TIME

George Porter
Nov 6 and 8, 2018

ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Kyle Jamieson, Princeton University (also under a CC BY-NC-SA 3.0 Creative Commons license)
ANNOUNCEMENTS
Reading: van Steen 6.1 through 6.4

Outline

1. Time synchronization
   • Cristian’s algorithm
   • Berkeley algorithm
   • NTP
2. Lamport clocks
3. Vector clocks
A DISTRIBUTED EDIT-COMPILE WORKFLOW

• 2143 < 2144 \(\Rightarrow\) 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 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
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?

Outline

1. Time synchronization
   - Cristian’s algorithm
   - Berkeley algorithm
   - NTP
2. Lamport clocks
3. Vector clocks
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:

\[ \text{Client} \rightarrow \text{Server} \]

\[ \text{Time of day?} \]

\[ 2:50 \text{ PM} \]

\[ \text{Time ↓} \]

- 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  - **Message delays** will have outdated server’s answer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?**
CRISTIAN’S ALGORITHM: OFFSET SAMPLE CALCULATION

Goal: Client sets clock \( T_3 + \delta_{\text{resp}} \)

- **Client samples** *round trip time*
  \[ \delta = \delta_{\text{req}} + \delta_{\text{resp}} = (T_4 - T_1) - (T_3 - T_2) \]
- **But client knows** \( \delta \), **not** \( \delta_{\text{resp}} \)

Assume: \( \delta_{\text{req}} \approx \delta_{\text{resp}} \)

Client sets clock \( T_3 + \frac{1}{2} \delta \)

Outline

1. Time synchronization
   - Cristian’s algorithm
   - Berkeley algorithm
   - NTP
2. Lamport clocks
3. Vector clocks
• 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

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

1. Time synchronization
   • Cristian’s algorithm
   • Berkeley algorithm
   • NTP
2. Lamport clocks
3. Vector 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 $\sigma_k = \max_i \delta_i - \min_i \delta_i$
  - Client uses the server with minimum dispersion
  - Outliers are discarded
NTP OPERATION: CLOCK OFFSET CALCULATION

- Client tracks minimum round trip time and associated offset over the last eight message exchanges \((\delta_0, \theta_0)\)
- \(\theta_0\) is the best estimate of offset: client adjusts its clock by \(\theta_0\) to synchronize to server

![Diagram showing offset vs. round trip time](image)

Each point represents one sample

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 $\mu$s–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

Outline

1. Time synchronization
   - Cristian’s algorithm
   - Berkeley algorithm
   - NTP
2. Lamport clocks
3. Vector clocks
MOTIVATION: MULTI-SITE DATABASE REPLICA TION

• 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

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
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”

- Consider three processes: **P1, P2, and P3**

- **Notation:** Event a *happens before* event b (a → b)

---

![Diagram showing physical time for processes P1, P2, and P3]

Physical time ↓
**DEFINING “HAPPENS-BEFORE”**

1. Can observe event order at a single process

   ![Diagram showing event order at a single process]

   - P1
     - a
     - b
   - P2
   - P3
     - Physical time ↓

**DEFINING “HAPPENS-BEFORE”**

1. If *same process* and a occurs before b, then a → b

   ![Diagram showing event order at a single process]

   - P1
     - a
     - b
   - P2
   - P3
     - Physical time ↓
DEFINING “HAPPENS-BEFORE”

1. If same process and a occurs before b, then $a \rightarrow b$

2. Can observe ordering when processes communicate

---

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 $b \rightarrow c$
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 \( b \rightarrow c \)
3. Can observe ordering transitively

\[ P1 \rightarrow P2 \rightarrow P3 \]

Physical time ↓

TRANSITIVE “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 \( b \rightarrow c \)
3. If \( a \rightarrow b \) and \( b \rightarrow c \), then \( a \rightarrow c \)

\[ P1 \rightarrow P2 \rightarrow P3 \]

Physical time ↓
CONCURRENT EVENTS

• 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 showing the Lamport clock algorithm with three processes P1, P2, P3, and events a, b, c and physical time moving downwards.](image)
1. Before executing an event $a$, $C_i \leftarrow C_i + 1$:
   - Set event time $C(a) \leftarrow C_i$

![Diagram of the Lamport Clock Algorithm for event a]

Physical time ↓

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

![Diagram of the Lamport Clock Algorithm for event b]

