Programming Distributed Systems

03 Causality and Vector clocks

Annette Bieniusa, Peter Zeller

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019
Motivation

- Causality is fundamental to many problems occurring in distributed computing
- **Examples**: Determining a consistent recovery point, detecting race conditions, exploitation of parallelism
- The happens-before relation of events is often also called *causality relation* [1].

An event \( e \) may causally affect another event \( e' \) if and only if \( e \rightarrow e' \).

- The happens-before order \( \rightarrow \) indicates only *potential* causal relationship.
- Tracking whether an event indeed is a cause of another event is much more involved and requires more complex dependency analysis.
Overview

- Causal Broadcast
- Causality Tracking with Vector clocks
- Causal Broadcast revisited
(Reliable) Causal Broadcast (RCO): Specification

- **RB1 - RB4** from reliable broadcast
- **CB (Causal delivery):** No process $p$ delivers a message $m'$ unless $p$ has already delivered every message $m$ such that $m \rightarrow m'$. 
Causal Broadcast (RCO): Algorithm 1 (No-waiting)

**State:**
- `delivered` //set of messages ids that were already rcoDelivered
- `past` // ordered set that it has rco-Broadcast or rco-Delivered

**Upon Init do:**
- `delivered` <- ∅;
- `past` <- ∅;

**Upon rco-Broadcast(m) do**
- `mid` <- generateUniqueID(m);
- `trigger` rb-Broadcast([`mid`, past, m]);
- `past` <- `past` ∪ {(`self`, `mid`, m)}; // ordered after prior entries

**Upon rb-Deliver(p, [`mid`, pastm, m]) do**
- if ( `mid` /∈ `delivered` ) then
  -forall (s_n, n_id, n) in pastm do // deterministic order!
    - if (n_id /∈ `delivered`) then
      - `trigger` rco-Deliver(s_n, n);
      - `delivered` <- `delivered` ∪ {n_id};
      - `past` <- `past` ∪ {(s_n, n_id, n)};
    - `trigger` rco-Deliver(p, m);
    - `delivered` <- `delivered` ∪ {`mid`};
    - `past` <- `past` ∪ {(p, `mid`, m)};
Causal Broadcast: Scenario 1

Process A

Process B

Process C

m_1

m_2

m_1m_2
Remarks

- Message id’s could be reused for RB broadcast
- $past_m$ of a message includes all messages that causally precede $m$
- Message from causal past of $m$ are delivered before message $m$
- Size of messages grows linearly with every message that is broadcast since it includes the complete causal past
- **Idea**: Garbage collect the causal past
  - If we know when a process fails (i.e., under the Fail-stop model), we can remove messages from the causal past
  - When a process rb-Delivers a message $m$, it rb-Broadcasts an acknowledgement message to all other processes
  - When an acknowledgement for message $m$ has been rbDelivered by all correct processes, $m$ is removed from $past$
  - $N^2$ additional ack messages for each data message
  - Typically, acknowledgements are grouped and processed in batch mode
Causality tracking with Vector clocks
Causal Histories

- We here distinguish three types of events occurring in a process:
  - Send events
  - Receive events
  - Local / internal events

- Let $E_i$ denote the set of events occurring at process $p_i$ and $E$ the set of all executed events:

$$E = E_1 \cup \cdots \cup E_n$$

- The causal history of an event $e \in E$ is defined as

$$C(e) = \{e' \in E \mid e' \rightarrow e\} \cup \{e\}$$

- Note: Just a different representation of happens-before:

$$e' \rightarrow e \iff e' \neq e \land e' \in C(e)$$
Example: Causal history of $b_3$

$C(b_3) = \{ a_1, b_1, b_2, b_3, c_1, c_2 \}$
Tracking causal histories

Each process $p_i$ stores current causal history as set of events $C_i$.

- Initially, $C_i \leftarrow \emptyset$
- On each local event $e$ at process $p_i$, the event is added to the set:

  $$C_i \leftarrow C_i \cup \{e\}$$

- On sending a message $m$, $p_i$ updates $C_i$ as for a local event and attaches the new value of $C_i$ to $m$.
- On receiving message $m$ with causal history $C(m)$, $p_i$ updates $C$ as for a local event. Next, $p_i$ adds the causal history from $C(m)$:

  $$C_i \leftarrow C_i \cup C(m)$$
Example: Causal histories
Example: Causal histories

Process A

Process B

Process C

Annette Bieniusa, Peter Zeller
Programming Distributed Systems
Summer Term 2019
Example: Causal histories
Example: Causal histories

Process A

Process B

Process C
Example: Causal histories

Process A
- $a_1$
- $\{a_1\}$
- $b_1$
- $b_2$ $\{a_1, a_2\}$
- $b_3$
- $b_4$
- $a_3$

Process B
- $c_1$ $\{a_1, b_1\}$
- $c_2$ $\{a_1, b_1, b_2, c_1, c_2\}$
- $c_3$
- $c_4$

