Distributed Systems

Coordination and Synchronization

WEEK 7: Clock Synchronization, Mutual Exclusion, Election Algorithms, Consensus, Coordination Protocol.

Online Lecture Series - 7

Felix Edesa

Addis Ababa Science and Technology University

AASTU

Coordination and Synchronization

Topics We Will Cover

- **Clocks** — Physical Logical (Lamport Vector)
- **Mutual Exclusion** — Centralized, Distributed, Token-ring
- **Election Algorithms** — Bully, Ring, Raft, ZooKeeper
- **Gossip Event Matching** — Peer-sampling, Pub-Sub
- **Location Systems** — GPS logical positioning
- **Summary** — Key coordination principles

AASTU

Physical Clocks

Key Points

- **Problem** — Distributed systems need precise time for event ordering
- **Solution** — Universal Coordinated Time (UTC)
- Based on **cesium-133 atomic clocks**; global time averaged over 50 clocks
- **Example:** Google Spanner uses UTC-based TrueTime API to synchronize transactions
- Leap seconds are added to adjust Earth's rotation
- UTC broadcast via GPS; provides accuracy ± 0.5 ms

AASTU

Clock Synchronization: Precision

Key Points

- **Precision** — Ensures the deviation between clocks on different machines stays within a bound ()
- Mathematically: $\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$
- Keeps distributed systems consistent in event ordering and coordination
- Important in industrial systems: e.g., high-frequency trading servers, telecom networks, and data centers

AASTU

Clock Synchronization: Accuracy Methods

Key Points

- **Accuracy** — Ensures each clock stays close to true time ()
- Mathematically: $\forall t, \forall p : |C_p(t) - t| \leq \alpha$
- **Internal Synchronization** — Maintains precise relative timing between clocks
- **External Synchronization** — Aligns clocks with an external reference (e.g., UTC or GPS)
- Practical examples: Coordinating industrial IoT sensors, distributed database timestamps, and cloud infrastructure

AASTU

Clock Synchronization: Practical Examples

Key Points

- **Data Centers** — Servers synchronize clocks using NTP to maintain consistent timestamps
- **Telecom Networks** — Base stations synchronize with GPS for accurate call timing
- **Financial Systems** — High-frequency trading systems use UTC-synchronized clocks for transaction ordering

AASTU

Clock Synchronization: Solution Approaches

Key Points

- **Network Time Protocol (NTP)**
  - Millisecond-level accuracy for LAN/WAN environments
- **Precision Time Protocol (PTP)**
  - Sub-microsecond synchronization for industrial systems
- **GPS-based synchronization**
  - Absolute time reference with high accuracy

AASTU

Clock Synchronization: Mathematical Problems & Solutions

Key Points

- **Precision Problem** — Keep clocks of any two machines within a bound π
  - Example: $\pi = 5$ ms $\rightarrow$ clocks of two servers differ by at most 5 milliseconds
  - $\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$
  - Ensures consistent event ordering in distributed databases
- **Accuracy Problem** — Keep clocks close to UTC within bound α
  - Example: $\alpha = 2$ ms $\rightarrow$ server clock is within ± 2 ms of true UTC
  - $\forall t, \forall p : |C_p(t) - t| \leq \alpha$
  - Critical for GPS, financial trading, and industrial automation

AASTU

Clock Synchronization: Precision Calculations

Key Points

- **Precision Definition** — Maximum allowed deviation between two clocks: π
  - $\forall t, \forall p, q : |C_p(t) - C_q(t)| \leq \pi$
- **Example Problem** — $\pi = 5$ ms
  - Clock A: $C_A(t) = 12:00:00.003$
  - Clock B: $C_B(t) = 12:00:00.007$
  - Deviation: $|C_A - C_B| = 4$ ms $\leq \pi \rightarrow$ Precision satisfied

