# Causal Broadcast: How to Forget?

Brice Nédelec, Pascal Molli, Achour Mostefaoui

# Causal Broadcast: How to Forget?

Brice Nédelec, Pascal Molli, and Achour Mostéfaoui
LS2N, University of Nantes,
2 rue de la Houssinière,
BP 92208, 44322 Nantes Cedex 3, France
first.last@univ-nantes.fr

## ABSTRACT

Causal broadcast constitutes a fundamental communication primitive of many distributed protocols and applications. However, state-of-the-art implementations fail to forget obsolete control information about already delivered messages. They do not scale in large and dynamic systems. In this paper, we propose a novel implementation of causal broadcast. We prove that all and only obsolete control information is safely removed, at cost of a few lightweight control messages. The local space complexity of this protocol does not monotonically increase and depends at each moment on the number of messages still in transit and the degree of the communication graph. Moreover, messages only carry a scalar clock. Our implementation constitutes a sustainable communication primitive for causal broadcast in large and dynamic systems.

***Keywords:*** *Causal broadcast, complexity trade-off, large and dynamic systems*

## 1 INTRODUCTION

Causal broadcast constitutes the core communication primitive of many distributed systems [9]. Applications such as distributed social networks [3], distributed collaborative software [1, 1], or distributed data stores [2, 4, 6, 1, 2] use causal broadcast to ensure consistency criteria. Causal broadcast ensures reliable receipt of broadcast messages, exactly-once delivery, and causal delivery following Lamport's happen before relationship [1]. When Alice comments on Bob's picture, nobody sees Alice's comment without Bob's picture, and nobody sees multiple occurrences of Alice's comment or Bob's picture.

Vector clock-based approaches [1, 1, 2] need to keep all their control information forever. They cannot forget any control information. The consumed memory monotonically increases with the number of processes that ever broadcast a message $O(N)$. They become unpractical in large and dynamic system comprising from hundreds to millions of processes joining, leaving, self-reconfiguring, or crashing at any time.

In this paper, we propose a novel implementation of causal broadcast that forgets all and only obsolete control information. A process $p$ that has $i$ incoming links receives each message $i$ times. A message $m$ is active for $p$ between its first and last reception by $p$. Process $p$ keeps control information about all its active messages. As soon as a message becomes inactive, the process can forget all control information related to it. Consequently, processes do no store any permanent control information about messages. When no message is active, no control information is stored in the system. Our contribution is threefold:

- We define the notion of *link memory* as a mean for each process to forbid multiple delivery. Link memory allows each process to identify processes from which it will receive a copy of an already delivered message. This allows each process to safely remove obsolete control information about broadcast messages that will never be received again. We prove that using causal delivery, each process can build such knowledge even in dynamic systems where processes may join, leave, self-reconfigure, or crash at any time.

- We propose an implementation of causal broadcast that uses the notion of link memory, where each process manages a local data the size of which is $O(i \cdot A)$ where $i$ is the number of incoming links and $A$ is the number of active messages. Moreover, the only control information piggybacked on messages is a scalar Lamport clock.

- We evaluate our implementation using large scale simulations. The experiments highlight the space consumed and the traffic generated by our protocol in dynamic systems with varying latency. The results confirm that the proposed approach scales with system settings and use.

The rest of this paper is organized as follows. Section 2 describes the model, highlights the issue, introduces the principle solving the issue, provides an implementation solving the issue along with its complexity analysis. Section 3 shows the experiments. Section 4 reviews related work. We conclude and discuss about perspectives in Section 5.

## 2 PROPOSAL

In this section, we present a causal broadcast protocol providing a novel trade-off between speed, memory, and traffic. Among others, it safely removes obsolete control information about broadcast messages. Its memory consumption increases and decreases over receipts. The key ideas are: (1) Every process broadcasts or forwards a message once, hence every link carries a message once. A process expects to receive as many copies of a message as its number of links. Once a process received all expected copies, it can forget about this broadcast message, i.e., it can safely remove the control information associated to this broadcast message. (2) Adding links between pair of processes adds uncertainty. The receiver cannot state any longer when it should expect a message from a link. (3) By exploiting causal order, processes remove this uncertainty. Causal order allows processes to remove batches of obsolete information while reasoning about temporarily buffered broadcast messages.

### 2.1 Model

A distributed system comprises a set of processes that can communicate with each other using messages. Processes may not have the knowledge of all processes in the system. Instead, processes build and maintain overlay networks: each process updates a local partial view of logical communication links, i.e., a set of processes to communicate with. The partial view is usually much smaller than the actual system size. We use the terms of overlay networks, and distributed systems interchangeably.

**Definition 1** (Overlay network)**.** *An overlay network $G = (P, E)$ comprises a set of processes $P$ and a set of directed links $E \subseteq P \times P$. An overlay network is static if both sets $P$ and $E$ are immutable. Otherwise, the overlay network is dynamic. An overlay network is strongly connected if there exists a path – i.e. a link or an sequence of links – from any process to any other process. We only consider strongly connected overlay networks.*

**Definition 2** (Process)**.** *A process runs a set of instructions sequentially. Processes communicate with each other using asynchronous message passing. A process A can send a message to a process B $s_{AB}(m)$, or to any process $s_A(m)$; receive a message from a process B $r_{AB}(m)$, or from any process $r_A(m)$. A process sends messages using the set of links departing from it, called out-view $Q_o$. Processes reachable via these links are called neighbors. A process receives messages from the set of links arriving to it, called in-view $Q_i$. Processes are faulty if they crash, otherwise they are correct. We do not consider Byzantine processes.*

Causal broadcast ensures properties similar to those of reliable broadcast. Each process may receive each broadcast message multiple times but delivers it once. In this paper, we tackle the issue of implementing these properties.

**Definition 3** (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 $d_B(m)$. Uniform reliable broadcast guarantees 3 properties: (1) Validity: If a correct process broadcasts a message, then it eventually delivers it. (2) Uniform Agreement: If a process – correct or not – delivers a message, then all correct processes eventually deliver it. (3) Uniform Integrity: A process delivers a message at most once, and only if it was previously broadcast.*

In static systems, implementing these properties only requires a local structure the size of which grows and shrinks over receipts [2]. Every process knows the number of copies of each delivered message it should expect. When this number drops down to 0, the process safely removes the control information associated to the delivered messages. This forbids multiple delivery for the process will never receive – hence deliver – this message again.

However, in dynamic systems where processes join, leave, or self-reconfigure their out-view at any time, the removal of a link may impair the consistency of the number of expected messages. Either processes cannot safely garbage collect obsolete control information, or processes suffer from multiple delivery.

In this paper, we solve this issue by exploiting causal broadcast's ability to ensure a specific order on message delivery. To characterize the order among events such as send, or receive, we define time in a logical sense using Lamports definition.

