Causal Broadcast: How to Forget?

Brice Nédelec, Pascal Molli, and Achour Mostéfaoui
OPODIS’18, Hong Kong, December 17–19.
Causal broadcast is the core of many distributed applications such as distributed social networks, distributed collaborative software, or distributed data stores.

What if... I want to develop fully decentralized applications?
Causal broadcast ensures causal order

Causal broadcast is a **reliable broadcast** that ensures a **specific ordering** among message deliveries.

**Definition (Causal order)**

The delivery order of messages follows the Lamport’s happen before relationships $\rightarrow$ of the corresponding broadcasts.

$$\forall A, B, C, b_A(m) \rightarrow b_B(m') \implies d_C(m) \rightarrow d_C(m')$$
Causal broadcast is a **reliable broadcast** that ensures a **specific ordering** among message deliveries.

**Definition (Uniform reliable broadcast)**

When a process A broadcasts a message to all processes of its system $b_A(m)$, each correct process B eventually receives it and delivers it $d_B(m)$. URB guarantees validity, uniform agreement, and uniform integrity.

A process must receive every message and deliver each message once.
How to forget obsolete control information about broadcast messages?

- forget all ✓
- no ✗
- ring, tree...
- control information
- tolerate failures
- ✓ yes
- forget none ✗
- vector clocks
Proposal: PRC-broadcast

PRC-broadcast exploits the topology without constraining it. It maintains a local data structure the size of which does not monotonically increase.

This implementation of causal broadcast uses the causal order to improve on the underlying reliable broadcast implementation.
**Definition (Link memory)**

A link from Process A to Process B remembers among Process B’s delivered messages those that will be received from Process A; and forgets among Process B’s delivered messages those that will never be received from Process A. \( \forall m, \ remember_{BA}(m) \equiv d_B(m) \land \neg r_{BA}(m) \)

**Theorem (Link memory forbids multiple delivery)**

Assuming that each link conveys each message at most once, link memory is sufficient to forbid multiple delivery.
Definition (Link memory)
A link from Process A to Process B remembers among Process B’s delivered messages those that will be received from Process A; and forgets among Process B’s delivered messages those that will never be received from Process A. ∀m, remember_{BA}(m) \equiv d_B(m) \land \neg r_{BA}(m)

Theorem (Link memory forbids multiple delivery)
Assuming that each link conveys each message at most once, link memory is sufficient to forbid multiple delivery.
Definition (Link memory)
A link from Process A to Process B remembers among Process B’s delivered messages those that will be received from Process A; and forgets among Process B’s delivered messages those that will never be received from Process A. ∀m, \( \text{remember}_{BA}(m) \equiv d_B(m) \land \neg r_{BA}(m) \)

Theorem (Link memory forbids multiple delivery)
Assuming that each link conveys each message at most once, link memory is sufficient to forbid multiple delivery.
Definition (Link memory)
A link from Process A to Process B remembers among Process B’s delivered messages those that will be received from Process A; and forgets among Process B’s delivered messages those that will never be received from Process A. \( \forall m, \text{remember}_{BA}(m) \equiv d_B(m) \land \neg r_{BA}(m) \)

Theorem (Link memory forbids multiple delivery)
Assuming that each link conveys each message at most once, link memory is sufficient to forbid multiple delivery.
Definition (Link memory)
A link from Process A to Process B remembers among Process B’s delivered messages those that will be received from Process A; and forgets among Process B’s delivered messages those that will never be received from Process A. \( \forall m, \text{remember}_{BA}(m) \equiv d_B(m) \land \neg r_{BA}(m) \)

Theorem (Link memory forbids multiple delivery)
Assuming that each link conveys each message at most once, link memory is sufficient to forbid multiple delivery.
Link memory in dynamic systems: \( X \)
Link memory in dynamic systems: X
Link memory in dynamic systems: X

should I expect \( b \) from the new link?
Link memory in dynamic systems: $X$

should I expect $b$ from the new link?

infinitely increasing memory consumption

Multiple delivery and cascading effect over the whole system
How can processes initialize link memory?
Example of link memory initialization

(using FIFO links)
Example of link memory initialization (using FIFO links)
Example of link memory initialization

