Today
1. The need for time synchronization
2. “Wall clock time” synchronization
3. Logical Time: Lamport Clocks

A distributed edit-compile workflow

• 2143 < 2144 → make doesn’t call compiler

Lack of time synchronization result – a possible object file mismatch

What makes time synchronization hard?
1. Quartz oscillator sensitive to temperature, age, vibration, radiation
   - Accuracy ca. one part per million (one second of clock drift over 12 days)
2. The internet is:
   • Asynchronous: arbitrary message delays
   • Best-effort: messages don’t always arrive
Today
1. The need for time synchronization
2. “Wall clock time” synchronization
   – Cristian’s algorithm, Berkeley algorithm, NTP
3. Logical Time: Lamport clocks

Just use Coordinated Universal Time?
• UTC is broadcast from radio stations on land and satellite
  (e.g., the Global Positioning System)
  – Computers with receivers can synchronize their clocks
    with these timing signals
• Signals from land-based stations are accurate to about
  0.1–10 milliseconds
• Signals from GPS are accurate to about one microsecond
  – Why can’t we put GPS receivers on all our computers?

Synchronization to a time server
• Suppose a server with an accurate clock (e.g., GPS-disciplined crystal oscillator)
  – Could simply issue an RPC to obtain the time:

  Client
  Time of day? 2:50 PM
  Server

  Time ↓

• But this doesn’t account for network latency
  – Message delays will have outdated server’s answer

Cristian’s algorithm: Outline
1. Client sends a request packet, timestamped with its local clock $T_1$
2. Server timestamps its receipt of the request $T_2$ with its local clock
3. Server sends a response packet with its local clock $T_3$ and $T_2$
4. Client locally timestamps its receipt of the server’s response $T_4$

How the client can use these timestamps to synchronize its local clock to the server’s local clock?
Today

1. The need for time synchronization

2. “Wall clock time” synchronization
   - Cristian’s algorithm, Berkeley algorithm, NTP

3. Logical Time: Lamport clocks

Berkeley algorithm

• A single time server can fail, blocking timekeeping

• The Berkeley algorithm is a distributed algorithm for timekeeping
  – Assumes all machines have equally-accurate local clocks
  – Obtains average from participating computers and synchronizes clocks to that average

Berkeley algorithm

• Master machine: polls L other machines using Cristian’s algorithm → \( \{ \theta_i \} (i = 1 \ldots L) \)
Today

1. The need for time synchronization
2. “Wall clock time” synchronization
   – Cristian’s algorithm, Berkeley algorithm, NTP
3. Logical Time: Lamport clocks

The Network Time Protocol (NTP)

- Enables clients to be accurately synchronized to UTC despite message delays
- Provides reliable service
  – Survives lengthy losses of connectivity
  – Communicates over redundant network paths
- Provides an accurate service
  – Unlike the Berkeley algorithm, leverages heterogeneous accuracy in clocks

NTP: System structure

- Servers and time sources are arranged in layers (strata)
  – Stratum 0: High-precision time sources themselves
    • e.g., atomic clocks, shortwave radio time receivers
  – Stratum 1: NTP servers directly connected to Stratum 0
  – Stratum 2: NTP servers that synchronize with Stratum 1
    • Stratum 2 servers are clients of Stratum 1 servers
  – Stratum 3: NTP servers that synchronize with Stratum 2
    • Stratum 3 servers are clients of Stratum 2 servers
- Users’ computers synchronize with Stratum 3 servers

NTP operation: Server selection

- Messages between an NTP client and server are exchanged in pairs: request and response
  • Use Cristian’s algorithm
- For \( i \)th message exchange with a particular server, calculate:
  1. Clock offset \( \theta_i \) from client to server
  2. Round trip time \( \delta_i \) between client and server
- Over last eight exchanges with server \( k \), the client computes its dispersion \( \sigma_k = \max \delta_i - \min \delta_i \)
  – Client uses the server with minimum dispersion
NTP operation: How to change time

