Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems

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@ls2n.fr

ABSTRACT

Many distributed protocols and applications rely on causal broadcast to ensure consistency criteria. However, none of causality tracking state-of-the-art approaches scale in large and dynamic systems. This paper presents a new non-blocking causal broadcast protocol suited for such systems. The proposed protocol outperforms state-of-the-art in size of messages, execution time complexity, and local space complexity. Most importantly, messages piggyback control information the size of which is constant. We prove that for both static and dynamic systems. Consequently, large and dynamic systems can finally afford causal broadcast.

1 INTRODUCTION

Causal broadcast [13] is a fundamental building block of many distributed applications [24] such as distributed social networks [6], distributed collaborative softwares [25, 14], or distributed data stores [9, 29, 2, 18, 7]. Causal broadcast is a reliable broadcast where every connected process delivers each broadcast message exactly once following the happen before relationship [17, 28]: when Alice comments on Bob's picture, everyone receives the comment after the picture; unrelated events are delivered in any order.

Unfortunately, causal broadcast has proven expensive in dynamic environments where any process can broadcast a message at any time [8]. While gossiping constitutes an efficient mean to disseminate messages to millions of processes [9, 4], ensuring causal delivery of these messages remains overcostly. Using state-of-the-art protocols, each message piggybacks a – possibly compressed – vector of Lamport's clocks [1, 10, 19, 30]. The message overhead increases monotonically, for entries cannot be reclaimed without consensus. The message overhead increases linearly with the number of processes N that ever broadcast a message in the system. Several messages W may deffer their delivery, for preceding messages did not arrive yet [20]. The delivery execution time takes linear time $O(W.N)$ as well. Vector-based protocols eventually become overcostly and inefficient.

To provide causal order, [11] employs a different strategy. Instead of piggybacking a vector in each message, processes forward all messages exactly once using FIFO communication means. Gossip encompasses forwarding so this does not constitute an overhead of the approach. Messages arrive ready so they are delivered immediately. This approach is both lightweight and efficient. However, its scope is restricted to static systems. In dynamic systems where processes can join, leave, add or remove communication means, using this approach may lead to causal order violations.

In this paper, we break the scalability barrier of causal broadcast for large and dynamic systems. Our contribution is threefold:

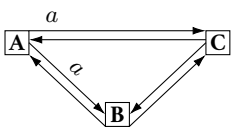
- We provide a powerful extension of [11] that extends its scope to dynamic systems. We prove that adding new communication means between processes constitutes the sole factor in causal order violation. Our approach solves this using bounded buffers and few control messages.
- We provide the complexity analysis of our broadcast protocol. Table 1 compares our protocol to two representative solutions. Our approach handles dynamic systems while providing constant size overhead on messages, and constant delivery execution time.
- We provide an experimentation highlighting the impact of our protocol on transmission delays before delivery. Indeed, to tolerate dynamicity our protocol temporarily disables new communication means. The experiment shows that even under bad network conditions and high dynamicity, our protocol hardly degrades the mean transmission time before delivery.

Consequently, causal broadcast finally becomes an affordable and efficient middleware for distributed protocols and applications in large and dynamic systems.

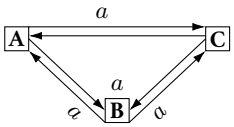
The rest of this paper is organized as follows: Section 2 shows the background and motivations. Section 3 details the model, our proposal, the proofs, and the complexity. Section 4 explains the results of experimentation. Section 5 reviews the related work. We conclude in Section 6.

Table 1: Complexity of causal broadcast protocols. N is the number of processes that ever broadcast a message. W is the number of received messages awaiting delivery. P is the number of delivered messages that are temporarily kept before being safely purged to forbid double delivery.

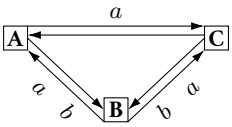
	dynamic systems	message overhead	local space consumption	delivery execution time
vector-based [28]	✓	$O(N)$	$O(N+W.N)$	$O(W.N)$
FIFO+forward [11]	✗	$O(1)$	$O(P)$	$O(1)$
this paper	✓	$\mathbf{O(1)}$	$\mathbf{O(N)}$	$\mathbf{O(1)}$



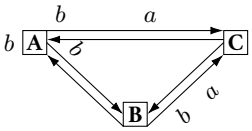
(a) Process A broadcasts a.



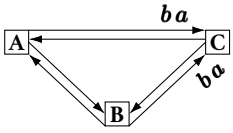
(b) Process B receives and delivers a. Process B forwards a using its FIFO links.



(c) Process B broadcasts b.



(d) Process A receives and delivers b. Process A forwards b using its FIFO links.



(e) Process C cannot receive b without having received a beforehand.

Figure 1: Causal broadcast [11] ensures causal order.

2 BACKGROUND AND MOTIVATIONS

Causal broadcast ensures that every connected process delivers each broadcast message exactly once [13] following the happen before relationship [17]. If the sending of a message m precedes the sending of a message m' then every process that delivers these two messages needs to deliver m before m'. Otherwise it can deliver them in any order.

Encoding the logical time at broadcast regarding all other broadcasts then piggybacking this control information in each broadcast message allows processes to ensure causal order on message delivery. Instead, [11] uses FIFO links and systematically forwards delivered messages. Intuitively, the dissemination pattern asserts that no paths from a process to another process carry messages out of causal order.

Figure 1 depicts this principle. The system comprises 3 processes connected to each other with FIFO links. In Figure 1a, Process A broadcasts a. It sends a to Process B and Process C. In Figure 1b, Process B receives, delivers, and forwards a. In Figure 1c, it broadcasts b. Consequently, all processes must deliver a before delivering b. In Figure 1d, Process A receives, delivers, and forwards b. Process A fulfills the causal order constraint between a and b. In Figure 1e, we see that either directly via Process B or indirectly via Process A, Process C cannot receive b before a. Thus, it eventually receives, delivers, and forwards the messages following causal order.

