Topics covered in this lecture

- Logical clocks
- Vector clocks
- Matrix clocks
Physical time in a distributed system is problematic

- This is not because of the effects of special relativity, which are negligible or non-existent for normal computers
  - Unless you count computers travelling in spaceships

- It is because of the inability to accurately timestamp events at different nodes
  - We need this to order any pairs of events
If two processes do not interact with each other?

- Their clocks need not be synchronized
- Lack of synchronization is not observable
  - Does not cause problems

Logical clocks

- Within a single process, events are ordered uniquely by times shown on local clock
- But we cannot synchronize clocks perfectly across a distributed system [Lamport 1978]
  - We cannot use physical time to find out the order of an arbitrary pair of events in a distributed system
We can use a scheme that is similar to physical causality to order events

1. If two events occurred at the same process $p_i$ ($i=1, 2, \ldots, N$)?
   - Then they occurred in the order in which $p_i$ observes them
     - This is the order $\rightarrow_i$

2. When a message is sent between processes?
   - The event of sending the message occurred before the event of receiving the message

The $\rightarrow$ relation

- Lamport called the **partial ordering** obtained by generalizing the previous 2 relationships
  - The *happened-before* or *happens-before* relation

- Sometimes also known as the relation of *causal ordering* or *potential causal ordering*
Lamport’s logical clocks

- The **happens-before** relation
  - $a$ and $b$ are events in the process; and $a$ occurs before $b$
    - Then $a \Rightarrow b$ is true
  - $a$ is event of message sent by one process;
    $b$ is event of message being received in another process
    - Then $a \Rightarrow b$ is true

Some more things about the happens-before relation

- If $a \Rightarrow b$ and $b \Rightarrow c$; then $a \Rightarrow c$
  - **Transitive**

- If events $x$ and $y$ occur in processes that do not exchange messages,
  then ...
  - $x \Rightarrow y$ is not true
  - But, neither is $y \Rightarrow x$
  - These events are said to be **concurrent**
Events occurring at three processes

- $a \rightarrow b$ and $c \rightarrow d$
  - These occur within the same process
- $b \rightarrow c$ and $d \rightarrow f$
  - Events that correspond to sending and receiving messages
- We can use transitivity to say $a \rightarrow f$
- No relationship between $a$ and $e$; these are concurrent $a \parallel e$

If the $\rightarrow$ relation holds between two processes

- The first event might or might-not have caused the second
  - The $\rightarrow$ relation only captures potential causality
    - i.e. two events can be related by $\rightarrow$ without a real connection between them
- EXAMPLE 1: If the server receives a request and sends a response?
  - Then reply is caused by the request
- EXAMPLE 2: A process might receive a request and subsequently issue another message
  - But this could be one that it issues every 5 minutes anyway
A simple example of Lamport timestamps

![Diagram showing Lamport timestamps]

An example of Lamport’s algorithm:

![Diagram showing Lamport's algorithm]

Each message carries the sending time according to the sender’s clock.

Each clock runs at a constant (but different rate).
An example of Lamport’s algorithm:

Each clock runs at a constant (but different rate)

Implementing Lamport’s clocks

1. Before executing an event; \( P_i \) executes
   \[ C_i = C_i + 1 \]

2. When \( P_i \) sends a message \( m \) to \( P_j \); it sets \( m \)’s timestamp \( ts(m) \) to \( C_i \) in previous step

3. Upon receipt of message \( m \), \( P_j \) adjusts its own local counter
   \[ C_j = \max \{ C_j, ts(m) \} \]
   do step 1 and deliver message
The positioning of Lamport’s clocks in distributed systems