**Definition 4** (Happens-before [1])**.** *The happens-before relationship defines a strict partial order of events. The happens-before relationship $\rightarrow$ is transitive ($e_1 \rightarrow e_2 \wedge e_2 \rightarrow e_3 \implies e_1 \rightarrow e_3$), irreflexive ($e_1 \not\rightarrow e_1$), and antisymmetric ($e_1 \rightarrow e_2 \implies e_2 \not\rightarrow e_1$). The sending of a message always precedes its receipt $s_{AB}(m) \rightarrow r_{BA}(m)$. Two messages are concurrent if none happens before the other ($r_A(m_1) \not\rightarrow s_A(m_2) \wedge r_A(m_2) \not\rightarrow s_A(m_1)$).*

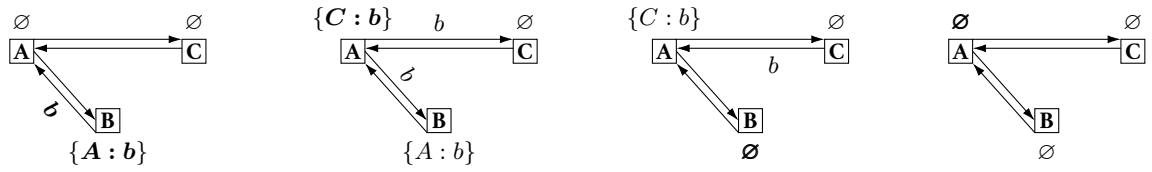
**Definition 5** (Causal order)**.** *The delivery order of messages follows the happen before relationships of the corresponding broadcasts. $d_A(m) \rightarrow b_A(m') \implies d_B(m) \rightarrow d_B(m')$*

**Definition 6** (Causal broadcast)**.** *Causal broadcast is a uniform reliable broadcast ensuring causal order.*

### 2.2 Link memory

We define link memory as a mean for processes to forbid multiple delivery while safely removing obsolete control information. Processes attach control information about expected messages to each link of their respective in-view.

**Definition 7** (Link memory)**.** *Assuming a link $(A, B)$, Process B remembers among its delivered messages those that*




(a) Process B broadcasts $b$ and awaits a copy of $b$ from Process A.

(b) Process A receives, delivers, and forwards $b$. It expects a copy from Process C.

(c) Process B and Process C receive $b$. They do not expect other copies. Process C delivers and forwards $b$.

(d) Process A receives its last expected copy of $b$. The message $b$ is completely removed from the system.

Figure 1: Link memory allows to safely remove obsolete control information in static systems.

*it will receive from this link; and forgets among its delivered messages those that it will never receive from this link.* $remember_{BA}(m) \equiv d_B(m) \wedge \neg r_{BA}(m)$

**Theorem 1** (Link memory forbids multiple delivery)**.** *A process that delivers only messages it does not remember using link memory delivers each broadcast message exactly once.*

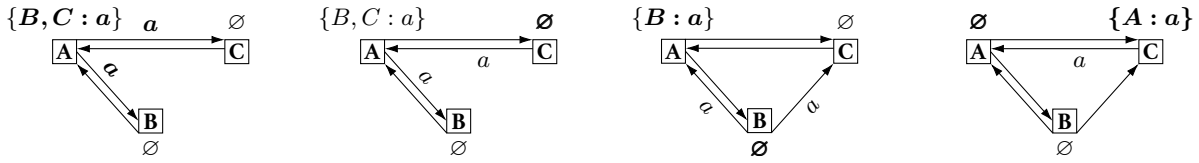
*Proof.* We must show that, for any message $m$, its receipt cannot lead to its delivery if a copy of $m$ has already been delivered before: $\nexists m,\ d_B(m) \rightarrow r_{BA}(m) \wedge r_{BA}(m) \rightarrow d_B(m)$.
The delivery $d_B(m)$ implies a prior receipt $r_B(m) \rightarrow d_B(m)$. Assuming that each link carries each messages once, and this receipt comes from link $(A,\ B)$, then Process B cannot receive, hence deliver, $m$ from $(A,\ B)$ again: $\nexists m,\ r_{BA}(m) \rightarrow d_B(m) \wedge d_B(m) \rightarrow r_{BA}(m) \wedge r_{BA}(m) \rightarrow d_{BA}(m)$.
If this receipt comes from any other link $(C,\ B)$, $C \neq A$, Process B remembers $m$ on other links, and among others, on link $(A,\ B)$: $\forall m,\ r_{BC}(m) \rightarrow remember_{BA}(m)$. Assuming that each link carries each message once, Process B eventually receives a copy of $m$ from link $(A,\ B)$: $remember_{BA}(m) \rightarrow r_{BA}(m)$. Since remembering a message forbids its delivery, this cannot lead to another delivery of $m$: $\nexists m,\ r_{BA}(m) \rightarrow d_B(m)$.
Finally, since every process delivers, hence forwards each message once, each link carries each messages once. □

Algorithm 1 shows a set of instructions that implements causal broadcast for static systems. It uses reliable FIFO links to ensure causal order [7], and implements link memory to forbid multiple delivery. Every process maintains a local structure the size of which increases and decreases over receipts. The first receipt of a broadcast message from a link tags the other links (see Line 15). The receipt on other links of this broadcast message removes the corresponding tag (see Line 16). Figure 1 depicts its functioning in a system comprising 3 processes. In Figure 1a, Process B broadcasts $b$. It awaits a copy of $b$ from the only link in its in-view. In Figure 1b, Process A receives $b$. It delivers it, for no link in its in-view is tagged with $b$, meaning this is a first receipt. It tags the other link in its in-view with $b$ and forwards $b$ to its out-view. In Figure 1c, Process B receives the awaited copy of $b$ from Process A. It removes the corresponding entry. The broadcast protocol at Process B does not consume space anymore. Process C receives $b$. It detects a first receipt so it delivers and forwards $b$. It does not tag any link, for the only link from its in-view is the link from which it just received $b$. In Figure 1d, the last process to await a copy of $b$ finally receives it. None of processes remembers about $b$. No copy of $b$ travels in the system. This implementation forbids multiple delivery in static systems while safely removing obsolete control information.

