Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

This text assumes the reader has a basic understanding of computer science and programming. It will also assume they are somewhat familiar with computer architecture and operating systems at a high level. The text will review these concepts briefly for completeness, but it will not try to teach them from scratch or provide a full understanding of these topics.

The main focus will be on distributed systems, and will touch on:

- **Concurrency and Parallelism**
  - Concurrency, Parallelism, Threads
- **Consistency and Fault Tolerance**
  - Consistency, Fault-tolerance, Atomicity
- **Distributed Systems and Coordination**
  - Asynchrony, Coordination, Logical Time, Snapshots
- **Consensus Algorithms**
  - Raft, Paxos, Consensus
- **Replication and Data Management**
  - Replication, Sharding, Cluster
- **Protocols and Computing Models**
  - RPC, 2PC, Broadcast
- **Technologies and Tools**
  - MapReduce, Spanner, Dynamo, GFS, TLA+, Golang

— 1 —
Introduction

1.1 Time, Clocks, and Logical Ordering

Accuracy of Time: Atomic Clocks & NTP

Time allows us to order and identify events. Say we ran `time.Now()` on two different machines:

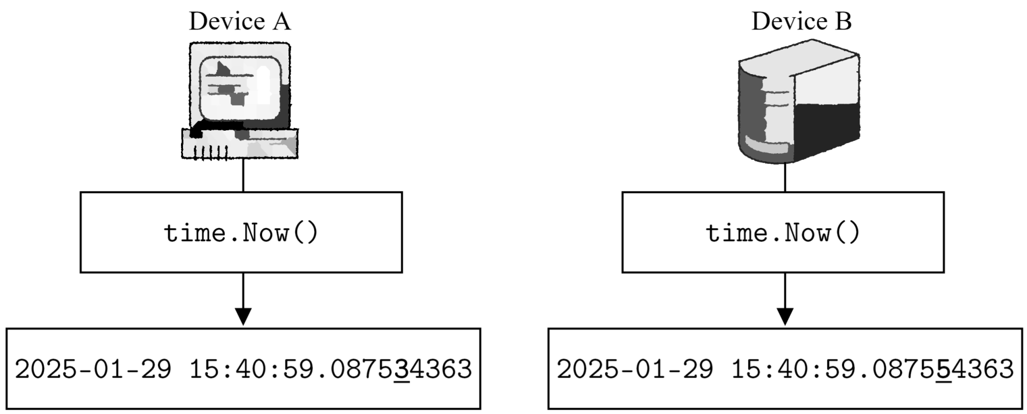


Figure 1.1: Using `time.Now()` on two different machines

Despite Device A appearing to be ahead of Device B, we cannot be certain via the following reasons:

Theorem 1.1: Clock Synchronization Impossibility

There are two key reasons why perfect clock synchronization is impossible:

- **Clock Skew:** There's a difference between every system clock (ideally 0), as they maintain their own local clock via a hardware oscillator incrementing a counter register.
- **Clock Drift:** Even if systems initialize with a reference time, their clocks will inevitably diverge due to variations in manufacturing, age, or environmental factors such as temperature. We measure the deviation by, $\frac{dC}{dt} = 1 + \rho$, where C is the clock time and t is the real time, and ρ (rho) is the drift rate (ideally 0).

We may formalize what we may consider synchronized clocks as follows:

Definition 1.1: Clock Synchronization Threshold

Let there be two clocks C_i and C_j. They are (delta) δ-synchronized if for all t time units:

$$|C_i(t) - C_j(t)| \leq \delta$$

E.g., C_i and C_j are δ-synchronized within 10ms if $|C_i(t) - C_j(t)| \leq 10ms$

In practice we to achieve semi-synchronized clocks, we developed the following protocol:

Definition 1.2: Network Time Protocol (NTP)

The NTP is a protocol synchronizes network clocks via a ground-truth time distribution system. The ground-truth time is are GPS satellite **atomic clocks**, which exhibit negligible drift over millions of years.