(using FIFO links)

\[ B_\beta \]

\[ C_2 \cup B_3 \]

\[ B_3' \cup C_3 \text{ with } B_3' \subseteq B_3 \]

\[ B_\alpha : \{c_1\} \]
Example of link memory initialization

(using FIFO links)

\[ C_2 \cup B_3 \]

\[ B_\beta \]

\[ B_\alpha : \{ c_1 \} \]

\[ B_\pi \]

\[ B' \]

\[ B'_3 \subset B_3 \]

\[ \emptyset \]

\[ B \]

\[ C \]

\[ A \]

\[ B \]

\[ c_1 \]

\[ \beta \]

\[ \alpha \]
Example of link memory initialization (using FIFO links)

\[ \emptyset \]

\[ B \]

\[ \emptyset \]

\[ c_1, b_1 \]

\[ B_\beta \]

\[ C \]

\[ \emptyset \]

\[ c_1 \]

\[ B_\alpha \]

\[ B_\alpha : \{c_1\} \]

\[ B_\beta : [c_1, b_1] \]

\[ B_3' \cup C_3 \text{ with } B_3' \subseteq B_3 \]
Example of link memory initialization (using FIFO links)

- $B \alpha : \{c_1, c_2\}$
- $B_\beta : [c_1, b_1]$
Example of link memory initialization (using FIFO links)
Example of link memory initialization (using FIFO links)

\[ B_\beta : [c_1, b_1, b_2, c_2] \]

\[ B_\alpha : \{c_1, c_2\} \]

\[ B_\pi : \{b_1, c_3\} \]
Example of link memory initialization

(uses FIFO links)

\[B_{\beta} : \{c_1, b_1, b_2, c_2\}\]

\[B_{\alpha} : \{c_1, c_2\}\]

\[B_{\pi} : \{b_1, c_3\}\]

\[B_{\beta} : [c_1, b_1, b_2, c_2]\]
Disambiguation: to deliver, to expect, to ignore

\[ B_\alpha : \{ c_1, c_2 \} \]
\[ B_\pi : \{ b_1, c_3 \} \]
\[ B_\beta : [c_1, b_1, b_2, c_2] \]

Delivered by Process B:
- To deliver: \( B_\beta \setminus B_\alpha \setminus B_\pi \)
- \([b_2]:\ maintain\)

Delivered by Process C:
- To ignore: \( B_\beta \land (B_\alpha \cup B_\pi) \)
- \( \{c_1, b_1, c_2\} \)

To expect from B:
- \( B_\pi \setminus B_\beta \)
- \( \text{initialize} : \{c_3\} \)
Order to the help of link memory initialization

Lemma (Buffer $B_\alpha$)

$B_\alpha$ contains $B_2$ and $C_2$.

Lemma (Buffer $B_\beta$)

$B_\beta$ contains $C_2$ and $B_3$.

Lemma (Buffer $B_\pi$)

$B_\pi$ contains $B'_3$ and $C_3$ with $B'_3 \subseteq B_3$.

**Lemma $B_\alpha, B_\beta, B_\pi$ initialize link memory**

The memory of a new link becomes correct at receipt of $B_\beta$.

To deliver: $B_\beta \setminus B_\alpha \setminus B_\pi = (C_2 \cup B_3) \setminus (B_2 \cup C_2) \setminus (B'_3 \cup C_3) = B_3 \setminus B'_3$

To expect: $B_\pi \setminus B_\beta = (B'_3 \cup C_3) \setminus (C_2 \cup B_3) = C_3$
### Complexity: a new trade-off

<table>
<thead>
<tr>
<th>message overhead</th>
<th>delivery execution time</th>
<th>local space consumption</th>
<th># control messages per added link</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(</td>
<td>Q_i</td>
<td>)$</td>
</tr>
</tbody>
</table>

![Diagram of message delivery](image)

We can achieve $\frac{|Q_i| - 1}{2} \cdot 2 \cdot N$ in space consumed. Where $\frac{1}{2}$ is a switch between “already received” and “expected” tags. And $2 \cdot N$ is a vector of intervals instead of all messages.

Most importantly, this structure is not grow-only.
Overhead: depends on the overlay network

<table>
<thead>
<tr>
<th>message overhead</th>
<th>delivery execution time</th>
<th>local space consumption</th>
<th># control messages per added link</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>$O(</td>
<td>Q_i</td>
<td>)$</td>
</tr>
</tbody>
</table>

Most importantly, the number of messages can be **small** and **constant**.
**Goal:** Show the trade-off proposed by PRC-broadcast experimentally.

A random peer-sampling protocol builds the overlay network. The topology has properties close to those of random graphs.

The system is very dynamic. Processes periodically exchange their outgoing links.