**Algorithm 1:** Causal broadcast for static systems.

```
1  Q_o                                                  // Out-view
2  Q_i                                                  // In-view
3  E ← ∅          // Map of expected messages Q_i : M*
4  DISSEMINATION:
5      function C-broadcast(m)
6          receive(m, _)
7      upon receive(m, l)
8          if ¬received(m, l) then
9              foreach q ∈ Q_o do sendTo(q, m)
10             C-deliver(m)
11     function received(m, l)
12         rcvd ← ∃q ∈ E with m ∈ E[q]
13         if ¬rcvd then
14             foreach q ∈ Q_i do
15                 E[q] ← E[q] ∪ m
16         E[l] ← E[l] \ m
17         return rcvd
```

However, implementing link memory becomes more challenging in dynamic systems where processes can start sending messages to any other process at any time. Any process can receive an already forgotten message from any other process. Figure 2 illustrates the issue. In Figure 2a, Process A broadcasts $a$. It expects a copy from both Process B and Pro-




(a) Process A broadcasts $a$ and expects a copy from both Process B and Process C.

(b) Process C receives, delivers, and forwards $a$. It does not expect additional copies.

(c) Process B adds a link to Process C. Then it receives, delivers, and forwards $a$.

(d) Process C receives and mistakes $a$ for a new message. It delivers, and forwards $a$.

Figure 2: Causal broadcast (Algorithm 1) fails to forbid multiple delivery in dynamic systems.

cess C. In Figure 2b, Process C immediately receives, delivers, and forwards $a$. It does not tag any link and expects to never receive this message again. However, network condition delays the receipt of $b$ from Process B. In Figure 2c, Process B adds a communication link towards Process C. Then it receives, delivers, and forwards $b$. Since Process C now belongs to its out-view, the forwarding includes Process C. In Figure 2d, Process C receives $a$ again. However, it did not keep control information about this message. It mistakes it for a first receipt. It delivers and forwards $a$. Not only Process C suffers multiple delivery but this has cascading effects over the whole system.

The rest of this section describes how causal broadcast can exploit causal order to initialize link memory, an implementation of such broadcast, and its complexity analysis.

## 2.3 Link memory for dynamic systems

This section demonstrates that causal broadcast can use causal order to initialize link memory, thereby enabling the use of link memory in dynamic systems.

Algorithm 1 already implements the maintenance of link memory over receipts. Every process safely removes obsolete control information over receipts. However, Figure 2 highlights that new links lack of consistent initialization. The challenge consists in initializing such memory without history of past messages. Causal broadcast starts to build the knowledge on-demand, i.e., when a process wants to add a link to another process. The protocol disables the new link until initialized. This initialization requires round-trips of control messages and message buffering. Causal broadcast takes advantage of causal order to provide guarantees on messages included in buffers.

Figure 3 depicts the principle of the approach. When a Process A adds a link to Process B, Process A notifies Process B using a control message $\alpha$. This control message $\alpha$, as all control messages that will follow ($\beta$, $\pi$, $\rho$), must be delivered after all its preceding messages. Hence, at receipt, Process B implicitly removes obsolete information: messages delivered by Process A before the sending of the notification $\mathcal{A}_1$. At receipt of $\alpha$, Process B can start gathering control information about its delivered messages in a buffer $B_\alpha$. Among other, Process B wants to identify messages concurrent to the correct establishment of the new link. Process B acknowledges Process A's notification using a control message $\beta$. At receipt of $\beta$, Process A removes obsolete information: messages delivered by Process B before the sending of the acknowledgment $\mathcal{A}_1 \cup \mathcal{B}_1$. This solves the issue identified in Figure 2, for $a$ would belong to $\mathcal{A}_1$ or $\mathcal{B}_1$. However, this is not sufficient to initialize link memory. Process A sends a control message $\pi$ to Process B, and starts to gather control information about its delivered messages in a buffer $B_\beta$. Upon receipt of $\pi$, Process B closes its first buffer $B_\alpha$.

**Lemma 1** (Messages in buffer $B_\alpha$)**.** *The buffer $B_\alpha$ contains messages delivered by Process B after the sending of $\beta$ and before the receipt of $\pi$.*
*This includes all messages delivered by Process A before the sending of $\pi$ that were not delivered by Process B before the sending of $\beta$: $\mathcal{A}_2$. Above all, this also includes all messages delivered by Process B that were not delivered by Process A at the sending of $\pi$: $\mathcal{B}_2$.*

*Proof.* Since control messages are delivered after preceding messages, all broadcast messages delivered by Process A before the sending of $\alpha$ precede the buffering: $\forall m,\ d_A(m) \rightarrow s_A(\alpha_{AB}) \implies m \notin B_\alpha$. Since messages are delivered once, $\forall m,\ d_A(m) \rightarrow s_A(\pi_{AB}) \wedge d_B(m) \rightarrow s_B(\beta_{AB}) \implies m \notin B_\alpha$. This removes $\mathcal{A}_1$ and $\mathcal{B}_1$.
The buffer $B_\alpha$ contains the rest of messages delivered by Process B before the receipt of $\pi$. This includes messages delivered by Process A between the sending of $\alpha$ and $\pi$ but not delivered by Process B before the sending of $\beta$ ($\mathcal{A}_2$); and messages delivered by Process B but not delivered by Process A before the sending of $\pi$ ($\mathcal{B}_2$). □

Upon receipt of $\pi$, Process B continues to gather control information about its delivered messages in another buffer $B_\pi$. Some messages in this buffer will be expected from Process A, but Process B cannot determine which ones just yet. It sends the last acknowledgment $\rho$ to Process A. Upon receipt of this acknowledgment, Process A closes its buffer and sends it using the new link. Afterwards, Process A uses the

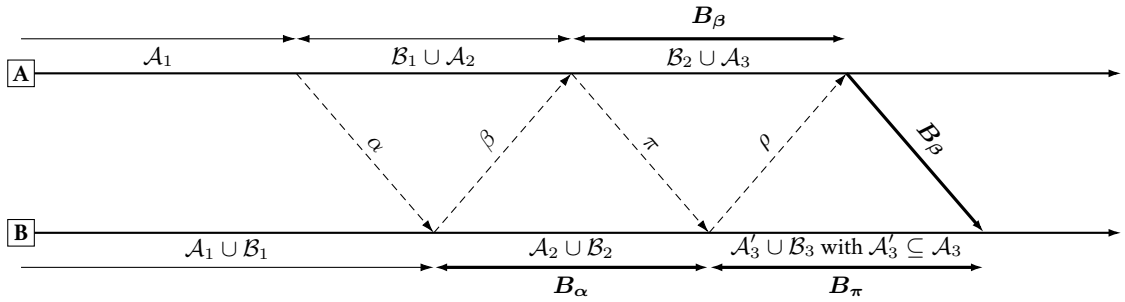



Figure 3: Initializing the link memory from Process A to Process B. Control messages $\alpha$, $\beta$, $\pi$, and $\rho$ are delivered after all preceding messages while $B_\beta$ is not. At receipt of $B_\beta$, Process B classifies the messages: $B_\beta \cap (B_\alpha \cup B_\pi) = \mathcal{B}_2 \cup \mathcal{A}'_3$ are messages to ignore; $B_\beta \setminus B_\alpha \setminus B_\pi = \mathcal{A}_3 \setminus \mathcal{A}'_3$ are messages to deliver; $B_\pi \setminus B_\beta = \mathcal{B}_3$ are messages to expect from Process A.

new link for causal broadcast, for it knows that Process B will receive $B_\beta$ before upcoming broadcast messages on this new link, and the receipt of $B_\beta$ will allow Process B to initialize this new link memory.
Upon receipt of $B_\beta$, Process B stops buffering in $B_\pi$.