AASTU

Clock Synchronization: Accuracy Calculations

Key Points

- **Accuracy Definition** — Maximum deviation from true UTC: α
  - $\forall t, \forall p : |C_p(t) - t| \leq \alpha$
- **Example Problem** — $\alpha = 2$ ms
  - Computed clock: $C_A(t) = 12:00:00.001$
  - True UTC: $t = 12:00:00.000$
  - Deviation: $|C_A - t| = 1$ ms $\leq \alpha \rightarrow$ Accuracy satisfied
- **Solution Approach** — NTP/PTP or GPS synchronization to maintain bounds

AASTU

Clock Drift

Key Points

- **Definition** — Deviation from ideal clock frequency over time
- **Maximum Drift Rate** — ρ, defines worst-case deviation per unit time
- **Clock Frequency** — $F(t)$ actual oscillator vs F ideal constant frequency
- **Impact** — Drift causes clocks to gradually diverge, affecting synchronization
- **Example**
  - $\rho = 10^{-5} \rightarrow \pm 1$ ms drift per 100 s
- **Correction** — Requires periodic synchronization using NTP/PTP or GPS

AASTU

Clock Drift: Frequency Bounds

Key Formula

$$\forall t : (1 - \rho) \leq \frac{F(t)}{F} \leq (1 + \rho)$$

- **Interpretation** — Actual clock frequency $F(t)$ can vary $\pm\rho$ from ideal F
- **Example** — $F = 1$ MHz, $\rho = 10^{-5}$ - Allowed range: $0.99999 \leq F(t)/F \leq 1.00001$ - $F(t) = 1.000008$ MHz $\rightarrow$ within bounds - $F(t) = 0.999985$ MHz $\rightarrow$ exceeds drift $\rightarrow$ needs correction

AASTU

Software Clock via Hardware Interrupts

Key Points

- **Concept** — Software clock can be coupled to hardware clock using hardware interrupts
- **Clock Integration** — Software clock computed as:

$$C_p(t) = \frac{1}{F} \int_0^t F(t)\, dt$$

- **Interpretation** — Software clock directly accumulates hardware clock ticks

Bounded Software Clock Drift Rate

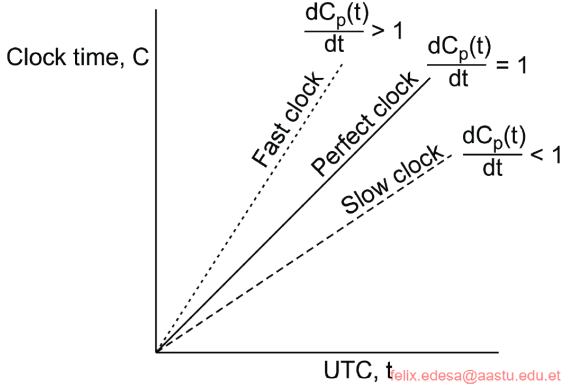
Drift Rate Bound

$$\forall t : 1 - \rho \leq \frac{dC_p(t)}{dt} \leq 1 + \rho$$

- **Interpretation** — Software clock's instantaneous rate must stay within $\pm\rho$ of ideal
- **Example** — If $\rho = 10^{-5}$, the drift rate must remain within 0.99999 to 1.00001 per unit time
- **Significance** — Ensures software clocks stay precise and maintain synchronization

AASTU

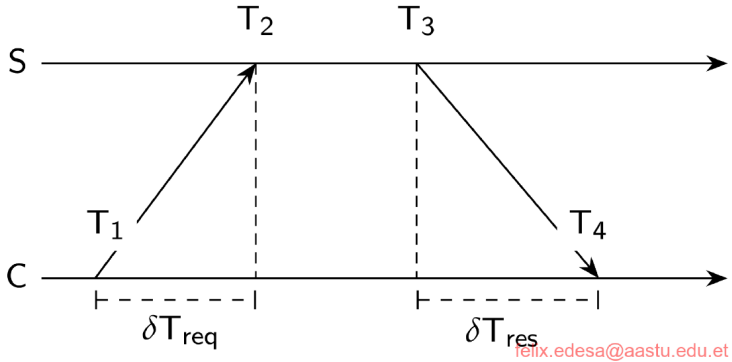
Fast, Perfect, Slow Clocks



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

felix.edesa@aastu.edu.et

Detecting and adjusting incorrect times



felix.edesa@aastu.edu.et

M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Detecting and adjusting incorrect times