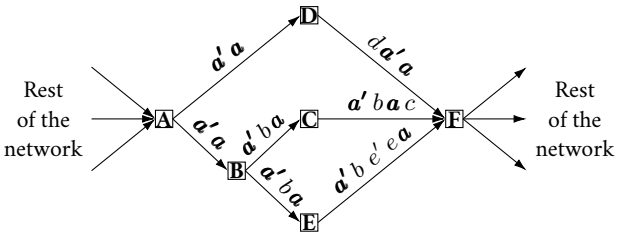


Figure 2: The principle of [11] works in large systems where processes have partial knowledge of the membership.

In large systems comprising from hundreds to millions of processes, no process can afford to maintain the full membership to communicate with. Instead, processes have a much smaller view called neighborhood. Forwarding messages allows them to reach all members of the system, either directly or transitively in a gossip fashion [9, 4]. In large systems, forwarding is mandatory. Processes pay the price of gossiping whatever broadcast protocol. They must create and send copies of the original broadcast message. Since

gossiping already encompasses forwarding of messages, it does not constitute an additional overhead of [11].

Figure 2 shows that such causal broadcast ensures causal order in larger systems where processes have limited knowledge of the membership. Process A only knows about Process B and Process D. Yet, Process A's broadcast messages a and a' arrive to all other processes either directly or transitively. In addition, a and a' always arrive in causal order at all processes despite concurrency and whatever the dissemination path.

Unfortunately, [11] ensures causal order only in static systems where the membership does not change and no links are added or removed. These are not practical assumptions. In practice, processes may join and leave the system at any time; and processes may reconfigure their neighborhood at any time [25]. Figure 3 shows an example of message dissemination in dynamic settings where causal delivery is violated. In Figure 3a, Process A broadcasts a. It sends a to all its neighbors. Here, it sends a to Process B only. Afterwards, in Figure 3b, Process A adds a link to Process C. Message a is still traveling. In particular, it did not reach Process C yet. In Figure 3c, Process A broadcasts a'. In this example, messages travel faster using the direct link from A to C than using B as intermediate. We see in Figure 3d that a' arrives at Process C before a. Figure 3e shows that not only it violates causal delivery but also propagates the violation to all processes downstream.

The causal broadcast presented in this paper extends [11] and solves the causal order violation issue of dynamic systems. Table 1 shows its complexity. Most importantly, message overhead and delivery execution time remain constant, i.e., our approach is both lightweight in terms of generated traffic and efficient. The local space complexity is linear in terms of number of processes that ever broadcast a message, and awaiting messages. The local space complexity also comprises a data structure to ensure causal order. We show in Algorithm 3 that the size of this structure can be bounded even in presence of system failures, such as crashes. This makes causal broadcast an affordable and efficient middleware for distributed protocols and applications even in large and dynamic systems.

The next section describes the proposed causal broadcast. It details its operation, provides the proofs that it works in both static and dynamic settings, and shows its complexity analysis.

3 CAUSAL BROADCAST FOR LARGE AND DYNAMIC SYSTEMS

In this section, we introduce PC-broadcast (stands for Preventive Causal broadcast), a causal broadcast protocol that breaks the scalability barrier for large and dynamic systems. Our approach is preventive: instead of repairing causal order violations or reordering received messages, it simply makes sure that messages never arrive out of causal order. Processes can immediately deliver messages upon receipt. This not only removes most of control information piggybacked in broadcast messages, but also leads to constant delivery execution time. Protocols and applications can finally afford causal broadcast in large and dynamic systems without loss of efficiency.

3.1 Model

A distributed system comprises processes. Processes can communicate with each other using messages. They may not have full knowledge of the membership, for maintenance is too costly in large and dynamic systems. Instead, processes build overlay networks with local partial view the size of which is generally much smaller than the actual size of the system [3, 15, 16]. Overlay networks can be built on top of other overlay networks. For the rest of this paper, we will use the terms of distributed systems, overlay networks, or networks indifferently.

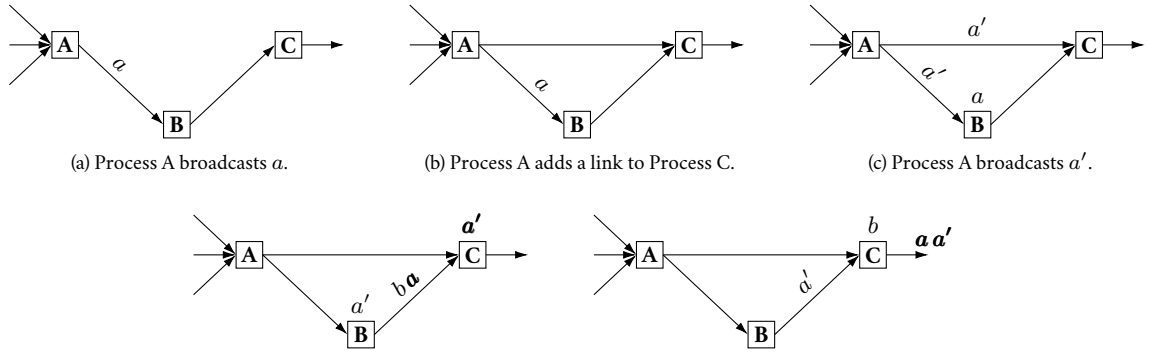
Definition 1 (Overlay network)**.** *An overlay network G is a pair $\langle P, E\rangle$ where P is a set of processes, and E is a set of links $E : P \times P$. An overlay network is static when P and E are immutable, otherwise it is dynamic.*

Definition 2 (Process)**.** *A process runs a set of instructions sequentially and communicates with other processes using message passing.*
A process' neighborhood is the set of links departing from it.
A process A can send messages to another process B in its neighborhood: $s_{AB}(m)$; receive a message from another process C that has Process A as neighbor: $r_{AC}(m)$.
A process is faulty if it crashes, otherwise it is correct. We do not consider byzantine processes.