NTP employs a **round-trip time (RTT)** calculation to estimate the clock offset request latency. It also organizes synchronization strength into a **stratum hierarchy**, where lower-numbered stratums indicate more accurate time sources:

- **Stratum 0:** Ground truth **atomic clocks/GPS receivers**.
- **Stratum 1:** NTP servers that directly synchronize with Stratum 0 reference clocks.
- **Stratum 2:** NTP servers synchronized to Stratum 1 servers.
- **Stratum 3 and beyond:** Weaker NTP servers synchronized to higher-stratum servers.
- **Stratum 16:** A system considered *unsynchronized* (e.g., a freshly booted system).

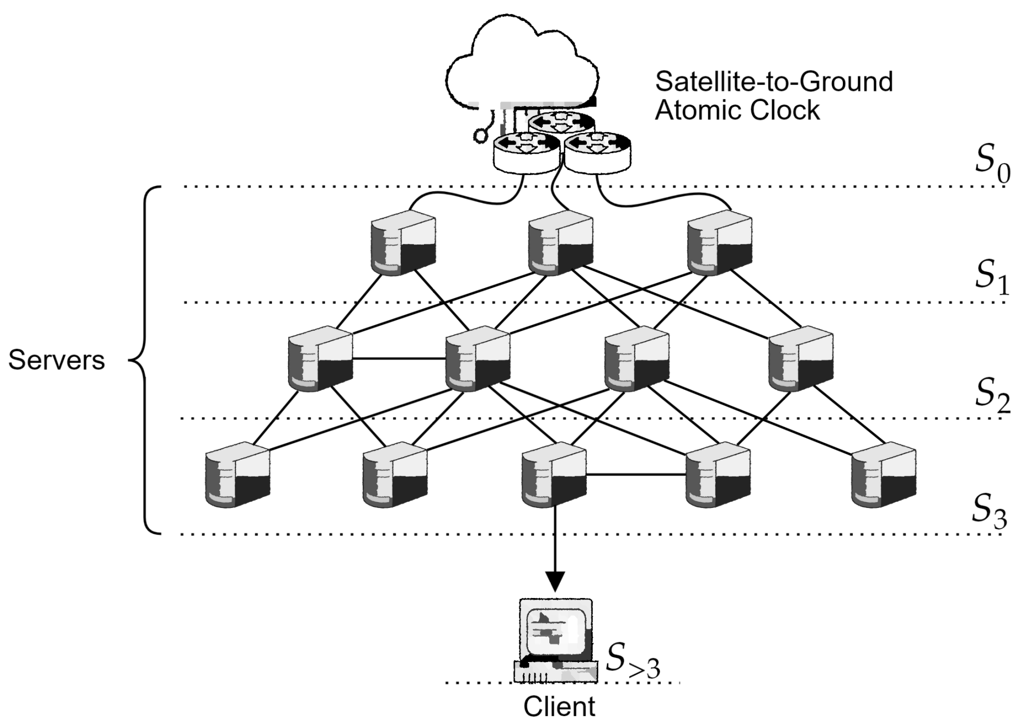


Figure 1.2: NTP Stratum Hierarchy From GPS Satellite Atomic Clock to Client.

Logical Clocks: Lamport & Vector Clocks

To get away from the limitations of physical clocks, we may use logical clocks to order events.

Definition 1.3: Logical Clocks

Let a and b be two events part of a totally ordered set of events. Let function $t(x)$ denote the time of event x. Then,

$$a \to b \implies t(a) < t(b)$$

Where $a \to b$ denotes that event a happens before b, which implies $t(a) < t(b)$.

We may become more formal about cause and effect relationships with the following definition:

Definition 1.4: Causal Order

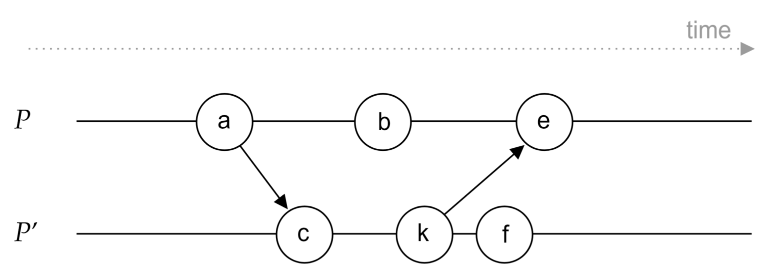
For r **execution trace** (sequence of events), the causal order relationship $\to_r$ is defined as:

- If a **happens before** b in the same process, then $a \to_r b$.
- If a is a **sender** and b the **receiver**, then $a \to_r b$.
- **Transitive Property**: if $a \to_r b$ and $b \to_r c$, then $a \to_r c$.
- Events a and b are **concurrent** (denoted as $a \parallel b$) if:
  - $a \not\to_r b$ and $b \not\to_r a$, meaning neither event happened before the other.

Example 1.1: Causal Order Example

Determine the causal order relationship between events in processes P and P':

- (a): $a \ ? \ b$; (b): $a \ ? \ k$; (c): $c \ ? \ b$; (d): $c \ ? \ e$



Answer on the next page. ■

Example (1.1) Answer: (a): $a \to_r b$; (b): $a \to_r k$; (c): $c \parallel b$; (d): $c \to_r e$.

Now we discuss a method that utilizes causal order, though assigns logical timestamps to events:

Definition 1.5: Lamport Clocks

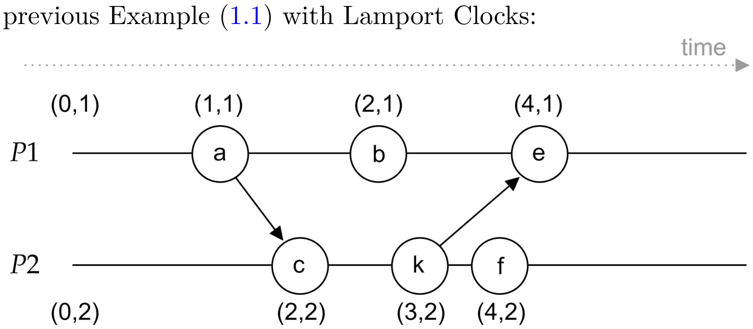
Named after Leslie Lamport, Lamport Clocks assign a logical timestamp to each event:

Let t_p store the logical time of process p. Then,

- **Initialization:** t_p is initialized to 0.
- **Timestamp Syntax:** Timestamps are tuples (t_p, p), assigned to each e event.
- **Incrementing:** For each e in process p, increment t_p by 1 and assign the (t_p, p) to e.
- **Sending:** If p sends a message m to process q, the timestamp included is $((t_p + 1), p)$.
- **Receiving:** Upon receiving message m, process q sets $t_q = \max((t_p + 1), t_q)$.

Example 1.2: Lamport Clocks Example (1.1) Extended

Consider the previous Example (1.1) with Lamport Clocks:



■

In practice the only thing we have access to are these logical timestamps, which we must evaluate:

Theorem 1.2: Comparing Lamport Timestamps

Given two events a and b with timestamps $t(a)$ and $t(b)$, with r trace, we only guarantee:

- If $a \to_r b$, then $t(a) < t(b)$.
- If $t(a) \geq t(b)$, then $a \not\to_r b$.

We may now derive the following about concurrency:

Theorem 1.3: Non-causality

Two events a and b are concurrent ($a \parallel b$) under r trace if **both** conditions hold:

- $a \not\to_r b$ (a does not happen before b).
- $b \not\to_r a$ (b does not happen before a).

Lamport Clocks are useful for causal ordering, but they do not capture the full context of events:

Definition 1.6: Vector Clocks

Let there be $p_1, p_2, \ldots, p_n$ processes each with a vector (array) v of size n. Each index $v[i]$ stores the logical time of process p_i. Then, the following rules apply:

- **Initialization:** Each $v[i]$ of p_i is initialized to 0 (e.g., $[0, 0, \ldots, 0]$).
- **Incrementing:** For each event e in process p_i, increment $v[i]$ by 1.
- **Sending:** When p_i sends a message m to p_j, include v in m (**no increment**).
- **Receiving:** Upon receiving message m, process p_j sets $v[j] = \max(v[j], m[j])$.

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.