Global state

To do ...

- Global snapshot and
- applications
How can we determine if a given property holds in a distributed system in execution?
An object is garbage if there are no longer any references to it anywhere in the system

- If garbage, you can reclaim the memory
- To decide, check there are no references

An example

- Process $p_1$ has 2 objects, one with a local, one with a remote ref
- Process $p_2$ has one garbage object (gray) and one with a reference in a message in transit

To decide, need both state of processes and comm. channels
Global state – Deadlock detection

- Deadlock detection

- Distributed termination detection
  - Checking each process, but a message in the channel can get them running again

- Debugging
  - Check the state of a related set of variables held by different processes
Processes’ and channels’ states

- A distributed system – *processes* that communicate by passing *msgs* over *channels*
- Each process can record its own states
  - $LS_i^k$ – state of $p_i$ right before $kth$ event
  - $LS_i^0$ is the initial event
- To capture the state of the channel $SC_{ij}$
  - Process record send/receiving of msgs as part of their state
  - If a msg was sent but not yet received, then it must be in the channel
Global state and consistent global state

- Global state – can be made of any set of states of all processes and channels
  \[ S = \{ U_i \text{ LS}_i, \; U_{ij} \text{ SC}_{ij} \} \]
  - *But which global states are meaningful?*
  - Which processes states could have occurred at the same time?
Global state and consistent global state

- A consistent global state, iff
  - $C_1$: $send(m_{ij}) \in LS_i \rightarrow mij \in SC_i \ XOR \ rec(m_{ij}) \in LS_j$
  - $C_2$: $send(m_{ij}) \notin LS_i \rightarrow mij \notin SC_{ij} \ AND \ rec(m_{ij}) \notin LS_j$

- $C_1$: A msg sent is either in the channel or has been received
- $C_2$: A msg not yet sent cannot be recorded in a channel nor received
Global snapshot with a global physical clock

- Recording a consistent global snapshot with a global clock (or perfectly synchronized clocks)
  - Initiator picks a future time for snapshot, $T$, broadcast it to all
  - Each process take their local snapshot at time $T$
  - Snapshot of $C_{ij}$ includes all messages received by $p_j$ after taken snapshot ($LS_j$) with timestamp $< T$

Take snapshot at 6:25:00PM CST

But you know there’s nothing like it
An (inconsistent) global state

- A system that maintains bank accounts A and B

Initially,
A = $500  B = $200  C_{12} = 0  C_{21} = 0
$700 in the system

X: Record state of A at t1: A: $500

Loc 1 initiate a transfer of $100 from A to B

Loc 2 initiate a transfer of $50 from B to A
An (inconsistent) global state

- A system that maintains bank accounts A and B

<table>
<thead>
<tr>
<th>Location 1: Account A</th>
<th>Location 2: Account B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t3 $400</td>
<td>C_{21}: $50</td>
</tr>
<tr>
<td></td>
<td>C_{12}: $100</td>
</tr>
<tr>
<td></td>
<td>$150</td>
</tr>
</tbody>
</table>

Global state shows $800 in the system!

X: Record state of B and channels C_{12} and C_{21} at t3:
B: $150 C_{21}: $50 C_{12}: $100

State of A was recorded before sending $100 but state of C_{12} after $100 transfer was initiated

<table>
<thead>
<tr>
<th>Location 1: Account A</th>
<th>Location 2: Account B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t4 $450</td>
<td>C_{21}: $0</td>
</tr>
<tr>
<td></td>
<td>C_{12}: $100</td>
</tr>
<tr>
<td></td>
<td>$150</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Location 1: Account A</th>
<th>Location 2: Account B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t5 $450</td>
<td>C_{21}: $0</td>
</tr>
<tr>
<td></td>
<td>C_{12}: $0</td>
</tr>
<tr>
<td></td>
<td>$250</td>
</tr>
</tbody>
</table>
Global states, cuts and frontiers

- **Cut** of a system’s execution – a subset of its global history (which is a union of prefixes of processes histories)
  
  \[ C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N} \]

![Diagram showing a physical timeline with two processes, \( p_1 \) and \( p_2 \), and their histories. The cut is marked as \( C \) and includes the prefixes up to \( e_1^2 \) and \( e_2^2 \). The past and future are indicated by arrows.](https://example.com/diagram.png)
The set of events \( \{ e^C_i : i = 1, 2, \ldots N \} \) is called the frontier of the cut.
Consistent cuts and global states

- A cut $C$ is consistent if, \textit{for each event}, it contains all the events that happened-before that event.
- A consistent global state is one that corresponds to a consistent cut.