An application of Lamport’s clock:
User has $1000 in bank account initially

Add $100 to account
San Francisco

Update with 1% interest
New York

Add $100 ... Total: $1100
Give 1% interest on total: $11
Balance: $1111

Add $100
Balance: $1110

Add $100 ... Total: $1010
Give 1% interest ...
Balance: $1110

COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT
LOGICAL CLOCKS
L34.17

COLORADO STATE UNIVERSITY
Professor: SHRIDEEP PALICKARA
COMPUTER SCIENCE DEPARTMENT
LOGICAL CLOCKS
L34.18
There is a difference when the orders are reversed

- Our objective for now is consistency
- Both copies must be exactly the same

Use Lamport’s clock to order messages

- Process puts received messages into local queue
  - Ordered according to the message’s timestamp
- Message can be delivered only if it is acknowledged by all the other processes
- If a message is at the head of the queue, and acknowledged by all processes
  - It is delivered and processed
Lamport’s Clocks order events based on the happened-before relationship

- If \( a \) happened before \( b \), then \( C(a) < C(b) \)
- But nothing can be said about two events \( a \) and \( b \) by merely comparing their values
- \( C(a) < C(b) \)?
  - Does not mean \( a \) happened before \( b \)

Let’s look a little closer

- \( T_{snd}(m_i) \): Time \( m_i \) was sent
- \( T_{rcv}(m_i) \): Time \( m_i \) was received
- \( T_{snd}(m_i) < T_{rcv}(m_j) \)
- BUT
  - \( T_{snd}(m_i) < T_{rcv}(m_j) \) ?
  - NO
Concurrent message transmissions

Sending $m_3$ MAY HAVE depended on $m_1$

$T_{\text{rcv}}(m_1) < T_{\text{snd}}(m_2)$

But sending of $m_2$ has nothing to do with receipt of $m_1$

Lamport clocks do not capture causality

Vector Clocks
Lamport’s Clocks order events based on the happened-before relationship

- If \( a \) happened before \( b \), then \( C(a) < C(b) \)
- But nothing can be said about two events \( a \) and \( b \) by merely comparing their values
- \( C(a) < C(b) ? \)
  - Does not mean \( a \) happened before \( b \)

Let’s look a little closer

- \( T_{snd}(m_i) : \) Time \( m_i \) was sent
- \( T_{rcv}(m_i) : \) Time \( m_i \) was received
- \( T_{snd}(m_i) < T_{rcv}(m_j) \)
- BUT
  - \( T_{snd}(m_i) < T_{rcv}(m_j) ? \)
    - NO
Concurrent message transmissions

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>12</th>
<th>18</th>
<th>24</th>
<th>30</th>
<th>36</th>
<th>42</th>
<th>48</th>
<th>70</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>61</td>
<td>69</td>
<td>77</td>
<td>85</td>
</tr>
<tr>
<td>m2</td>
<td>0</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
<td>80</td>
<td>90</td>
<td>100</td>
</tr>
</tbody>
</table>

Sending m3 MAY HAVE depended on m1

But sending of m2 has nothing to do with receipt of m1

\(T_{rcv}(m1) < T_{snd}(m2)\)

Lamport clocks do not capture causality

Vector clocks

- Developed by Mattern [1989] and Fidge [1991] to **overcome shortcomings** of Lamport’s clocks
  - i.e., if \(C(a) < C(b)\) then we cannot conclude \(a \rightarrow b\)

- A vector clock for a system of \(N\) processes is an **array** of \(N\) integers

- Each process keeps its own vector clock \(VC_i\)
  - Process uses it vector clock to timestamp messages
Causal precedence can be captured by Vector clocks

- Event $a$ is known to causally precede event $b$ iff $\text{VC}(a) < \text{VC}(b)$
  - $\text{VC}(a) < \text{VC}(b)$ iff $\text{VC}(a)[k] \leq \text{VC}(b)[k]$ for all $k$ and at least one of those relationships is strictly smaller

- Each process $P_i$ maintains a vector $\text{VC}_i$
- $\text{VC}_i[i]$ is number of events so far at $P_i$
- If $\text{VC}_i[j] = k$
  - $P_i$ knows $k$ events occurred at $P_j$
  - $P_i$'s knowledge of local time at $P_j$