Computing the relative offset θ and delay δ

Assumption: $\delta T_{req} = T_2 - T_1 \approx T_4 - T_3 = \delta T_{res}$

$$\theta = T_3 + ((T_2 - T_1) + (T_4 - T_3))/2 - T_4 = ((T_2 - T_1) + (T_3 - T_4))/2$$

$$\delta = ((T_4 - T_1) - (T_3 - T_2))/2$$

Network Time Protocol

Collect (θ, δ) pairs. Choose θ for which associated delay δ was minimal.

M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

AASTU

Reference Broadcast Synchronization

Key Points

- **Essence** — A node sends a reference message m; each receiver p records the reception time $T_{p,m}$ using its local clock
- **Problem** — Simple averaging of timestamps does not eliminate clock drift
- **Solution** — Reference Broadcast Synchronization (RBS) reduces critical path drift by comparing local timestamps across nodes
- **Technique** — Linear regression can be applied on received timestamps to improve synchronization accuracy

AASTU

RBS: Relative Offsets

Offset Computation

- **Relative Offset Between Nodes**:

$$\text{Offset}[p, q] = \frac{1}{M} \sum_{k=1}^{M} (T_{p,k} - T_{q,k})$$

- M = Total number of reference messages sent
- **Problem** — Averaging becomes inaccurate over time due to clock drift
- **Benefit** — Provides initial estimate of node clocks relative to each other

AASTU

RBS: Drift Compensation

Linear Regression Solution

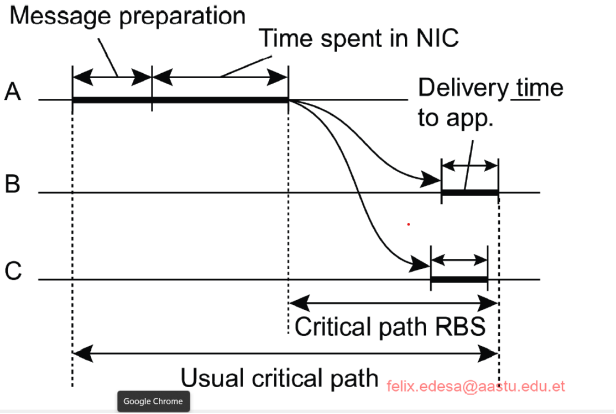
- **Linear Regression Model**:

$$\text{Offset}[p, q](t) = \alpha t + \beta$$

- α, β computed from timestamp pairs $(T_{p,k}, T_{q,k})$
- **Purpose** — Compensates for drift over time
- **Benefit** — Nodes can accurately estimate each other's current clock values dynamically

AASTU

Detecting and adjusting incorrect times



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

The Happened-Before Relationship

Key Points

- **Issue** — In distributed systems, the exact global time is less important than the order of events; a formal ordering is required
- **Happened-Before Relation ($\rightarrow$)**:
  - If events a and b occur in the same process and a precedes b, then $a \rightarrow b$
  - If a is the sending of a message and b is its receipt, then $a \rightarrow b$
  - Transitivity: if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
- **Note** — This defines a partial ordering of events in systems with concurrent processes

AASTU

Logical Clocks: Motivation

Problem Statement

- **Challenge** — How can we maintain a global view of events consistent with the happened-before relation in a distributed system?
- **Requirement** — Each event e should be assigned a timestamp $C(e)$ that respects event ordering.
- **Constraint** — In distributed systems, there is no global physical clock to rely on.
- **Solution Idea** — Introduce logical clocks: maintain a consistent clock per process to track event ordering.