**Lemma 2** (Messages in buffer $B_\beta$)**.** *The buffer $B_\beta$ contains messages delivered by Process A after the sending of $\pi$ and before the receipt of $\rho$.*
*This includes all messages delivered by Process B before the sending of $\rho$ that were not delivered by Process A before the sending of $\pi$: $\mathcal{B}_2$. This also includes all messages delivered by Process A that were not delivered by Process B at the sending of $\rho$: $\mathcal{A}_3$.*

*Proof.* The proof is similar to that of Lemma 9. Control messages shift roles. $\pi$ becomes $\rho$; $\beta$ becomes $\pi$; $\alpha$ becomes $\beta$. □

**Lemma 3** (Messages in buffer $B_\pi$)**.** *The buffer $B_\pi$ contains messages delivered by Process B after the sending of $\pi$ and before the receipt of $B_\beta$.*
*This may includes messages delivered by Process A before the sending of $B_\beta$ that were not delivered by Process B before the sending of $\rho$: $\mathcal{A}'_3$. This also includes all messages delivered by Process B that were not delivered by Process A at the sending of $B_\beta$: $\mathcal{B}_3$.*

*Proof.* The proof is similar to that of Lemmas 10 and 11. The difference being that $B_\beta$ is not necessarily delivered after preceding messages. Hence, the receipt of $B_\beta$ follows the sending of $\rho$ but Process B cannot state if it received all, part, or none of messages in $\mathcal{A}_3$. Thus, $\mathcal{A}'_3 \subseteq \mathcal{A}_3$. □

Using $B_\alpha$, $B_\beta$, and $B_\pi$ buffers, Process B identifies messages in $B_\beta$ it must deliver against messages it must ignore, and messages in $B_\pi$ it must receive from Process A. This allows Process B to initialize link memory.

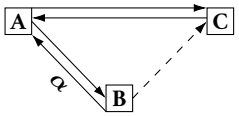
**Theorem 2** ($B_\alpha$, $B_\beta$, and $B_\pi$ initialize link memory)**.** *A process consistently initializes link memory at receipt of $B_\beta$ using $B_\alpha$ and $B_\pi$.*

*Proof.* From Lemma 9, $B_\alpha = \mathcal{A}_2 \cup \mathcal{B}_2$. From Lemma 10, $B_\beta = \mathcal{B}_2 \cup \mathcal{A}_3$. From Lemma 11, $B_\pi = \mathcal{A}'_3 \cup \mathcal{B}_3$.
First, we must show that Process B delivers all and only messages from $B_\beta$ it did not deliver yet: $m \in \mathcal{A}_3 \setminus \mathcal{A}'_3$.
Since $B_\beta \setminus B_\alpha \setminus B_\pi = (\mathcal{B}_2 \cup \mathcal{A}_3) \setminus (\mathcal{A}_2 \cup \mathcal{B}_2) \setminus (\mathcal{A}'_3 \cup \mathcal{B}_3) = \mathcal{A}_3 \setminus (\mathcal{A}'_3 \cup \mathcal{B}_3)$. Since $\mathcal{B}_3 \cap \mathcal{A}_3 = \varnothing$, we have $B_\beta \setminus B_\alpha \setminus B_\pi = \mathcal{A}_3 \setminus \mathcal{A}'_3$.
Second, we must show that Process B initializes the new link memory with all and only messages from $B_\pi$ that Process A did not deliver at the sending of $B_\beta$: $m \in \mathcal{B}_3$.
$B_\pi \setminus B_\beta = (\mathcal{A}'_3 \cup \mathcal{B}_3) \setminus (\mathcal{B}_2 \cup \mathcal{A}_3)$. Since $\mathcal{B}_3 \cap \mathcal{B}_2 = \varnothing$ and $\mathcal{A}'_3 \subseteq \mathcal{A}_3$, $B_\pi \setminus (B_\beta \setminus B_\alpha) = \mathcal{B}_3$. □

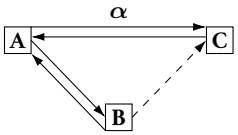
## 2.4 Implementation

PRC-broadcast stands for Preventive Reliable Causal broadcast. It prevents both causal order violations and multiple delivery by using all and only links that are safe [2], and the memory of which is correctly initialized and maintained. PRC-broadcast ensures that control messages are delivered after all their preceding messages by sending them on reliable FIFO links used for causal broadcast. PRC-broadcast uses a local structure the size of which increases and decreases over receipts. Every process safely removes obsolete control information about past broadcast messages.

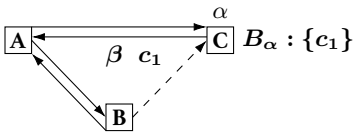
Algorithm 2 shows the instructions of PRC-broadcast. Figure 4 illustrates its operation in a scenario involving 3 processes. In this example, Process B adds a link to Process C. Process B disables the new link for causal broadcast until it is safe and guaranteed that Process C correctly initialized its memory.
Process B sends a first control message $\alpha$ to Process B using safe links (see Line 17 and Figure 4a).
After being routed to Process C by intermediary processes (see Figure 4b), $\alpha$ reaches Process C (see Figure 4c). Process C starts to register messages it delivers in a buffer $B_\alpha$. Process C acknowledges the receipt of $\alpha$ by sending a second control message $\beta$ to Process B using safe links (see Line 25).



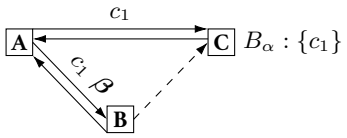

(a) Process B adds a link to Process C. PRC-broadcast ensures its safety. Process B sends a first control message $\alpha$ to Process C using Process A as mediator.



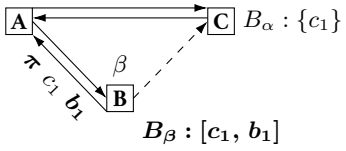

(b) Process A receives $\alpha$ and routes it to Process C.



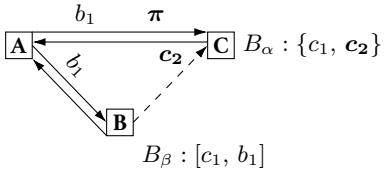

(c) Process C receives $\alpha$ and answers by sending $\beta$ to Process B using Process A as mediator. Then, Process C broadcasts $c_1$ and registers it in $B_\alpha$.