Definition 3 (Unpartitioned network)**.** *A network is unpartitioned if and only if for any pair of correct processes, there exist a path – a link or a sequence of links – of correct processes between them. We only consider unpartitioned overlay networks.*

Causal broadcast is a communication primitive that relies on reliable broadcast to send messages to all processes in the system.

Definition 4 (Uniform reliable broadcast)**.** *When a process A broadcasts a message $b_A(m)$, each correct process B in the network eventually receives it $r_B(m)$ and delivers it $d_B(m)$. Uniform reliable broadcast guarantees 3 properties:*
***Validity:** If a correct process broadcasts a message, then it eventually delivers it.*
***Uniform Agreement:** If a process – correct or not – delivers a message, then all correct processes eventually deliver it.*
***Uniform Integrity:** A process delivers a message at most once, and only if it was previously broadcast.*



(a) Process A broadcasts a.
(b) Process A adds a link to Process C.
(c) Process A broadcasts a'.
(d) Process C receives and delivers a' before a. This violates causal order.
(e) Process C propagates the causal order violation.

Figure 3: Causal broadcast [11] may violate causal order in dynamic settings.

Algorithm 1: R-broadcast at Process p.

```
1  INITIALLY:
2      Q     // Set of processes, p's neighborhood
3      received ← ∅     // Set of received messages
4  DISSEMINATION:
5      function R-broadcast(m)
6          received ← received ∪ m
7          foreach q ∈ Q do sendTo(q, m)    // broadcast
8          R-deliver(m)
9      upon receive(m)
10         if m ∉ received then
11             received ← received ∪ m
12             foreach q ∈ Q do sendTo(q, m)  // forward
13             R-deliver(m)
```

Algorithm 1 shows the instructions of a uniform reliable broadcast. It uses a structure that keeps track of received messages in order to deliver them at most once. Since processes may not have full membership knowledge, processes must forward broadcast messages. Since the network does not have partitions, processes either receive the message directly from the broadcaster or transitively. Thus, all correct processes eventually deliver all messages exactly once. R-broadcast ensures validity, uniform agreement, and uniform integrity.

Causal broadcast is a reliable broadcast that also ensures a specific ordering among message deliveries. To define a delivery order among messages, we define time in a logical sense using Lamport's definition [17].

Definition 5 (Happen before [17])**.** *Happen before is a transitive, irreflexive, and antisymmetric relation $\rightarrow$ that defines a strict partial order of events. The sending of a message always precedes its receipt.*

To order messages broadcast by every process, we define causal order.

Definition 6 (Causal order)**.** *The delivery order of messages follows the happen before relationships of corresponding broadcasts:* $\forall A, B, C, b_A(m) \rightarrow b_B(m') \implies d_C(m) \rightarrow d_C(m')$

Definition 7 (Causal broadcast)**.** *Causal broadcast is a uniform reliable broadcast ensuring causal order.*

Theorem 1 (Constraint flooding in deterministic overlay networks is causal [11])**.** *In static networks, a broadcast protocol is causal if it uses FIFO links, forwards all broadcast messages exactly once, and uses all its outgoing links.*

From Theorem 1, reliable broadcast from Algorithm 1 is causal if communication links employed to communicate with neighbors in Q are FIFO. This holds only for static networks where Q is immutable. In practice, processes can join, leave, add or remove links to neighbors from Q at any time.

Lemma 1 (R-broadcast is causal in dynamic systems subject to removals)**.** *R-broadcast using FIFO links is a causal broadcast in dynamic systems where processes can leave the system or links can be removed.*

Proof. Removing a process from the network and removing all the incoming and outgoing links of this process is equivalent. Since we assume that removals do not create network partitions[1], all correct processes eventually receive all broadcast messages. In addition, removing a link or a process does not reorder causally related messages. Hence, each process receives and delivers each broadcast message in causal order as in static systems. □

[1] It may create partitions infringing the uniform agreement property. Network partitioning constitutes an orthogonal problem that we do not address in this paper.

Link removals and process departures do not endanger broadcast properties. However, Figure 3 shows that adding links may lead to causal order violations. The next section describes PC-broadcast, a causal broadcast that handles all dynamicity.

3.2 Causal order in dynamic systems

PC-broadcast stands for Preventive Causal broadcast. It prevents causal order violations by forbidding the usage of new links until proven safe. It constitutes a powerful yet simple extension of [11]. Table 1 shows that it preserves both constant message overhead and constant delivery execution time in dynamic settings.

Figure 3 shows that adding links may infringe the causal order property of causal broadcast. New links may act as shortcuts for new messages: new messages that travel through new links may arrive before preceding messages that took longer paths. To prevent this behavior, we define the safety of a link. PC-broadcast uses all and only safe links to disseminate messages.

Definition 8 (Safe link)**.** *A link from Process A to Process B is safe if and only if Process B received or will receive all messages delivered by Process A before receiving any message that Process A will deliver:* $safe_{AB} \equiv \forall m, m', d_A(m) \rightarrow s_{AB}(m') \implies r_B(m) \rightarrow r_{BA}(m')$

Added links start unsafe. In Figure 3, Process A uses the link to broadcast a' while it is unsafe: Process B did not receive a yet, and there was no guarantee that Process B would receive a before receiving a' from the new link. In this example, the worst happens and Process B receives then delivers a' before a which violates causal order.