Links are bidirectional. Each new bidirectional link still costs 8 control messages.
Experiments: new complexity in local space consumed

- $|P| = 10000$, $|Q| ≈ 15.0$
- $|P| = 1000$, $|Q| ≈ 13.5$
- $|P| = 100$, $|Q| ≈ 10.0$

10 broadcasts per second

Average local space overhead per process

Latency (s)

Time (min)

100-entries vector

300 ms
Experiments: new complexity in local space consumed

- 10 broadcasts per second
- 100-entries vector
- Average local space overhead per process

- $|P| = 10000, |Q| \approx 15.0$
- $|P| = 1000, |Q| \approx 13.5$
- $|P| = 100, |Q| \approx 10.0$

$Q_i$ notation is used to denote the vector's index.
Experiments: new complexity in local space consumed

- \(|P|=10000, |Q|\approx 15.0\)
- \(|P|=1000, |Q|\approx 13.5\)
- \(|P|=100, |Q|\approx 10.0\)

10 broadcasts per second

average local space overhead per process

latency (s)

time (min)

100-entries vector

\(Q_i\)

300 ms
Experiments: new complexity in local space consumed

- |P| = 10000, |Q| ≈ 15.0
- |P| = 1000, |Q| ≈ 13.5
- |P| = 100, |Q| ≈ 10.0

10 broadcasts per second

Average local space overhead per process

Latency (s)

Time (min)

100-entries vector

300 ms
Experiments: new complexity in local space consumed

- |P|=10000, |Q|≈15.0
- |P|=1000, |Q|≈13.5
- |P|=100, |Q|≈10.0

10 broadcasts per second

average local space overhead per process

latency (s)

100-entries vector

300 ms

time (min)
Experiments: new complexity in local space consumed

10 broadcasts per second
100-entries vector
average local space overhead per process

|P|=10000, |Q|≈15.0
|P|=1000, |Q|≈13.5
|P|=100, |Q|≈10.0

100-entries vector

latency (s)

0 100 200 300 400 500 600 700 800

300 ms

0 10 20 30 40 50 60

0.5 1.5 2.5

time (min)
Experiments: small traffic overhead

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>15.0</td>
</tr>
<tr>
<td>1000</td>
<td>13.5</td>
</tr>
<tr>
<td>100</td>
<td>10.0</td>
</tr>
</tbody>
</table>

average number of control messages per process per second

latency (s)

1 shuffle per process per minute

no shuffles

300 ms

time (min)
Experiments: small traffic overhead

- Average number of control messages per process per second
- Latency (s)
- Time (min)

| |P| = 10000, |Q| ≈ 15.0
|---|---|
| |P| = 1000, |Q| ≈ 13.5
| |P| = 100, |Q| ≈ 10.0

- 1 shuffle per process per minute
- No shuffles

- 300 ms latency
Experiments: small traffic overhead

| |P| = 10000, |Q| ≈ 15.0
| |P| = 1000, |Q| ≈ 13.5
| |P| = 100, |Q| ≈ 10.0

average number of control messages per process per second

latency (s)

time (min)

1 shuffle per process per minute
no shuffles

300 ms
Experiments: small traffic overhead

- |P| = 10000, |Q| ≈ 15.0
- |P| = 1000, |Q| ≈ 13.5
- |P| = 100, |Q| ≈ 10.0

Average number of control messages per process per second over time (min).

- No shuffles
- 1 shuffle per process per minute

Latency (s) from 0 to 300 ms.

Graph showing the average number of control messages and latency over time for different process counts and traffic loads.
PRC-broadcast constitutes a new trade-off for causal broadcast.

<table>
<thead>
<tr>
<th></th>
<th>message overhead</th>
<th>delivery execution time</th>
<th>local space consumption</th>
<th># control messages per added link</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector-based R-broadcast</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>0</td>
</tr>
<tr>
<td>vector-based C-broadcast</td>
<td>$O(N)$</td>
<td>$O(W \cdot N)$</td>
<td>$O(N + W \cdot N)$</td>
<td>0</td>
</tr>
<tr>
<td>PC-broadcast</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
<td>$3$ to $2 \cdot</td>
</tr>
<tr>
<td>PRC-broadcast</td>
<td>$O(1)$</td>
<td>$O(</td>
<td>Q_i</td>
<td>)$</td>
</tr>
</tbody>
</table>

💡 Fun fact! This implementation of causal broadcast even makes an efficient implementation for reliable broadcast!
To be continued... Retrieving partial order out of flattened orders.
Thanks!

brice.nedelec@ls2n.fr