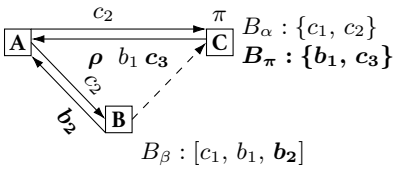

(d) Process A receives $\beta$ and routes it to Process B. Process A receives $c_1$ and forwards it to both its neighbors.



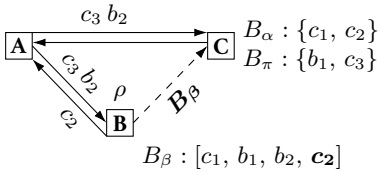

(e) Process C receives and discards $c_1$. Process B receives $\beta$ and replies $\pi$ to Process C using Process A as mediator. Process B receives $c_1$ and forwards it to its neighbor. Process B broadcasts $b_1$. It registers $c_1$ and $b_1$ in $B_\beta$.



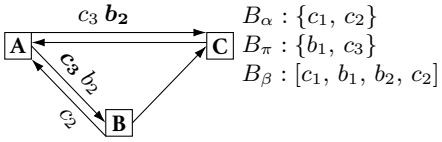

(f) Process A receives $c_1$ and discards it. Process A receives $\pi$ and routes it to Process C. Process A receives $b_1$ and forwards it to its neighbors. Process C broadcasts $c_2$ and registers it in $B_\alpha$.



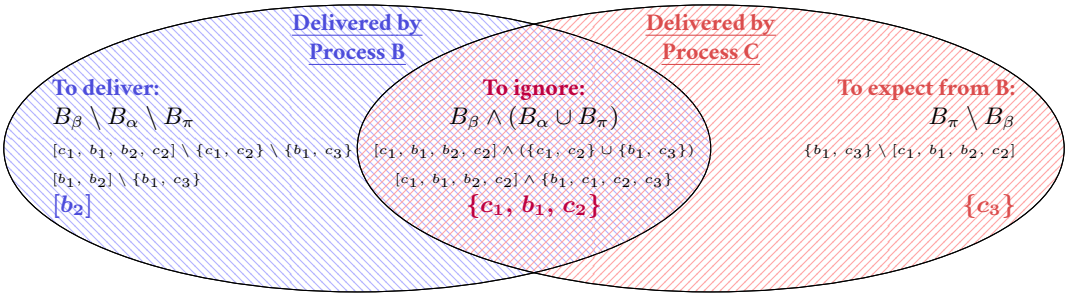

(g) Process A receives $c_2$ and forwards it to its neighbors. Process B broadcasts $b_2$ and registers it in $B_\beta$. Process C receives $\pi$ and replies $\rho$ to Process B using Process A as mediator. Then it receives and forwards $b_1$. Then it broadcasts $c_3$. It registers $b_1$ and $c_3$ in $B_\pi$.




(h) Process A receives and discards $b_1$. Process A receives and routes $\rho$ to Process B. Process A receives and forwards $b_2$ then $c_3$. Process B receives, forwards, and registers $c_2$. Then Process B receives $\rho$ and sends $B_\beta$ to Process C using the new link.




(i) Once Process A sent $B_\beta$, the new link is safe. Process C receives $B_\beta$. Process C does not deliver $c_1$, $b_1$ and $c_2$, for it already delivered them. Process C delivers $b_2$ and expects another copy from Process A, for it constitutes a new message. Process C expects to eventually receive $c_3$ from Process B.




(j) Process C categorizes each message of $B_\beta$ and $B_\pi$.

Figure 4: Using buffers and control messages, PRC-broadcast provides reliable causal broadcast.

**Algorithm 2:** PRC-broadcast at Process $p$.

```
1  B ← ∅                                          // @Sender Map of buffers Q_o : M*
2  S ← ∅                                          // @Receiver Map of buffers Q_i : M* × M* × bool
3  DISSEMINATION:
4      function PRC-broadcast(m)
5          C-broadcast(m)
6      upon C-deliver(m)
7          buffering(m)
8          PRC-deliver(m)
9      function buffering(m)
10         foreach q ∈ B do B[q] ← B[q] ∪ m
11         foreach ⟨B_α, B_π, received_π⟩ ∈ S do
12             if received_π then B_π ← B_π ∪ m
13             else B_α ← B_α ∪ m
14 LINK MEMORY:
       @Sender                                    @Receiver
15     upon open_o(to)                            upon open_i(from)
16         Q_o ← Q_o \ to                         18
17         send-α(p, to)                          19     Q_i ← Q_i \ from
20     upon receive-β(from, to)                   23     upon receive-α(from, to)
21         B[to] ← ∅                              24     S[from] ← ⟨∅, ∅, false⟩
22         send-π(from, to)                       25     send-β(from, to)
26     upon receive-ρ(from, to)                   30     upon receive-π(from, to)
27         send-B_β(from, to, B[to])              31     ⟨B_α, B_π, _⟩ ← S[from]
28         B ← B \ to                             32     S[from] ← ⟨B_α, B_π, true⟩
29         Q_o ← Q_o ∪ to                         33     send-ρ(from, to)
                                                  34     upon receive-B_β(from, to, B_β)
                                                  35     ⟨B_α, B_π, _⟩ ← S[from]
                                                  36     S ← S \ from
                                  filter messages to ignore →
                                                  37     foreach m ∈ B_β \ B_α \ B_π do
                                        to deliver →
                                                  38         receive(m, from)
                                   to expect →
                                                  39     E[from] ← B_π \ B_β
                                                  40     Q_i ← Q_i ∪ from
41     upon close_o(to)                           43     upon close_i(from)
42         B ← B \ to                             44     S ← S \ from
                                                  45     E ← E \ from
```

In Figure 4c, Process C broadcasts $c_1$ and registers it in $B_\alpha$. After being routed to Process B (see Figure 4d), $\beta$ reaches Process B. Process B starts to register messages it delivers in a buffer $B_\beta$. Process B sends a third control message $\pi$ to Process C using safe links (see Line 22). In Figure 4e, Process B delivers $c_1$ then broadcasts $b_1$. It registers them in $B_\beta$. In Figure 4f, Process C broadcasts $c_2$ and registers it in $B_\alpha$. After being routed to Process C by intermediary processes (see Figure 4f), $\pi$ reaches Process C. Process C ends its first buffer $B_\alpha$. Process C starts to register messages it delivers in $B_\pi$. Process C sends a fourth and last control message $\rho$ to Process B using safe links (see Line 33). In Figure 4g Process C delivers $b_1$, broadcasts $c_3$, and registers them in $B_\pi$. In the meantime, Process B broadcasts $b_2$ and registers it in $B_\beta$.

