Distributed Systems

CS425/ECE428

Jan 30 2023

Instructor: Radhika Mittal
Logistics Related

• VM clusters have been assigned
  • for those who filled out the forms and are enrolled in / planning to switch to 4 credits.
  • you might have received (or will receive by tonight) some instructions over email.

• HW1 will be released on Wednesday.

• Late registration can be permitted if there is capacity in the course. Please talk to CS/ECE advising office.
Quick Recap: Clock Synchronization

- Synchronization in synchronous systems:
  - Synchronization bound = \((\text{max} - \text{min})/2\)

- Synchronization in asynchronous systems:
  - Cristian Algorithm: Synchronization between a client and a server.
    - Synchronization bound = \((T_{\text{round}} / 2) - \text{min} \leq T_{\text{round}} / 2\)
  - Berkeley Algorithm: internal synchronization between clocks.
    - A central server picks the average time and disseminates offsets.
  - Network Time Protocol
    - Hierarchical time synchronization over the Internet.
    - Symmetric mode synchronization.
Today’s agenda

• Logical Clocks and Timestamps
  • Chapter 14.4

• Global State (if time)
  • Chapter 14.5
Event Ordering

• A usecase of synchronized clocks:
  • Reasoning about order of events.

• Why is it useful?
  • Reconciling updates made to an object in a distributed datastore.
  • Rollback recovery during failures:
    1. Checkpoint state of the system; 2. Log events (with timestamps);
    3. Rollback to checkpoint and replay events in order if system crashes.

• …

• Can we reason about order of events without synchronized clocks?
Process, state, events

• Consider a system with \( n \) processes: \(<p_1, p_2, p_3, \ldots, p_n>\)

• Each process \( p_i \) is described by its state \( s_i \) that gets transformed over time.
  • State includes values of all local variables, affected files, etc.

• \( s_i \) gets transformed when an event occurs.

• Three types of events:
  • Local computation.
  • Sending a message.
  • Receiving a message.
Event Ordering

• Easy to order events within a single process $p_i$, based on their time of occurrence.

• How do we reason about events across processes?
  • A message must be sent before it gets received at another process.

• These two notions help define happened-before (HB) relationship denoted by $\rightarrow$.
  • $e \rightarrow e'$ means $e$ happened before $e'$. 
Happened-Before Relationship

• Happened-before (HB) relationship denoted by $\rightarrow$.
  • $e \rightarrow e'$ means $e$ happened before $e'$.
  • $e \rightarrow_i e'$ means $e$ happened before $e'$, as observed by $p_i$.

• HB rules:
  • If $\exists p_i, e \rightarrow_i e'$ then $e \rightarrow e'$.
  • For any message $m$, $send(m) \rightarrow receive(m)$
  • If $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

• Also called “causal” or “potentially causal” ordering.
Event Ordering: Example

Which event happened first?

- \( a \rightarrow b \) and \( b \rightarrow c \) and \( c \rightarrow d \) and \( d \rightarrow f \)
- \( a \rightarrow b \) and \( a \rightarrow c \) and \( a \rightarrow d \) and \( a \rightarrow f \)
Event Ordering: Example

What can we say about $e$?

- $e \rightarrow f$
- $a \leftrightarrow e$ and $e \leftrightarrow a$
- $a \parallel e$
- $a$ and $e$ are concurrent.
Event Ordering: Example

What can we say about e and d?

\[ e \parallel d \]
Logical Timestamps: Example

What can we say about $e$ and $d$?

$e \rightarrow d$
Lamport’s Logical Clock

• Logical timestamp for each event that captures the happened-before relationship.

• Algorithm: Each process $p_i$
  1. initializes local clock $L_i = 0$.
  2. increments $L_i$ before timestamping each event.
  3. piggybacks $L_i$ when sending a message.
  4. upon receiving a message with clock value $t$
     • sets $L_i = \max(t, L_i)$
     • increments $L_i$ before timestamping the receive event (as per step 2).
Logical Timestamps: Example

Physical time

- $p_1$: 0, 1, 2
  - a, b, $m_1$ (2)
- $p_2$: 0, 4
  - (2 > 0), c, d, $m_2$ (4)
- $p_3$: 0, 1, e, f, 5
  - (4 > 1)
Lamport’s Logical Clock

• Logical timestamp for each event that captures the *happened-before* relationship.

• *Algorithm*: Each process $p_i$
  1. initializes local clock $L_i = 0$.
  2. increments $L_i$ before timestamping each event.
  3. piggybacks $L_i$ when sending a message.
  4. upon receiving a message with clock value $t$
     • sets $L_i = \text{max}(t, L_i)$
     • increments $L_i$ before timestamping the receive event (as per step 2).
Logical Timestamps: Example

Physical time

p1

p2

p3

e 2 g

f 6

0 1 2

0 0

0 0

m1 (2)

m2 (5)

Logical Timestamps: Example
Lamport’s Logical Clock

• Logical timestamp for each event that captures the happened-before relationship.

