University of Freiburg, Germany
Department of Computer Science

Distributed Systems

Chapter 3 Time and Global States

Christian Schindelhauer

12. May 2014
2.2: Logical Time

Why?

- Getting physical clocks absolutely synchronized is not possible.
- Thus it is not always possible to determine the order of two events.
- For such cases logical time can be used as a solution.
  - If two events happen in the same process they are ordered as observed.
  - If two processes interchange messages, then the sending event is always considered to be before the receiving event.
Lamport’s happened-before relation (causal ordering)

- If two events $a, b$ happen in the same process $p_i$ they are ordered as observed and we write $a \rightarrow_i b$. Moreover, this implies $a \rightarrow b$ systemwide.
- If two processes interchange messages, then the sending event $a$ is always considered to be before the receiving event $b$, thus $a \rightarrow b$.
- Whenever $a \rightarrow b$ and $b \rightarrow c$, then also $a \rightarrow c$.

Events not being ordered by $\rightarrow$ are called concurrent.
Example

We conclude $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow d$, $d \rightarrow f$, $a \rightarrow f$, however not $a \rightarrow e$; $a, e$ are concurrent.

from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg
Algorithm of Leslie Lamport

- Let $L_i(e)$ denote the time stamp of event $e$ at process $P_i$.
- When a new event $a$ occurs in process $P_i$:
  \[ L_i := L_i + 1 \]
- Each message $m$ sent from $P_i$ to $P_j$ is piggybacked by the timestamp $L_i(a)$ of the send-event $a$.
- When $(m, t_a)$ is received by $P_j$, $P_j$ adjusts its logical clock $L_j$ to the logical clock of $P_j$.
  \[ L_j := \max\{L_j, t_a\} \]
  and increments $L_j$ for the received message event.
Three clocks with application of Lamport’s algorithm.

from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg
Totally ordered logical clocks

- Extend the Lamport clock for each process $P_i$:
- Clock values must be systemwide unique
  - for this the clock value $L_i$ is referred to with the process id $i$, i.e. $(L_i, i)$
  - all distinct clocks $L_i$ can be unified into a system clock $L$.
- Define the total ordering

\[ (T_i, i) < (T_j, j) :\iff \begin{cases} 
  i < j & \text{if } T_i = T_j \\
  T_i < T_j & \text{else}
\end{cases} \]

- So, we translate a partial ordering into a total ordering
- However from the total ordering $L(a) < L(b)$ one cannot conclude $a \rightarrow b$. 
Mattern’s Vector Clocks

- Vector clock for a system of $n$ processes: array of $n$ integers.
- Each process $P_i$ keeps its own vector clock $V_i$ which is used to timestamp local events.
- Processes piggyback their own vector clock on messages they send.
- Update rules for vector clocks:
  
  **VC1:** Initially, $V_i[j] := 0$ for $i, j \in \{1, \ldots, n\}$
  
  **VC2:** $P_i$ timestamps prior to each event: $V_i[i] := V_i[i] + 1$.
  
  **VC3:** $P_i$ sends the value $t = V_i$ with each message.
  
  **VC4:** When $P_i$ receives some message piggybacked with timestamp $t$, it sets

  $$V_i[j] := \max\{V_i[j], t[j]\} \quad \text{for } i = 1, 2, \ldots, n$$

- $V_i[i]$ is the number of events that $P_i$ has timestamped.
- $V_i[j]$ for $i \neq j$ is the number of events that have occurred at $P_j$ to the knowledge of $P_i$. 
Vector Clock Example

from Distributed Systems – Concepts and Design, Coulouris, Dollimore, Kindberg
Comparing vector timestamps

- The clock vectors define a partial ordering
  - $V = V'$ iff $V[j] = V'[j]$ for all $j \in \{1, \ldots, n\}$
  - $V \leq V'$ iff $V[j] \leq V'[j]$ for all $j \in \{1, \ldots, n\}$
  - $V < V'$ iff $V \leq V' \land V \neq V'$.
- If for events $a, b$ neither $V(a) \leq V(b)$ nor $V(a) \geq V(b)$ the events are called concurrent, i.e. $a \parallel b$

