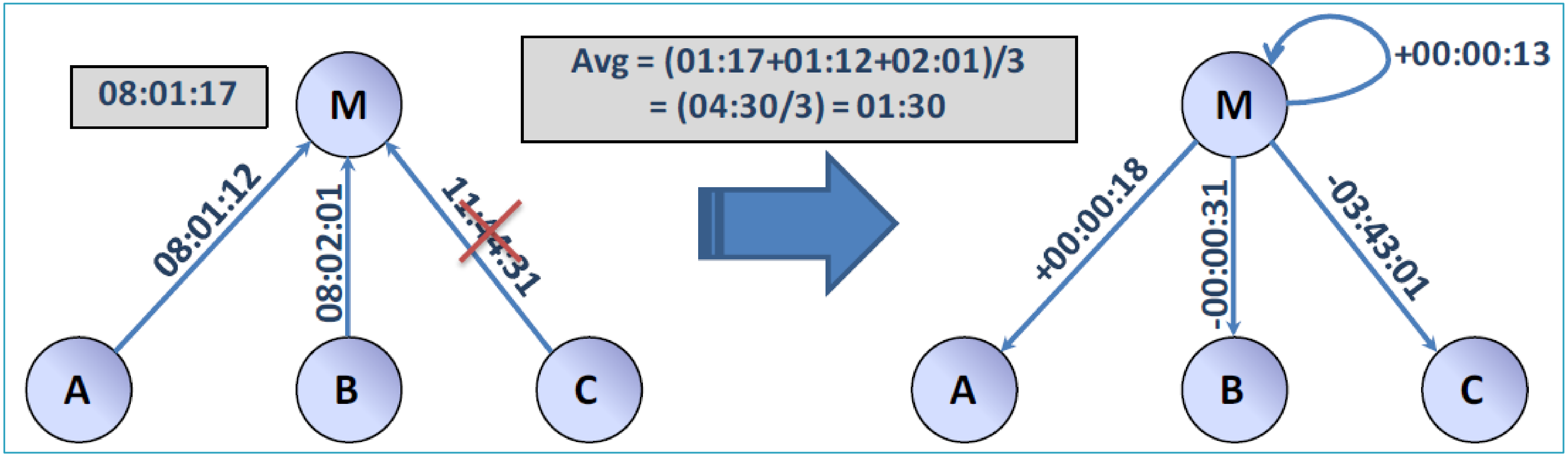
Berkeley Algorithm Example Var 1

- Master ("Time daemon") sends timestamp to all clients (left image)
- Each client return their relative offset to master (centre image)
- Master calculates average offset (i.e., (-10 + 0 + 25) / 3 = 5 minutes), determines the local time estimate (3:00 + 5), calculates the relative offset for each client clock, and sends adjustments to clients (right image)



Berkeley Algorithm Example Var 2

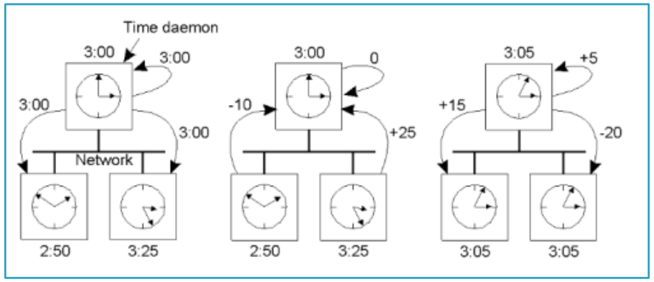
- Master requests timestamps from A, B and C, which they duly return (left image)
- Master discards outliers (C's timestamp), calculates the average time (Avg) as well as the clients' relative offsets, which are send to the clients (right image)



In-Class Activity: Menti

- Consider the following timestamps by computers M, A, B, C, D:
  - M: 8:00:13
  - A: 7:59:59
  - B: 8:00:01
  - C: 7:59:55
  - D: 8:00:05
- Which of those values is an outlier?
- Calculate the average time

Berkeley Algorithm



- Client clocks are adjusted to run faster or slower, to be synched to overall agreed system time
- The client network is an intranet, i.e., an isolated system
- This makes the Berkeley algorithm an **internal clock synchronisation algorithm**
- The Berkeley algorithm was implemented in the TEMPO time synchronisation protocol, which was part of the Berkeley UNIX 4.3BSD system (a remote uncle of today's Linux)