Process C
- $\{c_1\}$
- $\{c_1, c_2\}$
Example: Causal histories

Process A

\(a_1\) \[\{a_1\}\] \(b_1\) \(b_2\) \[\{a_1, b_1\}\] \(b_3\) \(b_4\) \[\{a_1, b_1, b_2, b_3, c_1, c_2\}\] \(a_3\)

Process B

\(c_1\) \[\{c_1\}\] \(\{a_1, b_1\}\) \[\{c_1, b_1\}\] \(\{a_1, b_1, b_2, c_1, c_2\}\) \(c_2\) \(\{c_2\}\) \[\{c_1, c_2\}\]

Process C
Example: Causal histories

\[
\begin{align*}
\text{Process A} & : & a_1 & \rightarrow & b_1 & \rightarrow & b_2 & \rightarrow & a_2 & \rightarrow & b_3 & \rightarrow & b_4 & \rightarrow & a_3 \\
\text{Process B} & : & c_1 & \rightarrow & \{a_1, b_1\} & \rightarrow & \{a_1, b_1, b_2, c_1, c_2\} & \rightarrow & \{a_1, b_1, b_2, b_3, c_1, c_2\} \\
\text{Process C} & : & \{c_1\} & \rightarrow & \{c_1, c_2\} & \rightarrow & \{a_1, b_1, b_2, b_3, c_1, c_2\} & \rightarrow & \{a_1, b_1, b_2, b_3, b_4, c_1, c_2, c_3, c_4\}
\end{align*}
\]
Example: Causal histories

Can we represent causal histories more efficiently?
Example: Efficient representation of causal histories

- Process A
  - $a_1\quad [1, 0, 0] \quad b_1 \quad a_2 \quad [2, 0, 0] \quad a_3$
- Process B
  - $b_1\quad [1, 1, 0] \quad b_2\quad [1, 2, 2] \quad b_3\quad [1, 3, 2] \quad b_4$
- Process C
  - $c_1\quad [0, 0, 1] \quad c_2\quad [0, 0, 2] \quad c_3\quad [1, 4, 4]$
Efficient representation of causal histories

- Vector clock $V(e)$ as efficient representation of $C(e)$.
- Vector clock is a mapping from processes to natural numbers:
  - Example: $[p_1 \mapsto 3, p_2 \mapsto 4, p_3 \mapsto 1]$
  - If processes are numbered $1, \ldots, n$, this mapping can be represented as a vector, e.g. $[3, 4, 1]$
  - Intuitively: $p_1 \mapsto 3$ means “observed 3 events from process $p_1$”
Formal Construction

- Assume processes are numbered 1, . . . , n
- Let $E_k = \{e_{k1}, e_{k2}, \ldots\}$ be the events of process $k$
  - Totally ordered: $e_{k1} \rightarrow e_{k2}, e_{k2} \rightarrow e_{k3}, \ldots$
- Let $C(e)[k] = C(e) \cap E_k$ denote the projection of $C(E)$ on process $k$.
  
  \[ C(e) = C(e)[1] \cup \cdots \cup C(e)[n] \]

- Now, if $e_{kj} \in C(e)[k]$, then by definition it holds that
  $e_{k1}, \ldots, e_{kj} \in C(e)[k]$
- The set $C(e)[k]$ is thus sufficiently characterized by the largest index of its events, i.e. its cardinality!
- Summarize $C(e)$ by an n-dimensional vector $V(e)$ such that for $k = 1, \ldots, n$:
  \[ V(e)[k] = |C(e)[k]| \]
Note: Both representations are lattices with a lower bound

<table>
<thead>
<tr>
<th>Operator</th>
<th>Causal history</th>
<th>Vector clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊥</td>
<td>∅</td>
<td>λi. 0</td>
</tr>
<tr>
<td>$A \leq B$</td>
<td>$A \subseteq B$</td>
<td>∀i. $A[i] \leq B[i]$</td>
</tr>
<tr>
<td>$A \geq B$</td>
<td>$A \supseteq B$</td>
<td>∀i. $A[i] \geq B[i]$</td>
</tr>
<tr>
<td>$A \sqcup B$</td>
<td>$A \cup B$</td>
<td>$\lambda i. \max(A[i], B[i])$</td>
</tr>
<tr>
<td>$A \sqcap B$</td>
<td>$A \cap B$</td>
<td>$\lambda i. \min(A[i], B[i])$</td>
</tr>
</tbody>
</table>

- ⊥: bottom, or smallest element
- $A \sqcup B$: least upper bound, or join, or supremum
- $A \sqcap B$: greatest lower bound, or meet, or infimum
Tracking causal histories

Each process $p_i$ stores current causal history as set of events $C_i$.