After being routed to Process B, $\rho$ reaches Process B (see Figure 4h). Process B stops buffering and sends its buffer of messages $B_\beta$ using the new link $s_{BC}(B_\beta)$ (see Line 27). In Figure 4i, this buffer contains $b_1$, $b_2$, and $c_2$. The new link is safe. Process B starts to use this link normally for causal broadcast using Algorithm 1.

Once Process C receives the buffer, it ends its buffer $B_\pi$ (see Figure 4i). Using $B_\alpha$, and $B_\pi$, Process C identifies among messages from $B_\beta$ the array of messages to deliver (see Line 38). In Figure 4j, this array only includes $b_2$. Using $B_\alpha$, and $B_\pi$, Process C also identifies the set of messages to ignore which is the rest of the buffer. In Figure 4j, this set includes $c_1$, $b_1$ and $c_2$. Finally, Process C identifies among its own delivered messages the messages to expect from Process B (see Line 39). In Figure 4j, this set includes $c_3$. This set constitutes the memory of the new safe link. Afterwards, messages received by this new link are processed normally.

PRC-broadcast builds link memory using control messages that acknowledges the delivery of preceding messages. Every process safely removes obsolete control information about past broadcast messages. The size of the local structure increases and decreases over receipts. In the next section, we analyze the complexity of this causal broadcast implementation.

### 2.5 Complexity

In this section, we analyze the complexity of PRC-broadcast in terms of broadcast message overhead, delivery execution time, local space consumption, and number of control messages.

The **broadcast message overhead** is constant $O(1)$. The protocol uses reliable FIFO links to transmit messages.

The **delivery execution time**, i.e., the time complexity of the receipt function is $O(|Q_i|)$. The protocol checks and updates control information associated to each link in the in-view $Q_i$. The size of in-views can be much smaller than the number of processes in the system $|P|$. For instance, peer-sampling approaches [8, 2] provides every process with an in-view the size of which is logarithmically scaling with the number of processes $O(\ln(|P|))$.

The **local space consumption** depends on the size of buffers and the size of the in-view. Each link in the in-view has its buffer of control information about messages. A message appears in the structure after its first receipt and disappears at its last receipt. So the local space complexity is $O(|Q_i| \cdot M)$ where $M$ is the number of messages already delivered that will be received again from at least a link in the in-view $Q_i$. The local space consumption depends on system settings (e.g. processes do not consume space when the system topology is a ring or a tree) and use (e.g. processes do not consume space when no process broadcasts any message).

The overhead in terms of **number of control messages** per added link in an out-view varies from 6 to $4 \cdot |P|^2$ depending on the overlay network; $P$ being the set of processes currently in the system. It achieves 6 messages when Process A adds Process B using Process C as mediator, and Process B has Process A in its out-view. It achieves 8 control messages when peer-sampling protocols build out-views using neighbor-to-neighbor interactions [1, 2]. It achieves $O(4 \cdot \log(|P|))$ control messages when peer-sampling protocols allows processes to route their messages [1, 2]. It achieves $O(4 \cdot |P|^2)$ control messages when Process A adds Process B without knowledge of any route. Process A and Process B fall back to reliable broadcast instead of routing to disseminate control messages.

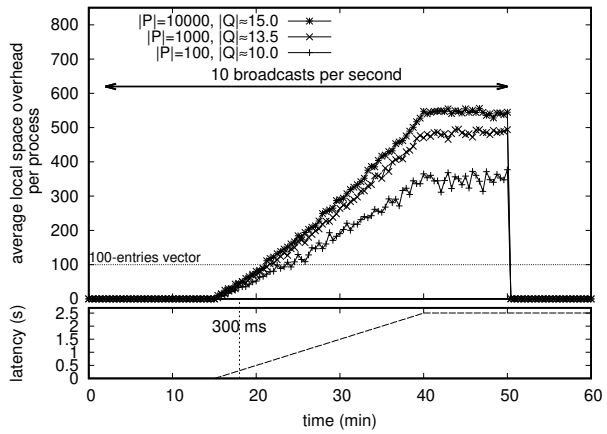
This complexity analysis shows that PRC-broadcast proposes a novel trade-off in terms of complexity. In systems allowing a form of routing, processes only send a few control messages to handle dynamicity. Every process maintains a local structure the size of which increases and decreases over receipts. Every process safely removes obsolete control information about past messages. It constitutes an advantageous trade-off that depends on the actual system settings and use instead of past deliveries. The next section describes an experiment highlighting the effects of the system settings on the space consumed by processes.

## 3 EXPERIMENTATION

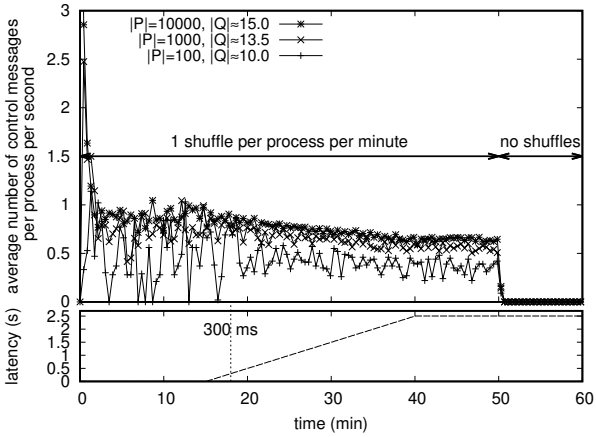
PRC-broadcast proposes a novel trade-off between speed, memory, and traffic. Most importantly, its space consumed varies over receipts. In this section, we evaluate the impact of the actual system on the space consumed and traffic generated by processes. The experiments run on the PeerSim simulator [1] that allows to build large and dynamic systems. Our implementation is available on the Github platform at `http://github.com/chat-wane/peersim-prcbroadcast`.