Logical Clocks

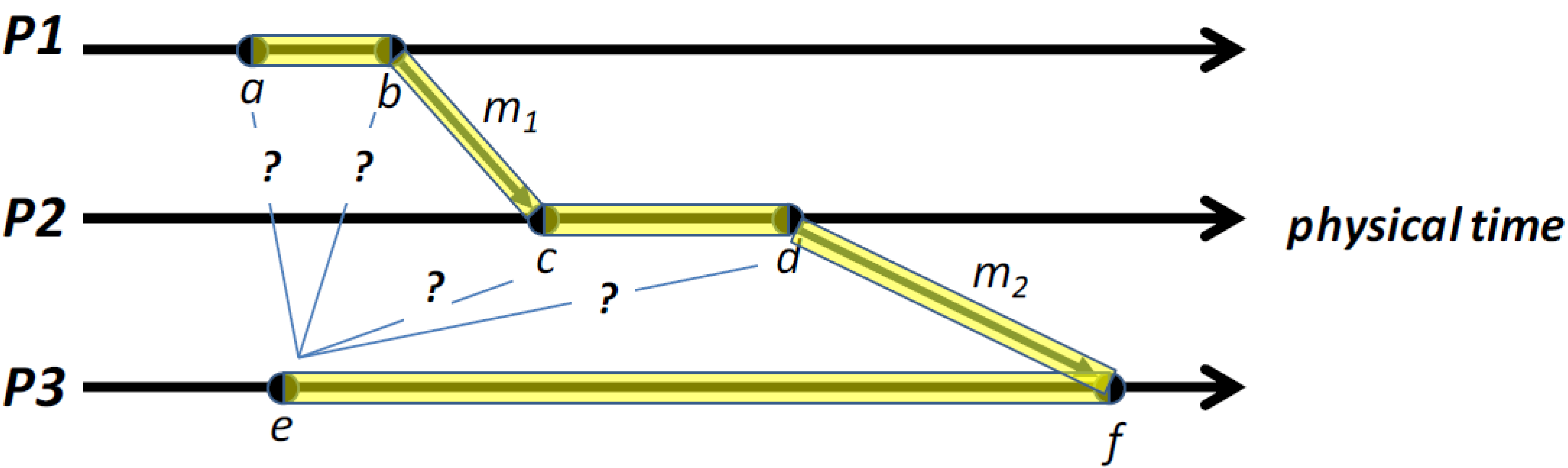
- Logical clocks is another concept linked to internal clock synchronisation
- Logical clocks only care about their internal consistency, but not about absolute (UTC) time
- Subsequently they do not need clock synchronisation and take into account the order in which events occur rather than the time at which they occurred
- In practice, if clients / processes only care about "event **a** happens before event **b**", but don't care about the time difference exactly, they can use logical clock

The "Happens-Before" Relation

- Some applications don't need to know exactly when event a occurred
  - Just need to know if **a** occurred before or after **b**
- Define the happens-before relation, **a** → **b**
  - If events **a** and **b** are within the same process, then **a** → **b**, if a occurs with an earlier local timestamp (process order)
  - If **a** is the event of a message being sent by one process, and **b** is the event of the message being received by another process , then **a** → **b** (causal order)
  - We have **transitivity**, i.e. if **a** → **b** and **b** → **c**, then **a** → **c**
- Note that this only provides a *partial order*:
  - If two events, **a** and **b**, happen in different processes that do not exchange messages (not even indirectly), then **a** → **b** is not true, but neither is **b** → **a**
  - We say that a and b are **concurrent** and write **a** ~ **b**
    - I.e. nothing can be said about when the events happened or which event happened first

Example

- Three processes P1, P2 and P3 (each with 6 events enumerated a … f), and 2 messages m_1 and m_2
  - Due to process order, we know a → b, c → d and e → f
  - Causal order tells us b→ c and d→ f
  - And by transitivity a → c, a → d, a → f, b → d, b → f, c → f
- However, event e is **concurrent** to a, b, c and d