• If \( e \rightarrow e' \) then
  • \( L(e) < L(e') \)

• What if \( L(e) < L(e') \)?
  • We cannot say that \( e \rightarrow e' \)
  • We can say: \( e' \not\rightarrow e \)
  • Either \( e \rightarrow e' \) or \( e \parallel e' \)
Logical Timestamps: Example

\[ L(e) < L(d), \ e \parallel d \]

\[ L(e) < L(f), \ e \rightarrow f \]
Vector Clocks

- Each event associated with a vector timestamp.
- Each process $p_i$ maintains vector of clocks $V_i$
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process $p_j$ as maintained by $p_i$
- Algorithm: each process $p_i$: 
Vector Clocks

- Each event associated with a vector timestamp.
- Each process $p_i$ maintains vector of clocks $V_i$.
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process $p_j$ as maintained by $p_i$.
- Algorithm: each process $p_i$:
  1. initializes local clock $V_i[j] = 0$.
Vector Clocks

• Each event associated with a vector timestamp.
• Each process $p_i$ maintains vector of clocks $V_i$
• The size of this vector is the same as the no. of processes.
  • $V_i[j]$ is the clock for process $p_j$ as maintained by $p_i$
• Algorithm: each process $p_i$:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
Vector Clocks

• Each event associated with a vector timestamp.

• Each process $p_i$ maintains vector of clocks $V_i$

• The size of this vector is the same as the no. of processes.
  • $V_i[j]$ is the clock for process $p_j$ as maintained by $p_i$

• Algorithm: each process $p_i$:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks $V_i$ when sending a message.
Vector Clocks

- Each event associated with a vector timestamp.
- Each process $p_i$ maintains vector of clocks $V_i$.
- The size of this vector is the same as the no. of processes.
  - $V_i[j]$ is the clock for process $p_j$ as maintained by $p_i$.
- Algorithm: each process $p_i$:
  1. initializes local clock $V_i[j] = 0$
  2. increments $V_i[i]$ before timestamping each event.
  3. piggybacks $V_i$ when sending a message.
  4. upon receiving a message with vector clock value $v$
     - sets $V_i[j] = \max(V_i[j], v[j])$ for all $j = 1 \ldots n$.
     - increments $V_i[i]$ before timestamping receive event (as per step 2).
Vector Timestamps: Example

[0,0,0] [1,0,0] [2,0,0]

p1 a b m1 ([2,0,0])

[0,0,0]

p2

c [2,1,0] d [2,2,0]

m2 ([2,2,0])

[0,0,0] [0,0,1]

p3 e

f [2,2,2]

Physical time
Vector Timestamps: Example

[p₁, 0, 0, 0] [1,0,0] [2,0,0]  
[0,0,0] a b m₁ ([2,0,0]) c d [2,1,0]  
[0,0,0] e g [0,0,2] f [2,3,3]  
[0,0,0,0] [2,2,2] [2,3,2]  
[0,0,0] [2,0,1] [0,0,2]  
[0,0,0] [2,0,1] [0,0,2]  

Physical time
Comparing Vector Timestamps

• Let \( V(e) = V \) and \( V(e') = V' \)

• \( V = V' \), iff \( V[i] = V'[i] \), for all \( i = 1, \ldots, n \)

• \( V \leq V' \), iff \( V[i] \leq V'[i] \), for all \( i = 1, \ldots, n \)

• \( V < V' \), iff \( V \leq V' \) \& \( V \neq V' \)
  \[ \text{iff } V \leq V' \text{ and } \exists j \text{ such that } (V[j] < V'[j]) \]

• \( e \rightarrow e' \) iff \( V < V' \)
  
  • \( (V < V' \text{ implies } e \rightarrow e') \) and \( (e \rightarrow e' \text{ implies } V < V') \)

• \( e \parallel e' \) iff \( V \nless V' \) and \( V' \nless V \)
What can we say about e & f based on their vector timestamps?
Vector Timestamps: Example

V(e) < V(f), e → f
What can we say about e & d based on their vector timestamps?
Vector Timestamps: Example

\[
\begin{align*}
&[0,0,0] \quad [1,0,0] \quad [2,0,0] \\
&\text{p_1} \quad \downarrow \quad \downarrow \quad \downarrow \\
&\text{a} \quad \text{b} \quad \text{m}_1 \quad ([2,0,0]) \\
&\text{p}_2 \quad \quad \quad \downarrow \quad \downarrow \\
&[0,0,0] \quad \text{c} [2,1,0] \quad \text{d} [2,2,0] \\
&\text{p}_3 \quad \quad \quad \quad \downarrow \\
&[0,0,0] \quad [0,0,1] \quad \downarrow \\
&\text{e} \quad \quad \quad \quad \text{f} [2,2,2] \\
\end{align*}
\]

\[V(e) \not\preceq V(d) \text{ and } V(d) \not\preceq V(e), \ e \parallel d\]
Vector Timestamps: Example

How about now?
Vector Timestamps: Example

- $V(e) < V(f)$, $e \rightarrow f$
- $V(e) < V(d)$, $e \rightarrow d$
Timestamps Summary

• Comparing timestamps across events is useful.
  • Reconciling updates made to an object in a distributed datastore.
  • Rollback recovery during failures:
    1. Checkpoint state of the system; 2. Log events (with timestamps);
    3. Rollback to checkpoint and replay events in order if system crashes.