AASTU

Logical Clocks: Rules

Key Principles

- **Rule P1** — For two events a and b in the same process, if $a \rightarrow b$, then $C(a) < C(b)$
- **Rule P2** — If a is the sending of a message m and b is its receipt, then $C(a) < C(b)$
- **Implementation** — Assign timestamps using logical clocks maintained independently in each process
- **Benefit** — Ensures a consistent ordering of events across distributed processes without a global clock
- **Practical Example** — Ordering of transactions in distributed databases

AASTU

Logical Clocks: Algorithm Steps

Key Steps

- **Local Counter** — Each process P_i keeps a counter C_i to track its events
- **Internal Events** — Increment C_i by 1 for every local event
- **Message Sending** — Attach current counter as timestamp: $ts(m) = C_i$
- **Message Receiving** — Upon receiving m, update local counter: $C_j = \max(C_j, ts(m))$ and then increment by 1
- **Tie-breaking** — If two events have the same timestamp, use process IDs to determine order

AASTU

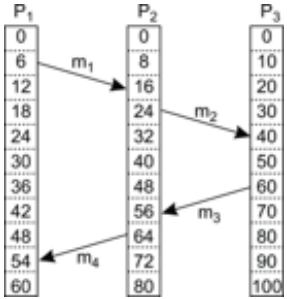
Logical Clocks: Practical Examples

Examples of Logical Clock Usage

- **Distributed Database Transactions** — Ensures operations across servers follow a consistent order
- **Messaging Systems** — Guarantees correct sequence of events in chat
- **Software Version Control** — Helps maintain chronological ordering of commits.
- **Event Ordering Illustration** — If P_1 sends m at $C_1 = 3$ and P_2 has $C_2 = 2$, receiving m updates $C_2 = \max(2, 3) + 1 = 4$
- **Benefit** — Avoids ambiguity and ensures causal consistency without a global clock

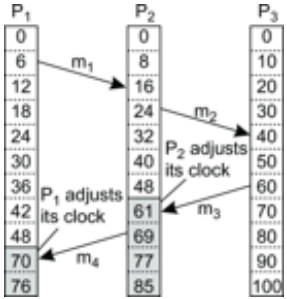
AASTU

Logical clocks: example



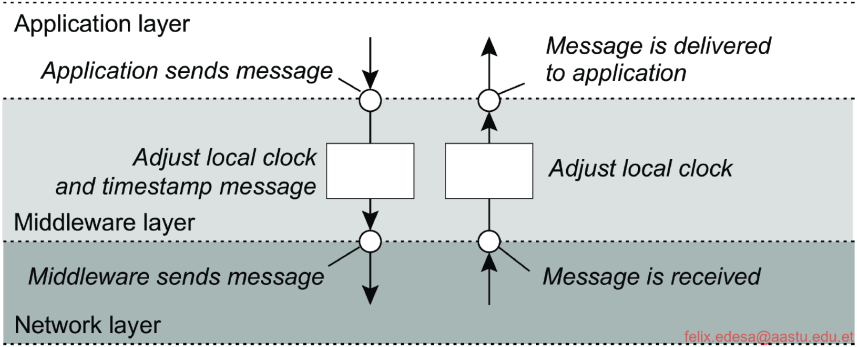
M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Logical clocks: example



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Logical clocks: where implemented Adjustments implemented in middleware



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Example: Totally Ordered Multicast

Scenario

- Multiple processes update a replicated database concurrently
- **Goal** — Ensure all replicas see updates in the same order
- Example operations on an account initially valued at \$1000:
  - P_1 adds \$100
  - P_2 applies 1% increment
- There are two replicas that must remain consistent