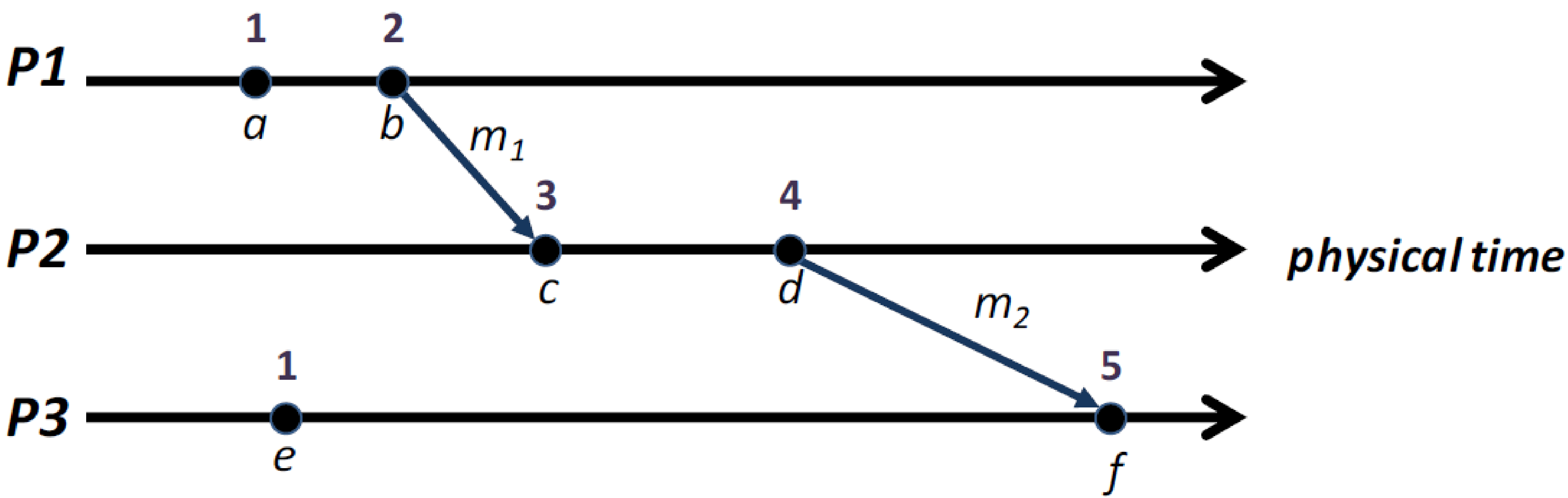


Implementing Happens-Before using the Lamport Scheme

- Each process P_i has a logical clock L_i
  - L_i can simply be an integer variable, initialised to 0
- L_i is incremented on every local event e
  - We write L_i (e) or L(e) as the timestamp of e
- When P_i sends a message, it increments L_i and copies its content into the packet
- When P_i receives a message from P_k, it extracts L_k and sets $L_i := \max(L_i, L_k)$, and then increments L_i
- This guarantees that if $a \rightarrow b$, then $L_i(a) < L_k(b)$
  - But nothing else!

