Logistics Related

• MP0 is due on Thursday.
  • If you are in the 4 credit section, and still do not have a VM cluster assigned to you, reach out to Sanchit (sv4) asap.
  • We will not give any extensions for this reason.
Recap: 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.
Recap: 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

- Global State
  
  - Chapter 14.5
  
  - Goal: reason about how to capture the state across all processes of a distributed system without requiring time synchronization.
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 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:

  \[
  \begin{align*}
  &X_1: 0 \\
  &Y_1: 0 \\
  &Z_1: 0 \\[empty]\end{align*}
  \]

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

  \[
  \begin{align*}
  &X_2: 1 \\
  &Y_2: 2 \\
  &Z_2: 3 \\[empty]\end{align*}
  \]

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

  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:

  $p_1$ 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:

  \[
  \begin{align*}
  X_1 &: 0 \\
  Y_1 &: 0 \\
  Z_1 &: 0
  \end{align*}
  \]

  \[
  \begin{align*}
  X_2 &: 1 \\
  Y_2 &: 2 \\
  Z_2 &: 3
  \end{align*}
  \]

  \( P_2 \) 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:

  - Process 1:
    - $X_1: 0$
    - $Y_1: 0$
    - $Z_1: 0$

  - Process 2:
    - $X_2: 4$
    - $Y_2: 2$
    - $Z_2: 3$

  - Channel $c_{12}: [\text{empty}]$
  - Channel $c_{21}: [\text{empty}]$

  Process 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).
  - Deadlock detection.
  - 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:
  \[
  \text{history}(p_i) = h_i = <e_i^0, e_i^1, \ldots>
  \]
  \[
  \text{prefix history}(p_{i}^{k}) = h_{i}^{k} = <e_i^0, e_i^1, \ldots,e_i^k>
  \]
  \[
  s_{i}^{k} : p_i's \text{ state immediately after } k^{th} \text{ event.}
  \]

- For a set of processes \(<p_1, p_2, p_3, \ldots., p_n>:\n  \text{global history} : H = \cup_{i} (h_i)\]
Some more notations and definitions

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

- For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:
  \[
  \text{global history} : H = \bigcup_i (h_i)
  \]
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 $C = \{e_i^{c_i}, i = 1,2, \ldots, n\}$
  
  global state $S$ that corresponds to cut $C = \bigcup_i (s_i^{c_i})$
Example: Cut

$C_A: <e_1^0, e_2^0>$

Frontier of $C_A$:

$C_B: <e_1^0, e_1^1, e_1^2, e_2^0, e_2^1, e_2^2>$

Frontier of $C_B$:
Some more notations and definitions

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

• For a set of processes $<p_1, p_2, p_3, \ldots, p_n>$:
  \[
  \text{global history: } H = \bigcup_i (h_i)
  \]
  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 $C = \bigcup_i (s_i^{c_i})$
Consistent cuts and snapshots

- A cut $C$ is **consistent** if and only if

  $$\forall e \in C \ (\text{if } f \rightarrow e \text{ then } f \in C)$$
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.
How to capture global state?

• Ideally: 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 a consistent global state that corresponds to a consistent cut?
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).
  - if $p_i$ sends $m$ before $m'$ to $p_j$, then $p_j$ receives $m$ before $m'$.
- All messages arrive intact, and are not duplicated.
- No failures: neither channel nor processes fail.
Chandy-Lamport Algorithm

• 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 Intuition

• First, initiator $p_i$:
  • records its own state.
  • creates a special marker message.
  • sends the marker to all other process.

• When a process receives a marker.
  • records its own state.
Chandy-Lamport Algorithm Intuition

• First, initiator $p_i$: 
  • records its own state.
  • creates a special marker message.
  • sends the marker to all other processes.

• When a process receives a marker:
  • records its own state.
Chandy-Lamport Algorithm Intuition

Cut frontier: \{e_1^2, e_2^2\}
Chandy-Lamport Algorithm Intuition

• First, initiator \( p_i \):
  • records its own state.
  • creates a special marker message.
  • sends the marker to all other processes.

• When a process receives a marker.
  • records its own state.

\[ \text{This captures the local state at each process.} \]
\[ \text{What about the channel state?} \]
Chandy-Lamport Algorithm Intuition

Cut frontier: \{e_1^2, e_2^2\}
Chandy-Lamport Algorithm Intuition

Cut frontier: \{e_1^2, e_2^2\}
Chandy-Lamport Algorithm Intuition

• First, initiator $p_i$:
  • records its own state.
  • creates a special marker message.
  • sends the marker to all other processes.
  • starts recording messages received on other channels.
    • until a marker is received on a channel.

