Distributed Systems

CS425/ECE428

Instructor: Radhika Mittal
Logistics Related

- Eng-IT is still working on assigning VM clusters
  - Will hopefully be done by the end of the day.
  - Watch for an email from us, and CampusWire post with instructions.

- All registered students have been added to Gradescope.
  - If you have registered late / plan on registering when a slot opens up, you can email Sarthak (sm106) to get added to Gradescope.
  - But please wait a week before doing so.
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?
  • Debugging distributed applications
  • 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 $m$, $\text{send}(m) \rightarrow \text{receive}(m)$
  - 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 \( e \)?

\[ e \rightarrow f \]

\[ a \leftrightarrow e \text{ and } e \leftrightarrow a \]

\[ a \parallel e \]

\( a \) 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

Logical timestamps:
- p1: a (0), b (1), m1 (2)
- p2: c (2 > 0), d (4)
- p3: e (0), f (5)

Physical timestamps:
- (2 > 0)
- (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 $L_i = \max(t, L_i)$
   • increments $L_i$ before timestamping the receive event (as per step 2).
Logical Timestamps: Example

Physical time

- $p_1$: a -> b -> m$_1$ (2)
- $p_2$: 0, 1, 2, 3, 4, 5, 6
- $p_3$: e -> g

- m$_2$ (5)
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 ⪯ d  
L(e) < L(f), e → 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

Physical time

p_1 [0,0,0] [1,0,0] [2,0,0]
a b m_1 ([2,0,0])

p_2 [0,0,0] [2,1,0] [2,2,0]
c d m_2 ([2,2,0])

p_3 [0,0,0] [0,0,1] e

f [2,2,2]
Vector Timestamps: Example

Physical time

[p_1] [p_2] [p_3]

[a] [b] [c]

[m_1] ([2,0,0])

[d] [e] [f] [g]

([0,0,2])

([[2,3,2]])
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 $V \leq V' \& V \neq V'$
  
  iff $V \leq V' \& \exists$ j such that $(V[j] < V'[j])$

- $e \rightarrow e'$ iff $V < V'$
  
  -(V < V' implies $e \rightarrow e'$) and $(e \rightarrow e'$ implies $V < V'$)

- $e \parallel e'$ iff $(V \not< V' \& V' \not< V)$
What can we say about e & f based on their vector timestamps?
Vector Timestamps: Example

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

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

How about now?
Vector Timestamps: Example

\[
\begin{align*}
V(e) &< V(f), \ e \rightarrow f \\
V(e) &< V(d), \ e \rightarrow d
\end{align*}
\]
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: \( \langle p_1, p_2, p_3, \ldots, p_n \rangle \).
- 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 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 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:

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

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

  **event 1**: \( p_1 \) sends a message to \( p_2 \) asking it to set \( X_2 = 4 \)
Global State (or Global Snapshot)

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

• Example:

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

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

\[ P_1 \]

\[ X_2 = 4 \]
\[ Y_2 : 2 \]
\[ Z_2 : 3 \]

\[ X_2 : 1 \]

\[ P_2 \]

\text{event 2: } p_2 \text{ 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:

  \[ \begin{align*}
  &c_{12}: \text{[empty]} \\
  &c_{21}: \text{[empty]}
  \end{align*} \]

  \[ \begin{array}{c|c|c|}
  \hline
  & P_1 & P_2 \\
  \hline
  X_1 & 0 & X_2: 4 \\
  Y_1 & 0 & Y_2: 2 \\
  Z_1 & 0 & Z_2: 3 \\
  \hline
  \end{array} \]

  **event 3**: \( p_2 \) 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?
Some more notations and definitions

- State of a process (or a channel) gets transformed when an event occurs.

- 3 types of events:
  - local computation, sending a message, receiving a message.

- $e_i^n$ is the $n^{th}$ event at $p_i$. 
Some more notations and definitions

- For a process $p_i$, where events $e_i^0, e_i^1, \ldots$ occur:
  \[
  \text{history}(p_i) = h_i = \langle e_i^0, e_i^1, \ldots \rangle \\
  \text{prefix history}(p_i^k) = h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle \\
  s_i^k: p_i$’s state immediately after $k^{th}$ event.
  \]
- For a set of processes $\langle p_1, p_2, p_3, \ldots, p_n \rangle$:
  \[
  \text{global history}: H = \bigcup_i (h_i) \\
  \text{global state}: S = \bigcup_i (s_i)
  \]
Some more notations and definitions

- For a process $p_i$, where events $e_i^0, e_i^1, \ldots$ occur:
  \[
  \text{history}(p_i) = h_i = <e_i^0, e_i^1, \ldots >
  \]
  \[
  \text{prefix history}(p_i^k) = h_i^k = <e_i^0, e_i^1, \ldots, e_i^k >
  \]
  \[
  s_i^k : p_i$'s state immediately after $k^{th}$ event.
  \]