![Diagram showing consistent and inconsistent cuts with event labels](image)
Runs and linearization

- The execution of a system can be characterized as a series of transitions between global states
  \[ S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \]

- A *run* – a total ordering of all events in a global history that is consistent with each local history’s ordering

- A *linearization or consistent run* – an ordering of the events in a global history that is consistent with the happened-before relation on \( H \)
  
  \( H \) is the union of individual processes’ histories \( (U_i h_i) \)

- All linearization (but not all runs) pass only through consistent global states
A state $S'$ is reachable from a state $S$ if there is a linearization that passes through $S$ and then $S'$

Detecting a condition (i.e., deadlock) = evaluating a global state predicate
- A function of set of global states of a system $\rightarrow \{\text{true, false}\}$
- Some predicates are stable (deadlock), others are not
Chandy & Lamport’s snapshot algorithm

- Goal – to record a consistent global state
- Algorithm records state locally at processes
  - BTW, it says nothing about how to collect them
- Assumptions
  - Neither channel nor processes fail; reliable, exactly-once communication
  - Channels are unidirectional and provide FIFO ordering
  - Graph of processes and channels is strongly connected
  - Any process may initiate the global snapshot at any time (just send a marker on a non-existing channel)
  - Processes may continue execution and communication while snapshot is taking place
Algorithm idea

- Each process records
  - Its state and, for each incoming channel, the set of messages sent to it
  - For each channel, any message that arrived after it records its state but before the sender records its own state

- Algorithm relies on special message, *markers*
  - A prompt for the receiver to save its own state, if it has not yet done it
  - A mean of determining which messages to include in the channel state
  - Since channels are FIFO, markers separate the messages in the channel to be included in the snapshot (process or channel state) from those not to be included
Algorithm idea

- Each process records
  - Its state and, for each incoming channel, the set of messages sent to it
  - For each channel, any message that arrived after it records its state but before the sender records its own state

- Algorithm relies on special message, *markers*
  - A prompt for the receiver to save its own state, if it has not yet done it
  - A mean of determining which messages to include in the channel state
  - Since channels are FIFO, markers separate the messages in the channel to be included in the snapshot (process or channel state) from those not to be included
Algorithm

Defined by two rules – marker sending and receiving

**Marker receiving rule for process \(p_i\)**

On receipt of a marker message at \(p_i\) over channel \(c\)

If \((p_i\) has not yet recorded its state) it
- records its process state
- records the state of \(c\) as the empty set
- turns on recording of messages arriving over other incoming channels

Else

\(p_i\) records the state of \(c\) as the set of messages it has received over \(c\) since it saved its state

endif
Algorithm is defined by two rules …

**Marker sending rule for process** $p_i$

After $p_i$ has recorded its state, for each outgoing channel $c$, before it sends any other message over $c$
- $p_i$ sends one marker message over $c$
Proving correctness

• That the algorithm satisfies C1 and C2
  – C2: \( send(m_{ij}) \notin LS_i \rightarrow mij \notin SC_{ij} \) AND \( rec(m_{ij}) \notin LS_j \)

• C2: Since \( p_j \) record snapshot upon receiving a maker on a channel \( c_{ij} \)
  – No message that follows the maker is part of \( LS_j \)
  – Since it stops recording the state of the channel at that point, it is not part of the \( SC_{ij} \) either
  – Given FIFO, no message sent after the marker (after \( p_i \) took its snapshot) is part of the \( SC_{ij} \)
Proving correctness

- That the algorithm satisfies C1 and C2
  - C1: \( \text{send}(m_{ij}) \in LSi \rightarrow mij \in SCij \) XOR \( \text{rec}(m_{ij}) \in LSj \)