<table>
<thead>
<tr>
<th>$V(a)$</th>
<th>$V(b)$</th>
<th>Relation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2, 1, 0)</td>
<td>(2, 1, 0)</td>
<td>$V(a) = V(b)$</td>
<td>all entries are the same</td>
</tr>
<tr>
<td>(1, 2, 3)</td>
<td>(2, 3, 4)</td>
<td>$V(a) &lt; V(b)$</td>
<td>all entries of $V$ are prior to $V'$</td>
</tr>
<tr>
<td>(1, 2, 3)</td>
<td>(3, 2, 1)</td>
<td>$a \parallel b$</td>
<td>two events are concurrent</td>
</tr>
</tbody>
</table>
Lamport Relationship and Vector Clocks

Theorem

For any two events \( e_j, e_i \):

\[
e_j \rightarrow e_i \iff V(e_j) < V(e_i).
\]

Proof sketch

- \( e_j \rightarrow e_i \iff V_j < V_i \).
  - If the events occur on the same process then \( V_j < V_i \) follow directly.
  - \( e_j \rightarrow e_i \) implies a message is sent after \( e_j \) to the process with event \( e_i \) or two succeeding events of a process.
  - Since each entry of the receiving process is updated to at least the maximum of the entries of the sending processes, \( V_j < V_i \).

- \( e_j \rightarrow e_i \iff V_j < V_i \).
  - If both events occur on the same process, \( e_j \rightarrow e_j \) follows straightforward.
  - An increase of the \( i \)-th row can only be caused by a message path sent from the process of \( e_j \) to \( e_i \).

- Complete proof is left as an exercise.
2.3. Global System States

Distributed Garbage Collection

- Non-referenced objects need to be erased
- $p_2$ has an object referenced in a message to $p_1$
- $p_1$ has an object referenced by $p_2$
- Neither one can be erased

- How to determine a global state in the absence of global time
2.3. Global System States

Distributed Deadlock Detection

- occurs when processes wait for each other to send a message
- and the processes form a cycle

from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg
2.3. Global System States

Distributed Termination Detection

- How to detect that a distributed algorithm has terminated
- Assume $p_1$ and $p_2$ request values from the other
- If they wait for a value they are passive, otherwise active
- Assume both processes are passive. Can we conclude the system has terminated?
- No, since there might be an activating message on its way
2.3. Global System States

Distributed Debugging

- Distributed systems are difficult to debug
- e.g. consider a program where each process has a changing variable $x_i$
- All variables are required to be in range $|x_i - x_j| \leq 1$
- How to be sure that this will never be violated?
Consider system $\mathcal{P}$ of $n$ processes $p_i$ for $i = 1, \ldots, n$.

The execution of a process is characterized by its history (of events $e_i^t$)

$$\text{history}(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

We denote a finite prefix

$$h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$$

An event is either

- an internal action or
- sending a message or
- receiving a message

Let $s_i^k$ denote the state of process $p_i$ immediately before event $e_i^k$.

The global history $H$ is

$$H = h_1 \cup h_2 \cup \ldots \cup h_n$$

A cut $C$ of the system’s execution is a set of prefaces

$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_n^{c_n}$$
Consistent Cuts

- A cut $C$ is consistent if,

$$\text{For all events } e \in C : \ f \rightarrow e \implies f \in C.$$ 

- i.e. for each event it also contains all the events that happened-before the event.
Global States

- A *consistent global state* corresponds to a consistent cut.
- A *run* is a total ordering of all events in a global history that is consistent with each local history’s ordering ($\rightarrow_i$, for $i = 1, \ldots, n$).
- A *consistent run (linearization)* is an ordering of the events in the global history that is consistent with the happened-before-relation ($\rightarrow$) on $H$.
- Consistent runs pass only through consistent global states.
Global State Predicates, Stability, Safety and Liveness

- A *global state predicate* is a function that maps from the set of global states to \{true, false\}.
- *Stability* of a global state predicate: A global state predicate is *stable* if once it has reached true it remains in this state for all states reachable from this state.
- *Safety* is the assertion that an undesired state predicate evaluates to false to all states \( S \) reachable from the starting state \( S_0 \).
- *Liveness* is the assertion that a desired state predicate evaluates to true to all states \( S \) reachable from the starting state \( S_0 \).
How to detect and record a global state