The challenge is to make unsafe links safe using local knowledge only. A straightforward mean for Process A to achieve this consists in sending all its delivered messages to Process B using this unsafe link. This guarantees that any message delivered by A will be received by B before A starts using the new – now safe – link for causal broadcast. However, this is costly both in local space and generated traffic. Performing an anti-entropy round to extract missing messages would also be overcostly in terms of generated traffic for it would require sending the vector of received messages [9]. Instead, Process A avoid sending most of messages by initiating a ping phase to Process B.

Definition 9 (Ping phase)**.** *Ping phase starts when Process A pings Process B. Ping messages π travel using safe links. When Process B receives this ping, it replies to Process A. Replies ρ travel using any communication mean. Ping phase ends when Process A receives the reply of Process B.*

Lemma 2 (Ping phases acknowledge broadcast receipts)**.** *A ping phase from Process A to Process B acknowledges the receipt by B of all messages delivered by Process A before this ping phase:* $\forall m, d_A(m) \rightarrow s_A(\pi_{AB}) \wedge r_A(\rho_{AB}) \implies r_B(m)$

Proof. Suppose a process A initiates a ping phase to a process B. Suppose a series of messages delivered by Process A. Process A sent every of these messages exactly once using all its outgoing safe links. Processes that will receive these messages either already forwarded them or will forward them in their receipt order. Since Process A's ping travels using safe links after these messages, when Process B receives the ping, it already received all messages delivered by Process A. Process A receives Process B's reply after Process B received the ping. Consequently, when Process A receives Process B's reply, Process B received all messages delivered by Process A before the start of this ping phase. □

Upon receipt of Process B's reply, Process A has the guarantee that Process B received all its delivered messages preceding the ping phase. This is not sufficient, for ping phases take time. Messages delivered during ping phase by Process A may not be received by Process B yet. To fill this gap, Process A sends to Process B the messages it buffered during the ping phase.

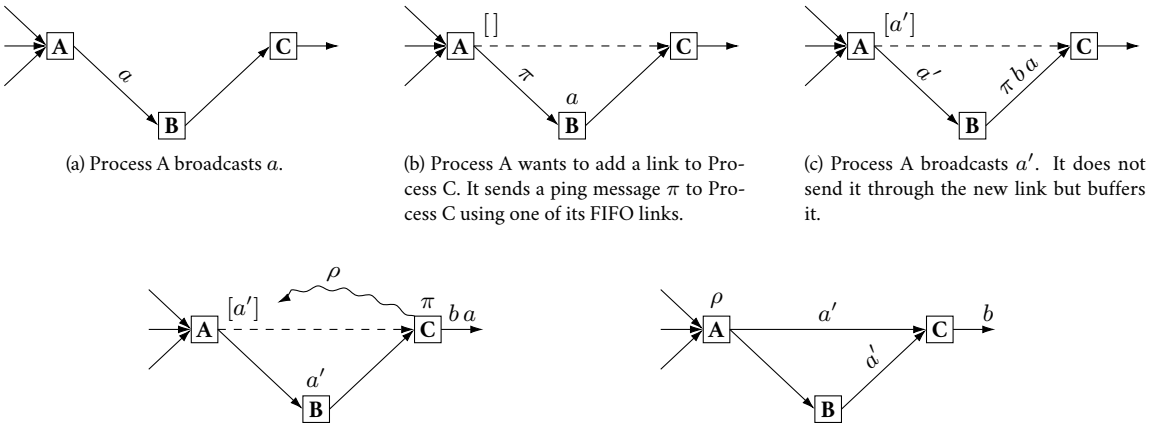
Definition 10 (Buffering)**.** *Process A records in a buffer $\mathcal{B}$ all its delivered messages during a ping phase to Process B.* $\forall m, s_A(\pi_{AB}) \rightarrow d_A(m) \wedge d_A(m) \rightarrow r_A(\rho_{AB}) \Leftrightarrow m \in \mathcal{B}$

Lemma 3 (Ping phase and buffering makes safe links)**.** *Process A makes an unsafe link to Process B safe by completing a ping phase to Process B then finalizing it by sending all delivered messages buffered during ping phase using the new link.*

Proof. Suppose a series of messages $m_1 \dots m_i \dots m_j$ delivered by a process A. Suppose Process A initiated a ping phase to a process B after delivering m_i. Suppose Process A receives Process B's reply after m_j. We must show that when Process A delivers a message after m_k, Process B received or will receive $m_1 \dots m_j$ before.
From Lemma 2, when Process A receives Process B's reply, Process B received $m_1 \dots m_i$.
Since Process A buffered all messages delivered since the beginning of the ping phase, the buffer contains $m_{i+1} \dots m_j$ when the ping phase ends. Since links are FIFO, sending messages of this buffer using the new link guarantees that Process B will receive them before receiving any m_k delivered after m_j. The link from Process A to Process B is safe. □

Lemma 4 (PC-broadcast is causal in dynamic systems subject to additions)**.** *In dynamic systems where processes can join or add links, broadcasting using all and only safe FIFO links ensures causal order. Without partition, the broadcast is causal.*

Proof. PC-broadcast ensures **validity**, **uniform agreement**, and **uniform integrity**, for it extends R-broadcast that ensures all 3 properties.
We must show that PC-broadcast ensures **causal order**:



(a) Process A broadcasts a.

(b) Process A wants to add a link to Process C. It sends a ping message π to Process C using one of its FIFO links.

(c) Process A broadcasts a'. It does not send it through the new link but buffers it.