AASTU

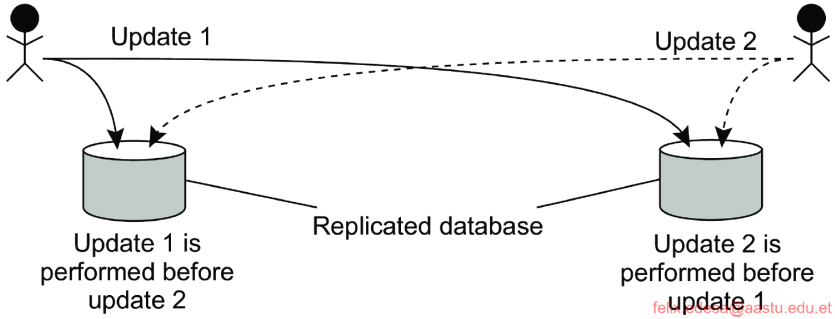
Example Result: Unsynchronized Updates

Impact of Missing Logical Clocks

- **Replica 1** sees updates in order: $\$1000 \rightarrow +\$100 \rightarrow +1\% \rightarrow \$1111$
- **Replica 2** sees updates in different order: $\$1000 \rightarrow +1\% \rightarrow +\$100 \rightarrow \$1110$
- **Observation** — Inconsistent replicas arise without proper ordering
- **Solution** — Use logical clocks and totally ordered multicast to synchronize updates across all replicas
- **Benefit** — All replicas maintain consistent, deterministic state despite concurrent updates

AASTU

Unsynchronized Updates



M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., 2024.

Totally Ordered Multicast: Message Handling

Key Steps

- Each process P_i sends a **timestamped message** m_i to all other processes.
- The message is added to its local message queue $queue_i$.
- Upon receiving m_i, process P_j:
  - Inserts m_i into its local queue $queue_j$ according to its timestamp.
  - Sends an acknowledgment to all other processes.
- **Example:** In a replicated banking system, when P_1 updates an account balance, the update message is timestamped and shared with all replicas before processing.

Totally Ordered Multicast: Delivery Conditions

Message Delivery Rules

- A process P_j delivers a message m_i to the application only if:
  - m_i is at the **head of** $queue_j$, and
  - For every process P_k, there exists a message m_k in $queue_j$ with a **larger timestamp**.
- Ensures that all processes deliver updates in the **same global order**.
- **Example:** If P_1 (deposit) and P_2 (interest update) issue concurrent transactions, timestamps ensure every replica applies them in the same sequence — avoiding balance mismatches.
- **Assumption:** Communication is reliable and follows FIFO ordering.

AASTU

Lamport's Logical Clocks: Process Initialization

Concept Overview

- Each process P_i joins the communication channel and identifies other processes.
- Initializes:
  - A unique process ID.
  - A request queue to manage critical section requests.
  - A logical clock $C_i = 0$.
- When P_i wants to enter the **critical section**:
  - It increments its logical clock.
  - Adds a request message to its local queue.

AASTU

Lamport's clocks for mutual exclusion

```python
import heapq

class Channel:
    def __init__(self):
        self.processes = {}

    def join(self, proc):
        self.processes[proc.procID] = proc

    def broadcast(self, senderID, message):
        for pid, proc in self.processes.items():
            proc.receive(message, senderID)

```

AASTU

Lamport's clocks for mutual exclusion

```
14 class Process:
15     def __init__(self, chan, procID, procIDSet):
16         self.chan = chan
17         self.procID = procID
18         self.otherProcs = [p for p in procIDSet if p != self.procID]
19         self.clock = 0
20         self.queue = []  # Priority queue (timestamp, procID)
21         self.replies = set()
22
23     def increment_clock(self, received_time=None):
24         """Update Lamport clock."""
25         if received_time is not None:
26             self.clock = max(self.clock, received_time) + 1
27         else:
28             self.clock += 1
29
```