- Can’t just change time: Don’t want time to run backwards
  – Recall the make example

- Instead, change the update rate for the clock
  – Changes time in a more gradual fashion
  – Prevents inconsistent local timestamps

Clock synchronization: Take-away points

- Clocks on different systems will always behave differently
  – Disagreement between machines can result in undesirable behavior

- NTP, Berkeley clock synchronization
  – Rely on timestamps to estimate network delays
  – 100s µs–ms accuracy
  – Clocks never exactly synchronized

- Often inadequate for distributed systems
  – Often need to reason about the order of events
  – Might need precision on the order of ns

Today

1. The need for time synchronization

2. “Wall clock time” synchronization
  – Cristian’s algorithm, Berkeley algorithm, NTP

3. Logical Time: Lamport clocks
Motivation: Multi-site database replication

- A New York-based bank wants to make its transaction ledger database resilient to whole-site failures
- Replicate the database, keep one copy in sf, one in nyc

The consequences of concurrent updates

- Replicate the database, keep one copy in sf, one in nyc
  - Client sends query to the nearest copy
  - Client sends update to both copies

Inconsistent replicas!

Updates should have been performed in the same order at each copy

Defining “happens-before” (→)

- Consider three processes: P1, P2, and P3
- Notation: Event a happens before event b (a → b)

Idea: Logical clocks

- Landmark 1978 paper by Leslie Lamport
- Insight: only the events themselves matter

Idea: Disregard the precise clock time
Instead, capture just a “happens before” relationship between a pair of events
Defining “happens-before” (→)
- Can observe event order at a single process

1. If same process and a occurs before b, then $a \rightarrow b$
2. Can observe ordering when processes communicate

Physical time ↓

Defining “happens-before” (→)
1. If same process and a occurs before b, then $a \rightarrow b$
2. If c is a message receipt of $b$, then $b \rightarrow c$
1. If same process and a occurs before b, then \( a \rightarrow b \)
2. If c is a message receipt of b, then \( b \rightarrow c \)
3. Can observe ordering transitively

---

**Defining “happens-before” (\( \rightarrow \))**

---

**Concurrent events**

- Not all events are related by \( \rightarrow \)
- \( a, d \) not related by \( \rightarrow \) so **concurrent**, written as \( a \parallel d \)

---

**Lamport clocks: Objective**

- We seek a **clock time** \( C(a) \) for every event \( a \)

**Plan:** Tag events with clock times; use clock times to make distributed system correct

- Clock condition: If \( a \rightarrow b \), then \( C(a) < C(b) \)
The Lamport Clock algorithm

- Each process $P_i$ maintains a local clock $C_i$

1. Before executing an event, $C_i \leftarrow C_i + 1$

**Diagram:**

![Diagram of the Lamport Clock algorithm](image)

1. Before executing an event $a$, $C_i \leftarrow C_i + 1$: 
   - Set event time $C(a) \leftarrow C_i$

2. Send the local clock in the message $m$

**Diagram:**

![Diagram of the Lamport Clock algorithm](image)
The Lamport Clock algorithm

3. On process \( P_j \) receiving a message \( m \):
   - Set \( C_j \) and receive event time \( C(c) \leftarrow 1 + \max\{ C_j, C(m) \} \)

Lamport Timesteps: Ordering all events

- Break ties by appending the process number to each event:
  1. Process \( P_j \) timestamps event \( e \) with \( C(e).i \)
  2. \( C(a).i < C(b).j \) when:
     - \( C(a) < C(b) \), or \( C(a) = C(b) \) and \( i < j \)
  - Now, for any two events \( a \) and \( b \), \( C(a) < C(b) \) or \( C(b) < C(a) \)
    - This is called a total ordering of events

Making concurrent updates consistent

- Recall multi-site database replication:
  - San Francisco (\( P_1 \)) deposited $100:
  - New York (\( P_2 \)) paid 1% interest:

  We reached an inconsistent state

Totally-Ordered Multicast

Goal: All sites apply updates in (same) Lamport clock order

- Client sends update to one replica site \( j \)
  - Replica assigns it Lamport timestamp \( C_j.j \)

Key idea: Place events into a sorted local queue
  - Sorted by increasing Lamport timestamps

Example: \( P_1 \)'s local queue: $1.3 \leftarrow 1.2 \leftarrow$ Timestamps
Totally-Ordered Multicast (Almost correct)

1. On receiving an update from client, broadcast to others (including yourself)

2. On receiving an update from replica:
   a) Add it to your local queue
   b) Broadcast an *acknowledgement message* to every replica (including yourself)

3. On receiving an acknowledgement:
   - Mark corresponding update *acknowledged* in your queue

4. Remove and process updates *everyone* has ack'ed from head of queue

Totally-Ordered Multicast (Correct version)

1. On receiving an update from client, broadcast to others (including yourself)

2. On receiving or processing an update:
   a) Add it to your local queue, if received update
   b) Broadcast an *acknowledgement message* to every replica (including yourself) only from head of queue

3. On receiving an acknowledgement:
   - Mark corresponding update *acknowledged* in your queue

4. Remove and process updates *everyone* has ack'ed from head of queue

(Ack's to self not shown here)
So, are we done?

• Does totally-ordered multicast solve the problem of multi-site replication in general?

• Not by a long shot!

1. Our protocol assumed:
   – No node failures
   – No message loss
   – No message corruption
2. All to all communication does not scale
3. Waits forever for message delays (performance?)

Take-away points: Lamport clocks

• Can totally-order events in a distributed system: that’s useful!
  – We saw an application of Lamport clocks for totally-ordered multicast

• But: while by construction, \(a \rightarrow b\) implies \(C(a) < C(b)\),
  – The converse is not necessarily true:
    • \(C(a) < C(b)\) does not imply \(a \rightarrow b\) (possibly, \(a || b\))

Can’t use Lamport clock timestamps to infer causal relationships between events

Why global timing?

• Suppose there were an infinitely-precise and globally consistent time standard

• That would be very handy. For example:
1. Who got last seat on airplane?
2. Mobile cloud gaming: Which was first, A shoots B or vice-versa?
3. Does this file need to be recompiled?

Wednesday Topic: Vector Clocks & Distributed Snapshots

Friday Precept: RPCs in Go
**Totally-Ordered Multicast (Attempt #1)**

- P1 queues $, P2 queues %
- P1 queues and ack's %
  - P1 marks % fully ack'ed
- P2 marks % fully ack'ed
  - P2 processes %
- P2 queues and ack's $
  - P2 processes $
- P1 marks $ fully ack'ed
  - P1 processes $, then %

Note: ack's to self not shown here

**Totally-Ordered Multicast (Correct version)**

- P1 queues $, P2 queues %
- P1 queues %
- P2 queues and ack's $
- P2 marks $ fully ack'ed
  - P2 processes $
- P1 marks $ fully ack'ed
  - P1 processes $%
- P1 marks % fully ack'ed
  - P1 processes $
- P2 marks % fully ack'ed
  - P2 processes %

(Ack's to self not shown here)

**Time standards**

- **Universal Time (UT1)**
  - In concept, based on astronomical observation of the sun at 0º longitude
  - Known as “Greenwich Mean Time”
- **International Atomic Time (TAI)**
  - Beginning of TAI is midnight on January 1, 1958
  - Each second is 9,192,631,770 cycles of radiation emitted by a Cesium atom
  - Has diverged from UT1 due to slowing of earth’s rotation
- **Coordinated Universal Time (UTC)**
  - TAI + leap seconds, to be within 0.9 seconds of UT1
  - Currently TAI – UTC = 36

**VC application: Order processing**

- Suppose we are running a distributed order processing system
  - Each process = a different user
  - Each event = an order
  - A user has seen all orders with V(order) < the user’s current vector