- Initially, $C_i \leftarrow \emptyset$
- On each local event $e$ at process $p_i$, the event is added to the set: $C_i \leftarrow C_i \cup \{e\}$
- On sending a message $m$, $p_i$ updates $C_i$ as for a local event and attaches the new value of $C_i$ to $m$.
- On receiving message $m$ with causal history $C(m)$, $p_i$ updates $C_i$ as for a local event. Next, $p_i$ adds the causal history from $C(m)$:

$$C_i \leftarrow C_i \cup C(m)$$
Tracking causal histories

Each process $p_i$ stores current causal history as set of events $C_i$.

- Initially, $C_i \leftarrow \bot$
- On each local event $e$ at process $p_i$, the event is added to the set: $C_i \leftarrow C_i \cup \{e\}$
- On sending a message $m$, $p_i$ updates $C_i$ as for a local event and attaches the new value of $C_i$ to $m$.
- On receiving message $m$ with causal history $C(m)$, $p_i$ updates $C_i$ as for a local event. Next, $p_i$ adds the causal history from $C(m)$:

$$C_i \leftarrow C_i \sqcup C(m)$$
Vector time

Each process $p_i$ stores current causal history as a vector clock $V_i$.

- Initially, $V_i[k] \leftarrow \bot$
- On each local event, process $p_i$ increments its own entry in $V_i$ as follows: $V_i[i] \leftarrow V_i[i] + 1$
- On sending a message $m$, $p_i$ updates $V_i$ as for a local event and attaches new value of $V_i$ to $m$.
- On receiving message $m$ with vector time $V(m)$, $p_i$ increments its own entry as for a local event. Next, $p_i$ updates its current $V_i$ by joining $V(m)$ and $V_i$:

$$V_i \leftarrow V_i[k] \sqcup V(m)$$
Relating vector times

Let $u, v$ denote time vectors. We say that

- $u \leq v$ iff $u[k] \leq u[k]$ for $k = 1, \ldots, n$
- $u < v$ iff $u \leq v$ and $u \neq v$
- $u \parallel v$ iff neither $u \leq v$ nor $v \leq u$

For two events $e$ and $e'$, it holds that $e \rightarrow e' \iff V(e) < V(e')$

Proof: By construction.
How does vector time relate to Lamport timestamps?

- Both are logical clocks, counting events.
- Lamport time (and real time) are insufficient to characterize causality and can’t be used to prove that events are not causally related
Causal Broadcast (RCO): Algorithm 2 (Waiting)

**State:**
- pending //set of messages that cannot be delivered yet
- VC // vector clock

**Upon Init do:**
- pending <- ∅;
- forall \( p_i \in \Pi \) do: VC\[p_i\] <- 0;

**Upon rco-Broadcast(\(m\)) do**
- trigger rco-Deliver(self, \(m\));
- trigger rb-Broadcast(VC, \(m\));
- VC[self] <- VC[self] + 1;

**Upon rb-Deliver(p, VC\(m\), \(m\)) do**
- if ( \(p \neq self\) ) then
  - pending <- pending \(\cup\) \{(p, VC\(m\), \(m\))\};
  - while exists (q, VC\(m\), \(m\)) \(\in\) pending, such that VC \(\geq\) VC\(m\) \(do\)
    - pending <- pending \(\setminus\) \{(q, VC\(m\), \(m\))\};
    - trigger rco-Deliver(q, \(m\));
  - VC[q] <- VC[q] + 1;
Causal Broadcast (RCO): Algorithm 2 (Waiting)

State:
pending // set of messages that cannot be delivered yet
VC // vector clock

Upon Init do:
pending ← ∅;
forall \( p_i \in \Pi \) do: VC\[p_i\] ← 0;

Upon rco-Broadcast\( (m) \) do
  trigger rco-Deliver\( (self, m) \);
  trigger rb-Broadcast\( (VC, m) \);
  VC[\( self \)] ← VC[\( self \)] + 1;

Upon rb-Deliver\( (p, VC_m, m) \) do
  if (p ≠ self) then
    pending ← pending ∪ \{ \( p, VC_m, m \) \};
  while exists \( (q, VC_{mq}, m_q) \in \) pending, such that VC ≥ VC\( _{mq} \) do
    pending ← pending \{ \( q, VC_{mq}, m_q \) \};
  trigger rco-Deliver\( (q, m_q) \);
  VC[q] ← VC[q] + 1;
Limitations of Causal Broadcast

Processes can observe messages in different order!

*Example*: Replicated database handling bank accounts

- Initially, account A holds 1000 Euro.
- User deposits 150 Euro, triggers broadcast of message $m_1 = \text{'add 150 Euro to A'}$
- Concurrently, bank initiates broadcast of message $m_2 = \text{'add 2% interest to A'}$
- Diverging state!

$\Rightarrow$ Later lecture: Atomic broadcast!
Summary

- Causality important for many scenarios
- Causality not always sufficient
- Vector clocks:
  - Efficient representation of causal histories / happens-before
  - How many events from which process?
- Causal broadcast: Use vector clocks to deliver in causal order
Further reading I