(d) Process C receives π and replies to A. The reply ρ can travel by any communication mean.

(e) Process A receives Process C's reply. The former safely empties its buffer to Process C. Using the new link cannot cause causal order violation anymore.

Figure 4: PC-broadcast does not violate causal order in dynamic settings.

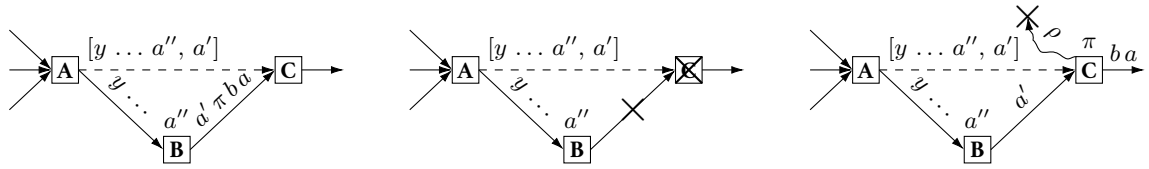
$\forall A, B, C, b_A(m) \to b_B(m') \implies d_C(m) \to d_C(m')$. $\forall A, B, b_A(m) \to b_B(m') \Leftrightarrow d_B(m) \to d_B(m') \Leftrightarrow d_B(m) \to s_B(m') \Leftrightarrow d_B(m) \to (\forall C \in Q, s_{BC}(m'))$. Since all links in Q are safe links, $r_c(m) \to r_{CB}(m')$ (see Definition 8). Since delivery order follows first receipt order, $d_C(m) \to d_C(m')$. This order on message delivery transitively reach all correct processes as long as the network remains unpartitioned. □

Algorithm 2 shows a small set of instructions that implement safe links. Figure 4 shows on an example how it solves causal order violations. In Figure 4a, Process A broadcasts a. In Figure 4b, Process A wants to add a link to Process C. It sends a ping message π to Process C (see Line 11) and awaits for the latter's reply. We leave aside the implementation of this send function (e.g. broadcast or routing). While awaiting, Process A keeps its normal functioning and maintain a buffer of messages associated with the unsafe link (see Line 25). In Figure 4c, Process A broadcasts another message a'. It sends it normally to Process B but does not send it to Process C directly. Instead, it buffers it. In Figure 4d, Process C receives Process A's ping message π. Since links are FIFO, it implicitly means that Process C also received a. Process C sends a reply ρ to Process A (see Line 13). ρ can travel through any communication mean. In Figure 4e, Process A receives ρ. Consequently, Process A knows that Process C received and delivered at least a and all preceding messages. It empties the buffer of messages to Process C (see Line 16). Afterwards, the new link is safe. Process A uses the new link normally.

Theorem 2 (PC-broadcast is a causal broadcast)**.** PC-broadcast *is a causal broadcast in both static and dynamic network settings.*

Algorithm 2: PC-broadcast at Process p.

```
1  INITIALLY:
2      Q                          // p's neighborhood, FIFO links
3      B ← ∅                      // Map link → buffered messages
4      counter ← 0                // Control message identifier
5  SAFETY:
6      upon open(q)
7          if |Q| > 1 then
8              counter ← counter + 1
9              Q ← Q \ q                          // is unsafe
10             B[q] ← ∅                  // initialize buffer
11             ping(p, q, counter)                // send π
12     upon receivePing(from, to, id)             // to = p
13         pong(from, to, id)                     // send ρ
14     upon receivePong(from, to, id)             // from = p
15         if to ∈ B then
16             foreach m ∈ B[to] do sendTo(to, m)
17             B ← B \ to                         // remove buffer
18             Q ← Q ∪ to                         // now safe
19     upon close(q)
20         B ← B \ q
21 DISSEMINATION:
22     function PC-broadcast(m)
23         R-broadcast(m)
24     upon R-deliver(m)
25         foreach q ∈ B do B[q] ← B[q] ∪ m // buffers
26         PC-deliver(m)
```



(a) Slow ping message π. (b) Process C crashes. (c) Reply ρ fails.

Figure 5: Buffers may grow unbounded due to network conditions.

Proof. For static networks, it comes from [11]. For dynamic networks, it comes from Lemmas 1 and 4. □

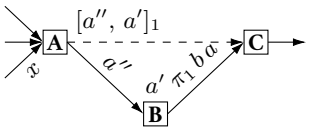
3.3 Bounding space consumption

Algorithm 2 ensures causal delivery of messages even in dynamic network settings. Compared to the original causal broadcast for static networks [11], it uses an additional local structure: buffers of messages. It associates a buffer to each new unsafe link. We assumed that the size of these buffer stays small in general, for it depends on the time taken by the ping phase which is assumed short. However, network conditions may invalidate this assumption. Figure 5 depicts scenarios where buffers grow out of acceptable boundaries. In Figure 5a, the issue comes from high transmission delays from Process A to Process B, and from Process B to Process C compared to the number of messages to broadcast and forward. The ping message π did not reach Process C yet that the buffer contains a lot of messages. In Figure 5b, the issue comes from the departure of Process C. Depending on network settings, Process A may not be able to detect Process C's departure. The former will never receive the awaited reply and the buffer will grow forever. In Figure 5c, the reply ρ itself fails to reach Process A. For the recall, this message can travel to Process A by any communication mean, including unreliable ones. If this fails, Process A's buffer to Process C will grow forever.