Vectors are piggybacked along with any messages that are sent

1. Before executing an event (sending, delivering, or internal) $P_i$ executes
   - $\text{VC}_i[i] = \text{VC}_i[i] + 1$
2. When $P_i$ sends a message $m$ to $P_j$
   - Set $m$'s timestamp $ts(m)$ to $\text{VC}_i$ after doing (1)
3. After receiving $m$, process $P_j$ adjusts its vector
   - $\text{VC}_j[k] = \max\{\text{VC}_j[k], \; ts(m)[k]\}$ for each $k$
   - Execute step (1) and deliver
Vector clocks example 1

\[
\begin{array}{c}
A \\
[1,0,0] \quad [2,0,0] \\
B \\
[2,1,0] \\
C \\
[0,0,1] \quad [2,2,2]
\end{array}
\]

Vector clocks example 2

\[
\begin{array}{c}
A \\
[1,0,0] \quad [5,4,0] \quad [7,4,4] \\
B \\
[1,2,0] \quad [1,4,0] \\
C \\
[1,3,3] \quad [1,3,4]
\end{array}
\]
Vector timestamps allow us to determine causality and concurrency

- Event $a$ happened before event $b$ iff
  - $ts(a) \leq ts(b)$ for each process $i$
  - And one of those relationships is strictly smaller

- If this is not true
  - Events $a$ and $b$ are concurrent

Vector Clocks: Other aspects

- If event $a$ has timestamp, $ts(a)$:
  - $ts(a)[i] - 1$
    - Denotes number of events at $P_i$ that precede $a$

- When $P_j$ receives message $m$ from $P_i$ with timestamp $ts(m) = VC_i$
  - $P_j$ knows about the number of events at $P_i$ that causally preceded $m$
  - Also, $P_j$ knows about how many events at other processes have preceded the sending of $m$, and on which $m$ may causally depend
Vector clocks: Disadvantages

- Storage and message payload is proportional to $N$, the number of processes
- It’s been shown ([Charron-Bost 1991]) that if we are to tell if two events are concurrent by inspecting timestamps?
  - The dimension of $N$ is unavoidable

Using Vector Clocks for Causally Ordered Multicasting
Contrasting totally-ordered and causally-ordered multicasting

- Causally-ordered multicasting is **weaker than** totally-ordered multicasting
- If two messages are **not in any way related** to each other?
  - We **do not care about the order** in which they are delivered to applications
  - Could be delivered in **different order at different applications**

Using Vector Clocks for causally-ordered **multicasting**

- Clocks are **ONLY adjusted when sending and receiving** messages
- Upon **sending** a message, process \( P_i \) will only increment \( VC_i[i] \) by 1
- When \( P_i \) **delivers** a message \( m \) with timestamp \( ts(m) \) it adjusts \( VC_i[k] \)
  - To \( \max(VC_i[k], ts(m)[k]) \) for each \( k \)
When process $P_j$ receives a message $m$ from $P_i$

- Delivery of the message $m$ to the application layer is delayed until 2 conditions are met:
  1. $ts(m)[i] = VC_j[i] + 1$
     - This means $m$ is the next message that $P_j$ was expecting from $P_i$
  2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$
     - This means that $P_j$ has seen all messages that have been seen by $P_i$ when it receives $m$

An example showing enforcement of causal communications

[Diagram showing causal communications with nodes A, B, and C, and messages m and m*.]

Delivery of $m^*$ is delayed until $m$ is delivered

[Errata fixed on this slide.]
Matrix clocks

- Generalizes the notion of vector clocks
- Processes keep estimates of other processes' vector time [Raynal & Singhal, 1996]
- Essentially, a vector of vector clocks for each of the communicating processes

The contents of this slide-set are based on the following references

- http://en.wikipedia.org/wiki/Matrix_clocks