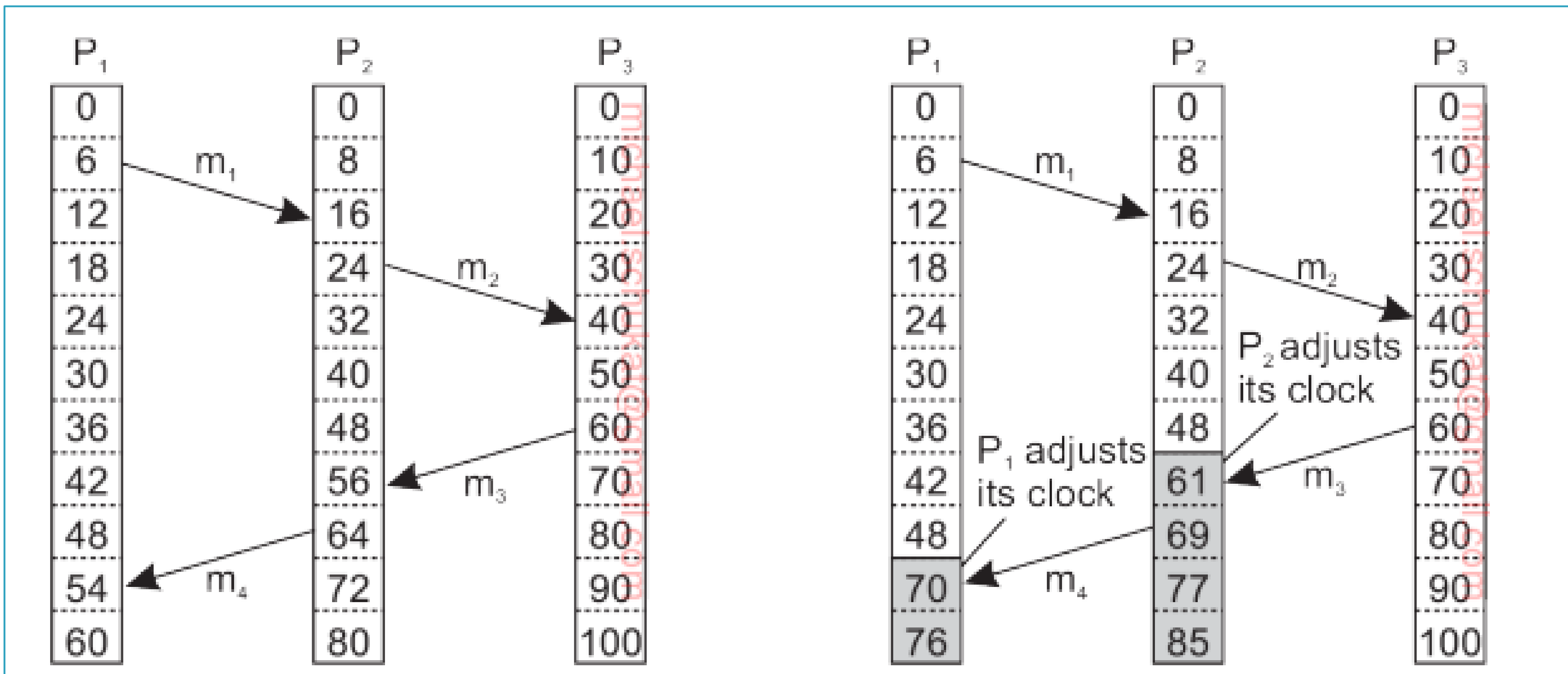
Lamport Clocks Example

- When P2 receives m_1, it extracts timestamp 2 and sets its clock to max(0, 2) before incrementing it, i.e. $L_2 = 3$
- It is possible for events to have the same timestamp
  - e.g. event e has the same timestamp as event a
  - If desired, unique timestamps can be created for example by adding a process identifier (PID), but there's no real benefit



Lamport Clocks Example

- 3 processes with their logical clocks before (left) and after applying Lamport's algorithm (right)



Identify incorrect timestamps by their X-Y position in the grid (e.g. “TA” for the top left timestamp)

0	0	0	
2	1	1	**A**
3	2	3	**B**
7	5	4	**C**
6	6	6	**D**
9	5	6	**E**
11	8	10	**F**
12	9	11	**G**
13	13	14	**H**
14	15	16	**I**
T	**U**	**V**	

Incorrect Timestamps

0	0	0	
2	1	1	**A**
3	2	3	**B**
7	5	4	**C**
6	6	5	**D**
9	5	6	**E**
11	8	10	**F**
12	9	11	**G**
13	13	14	**H**
14	15	16	**I**
T	**U**	**V**	