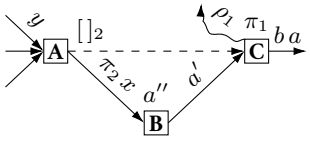
Algorithm 3 solves the unbounded growth issue of buffers. It solves the issue from the buffer owner's perspective. Figure 6 shows how this algorithm bounds the size of buffers. In Figure 6a, Process A broadcast a; then wanted to add a link to Process C so it sent a ping message; then broadcast a' and a'' so it buffered them. We see that the ping message π_1 carries a counter. The new buffer is identified by the same counter. In Figure 6b, Process A receives, delivers, and forwards the message x. Each message delivery increases the size of current buffers. The algorithm checks if the size of the buffer exceeds the configured bound (see Line 15). Adding x to the buffer would exceed the bound of 2 elements. This is the first ping phase failure. Process A simply restarts the ping phase: it resets the buffer and sends another ping message π_2 (see Line 24). The counter of the reset

Algorithm 3: Bounding the size of buffers.

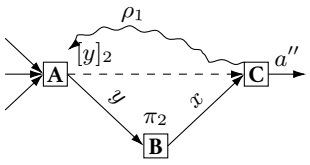
```
1  INITIALLY:
2      B                          // link → buffered messages
3      I ← ∅                      // message id ↔ link
4      R ← ∅                      // link → number of retries
5      maxSize ← ∞
6      maxRetry ← ∞
7  BOUNDING BUFFERS:
8      upon ping(from, to, id)
9          if q ∉ R then R[q] ← 0
10         I[id] ← to
11     upon receiveAck(from, to, id)
12         I ← I \ id
13         R ← R \ to
14     upon PC-deliver(m)
15         foreach q ∈ B such that |B[q]| > maxSize do
16             retry(q)
17     upon close(q)
18         for i ∈ I such that I[i] = q do I ← I \ i
19         R ← R \ q
20     function retry(q)
21         for i ∈ I such that I[i] = q do I ← I \ i
22         if q ∈ R then
23             R[q] ← R[q] + 1
24             if R[q] ≤ maxRetry then open(q)
25             else close(q)
26 HANDLING FAILURES:
27     upon timeout(from, to, id)
28         if id ∈ I then retry(to)
```



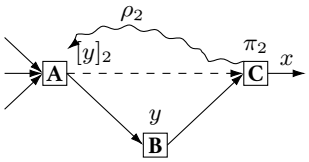
(a) Process A wanted to add a link to Process C after having broadcast a. Afterwards, it broadcast a' and a''.



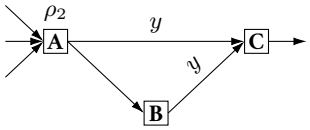
(b) Process A receives, delivers, and forwards x. Since the buffer size would exceed the maximal boundary, it resets the buffer with a new counter and sends a ping message with this counter. In the meantime, Process C receives π_1 and sends ρ_1.



(c) Process A forwards y and buffers it. It also receives ρ_1 but no buffers exist with this counter. Process A simply discards ρ_1.



(d) Process C receives π_2 and sends the corresponding reply ρ_2.



(e) Process A receives ρ_2. It empties the buffer with the corresponding counter. Now, the direct link to Process C is safe.

Figure 6: Buffers become bounded. We allow only 2 elements in each buffer.

buffer is the one of the new ping message. In the meantime, Process C receives π_1 and sends the corresponding reply ρ_1. In Figure 6c, Process A receives a broadcast message y. It delivers it, checks if the buffer can admit it, adds the message to the buffer, and forwards it. Process A also receives the first reply ρ_1 but discards it, for no buffers have such counter. In Figure 6d, Process C receives π_2 and sends the corresponding reply ρ_2 to Process A. In Figure 6e, Process A receives ρ_2. Since the corresponding buffer exists, it empties it. The new link is now safe to use for causal broadcast.

While it solves the issue of unbounded buffers, it also brings another issue. For instance, if the maximal size of buffers is too small, it could stuck the protocol in a loop of retries. We address this issue by bounding the number of retries. However, it means that the ping phase could fail entirely. Causal broadcast must not employ the new link. In extreme cases, it could cause partitions in the causal broadcast overlay network. It would violate the uniform agreement property of causal broadcast. Thus, we assume a sufficiently large maximal bound. It never creates partitions, for most links become safe, and the failing ones are replaced over time thanks to network dynamicity.

Other orthogonal improvements are possible. For instance, causal broadcast could use reliable communication means to acknowledge the receipt of the ping message. The time taken between the sending and the receipt of the reply would increase when failures occur. However, it would take less time than resetting the buffering phase.

3.4 Complexity

We review and discuss about the complexity of PC-broadcast. We distinguish the complexity brought by (i) the overlay network, (ii) reliable broadcast, (iii) and causal order.

Overlay network. Processes cannot afford the upkeep of full membership in large and dynamic systems. Instead, each process builds a partial view the size of which is considerably smaller than the actual network size. To maintain these partial views, each process runs a peer-sampling protocol [3, 15, 16]. Some peer-sampling protocols provides partial views the size of which scales logarithmically with the actual network size [26]. The number of messages forwarded by each process for each broadcast remains small, for this number is equal to their view size: $O(Q)$ where Q is the size of the partial view.

Reliable broadcast. Gossiping constitutes an efficient mean to disseminate messages to all processes [9, 4]. Algorithm 1 shows that it relies on a local structure to guarantee that messages are delivered exactly once. This structure grows linearly with the number of processes in the network: $O(N)$. In addition, each message piggybacks a pair $\langle process, counter \rangle$ that identifies it: $O(1)$. Checking if a message is a duplicate takes constant time: $O(1)$.

Causal ordering. Causal ordering primarily uses FIFO links to broadcast messages which implies a constant size overhead on messages $O(1)$.