- C1: When \( p_j \) receives a msg \( m_{ij} \) before the marker
  - If \( p_j \) has not recorded its snapshot yet, it includes \( m_{ij} \) in \( LS_j \)
  - Else, it records \( m_{ij} \) in the state of channel \( SC_{ij} \)
Two processes trading widgets; $p_1$ sends order over $c_{12}$ for widgets at $10$ per unit; $p_2$ sends widgets over $c_{21}$

Initial state; $p_2$ has already received an order for 5 widgets that’s about the send

- **Account**: $p_1$ has $1000$, $p_2$ has $50$
- **Widgets**: $p_1$ has none, $p_2$ has 2000
p₁ saves $LS₁$ in global state $S₀$

and sends a marker on its outgoing channel before sending a new order for 10 widgets (at $100$)
Example execution

$p_2$ receives the marker, emits order for 5 widgets from previous request; system enters $S_2$
Example execution

$p_1$ receives order, $p_2$ receives marker; $p_2$ saves $LS_2 <\$50, 1995>$
And $CS_{12}$ as empty; sends maker over $c_1$

$p_1$ receives marker over $c_{21}$ it records the $SC_{21}$ as the single message (five widgets) that has received after first recorded its state
Example execution

- Final recorded state:
  \[ p_1: <$1000,0> \]
  \[ p_2: <$50,1995> \]
  \[ c_{21}:<(five widgets)> \]
  \[ c_{12}: <> \]

- Note that this state differ from all the global states through which the system actually passed!

\[
S_0: \{<$1000,0>, <$50,2000>,<>,<>\}
\]
\[
S_1: \{<$900,0>, <$50,2000>,<>,<(Order 10, $100)M>\}
\]
\[
S_2: \{<$900,0>, <$50,1995>,<5 widgets>,<(Order 10, $100)M>\}
\]
\[
S_3: \{<$900,5>, <$50,1995>,<>,<(Order 10, $100)M>\}
\]
• Could have passed through it in an equivalent execution
  – Let seq = (e_i, 0<= i) be a distributed computation
  – Let’s S* be the recorded global state (the snapshot)
  – And S_Φ be the final state of the system
  – We can show that
    • S* can be reached from S_0 through permutations of seq (there’s a linearization)
    • We can reach S_Φ from there through permutations of seq

And if a stable property holds before, it will hold in the recorded global snapshot
Collecting global state and variants

- Collecting global state – some options
  - Each process send its local snapshot to the initiator
  - Each sends its along all outgoing channels, and forwards what it receives on the incoming channels

- Some variants to the algorithm
  - Optimizing concurrent initiation and distribution of the recorded snapshot – Spezialletti-Kearns ‘89
  - Optimizing incremental snapshots – Venkatesan ‘93
  - Handling non-FIFO channels – Helary ‘89, Lai-Yang ‘87
  - ...
Recording global state is an important paradigm in the design of distributed systems.

So it’s designing efficient methods to do it.

The challenge comes from the lack of shared memory or a global clock.

There are many alternative algorithms out there:

- Look at your application requirements to choose among them.
- E.g. checkpointing, termination detection, global state monitoring, …
A run with four processes

$p_2$ initiates a snapshot

Markers in outgoing channel
A run with four processes

$p_4$ takes a snapshot

Markers in outgoing channel $C_{43}$

Messages are part of $p_4$ state

$p_2$ takes a snapshot

$p_4$ takes a snapshot

Markers in outgoing channel $C_{43}$

Messages are part of $p_4$ state
A run with four processes

Markers in outgoing channels $C_{34}$, $C_{32}$ and $C_{31}$

$p_3$ takes a snapshot
A run with four processes

$p_1$ takes a snapshot

$p_2$ gets a second marker
$p_4$ gets a second marker

Markers in outgoing channels

Messages are part of $C_{32}$ state

Message is part of $C_{34}$ state
A run with four processes

$p_2$ gets a third marker
$p_3$ gets a third marker
$p_1$ gets a second marker