• How to compare timestamps across different processes?
  • **Physical timestamp:** requires clock synchronization.
    • Google's Spanner Distributed Database uses “TrueTime”.
  • **Lamport's timestamps:** cannot fully differentiate between causal
    and concurrent ordering of events.
    • Oracle uses “System Change Numbers” based on Lamport's clock.
  • **Vector timestamps:** larger message sizes.
    • Amazon’s DynamoDB uses vector clocks.
Timestamps Summary

• Comparing timestamps across events is useful.
  • Reconciling updates made to an object in a distributed datastore.
  • Rollback recovery during failures:
    1. Checkpoint state of the system; 2. Log events (with timestamps);
    3. Rollback to checkpoint and replay events in order if system crashes.

• How to compare timestamps across different processes?
  • Physical timestamp: requires clock synchronization.
    • Google's Spanner Distributed Database uses “TrueTime”.
  • Lamport’s timestamps: cannot fully differentiate between causal and concurrent ordering of events.
    • Oracle uses “System Change Numbers” based on Lamport's clock.
  • Vector timestamps: larger message sizes.
    • Amazon's DynamoDB uses vector clocks.
Today’s agenda

• Logical Clocks and Timestamps
  • Chapter 14.4

• Global State
  • Chapter 14.5
Process, state, events

- Consider a system with \( n \) processes: \(<p_1, p_2, p_3, \ldots, p_n>\).

- Each process \( p_i \) is associated with state \( s_i \).
  - State includes values of all local variables, affected files, etc.

- Each channel can also be associated with a state.
  - Which messages are currently \textit{pending} on the channel.
  - Can be computed from process’ state:
    - Record when a process sends and receives messages.
    - If \( p_i \) sends a message that \( p_j \) has not yet received, it is pending on the channel.

- State of a process (or a channel) gets transformed when an \textit{event} occurs. 3 types of events:
  - Local computation, sending a message, receiving a message.
Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:

Two processes: $p_1$ and $p_2$.

$c_{12}$: channel from $p_1$ to $p_2$.

$c_{21}$: channel from $p_2$ to $p_1$. 
Global State (or Global Snapshot)

• State of each process (and each channel) in the system at a given instant of time.

• Example:

Process state for $p_1$ and $p_2$.
No pending messages on the channels.
Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:

  \[ p_1 \text{ sends a message to } p_2 \text{ asking it to set } X_2 = 4 \]

  \[ X_1 : 0 \]
  \[ Y_1 : 0 \]
  \[ Z_1 : 0 \]

  \[ c_{12} : [X_2 = 4] \]

  \[ X_2 : 1 \]
  \[ Y_2 : 2 \]
  \[ Z_2 : 3 \]

  \[ c_{21} : [\text{empty}] \]

  Global Snapshot #2
Global State (or Global Snapshot)

- State of each process (and each channel) in the system at a given instant of time.

- Example:

  \[ P_1 \]
  \[
  \begin{array}{c}
  X_1: 0 \\
  Y_1: 0 \\
  Z_1: 0 \\
  \end{array}
  \]

  \[ P_2 \]
  \[
  \begin{array}{c}
  X_2: 4 \\
  Y_2: 2 \\
  Z_2: 3 \\
  \end{array}
  \]

  \( c_{12}: [\text{empty}] \)
  \( c_{21}: [\text{empty}] \)

  \( P_2 \) receives the message.
Global State (or Global Snapshot)

• State of each process (and each channel) in the system at a given instant of time.

• Example:

\[ p_1 \]
\[ p_2 \]
\[ c_{12}: \text{[empty]} \]
\[ c_{21}: \text{[empty]} \]

\[ p_2 \text{ changes the value of } X_2 \]
Capturing a global snapshot

• Useful to capture a global snapshot of the system:
  • *Checkpointing* the system state.
  • Reasoning about unreferenced objects (for garbage collection).
  • Distributed debugging.
Capturing a global snapshot

• Difficult to capture a global snapshot of the system.

• Global state or global snapshot is state of each process (and each channel) in the system at a given instant of time.

• Strawman:
  • Each process records its state at 3:15pm.
  • We get the global state of the system at 3:15pm.
  • But precise clock synchronization is difficult to achieve.

• How do we capture global snapshots without precise time synchronization across processes?
  • Our agenda for next class!