'Snapshot' algorithm of Chandy and Lamport

- **Goal**
  - record a set of events corresponding to a global state (consistent cut)
  - in a living system during run-time
  - without extra process

- **Requirements**
  - channels, processes do not fail. Communication is reliable
  - channels are uni-directional and have FIFO message delivery
  - graph of processes and channels is strongly connected
  - any process may initiate a snapshot
  - processes continue their execution (including messages)

- **Notations**
  - $p_i$’s incoming channel: where all messages for $p_i$ arrive
  - $p_i$’s outgoing channel: where $p_i$ sends all messages to other processes
  - Marker message: a special message distinct from every other message
Distributed Snapshot of Chandy and Lamport

Marker receiving rule for process $p_i$

On $p_i$’s receipt of a marker message over channel $c$:
- if ($p_i$ has not yet recorded its state) it
  records its process state now;
  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.
end if

Marker sending rule for process $p_i$

After $p_i$ has recorded its state, for each outgoing channel $c$:
- $p_i$ sends one marker message over $c$
  (before it sends any other message over $c$).

from Distributed Systems – Concepts and Design, Coulouris, Dollimore, Kindberg
**General remarks**

*A snapshot consists of the state of a process and states of all incoming channels.*

- **Starting a snapshot:**
  - Any process \( P \) can start a snapshot.
    1. Create a local snapshot of \( P \)'s state.
    2. Send marker message over all channels.
  - Upon receipt of a marker message, other processes participate in the snapshot.

- **Collecting the snapshot:**
  - Every process has created a local snapshot.
  - The local snapshot can be sent to a collector process.

- **Terminating a snapshot:**
  - If marker message has been received on all channels, then the snapshot terminates
  - Then the snapshot can be sent to a collector process.
2. Time and Global States
2.3. Global States

Distributed Snapshot of Chandy and Lamport

State S0
- mark process
- 1000€
- 0 T-Shirts

State S1
- mark process
- 900€
- 0 T-Shirts
Distributed Snapshot of Chandy and Lamport

State S1

- mark process
- 900€
- 0 T-Shirts

Order 10 shirts for 100€

p1

Marker

50€

2000 T-Shirts

old order 5 shirts for 50€

State S2

- mark process
- 900€
- 0 T-Shirts

Order 10 shirts for 100€

p1

Marker

50€

1995 T-Shirts

send 5 shirts

State S3

- mark process
- 900€
- 5 T-Shirts

Order 10 shirts for 100€

p1

Marker

50€

1995 T-Shirts

Marker
2. Time and Global States

2.3. Global States

Distributed Snapshot of Chandy and Lamport

Recorded state

<table>
<thead>
<tr>
<th>1000€</th>
<th>Order 10 shirts for 100€</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 T-Shirts</td>
<td></td>
</tr>
</tbody>
</table>

p1 \[c1\] \[\rightarrow\] p2

Order 10 shirts for 100€

1000€ 50€

1995 T-Shirts
Termination of the snapshot algorithm

- If marker message has been received on all channels, then the snapshot terminates
- If the communication graph induced by the messages is strongly connected then the marker eventually reaches all nodes
- \( \Rightarrow \) only a finite number of messages need to be recorded
The snapshot algorithm selects a Consistent Cut

- Consider two events $e_i \rightarrow e_j$ on processes $p_i$ and $p_j$
- If $e_j$ is in the cut of the snapshot, then $e_i$ should be, too
- If $e_j$ occurred before $p_j$ taking its snapshot, then $e_i$ should have occurred before $p_i$ has taking its snapshot
- If $p_i = p_j$ this is obvious.
- Now we consider $p_i \neq p_j$ and assume (*) that $e_i$ is not in the cut and $e_j$ is within the cut.
- Consider messages $m_1, m_2, \ldots m_h$ causing the happened-before relationship $e_i \rightarrow e_j$.
- So, $m_1$ must have sent after the snapshot, and $m_2$, and so forth. Each of this messages must have been sent after the marker message occurred on each channel (because of FIFO rules on the channel).
- Then, $e_j$ cannot be in the cut. This contradicts (*) and proofs the claim.
Reachability of the snapshot algorithm selects a Consistent Cut