Physical time ↓
THE LAMPORT CLOCK ALGORITHM

1. Before executing an event \( b \), \( C_i \leftarrow C_i + 1 \)

2. Send the local clock in the message \( m \)

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

- **Break ties** by appending the process number to each event:
  1. Process $P_i$ timestamps event $e$ with $C_i(e).i$
  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

*Could we design a system that uses Lamport Clock total order to make multi-site updates consistent?*
**TOTALLY-ORDERED MULTICAST**

- Client sends update to **one replica** → Lamport timestamp $C(x)$

- **Key idea:** Place events into a **local queue**
  - **Sorted** by increasing $C(x)$

- **Key diagram**

<table>
<thead>
<tr>
<th>P1's local queue:</th>
<th>P2's local queue:</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Event 1.1]</td>
<td>![Event 1.2]</td>
</tr>
<tr>
<td>![Event 1.1]</td>
<td>![Event 1.2]</td>
</tr>
</tbody>
</table>

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

---

**TOTALLY-ORDERED MULTICAST (ALMOST CORRECT)**

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

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

3. **Remove and process** events **everyone** has ack’ed from **head** of queue
11/4/2018

TOTALLY-ORDERED MULTICAST (ALMOST CORRECT)

- \( P_1 \) queues $, \( P_2 \) queues %
- \( P_1 \) queues and ack’s %
  - \( P_1 \) marks % fully ack’ed
- \( P_2 \) marks % fully ack’ed

(TOTALLY-ORDERED MULTICAST (CORRECT VERSION))

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

2. On receiving or processing an event:
   a) Add it to your local queue
   b) Broadcast an acknowledge message to every process (including yourself) only from head of queue

3. When you receive an acknowledgement:
   - Mark corresponding event acknowledged in your queue

4. Remove and process events everyone has ack’ed from head of queue
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!
- 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

Outline

1. Time synchronization
   - Cristian’s algorithm
   - Berkeley algorithm
   - NTP
2. Lamport clocks
3. Vector clocks
VECTOR CLOCK (VC)

- Label each event $e$ with a vector $V(e) = [c_1, c_2, ..., c_n]$
  - $c_i$ is a count of events in process $i$ that causally precede $e$
- Initially, all vectors are $[0, 0, ..., 0]$

- Two update rules:
  1. For each local event on process $i$, increment local entry $c_i$
  2. If process $j$ receives message with vector $[d_1, d_2, ..., d_n]$:
     - Set each local entry $c_k = \max\{c_k, d_k\}$
     - Increment local entry $c_j$

VECTOR CLOCK: EXAMPLE

- All counters start at $[0, 0, 0]$
- Applying local update rule
- Applying message rule
  - Local vector clock piggybacks on inter-process messages

[2,2,2]: Remember we have event $e$ at P3 with timestamp $[0,0,1]$. D’s message gets timestamp $[2,2,0]$, we take max to get $[2,2,1]$ then increment the local entry to get $[2,2,2]$.
VECTOR CLOCKS CAN ESTABLISH CAUSALITY

• Rule for comparing vector clocks:
  • \( V(a) = V(b) \) when \( a_k = b_k \) for all \( k \)
  • \( V(a) < V(b) \) when \( a_k \leq b_k \) for all \( k \) and \( V(a) \neq V(b) \)

• Concurrency: \( a \parallel b \) if \( a_i < b_i \) and \( a_j > b_j \), some \( i, j \)

• \( V(a) < V(z) \) when there is a chain of events linked by \( \rightarrow \) between \( a \) and \( z \)

Two events \( a, z \)

Lamport clocks: \( C(a) < C(z) \)

Conclusion: None

Vector clocks: \( V(a) < V(z) \)

Conclusion: \( a \rightarrow \ldots \rightarrow z \)

Vector clock timestamps tell us about causal event relationships
VC APPLICATION: CAUSALLY-ORDERED BULLETIN BOARD SYSTEM

- Distributed bulletin board application
- Each post → multicast of the post to all other users
- **Want**: No user to see a reply before the corresponding original message post
- Deliver message only after all messages that causally precede it have been delivered
- Otherwise, the user would see a reply to a message they could not find

VC APPLICATION: CAUSALLY-ORDERED BULLETIN BOARD SYSTEM

- User 0 posts, user 1 replies to 0’s post; user 2 observes