**Objective:** To confirm that local space complexity depends on in-views and message receipts.

**Description:** We measure the average size of buffers and arrays of expected messages. This constitutes the average local space overhead consumed by PRC-broadcast to detect and forbid multiple delivery in dynamic systems.
Runs involve 3 overlay networks comprising 100, 1k, and 10k processes. Spray [2] builds a highly dynamic overlay networks. The resulting topology has properties close to those of random graphs such as low diameter, or low clustering coefficient. Such systems are highly resilient to random crashes, and allows processes to balance the load of the traffic generated by broadcasting. Each process maintains an out-view logarithmically scaling with the number of processes in the system. Each process of the 100-processes system has an out-view of $\approx 10$ neighbors. Each process of the 1k-processes system has an out-view of $\approx 13.5$ neighbors. Each process of the 10k-processes system has a out-view of $\approx 15$ neighbors. Each process dynamically reconfigures its out-view: it gives half of its correctly initialized links to a chosen neighbor; the latter gives half of its correctly initialized links to the former as well. Each exchange leads to link memory initialization and safety checks of the new links, and removal of given links. Each process starts to reconfigures its out-view as soon as it joins the system and reconfigures its out-view every minute. This uniformly spreads reconfigurations over the duration of the experiment.
Links are bidirectional, their safety must be checked in both directions but the overhead remains minor. Since the peer-sampling protocol that builds the system uses neighbor-to-neighbor communication to establish new links, each control message is two hops away from its destination. Overall, a new link requires 8 control messages to be initialized properly. Links have transmission delay, i.e., the time between the sending of a message and its receipt is not null. The experiments start with 1 millisecond transmission delay. At 15 minutes, the delay starts to increase. At 17 minutes, links reach 300 milliseconds transmission delay. At 40 minutes, links reach 2.5 seconds transmission delay and it stops increasing.
From 2 minutes to 50 minutes, every second, 10 processes chosen uniformly at random among all processes broadcast a message.




(a) Local space overhead (number of control information about broadcast messages in all buffers).




(b) Generated traffic overhead (number of control messages transiting in the system including routed messages).

Figure 5: Overhead of PRC-broadcast required to ensure causal order and forbid multiple delivery in dynamic systems with varying latency.

**Results:** Figure 5a shows the results of this experiment. The x-axis denotes the time in minute. The top part of the figure shows the local space overhead while the bottom part of the figure shows the evolution of transmission delays.

Figure 5a confirms that the local space consumption depends on the in-view size. Systems with larger in-views consume more space. Each new delivered message adds control information on each link of the in-view (see Algorithm 1).

Figure 5a confirms that the local space consumption depends on network condition. The overhead increases as the latency increases. Latency increases the time between the first and the last receipt of each message. Processes store messages longer until their safe removal.

Figure 5a confirms that the local space consumption depends on broadcast messages. When processes stops broadcasting, the space consumed at each process drops to 0. Each process eventually receive each message and safely remove the corresponding entry.

Figure 5a shows that at a rate of 10 broadcasts per second and when latency stays under a realistic bound (300 milliseconds), the overhead is lower than vector-based approaches. Whatever system conditions, it would require a vector of 100, 1k entries, 10k entries to forbid multiple delivery in the 100-processes system, 1k-processes system, 10k-processes system respectively. However, it is worth noting that the overhead of PRC-broadcast increases linearly with the number of messages currently transiting. 100 broadcasts per second would multiply measurements made on PRC-broadcast by a factor of 10. In such case, the 100-entries vector would be better than PRC-broadcast even under a latency of 300 milliseconds.

PRC-broadcast provides a novel trade-off between speed, memory, and traffic. Among other, its space consumed increases and decreases depending on the system and its current use; instead of past use (see Section 4. This result means that it constitutes an advantageous trade-off in (i) dynamic systems (ii) comprising up to millions of processes (iii) that could broadcast at any time.

**Objective:** To confirm that the generated traffic overhead depends on the dynamicity of the system.

**Description:** We measure the average number of control messages received by each process during a second. This includes the routing of messages. The setup is identical to that of prior experiment.

**Results:** Figure 5b shows the results of this experiment. The top part of the figure depicts the traffic overhead generated by PRC-broadcast while the bottom part of the figure depicts the evolution of transmission delays.

Figure 5b shows that the number of control messages received by processes depends on the dynamicity of the system. The more dynamic the higher the traffic overhead. At the beginning of the experiment, processes join the system. Numerous links are established at once, hence the high number of control messages. Then processes shuffle their out-view during 50 minutes. The number of links to add and remove is roughly constant over time, hence the stabilization in number of control messages. Finally, processes stop shuffling at 50 minutes. Processes do not receive additional control messages.

Figure 5b confirms our traffic overhead complexity analysis. For instance, in the 10k-processes system, views comprises 15 processes which belong half from the out-view and half from the in-view. Each process shuffles every

Table 1: Complexity of broadcast algorithms at each process. $N$ the number of processes that ever broadcast a message. $P$ is the set of processes in the system. $w$ the number of messages received but not delivered yet. $Q_i$ is the set of incoming links. $M$ is the set of messages already delivered that will be received again from at least one link in $Q_i$.

| | message overhead | delivery execution time | local space consumption | # control messages per added link |
|---|---|---|---|---|
| reliable broadcast [9] | $O(1)$ | $O(1)$ | $O(N)$ | 0 |
| causal broadcast [2] | $O(N)$ | $O(W \cdot N)$ | $O(N + W \cdot N)$ | 0 |
| preventive broadcast [2] | $O(1)$ | $O(1)$ | $O(N)$ | 3 to $2 \cdot \|P\|^2$ |
| **this paper** | $\mathbf{O(1)}$ | $\mathbf{O(\|Q_i\|)}$ | $\mathbf{O(\|Q_i\| \cdot M)}$ | **6** to $\mathbf{4 \cdot \|P\|^2}$ |

minute. Each shuffle adds and removes 7.5 links (twice half of the out-view size). Since the peer-sampling protocol establishes links using neighbor-to-neighbor interactions, it allows a form of routing where only 8 control messages are required to initialize a new link. $|exchanged\_links| * |control\_messages|/60 \approx 7.5 * 8/60 \approx 1$ control message per second.

Figure 5b shows that latency smooth and decreases the number of control messages. The peer-sampling protocol only shuffles links already safe and the memory of which is initialized. Since increasing latency increases the initialization time of links, processes exchange less links at each shuffle. The generated traffic decreases accordingly. Latency also spreads control messages over time, hence the smoothing in measurements.

Assuming peer-sampling protocols that enable a form of routing, PRC-broadcast forbids multiple delivery at the cost of a few lightweight control messages in dynamic systems. In this experiment, the underlying peer-sampling protocol builds a random graph topology that has numerous desirable properties such as resilience to failures, quick dissemination of information, or load balancing [1]. It fits dynamic systems where numerous processes join and leave continuously. Nonetheless, other peer-sampling protocols could be used depending on the configuration of the system. One could minimize latency [5], or gather people based on user preferences [1].

Overall, this section showed that PRC-broadcast proposes a novel trade-off in terms of complexity. Its complexity actually depends on the system (its dynamicity, its latency, its topology) and current use (broadcasts per second). PRC-broadcast forbids multiple delivery and safely removes obsolete control information about broadcast messages. The next section reviews state-of-the-art approaches designed to forbid multiple delivery.

## 4 RELATED WORK

Causal broadcast ensures causal order and forbids multiple delivery. PRC-broadcast uses the former to improve on the complexity of the latter. This section reviews state-of-the-art broadcast protocols that forbid multiple delivery in asynchronous and dynamic systems.