• When a process receives a marker:
  • If marker is received for the first time.
    • records its own state.
    • sends marker on all other channels.
    • starts recording messages received on other channels.
      • until a marker is received on a channel.
Chandy-Lamport Algorithm

• First, initiator $p_i$:
  - records its own state.
  - creates a special marker message.
  - for $j = 1$ to $n$ except $i$
    - $p_i$ sends a marker message on outgoing channel $c_{ij}$.  
    - starts recording the incoming messages on each of the incoming channels at $p_i : c_{ji}$ (for $j = 1$ to $n$ except $i$).
Chandy-Lamport Algorithm

Whenever a process $p_i$ receives a marker message on an incoming channel $c_{ki}$

• if (this is the first marker $p_i$ is seeing)
  • $p_i$ records its own state first
  • marks the state of channel $c_{ki}$ as “empty”
  • for $j=1$ to $n$ except $i$
    • $p_i$ sends out a marker message on outgoing channel $c_{ij}$
    • starts recording the incoming messages on each of the incoming channels at $p_i: c_{ji}$ (for $j=1$ to $n$ except $i$ and $k$).

• else // already seen a marker message
  • mark the state of channel $c_{ki}$ as all the messages that have arrived on it since recording was turned on for $c_{ki}$
Chandy-Lamport Algorithm

The algorithm terminates when

- All processes have received a marker
  - To record their own state

- All processes have received a marker on all the \((n-1)\) incoming channels
  - To record the state of all channels
Example
Example

$p_1$ is initiator:
- Record local state $s_1$,
- Send out markers
- Start recording on channels $c_{21}, c_{31}$

---

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

---

Time
Example

- First marker!
- Record own state as $s_3$
- Mark $c_{13}$ state as empty
- Start recording on other incoming $c_{23}$
- Send out markers

$s_1$, Record $c_{21}, c_{31}$

Diagram:

- A, B, C, D, E
- P1, P2, P3
- Time
- Example
Example

\[
P1 \quad A \quad B \quad C \quad D \quad E
\]

\[
P2 \quad E \quad F \quad G
\]

\[
P3 \quad H \quad I \quad J
\]

\[s_1, \text{ Record } c_{21}, c_{31}\]

\[s_3, c_{13} = < >, \text{ Record } c_{23}\]
Example

State of channel $c_{31} = < >$

$\text{Duplicate marker!}$

$\text{s}_1, \text{Record } c_{21}, c_{3+}$

$\text{Record } c_{23}$

$\text{s}_3$

$\text{c}_{13} = < >$
Example

- First marker
- Record own state as $s_2$
- Mark $c_{32}$ state as empty
- Turn on recording on $c_{12}$
- Send out markers

$\begin{align*}
    s_1, \text{Record } c_{21}, c_{31} \\
    c_{31} = < >
\end{align*}$
Example

P1

P2

P3

Time

\( s_1, \text{Record } c_{21}, c_{31} \)
\( c_{31} = \langle \rangle \)
\( c_{31} = \langle \rangle \)

\( s_2 \)
\( c_{32} = \langle \rangle \)

\( c_{13} = \langle \rangle \)

Record \( c_{23} \)

Record \( c_{12} \)
Example

\[ s_1, \text{Record } c_{21}, c_{31} \]

\[ c_{31} = <> \]

\[ s_2 \]

\[ c_{32} = <> \]

\[ \text{Record } c_{42} \]

\[ \bullet \text{ Duplicate!} \]

\[ s_3 \]

\[ c_{13} = <> \]

\[ \text{Record } c_{23} \]
Example

\[ s_1, \text{Record } c_{23}, c_{33} \]

- Duplicate!
- \( c_{21} = \text{<message G to D>} \)

\[ s_2 \]
\[ c_{32} = <> \]

\[ s_3 \]
\[ c_{13} = <> \]

Record \( c_{23} \)

\[ c_{12} = <> \]
Example

- $s_1, \text{Record } \epsilon_{24}, \epsilon_{34}$
- $c_{24} = < >$
- $c_{34} = < >$
- $c_{13} = < >$
- $\text{Record } \epsilon_{23}$
- $s_2$
- $c_{32} = < >$
- $\text{Record } \epsilon_{42}$
- $c_{12} = < >$
- $c_{23} = < >$
- Duplicate!
Example

Algorithm has terminated!
Example

Frontier for the resulting cut: \{B, G, H\}

Channel state for the cut:
Only \(c_{21}\) has a pending message.
Global snapshots pieces can be collected at a central location.
Chandy-Lamport Algorithm: Properties

• Any run of the Chandy-Lamport Global Snapshot algorithm creates a consistent cut.

• Homework: why?
Global Snapshot Summary

• The ability to calculate global snapshots in a distributed system is very important.
• But don’t want to interrupt running distributed application.
• Chandy-Lamport algorithm calculates global snapshot.
• Obeys causality (creates a consistent cut).