PC-broadcast constitutes an advantageous tradeoff when the system features light routing capabilities. For instance, protocols building systems using neighbor-to-neighbor interactions [15, 16, 26] only require 3 control messages per added link (see Section 4); protocols featuring message rout-

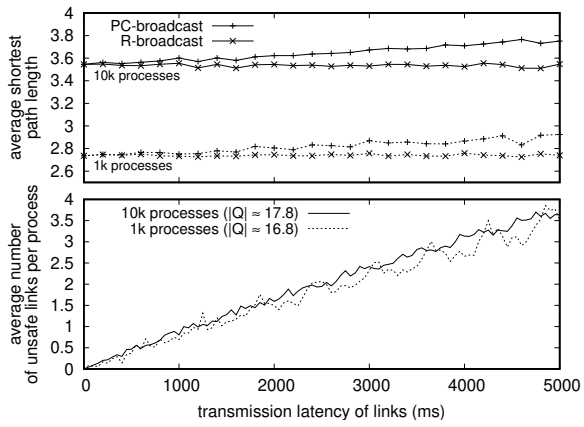


Figure 7: Impact of PC-broadcast on the overlay network.

ing such as DHT require $O(log(N))$ control messages per added link. However, in the worst case, PC-broadcast requires $O(N^2)$ control messages per added links, for it falls back to broadcasting control messages.

The space complexity of PC-broadcast is that of FIFO links plus buffered messages ensuring safety. The size of buffers depends on the ping phase duration. Assuming a system featuring light routing capabilities, we assume that this time remains short so the number of buffered messages stays small. As shown in Section 3, network conditions can make this assumption false. Algorithm 3 allows to bound the size of each buffer and handle network failures.

Overall. Generated traffic remains the most important criterion for scalability. The traffic generated by PC-broadcast for each process and for each broadcast only depends on the size of messages and the overlay network chosen to broadcast messages. The size of messages is an irreducible variable; and many protocols designed to build overlay networks achieve high scalability in terms of network size and dynamicity [3, 15, 16], [26, 31]. Consequently, PC-broadcast achieves high scalability in both these terms too. PC-broadcast is efficient, for the upper bound on the complexity of delivery execution time does not depend on any factor.

However, to ensure causal order, PC-broadcast may not use all outgoing links in dynamic settings, for some may be temporarily unsafe. This negatively impacts the overlay network properties. The next section shows an experiment that highlights the influence of PC-broadcast's way to ensure causal order on the underlying overlay network. In particular, it shows that the number of hops required by a broadcast message to reach all processes increases when delays on transmission increase.

4 EXPERIMENTATION

PC-broadcast provides causal order with constant size message overhead. This feature comes at a cost: at first, new communication means are disabled for causal broadcast. In this section, we evaluate the impact of PC-broadcast on the message delivery in a specific overlay network that corresponds to random graphs. The experiments run on the PeerSim simulator [21] that allows simulations to reach high scale in terms of number of processes. Our implementation is available on the Github platform at `http://github.com/chat-wane/peersim-pcbroadcast`.

Objective: To observe the transmission delay introduced by PC-broadcast on message delivery. We expect the delay to increase as the latency increases.

Description: We build an overlay network with a topology close to random graphs using Spray [26]. The overlay networks comprise 1k, and 10k processes respectively. Networks are dynamic. Each process' neighborhood Q changes at least once every 60 seconds; and on average twice every 60 seconds. Each exchange involves two processes that both add and remove half of their partial view. Links are FIFO, bidirectional, and have transmission delays. The delay increases over time up to 5 seconds. Consequently, the duration of ping phases increases during the experiment. Links become safe slower.

We measure the shortest path length from a random set of processes to all other processes. It represents the average number of hops taken by broadcast messages before being received and delivered by all. Multiplied by the transmission latency of links, it represents the transmission delay of broadcast messages before being received by all processes.

We perform measurements on 2 broadcast protocols: PC-broadcast and R-broadcast. R-broadcast uses all available links to broadcast messages in a gossip fashion. Transmission delays before delivery are similar to piggybacking approaches [1, 10, 19, 30, 5, 12, 22] without accounting for the time taken to send large messages (e.g. each message conveys a vector clock of 10k entries when the network comprises 10k broadcasters). Larger packets induce larger transmission time.

Results: Figure 7 shows the result of the experiment. The x-axis depicts the delay set on message transmission for each link. The top part of the figure shows the average shortest path length. The bottom part of the figure shows the average number of unsafe links per process that cannot be used for causal broadcast yet.

Figure 7 shows that both R-broadcast and PC-broadcast deliver message quickly to all processes. The overlay network guarantees that paths stay short and logarithmically scaling with the number of random neighbors in partial views.

The top part of Figure 7 shows that measurements made on PC-broadcast increase while measurements made on R-broadcast stay constant. R-broadcast uses all neighbors pro-

vided by Spray while PC-broadcast excludes links still in buffering phase. The more latency on transmission, the longer the buffering phase. The bottom part of Figure 7 shows that the number of elements in the buffers increases accordingly.
Figure 7 shows that the growth of path length stays small even when transmission delays become high. The number of elements in buffers stays small because the buffering phase takes a constant number of hops to complete: at most 3 hops. The path length grows even slower, for removing 3 among 17 links has restricted impact on overlay networks close to random graphs.

This experimentation shows that even under bad network conditions (high transmission delays) and using highly dynamic overlay networks (random peer-sampling), the number of unsafe links remains small. The negative impact expected on transmission time before message delivery remains small. In practice, we expect smaller network transmission delays, and overlay networks less subject to neighborhood changes (e.g. exploiting user preferences, or geolocalization). In such settings, we expect PC-broadcast to have a negligible negative impact on the overlay network properties.

The next section reviews state-of-the-art techniques designed to maintain causal order among messages.

5 RELATED WORK

This section reviews the related work of logical clocks. It goes from piggybacking approaches to vector-based approaches. Then, it reviews explicit dependency tracking and dissemination-based approaches.