Building **specific dissemination topologies** such as tree or ring guarantees that every process receives each message once [4, 2]. Processes deliver messages as soon as they arrive. They do not need to save any control information about messages, for they will never receive a copy of this message again. While these approaches are lightweight, they stay confined to systems where failures are uncommon, and where churn rate remains low [1]. PRC-broadcast generalizes on these specific topologies. It follows the same principle where the topology impacts on the number of receipts. Its space complexity scales linearly with this number of receipts. In turns, PRC-broadcast inherits from the resilience of the underlying topology maintained by processes. PRC-broadcast supports dynamic systems without assuming any specific topology.

Without any specific dissemination topology, each process may receive each broadcast message multiple times. Despite multiple receipts, a process must deliver a message once. Using local structures based on **logical clocks [1]**, every process differentiates between the first receipt of a broadcast message and the additional receipts of this message. It allows to deliver the former while ignoring the latter. Unfortunately, the size of these structures increases monotonically and linearly with the number of processes that ever broadcast a message [1, 1]. Processes cannot reclaim the space consumed, for it would require running an overcostly distributed garbage collection that is equivalent to a distributed consensus [1]. This limits their use to context where the number of broadcasters is known to be small. PRC-broadcast uses local structures based on logical clocks too. However, instead of saving the past deliveries of broadcasters, its saves the messages expected from direct neighbors. The set of expected messages varies over receipts, and the number of neighbors can be far smaller than the set of broadcasters. PRC-broadcast scales in large and dynamic systems. Among others, PRC-broadcast fits contexts where the number of participants is unknown, such as distributed collaborative editing [1].

Table 1 summarizes the complexity of broadcast imple-

Table 2: Complexity of broadcast algorithms at each process. $N$ the number of processes that ever broadcast a message. $P$ is the set of processes in the system. $W$ the number of messages received but not delivered yet. $Q_i$ is the set of incoming links. $M$ is the number of messages already delivered that will be received again from at least one link in $Q_i$.

| | message overhead | delivery execution time | local space consumption | # control messages per added link |
|---|---|---|---|---|
| reliable broadcast [9] | $O(1)$ | $O(1)$ | $O(N)$ | 0 |
| causal broadcast [2] | $O(N)$ | $O(W \cdot N)$ | $O(N + W \cdot N)$ | 0 |
| preventive broadcast [2] | $O(1)$ | $O(1)$ | $O(N)$ | 3 to $2 \cdot \lvert P \rvert^2$ |
| **this paper** | $\mathbf{O(1)}$ | $\mathbf{O(\lvert Q_i \rvert)}$ | $\mathbf{O(\lvert Q_i \rvert \cdot M)}$ | **6** to $\mathbf{4 \cdot \lvert P \rvert^2}$ |

mentations that handle asynchronous and dynamic systems. To the best of our knowledge, all causal broadcast implementations use an underlying reliable broadcast in order to forbid multiple delivery. Their local space complexity comprises $O(N)$ where $N$ is the number of processes that ever broadcast a message. Compared to preventive causal broadcast [2], PRC-broadcast slightly increases the delivery execution time, and doubles the number of control message per added link. In turns, PRC-broadcast keeps a constant overhead on broadcast message, and changes the terms of local space complexity. Most importantly, the local space consumed does not monotonically increase anymore.

## 5 CONCLUSION

In this paper, we proposed a causal broadcast implementation that provides a novel trade-off between speed, memory, and traffic. Our approach exploits causal order to improve on the space complexity of the implementation that forbids multiple delivery. The local space complexity of this protocol does not monotonically increase and depends at each moment on the number of messages still in transit and the degree of the communication graph. The overhead in terms of number of control messages depends on the dynamicity of the system and remains low upon the assumption that the overlay network allows a form of routing. This advantageous trade-off makes causal broadcast a lightweight and efficient middleware for group communication in distributed systems.

As future work, we plan to investigate on ways to retrieve the partial order of messages out of PRC-broadcast. Applications may require more than causal order, they also may need to identify concurrent messages [2]. PRC-broadcast discards a lot of information by ignoring multiple receipts altogether. Analyzing the receipt order could provide insight on the partial order. The cost could depend on the actual concurrency of the system.

## 6 ACKNOWLEDGEMENTS

This work was partially funded by the French ANR projects O'Browser (ANR-16-CE25-0005-01), and Descartes (ANR-16-CE40-0023).

## REFERENCES

[1] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, September 1998.

[2] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.

[3] Dhruba Borthakur. Petabyte scale databases and storage systems at facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1267–1268, New York, NY, USA, 2013. ACM.

[4] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA, 2017. ACM.

[5] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004.

[6] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[7] Roy Friedman and Shiri Manor. Causal ordering in deterministic overlay networks. *Israel Institute of Technology: Haifa, Israel*, 2004.

[8] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 44–55. Springer Berlin Heidelberg, 2001.

[9] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[10] Matthias Heinrich, Franz Lehmann, Thomas Springer, and Martin Gaedke. Exploiting single-user web applications for shared editing: a generic transformation approach. In *Proceedings of the 21st international conference on World Wide Web*, pages 1057–1066. ACM, 2012.

[11] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[12] Mrk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer Networks*, 53(13):2321 – 2339, 2009.

[13] Sveta Krasikova, Raziel C. Gómez, Heverson B. Ribeiro, Etienne Rivière, and Valerio Schiavoni. Evaluating the cost and robustness of self-organizing distributed hash tables. In *Proceedings of the 16th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 9687*, pages 16–31, Berlin, Heidelberg, 2016. Springer-Verlag.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[16] Dahlia Malkhi and Doug Terry. Concise version vectors in winfs. *Distributed Computing*, 20(3):209–219, 2007.

[17] Alberto Montresor and Márk Jelasity. Peersim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer (P2P'09)*, pages 99–100, Seattle, WA, September 2009.

[18] Madhavan Mukund, Gautham Shenoy R., and S.P. Suresh. Optimized or-sets without ordering constraints. In Mainak Chatterjee, Jian-nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2014.

[19] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. Crate: Writing stories together with our browsers. In *Proceedings of the 25th International Conference Companion on World Wide Web*, WWW '16 Companion, pages 231–234, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[20] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems. In *Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems*, SRDS '18. IEEE, 2018.

[21] Brice Nédelec, Julian Tanke, Davide Frey, Pascal Molli, and Achour Mostéfaoui. An adaptive peer-sampling protocol for building networks of browsers. *World Wide Web*, Aug 2017.

[22] Michel Raynal. *Distributed algorithms for message-passing systems*, volume 500. Springer, 2013.

[23] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, Mar 1994.

[24] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, January 2011.

[25] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, USA, 2001. ACM.

[26] David Sun and Chengzheng Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, Oct 2009.