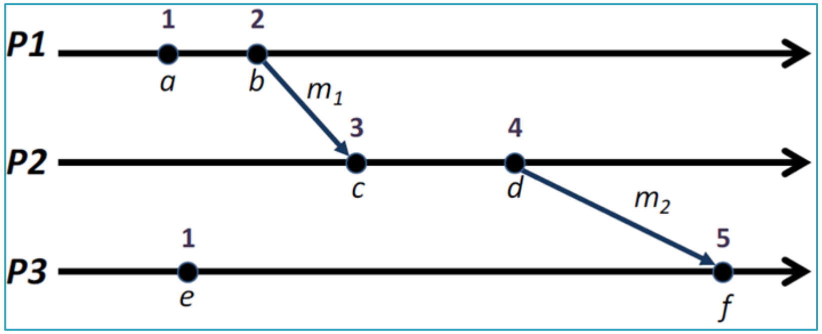
Limitations of Lamport's Logical Clocks

- Lamport's logical clocks lead to a situation where all events in a distributed system are ordered, so that if event **a** (linked to P_i) "happened before" event **b** (linked to P_k), i.e. **a** $\rightarrow$ **b**, then **a** will also be positioned in that ordering before **b**, i.e. L_i(**a**) < L_k(**b**) or simply L(**a**) < L(**b**)
- However, nothing can be said about the relationship between two events **a** and **b** by merely comparing their time values L_i(**a**) and L_k(**b**), iff i <> k, i.e. we can't tell if **a** $\rightarrow$ **b** / **b** $\rightarrow$ **a**, or **a** $\sim$ **b**

Limitations of Lamport's Logical Clocks: Example

- Each process keeps a list of time-stamped events following Lamport
- Examining these lists allows us (obviously) to determine that
  - L(a) < L(c)
  - L(e) < L(c)
- However (and we **only** know this from examining the diagram):
  - a → c, but
  - e ~ c

	a	b	c	d	e	f
P_1	1	2				
P_2			3	4		
P_3					1	5



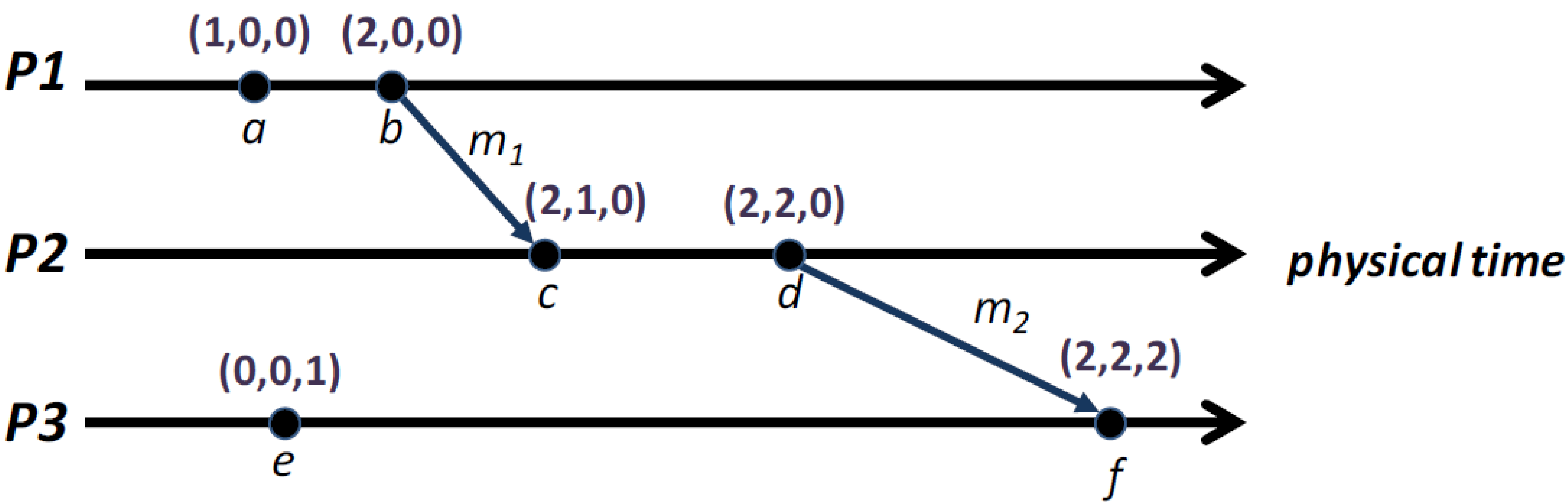
- I.e., comparing the timestamps of some events a and b alone does not allow us to determine if a → b, b → a, or a ~ b, unless they are happening on the same process
- The problem is that Lamport clocks do not capture **causality**