Piggybacking approaches [5, 12]. A trivial way to ensure causal ordering of messages is to piggyback all causally related messages since the last broadcast message along with the new broadcast message. Even by piggybacking the identifiers of messages instead of messages themselves, the broadcast message size may increase quickly depending on the application. PC-broadcast does not piggyback all preceding messages in broadcast messages. However, an accumulation of messages arises during buffering. As discussed in Section 3.4, we can assume that links quickly become safe so the buffer size stays small, and we can easily set a threshold on the buffer size.

Vector clock approaches [10, 19]. A vector clock is a vector of monotonically increasing counters. It encodes the partial order of messages using this vector: $VC(m) < VC(m') \implies m \rightarrow m'$. Before delivering a message, processes using vector-based broadcast check if the vector of the message is ready regarding their local vector. If it detects any missing preceding message, the process delays the delivery. To implement this vector-based broadcast (i) each process must maintain a vector locally; (ii) each message must piggyback such vector; (iii) there is 1 counter per process that ever broadcast a message. To accurately track causality, processes cannot share their entry. To safely track causality, processes cannot reclaim entries. Hence, even with **compaction approaches [30]**, the vectors grow linearly in terms of number of processes that ever broadcast a message. In [1], the complexity is reduced to the actual number of processes in the network. Still, these approaches do not scale, particularly in dynamic networks subject to churn and failures.
In comparison to these vector-based approaches, our approach reduces the generated traffic of causal broadcast by a factor of N in the most common context where processes have partial knowledge of the network membership.

Probabilistic approaches [22] sacrifices on causality tracking accuracy: messages may be delivered out of order under a computable boundary. The size of control information in messages depends on the desired boundary.
Unlike vector-based approach, our broadcast cannot state if two messages are concurrent, accurate causal delivery is a feature provided by default by the propagation scheme. Once safe, FIFO links deliver message in causal order without further delay. The speed of delivery is that of FIFO links.

Explicitly tracking **semantic dependencies** allows broadcast protocols to reduce the size of piggybacked control information [2, 18, 23]. For instance, when Alice comments on Bob's picture, everyone must receive the picture before the comment. The broadcast message only piggybacks one semantic dependency. When Alice comments on multiple pictures at once, the broadcast message piggybacks all dependencies. Message overhead increases linearly with the number of semantic dependencies. To track semantic dependencies, causal broadcast becomes application dependent. Instead our approach remains application-agnostic. Comments, pictures, etc. are events that relate to all preceding events. When Alice comments on Bob's picture, everyone will receive this event before the former event and all other preceding events. Whatever the number of preceding events, broadcast messages only piggyback constant size control information.

Preserving causal order using **dissemination paths** reduces generated traffic by keeping message overhead constant [7, 11]. State-of-the-art does not support dynamic systems [11], or supports it using epochs [7] that confines usability to small scale systems where failures are uncommon. In comparison, we designed PC-broadcast to handle large and dynamic systems. Our approach provides a lightweight and efficient mean to reconfigure dissemination paths using local knowledge without impairing causal order. Saturn [7] along with PC-broadcast could ease online changes in configuration while improving its resilience to failures and topology changes.

6 CONCLUSION

In this paper, we described a non-blocking causal broadcast protocol that breaks scalability barriers for large and dynamic systems. Using PC-broadcast, message overhead and delivery execution time remain constant. Causal broadcast finally becomes an affordable and efficient middleware for large scale distributed applications running in dynamic environments.

As future work, we plan to investigate on reducing the space complexity of reliable broadcast. Section 3.4 reviews structures with linearly increasing space consumption. We can reduce this complexity in static systems. We can prune the structure from already received messages, for we know that the number of duplicates is equal to the number of incoming links [27]. Unfortunately, this does not hold in dynamic systems. We would like to investigate on a way to prune the structure in such settings. This would make causal broadcast scalable as well on generated traffic as on space consumption.

We also plan to investigate on retrieving partial order of events. Section 5 states that vector-based approaches allows to compare an event with any other event. They can decide on whether one precedes the other, or they are concurrent. They can build the partial order of event using this knowledge. Our approach cannot by default. However, in extreme settings where the overlay network is fully connected, we can assign a vector to each received message using local knowledge only, and without message overhead. We would like to investigate on a way to build these vectors locally in more realistic settings where processes have partial knowledge of the membership.

ACKNOWLEDGMENTS

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] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 259–274, Berlin, Heidelberg, 2008. Springer-Verlag.

[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] M. Bertier, F. Bonnet, A. M. Kermarrec, V. Leroy, S. Peri, and M. Raynal. D2ht: The best of both worlds, integrating rps and dht. In *Dependable Computing Conference (EDCC), 2010 European*, pages 135–144, April 2010.

[4] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems*, 17(2):41–88, May 1999.

[5] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.

[6] 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.

[7] 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.

[8] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.

[9] 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 6th annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[10] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 10:56–66, 02 1988.

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

[12] Vassos Hadzilacos and Sam Toueg. Distributed systems. chapter Fault-tolerant Broadcasts and Related Problems, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

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

[14] 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.

[15] 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.

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

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

[18] 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.

[19] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[20] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 453–468, Berkeley, CA, USA, 2017. USENIX Association.

[21] 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.

[22] Achour Mostéfaoui and Stéphane Weiss. Probabilistic causal message ordering. In *Proceedings of Parallel Computing Technologies: 14th International Conference, PaCT 2017, Nizhny Novgorod, Russia, September 4-8, 2017*, pages 315–326, Cham, 2017. Springer International Publishing.

[23] 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.

[24] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 03 2009.

[25] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[31] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.