A snapshot characterizes events into two types

1. **pre-snap**: An event happening before marking the corresponding process
2. **post-snap**: An event happening after marking

Note that pre-snap events can take place after post-snap events

It is impossible that $e_i \rightarrow e_j$ if $e_i$ is a post-snap event and $e_j$ is a pre-snap event

---

From *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg
Distributed Debugging

Goal of algorithm of Marzullo and Neiger

- Testing properties post-hoc, e.g. safety conditions
- Capture traces rather than snapshots
- Gathered by a monitoring process (outside the system)
- How are process states collected
- How to extract consistent global states
- How to evaluate safety, stability and liveness conditions
Distributed Debugging

**Temporal operators**

Consider all linearizations of $H$

- **possible** $\phi$: There exists a consistent global state $S$ through a linearization such that $\phi(S)$ is true.

- **definitely** $\phi$: For all linearizations a consistent global state will be passed such that $\phi(S)$ is true.
Relationship of Definitely and Possibly

1. \( \forall S \in H : \phi(S) \implies \text{definitely } \phi \)
2. \( \forall S \in H : \phi(S) \implies \text{possible } \phi \)
3. \( \forall S \in H : \neg \phi(S) \implies \neg \text{definitely } \phi \)
4. \( \forall S \in H : \neg \phi(S) \implies \neg \text{possibly } \phi \)
5. \( \text{definitely } \phi \implies \text{possibly } \phi \)
6. \( \neg \text{possibly } \phi \implies \text{definitely } \neg \phi \)
7. \( \text{definitely } \neg \phi \iff \neg \text{possibly } \phi \)
Distributed Debugging: Definitely $|x_1 - x_2| \leq 50$

$S_{ij}$ = global state after $i$ events at process 1 and $j$ events at process 2

$x_1 = 105$

$x_L = 35$

$10 \leq 50$
\begin{aligned}
  \sum_{i=0}^{n} S_{i} & \quad i, j \in \mathbb{E}, n^3 \\
  (n+1) & \\
  \binom{m+1}{m} & 
\end{aligned}
Algorithm of Marzullo & Neiger

Collecting the states

- All initial states are sent to the monitor
- All state changes are sent to the monitor
- If only a predicate is monitored $\phi$ then only states are sent where $\phi$ changes
- With the states the corresponding vector clock is sent to the monitor
- The vector clocks will be used to establish the $\rightarrow$-relationship
- The monitor computes the DAG corresponding to the happened-before-relationship
- Arrange the graph in levels $L = 0, 1, \ldots$ such that no global state in level happened before a state in lower level.
- In Level 0 there is only the initial state.
1. Evaluating possibly $\phi$ for global history $H$ of $N$ processes

$L := 0$;

$States := \{ (s_1^0, s_2^0, ..., s_N^0) \}$;

while ($\phi(S) = False$ for all $S \in States$)

$L := L + 1$;

$Reachable := \{ S' : S' \text{ reachable in } H \text{ from some } S \in States \land level(S') = L \}$;

$States := Reachable$

end while

output "possibly $\phi$";

from Distributed Systems – Concepts and Design, Coulouris, Dollimore, Kindberg
2. Evaluating definitely $\phi$ for global history $H$ of $N$ processes

$L := 0$;
if ($\phi(s_1^0, s_2^0, \ldots, s_N^0)$) then $States := \{\}$ else $States := \{ (s_1^0, s_2^0, \ldots, s_N^0) \}$;
while ($States \neq \{\}$)
$L := L + 1$;
$Reachable := \{ S' : S'$ reachable in $H$ from some $S \in States \land \text{level}(S') = L \}$;
$States := \{ S \in Reachable : \phi(S) = False \}$
end while
output "definitely $\phi$";

from Distributed Systems – Concepts and Design, Coulouris, Dollimore, Kindberg
Evaluating Definitely $\phi(S)$

Cost

Let $n$ be the number of processes with $k$ events each.

- Time: $O(k^n)$
- Space: $O(kn)$.

Level 0

1

2

3

4

5

$F = (\phi(S) = \text{False})$; $T = (\phi(S) = \text{True})$