AASTU

Lamport's clocks for mutual exclusion

```
14 class Process:
30     def request_critical_section(self):
33         heapq.heappush(self.queue, request)
34         print(f"P{self.procID} sends request at time {self.clock}")
35         self.chan.broadcast(self.procID, request)
36
37     def receive(self, message, senderID):
38         timestamp, proc, msg_type = message
39         self.increment_clock(timestamp)
40         heapq.heappush(self.queue, message)
41
42         if msg_type == "REQUEST":
43             # Always reply immediately
44             reply = (self.clock, self.procID, "REPLY")
45             self.chan.processes[senderID].receive(reply, self.procID)
46
47         elif msg_type == "REPLY":
48             self.replies.add(senderID)
49             if self.can_enter_CS():
50                 self.enter_critical_section()
51
```

AASTU

Lamport's clocks for mutual exclusion-sIMULATION-PART

```
71 # ================================
72 # Simulation
73 # ================================
74 procIDs = [1, 2, 3]
75 chan = Channel()
76 procs = [Process(chan, pid, procIDs) for pid in procIDs]
77 for p in procs:
78     chan.join(p)
79
80 # P1 requests CS
81 procs[0].request_critical_section()
82
83 # P2 requests CS
84 procs[1].request_critical_section()
85
86 # Show queues
87 for p in procs:
88     print(f"P{p.procID} Queue: {p.queue}")
```

AASTU

Lamport's clocks for mutual exclusion-OUTPUT

```
P1 sends request at time 1

🔒 P1 enters CRITICAL SECTION at time 4

🔓 P1 leaves CRITICAL SECTION.

P2 sends request at time 3
P1 Queue: [(1, 1, 'REQUEST'), (2, 2, 'REPLY'), (2, 1, 'REPLY'), (2, 3, 'REPLY'), (3, 2, 'REQUEST')]
P2 Queue: [(1, 1, 'REQUEST'), (3, 2, 'REQUEST'), (4, 3, 'REPLY'), (3, 2, 'REQUEST'), (8, 2, 'REPLY'), (6, 1
, 'REPLY')]
P3 Queue: [(1, 1, 'REQUEST'), (3, 2, 'REQUEST')]
```

AASTU

Lamport's Clocks for Mutual Exclusion

Analogy with Totally Ordered Multicast

- In **totally ordered multicast**, all processes maintain **identical message queues** and deliver messages in the exact same order.
- Similarly, Lamport's clock ensures that **critical section requests** are ordered consistently across distributed processes.
- Instead of agreeing on message delivery order, the system agrees on **which process enters the critical section first**.
- This ensures **mutual exclusion** — only one process can access the shared resource.

Lamport's Clocks for Mutual Exclusion — Example

Example Scenario

- Consider three processes: P_1, P_2, and P_3 sharing a critical resource.
- P_1 and P_2 both request access to the resource nearly simultaneously.
- Each process assigns a **Lamport timestamp** to its request and multicasts it to all.
- All processes maintain a synchronized queue ordered by timestamp.
- The process with the **smallest timestamp** (e.g., P_1) enters the critical section first.
- Once P_1 exits, it sends a **RELEASE** message, allowing the next in queue (P_2) to proceed.

AASTU

Lamport's Clocks vs Totally Ordered Multicast

Comparison

- **Totally Ordered Multicast:**
  - Goal — Ensure all processes deliver messages in the same order.
  - Used for replicated databases and consistent updates.
- **Lamport's Mutual Exclusion:**
  - Goal — Ensure all processes agree on the entry order to a critical section.
  - Used for distributed synchronization and resource sharing.
- Both rely on **Lamport timestamps** to achieve global ordering in a distributed environment.

AASTU

Questions?

Thank you for your attention!

References

Suggested References

1. M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 4th ed., Version 4.02, February 2024.
2. G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed., Addison-Wesley, 2012.
3. N. Lynch, *Distributed Algorithms*, 2nd ed., Morgan Kaufmann, 1996.
4. R. E. Bryant and D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 4th ed., Pearson, 2024.

AASTU