Vector Clocks

- In practice, causality is captured by means of **vector clocks**
- Vector clocks work as follows:
  - There is an ordered list of logical clocks, with one per process
  - Each process P_i maintains vector V_i [], initially all zeroes at start
  - On a local event e, P_i increments V_i [i] (i^{th} vector component)
    - If the event is "message send", new V_i[] is copied into packet
  - If P_i receives a message from P_m then, for all k = 0, 1, …, it sets V_i [k] := max(V_m[k], V_i[k]), and increments V_i[i]
- Intuitively V_i[k] captures the number of events at process P_k that have been observed by P_i

Vector Clocks Example

- When P_2 receives m_1, it merges the entries from P_1's clock
  - choose the maximum value in each position
- Similarly when P_3 receives m_2, it merges in P_2's clock
  - this incorporates the changes from P_1 that P_2 already saw
- Vector clocks explicitly track the transitive causal order: f's timestamp captures the history of a, b, c & d



Using Vector Clocks for Ordering

- Can compare vector clocks piecewise:
  - $V_i = V_j$ iff $V_i[k] = V_j[k]$ for k = 0, 1, 2, …
  - $V_i \leq V_j$ iff $V_i[k] \leq V_j[k]$ for k = 0, 1, 2, …
  - $V_i < V_j$ iff $V_i \leq V_j$ and $V_i \neq V_j$
  - $V_i \sim V_j$ otherwise

e.g. [2,0,0] versus [0,0,1]

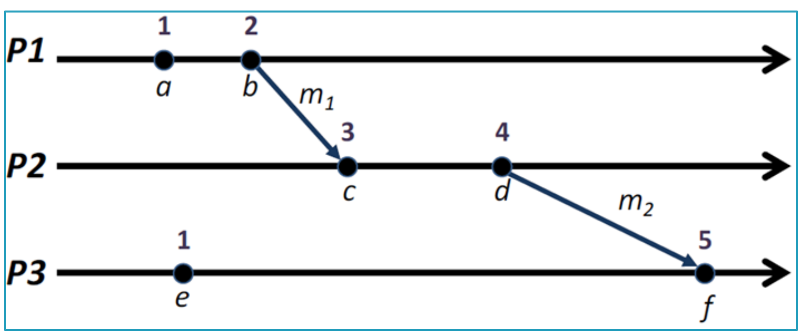
- For any two event timestamps T(a) and T(b)
  - if $a \rightarrow b$ then $T(a) < T(b)$; **and**
  - if $T(a) < T(b)$ then $a \rightarrow b$
- Hence can use timestamps to determine if there is a causal ordering between any two events
  - i.e. determine whether $a \rightarrow b$, $b \rightarrow a$ or $a \sim b$

Lamport Clocks versus Vector Clocks

Lamport Clocks

	a	b	c	d	e	f
P_1	1	2				
P_2			3	4		
P_3					1	5

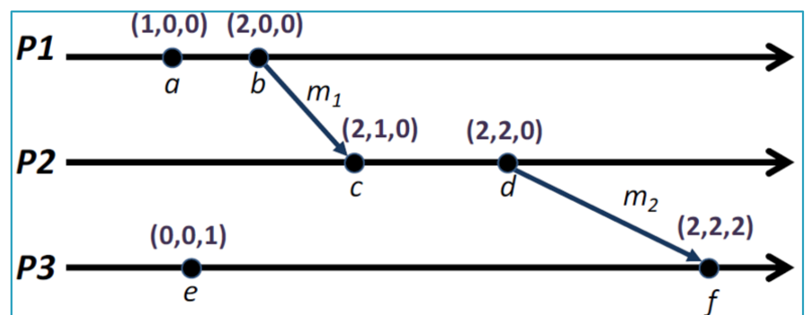
Is it e → c or e ~ c?



Vector Clocks

	a	b	c	d	e	f
P_1	(1,0,0)	(2,0,0)				
P_2			(2,1,0)	(2,2,0)		
P_3					(2,2,1)	(2,2,2)

It is e ~ c!



Summary

- Accurate clock synchronisation is an important task for many distributed systems
- We've looked at various approaches to achieve that by
  - using physical or logical clocks
  - applying different synchronisation algorithms / approaches
- In the next lecture we'll be looking at concrete time synchronisation network protocols, how they work, and their performance (i.e., Assignment 1)