Lecture 8: Logical Time

Why is time important?

- In real life, 
  - Just because 😊
- In distributed systems, 
  - Fairness in mutual exclusion 
  - Avoiding deadlocks 
  - Tracking dependence in case of failure 
  - Consistency of replicated data 
  - Knowledge about progress for garbage collection 
  - Concurrency measurements
Example

- Consider a bank system running on a distributed system.
  - The data is replicated on multiple servers.
- Some account initially has $200.
  - A deposit of $1000 is made to the account, and server A handles the transaction.
    - The deposit is time-stamped with A's local time of 9:22pm.
    - All data is synchronized at all servers.
- A withdrawal of $500 is then made, and server B handles the transaction.
  - The withdrawal is time-stamped with B's local time of 9:15pm.
- Server C queries the transaction logs of both servers.
  - What will happen?

We need to synchronize time

Here are some clock synchronization algorithm(s)
Christian’s algorithm

- Christian [1989] suggested using a time server that helps different machines (processes) adjust their clocks.

Problems?

- Delays in processing the messages and buffer delays and other network stuff.
- P can’t set the clock to t blindly
Problems?

- Delays in the actual message transfer from the network
- P can’t set the clock to \( t \) blindly

\[
P \text{ sets clock to } t + \frac{\text{RTT} + \text{min}_2 - \text{min}_1}{2}
\]

Problems?

- P sets clock to \( t + \frac{\text{RTT} + \text{min}_2 - \text{min}_1}{2} \)
All problems fixed?

- No!
  - We still have a single point of failure.
  - We still might have errors in computing the clock value \( t \).

- What can we do?
  - Distribute the responsibility of the time server.
  - Berkeley algorithm
  - Design an architecture for the time protocol, which would statistically minimize clock errors.
  - Network Time Protocol (NTP)

But do we really need to synchronize clocks?

We still have a chance of error, when working with time
Consider a bank system running on a distributed system.

The data is replicated on multiple servers.

Some account initially has $200.

A deposit of $1000 is made to the account.
- The deposit is time-stamped with A's local time of 9:22pm.
- All data is synchronized at all servers.

A withdrawal of $500 is then made, and server B handles the transaction.
- The withdrawal is time-stamped with B's local time of 9:15pm.

Server C queries the transaction logs of both servers.

What will happen?

Do we have to worry about the time stamp?

What is it that we really care about?

It's all about the sequence of events!
Definitions:
- Multiple processes running on independent machines.
- Events occurring at a process could be a step, send, or receive.

Ordering assumptions:
- Events occurring at the same process have a natural ordering.
  - The process can keep track of that using local clock.
- When a message is sent, the send event then the receive event, then the message is delivered to the other process.

Ordering of events

Potential causal ordering

- Or what we call the happened-before relationship (\(\rightarrow\)).

  - HB1: within a single process if \(e\) happened before \(e'\),
    - \(e \rightarrow e'\)
  - HB2: for any message \(m\),
    - \(send(m) \rightarrow receive(m)\)
  - HB3: if \(e \rightarrow e'\) and \(e' \rightarrow e''\),
    - \(e \rightarrow e''\)
  - HB4: if \(e (\rightarrow) e'\) and \(e' (\rightarrow) e\),
    - \(e \parallel e'\)
    - Which means that they are concurrent
To design a logical clock algorithm, you need to define:
- Data structures used locally by a process to define the logical time
- Protocol used among processes to update these data structures

The protocol must specify:
- R1: How a process adjusts its logical time when a local event occurs
  - Times are piggybacked with messages sent.
- R2: How a process adjusts its logical time when a message is received
Lamport’s algorithm (Scalar time)

- Data structures:
  - Each process \(i\) keeps track of a single non-negative integer \(C_i\)

- Protocol:
  - R1: before executing an event locally \(C_i = C_i + d\)
  - R2: when message is received with a time value of \(C_{msg}\)
    - \(C_i = \max(C_i, C_{msg})\)
    - Execute R1
    - Deliver the message

Example

- Group exercise time!
Initial counters (clocks):

- P1: 0
- P2: 0
- P3: 0

Lamport Timestamps:

- P1
  - Instruction or step
  - Message send: $ts = 1$

- P2
  - Message carries: $ts = 1$

- P3
  - $ts = 1$
  - Message send
Lamport Timestamps

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Message carries

\[ ts = 1 \]

\[ ts = \max(local, \text{msg}) + 1 \]

\[ = \max(0, 1) + 1 \]

\[ = 2 \]

\[ \text{ts} = 1 \]

Instruction or step

Message
Lamport Timestamps

max(3, 4) + 1 = 5

Instruction or step
Message
Obeying Causality

- A \rightarrow B :: 1 < 2
- B \rightarrow G :: 2 < 3
- A \rightarrow G :: 1 < 3

- ? C \rightarrow G ? :: 3 = 3
- ? J \rightarrow C ? :: 1 < 3
- (C, G) and (J, C) are pairs of concurrent events
Properties of scalar time

- Consistency
- Total ordering
- Event counting
- But...
  - No strong consistency

Vector time

- Data structures:
  - Each process (i) keeps track of a vector of n non-negative integers \( v_t[i] \)

- Protocol:
  - R1: before executing an event locally \( v_t[i] = v_t[i] + d \)
  - R2: when message is received with a time value of \( v_t \)
    - For all values in the vector \( v_t[k] = \max(v_t[k], v_t[k]) \)
    - Execute R1
    - Deliver the message
Example

- Group exercise!

Vector Timestamps

- Initial counters (clocks)
Vector Timestamps

P1
(0,0,0) (1,0,0)

P2
(0,0,0)

P3
(0,0,0) Message(0,0,1)
(0,0,1)

P1
(0,0,0) (1,0,0)

P2
(0,0,0) (0,1,1)

P3
(0,0,0) Message(0,0,1)
(0,0,1)
Vector Timestamps

P1 (0,0,0) (1,0,0) (2,0,0) 
Message (2,0,0) 
(0,0,1) (2,2,1) 
(0,0,0) (0,0,1) 

P2 (0,0,0) (0,1,1) (2,2,1) (2,3,1) 
(0,0,0) (0,0,1) (0,0,2) (5,3,3) 

P3 (0,0,0) (2,0,0) (3,0,0) (4,3,1) (5,3,1) 
Time