- For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:
  \[
  \text{global history}: H = \bigcup_i (h_i)
  \]
  \[
  \text{global state}: S = \bigcup_i (s_i)
  \]
  
  *But state at what time instant?*
Some more notations and definitions

• For a process $p_i$, where events $e_i^0, e_i^1, \ldots$ occur:
  \[
  \text{history}(p_i) = h_i = <e_i^0, e_i^1, \ldots >
  \]
  \[
  \text{prefix history}(p_i^k) = h_i^k = <e_i^0, e_i^1, \ldots, e_i^k >
  \]
  \[
  s_i^k : p_i's \text{ state immediately after } k^{th} \text{ event.}
  \]

• For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:
  \[
  \text{global history: } H = \bigcup_i (h_i)
  \]
  \[
  \text{global state: } S = \bigcup_i (s_i^{c_i})
  \]
  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$
  the frontier of $C = \{e_i^{c_i}, i = 1, 2, \ldots, n\}$
  global state $S$ that corresponds to cut $C = \bigcup_i (s_i^{c_i})$
Example: Cut

\[
C_A: \langle e_1^0, e_2^0 \rangle
\]
Frontier of \(C_A\):

\[
C_B: \langle e_1^0, e_1^1, e_1^2, e_2^0, e_2^1, e_2^2 \rangle
\]
Frontier of \(C_B\):
Some more notations and definitions

• For a process $p_i$, where events $e_i^0, e_i^1, \ldots$ occur:
  
  $\text{history}(p_i) = h_i = <e_i^0, e_i^1, \ldots>$
  
  $\text{prefix history}(p_i^k) = h_i^k = <e_i^0, e_i^1, \ldots, e_i^k>$
  
  $s_i^k: p_i$’s state immediately after $k^{th}$ event.

• For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:
  
  $\text{global history}: H = \bigcup_i (h_i)$
  
  a cut $C \subseteq H = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$
  
  the frontier of $C = \{e_i^{c_i}, i = 1,2, \ldots n\}$
  
  $\text{global state} S$ that corresponds to cut $C = \bigcup_i (s_i^{c_i})$
Consistent cuts and snapshots

• A cut $C$ is consistent if and only if
  \[ \forall e \in C \ (\text{if } f \to e \text{ then } f \in C) \]
Example: Cut

\[ C_A : < e_1^0, e_2^0 > \]
Frontier of \( C_A \): \{e_1^0, e_2^0\}

Inconsistent cut.

\[ C_B : < e_1^0, e_1^1, e_1^2, e_2^0, e_2^1, e_2^2 > \]
Frontier of \( C_B \): \{e_1^2, e_2^2\}

Consistent cut.
Consistent cuts and snapshots

- A cut $C$ is consistent if and only if
  \[ \forall e \in C \ (\text{if } f \rightarrow e \text{ then } f \in C) \]

- A global state $S$ is consistent if and only if it corresponds to a consistent cut.
Consistent cuts and snapshots

- A cut $C$ is **consistent** if and only if
  $$\forall e \in C \ (\text{if } f \rightarrow e \text{ then } f \in C)$$

- A global state $S$ is consistent if and only if it corresponds to a consistent cut.
How to capture global state?

• State of each process (and each channel) in the system at a given instant of time.
  • Difficult to capture -- requires precisely synchronized time.

• Relax the problem: find a consistent global state.
  • For a system with n processes \( <p_1, p_2, p_3, \ldots, p_n> \), capture the state of the system after the \( c_i \)th event at process \( p_i \).
    • State corresponding to the cut defined by frontier events \( \{e_i^{c_i}, \text{for } i = 1,2, \ldots n \} \).
  • We want the state to be consistent.
    • Must correspond to a consistent cut.

How to find consistent global state?
Chandy-Lamport Algorithm

• Goal:
  • Record a global snapshot
    • Process state (and channel state) for a set of processes.
    • The recorded global state is consistent.

• Identifies a consistent cut.

• Records corresponding state locally at each process.
Chandy-Lamport Algorithm

- **System model and assumptions:**
  - System of $n$ processes: $<p_1, p_2, p_3, \ldots, p_n>$.
  - There are two uni-directional communication channels between each ordered process pair: $p_j$ to $p_i$ and $p_i$ to $p_j$.
  - Communication channels are FIFO-ordered (first in first out).
  - All messages arrive intact, and are not duplicated.
  - No failures: neither channel nor processes fail.

- **Requirements:**
  - Snapshot should not interfere with normal application actions, and it should not require application to stop sending messages.
  - Any process may initiate algorithm.
Chandy-Lamport Algorithm

• To be continued in next class....