Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Networks (Regular)

Jane Doe*, John Doe*
*Affiliations
Street
City, Country
first.last@anonymous.com

***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 networks. We present a uniform reliable causal broadcast protocol that outperforms state-of-the-art in size of messages, execution time complexity, and local space complexity. Most importantly, it only overloads messages with constant size control information. We prove that this holds in both static and dynamic networks. Consequently, large and dynamic systems can finally afford causal broadcast.**

I. Introduction

Broadcasting [1] is a fundamental communication mechanism for numerous distributed protocols [2], [3] and applications [4]. ~~They generally run in highly dynamic environments where processes can join and leave the network at any time [5].~~ Broadcast protocols ensure that all connected processes receive and deliver each broadcast message. In this paper, we focus on causal broadcast: the order of message delivery follows the happen before relationship [6], [7]. This alleviates systems from the burden of tracking causality between its operations. ~~For instance, distributed data stores such as Riak, Dynamo, or Cassandra (REF) and collaborative editors such as Crate, or Peerpad (REF) make extensive use of conflict-free replicated data types [3], [8].~~ They feature addition and deletion of elements. A deletion must follow the corresponding addition, for the sake of eventual consistency [9]. Causal broadcast relieves these applications from continuously checking such constraints.

Unfortunately, accurate causality tracking has proven expensive especially in large and dynamic networks [10]. Most approaches are reactive: they check if preceding messages are missing at each receipt and deliver or delay the message accordingly. They either (i) piggyback preceding messages in new broadcast messages [11], [12], (ii) or maintain and send vector clocks along with new broadcast messages [13], [14]. Approaches reduce the size of vectors either by (a) maintaining local structures to send the least amount of changing data [15], or by (b) constraining the network topology (REF). None of these approaches scales in large and dynamic networks, for generated traffic and delivery time of messages increase linearly with the number of processes.

In static networks, causal broadcast can be preventive [16]: processes receive messages that are immediately ready for causal delivery. It overloads messages with constant size control information, for it uses FIFO communication links. Delivery time of messages is constant, for it only discards message duplicates. Nothing limits the scalability of the system. However, it does not handle network membership changes. ~~We would like a preventive causal broadcast that generalizes these complexity to dynamic networks. Causal broadcast would finally become affordable and efficient in large and dynamic systems.~~

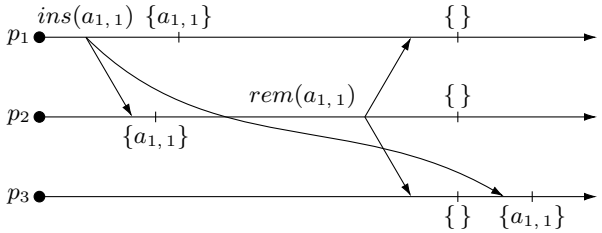


Fig. 1: Broadcast without causal order enforcement causes causal order violation. It breaks eventual consistency, for replicas do not converge.

~~In this~~ paper, we break the scalability barrier of causal broadcast for large and dynamic networks. We propose a uniform reliable broadcast that outperforms state-of-the-art in size of broadcast messages, execution time complexity, and local space complexity. ~~Our approach belongs to preventive causal broadcast.~~ Message overhead is constant. Message delivery time is constant. Local space complexity is that of reliable broadcast plus buffers of messages the size of which can be bounded. We prove that our causal broadcast protocol handles both static and dynamic networks. Experiment.

The rest of this paper is organized as follows: Section II shows the issues and motivations of our work. Section III defines our model, describes our proposal, provides the corresponding proofs, and details the complexity analysis. Section IV shows the results of experimentation. Section V reviews the related work. We conclude in Section VI.

II. ~~Issues~~ and Motivations

Broadcast protocols ensure that all connected processes receive and deliver each broadcast message [1]. Applications often require more guarantees on message delivery to ensure

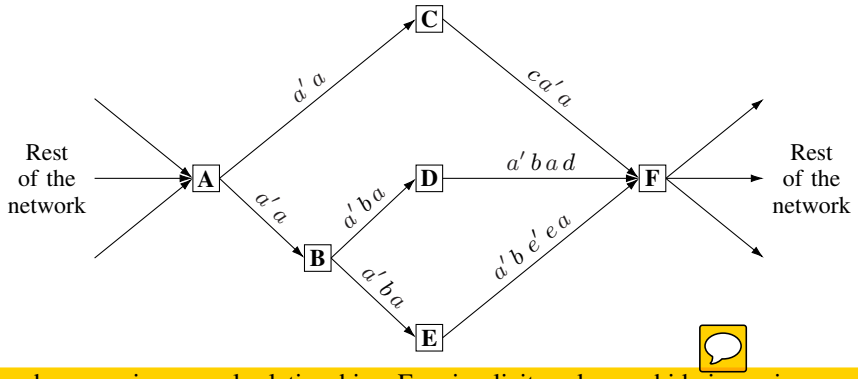


Fig. 2: Dissemination graph preserving causal relationships. For simplicity sake, we hide incoming messages of A and outgoing messages from F. Preventive causal broadcast preserves causal order in static settings. All dissemination paths arriving to F carry a before a'; and also a' before c, a before b, a before e, e before e'.

consistency criteria (REF). ~~For instance, conflict-free replicated data types [3] are replicated data structures ensuring eventual consistency [9]. Replicas eventually converge to an identical state. CRDTs for sets [17] and sequences [18] are extensively used in distributed data stores such as Riak, Dynamo, or Cassandra (REF) and distributed collaborative editors such as Crate, or Peerpad (REF). They feature basic operations such as insertion and deletion that require causal order: the deletion of an element must follow its insertion.~~

Figure 1 illustrates the need of a mechanism ensuring causal order in message deliveries. In this example, each process hosts the replica of a set. It starts empty. Process p_1 inserts the element a in the set along with its unique identifier: $ins(a_{1,1})$. It broadcasts the new element and its identifier. Process p_2 receives the operation and integrates $a_{1,1}$ to its own copy. Then, Process p_2 removes this element from its set and broadcast the operation: $rem(a_{1,1})$. Process p_3 receives the message stating that $a_{1,1}$ should be removed. Since its replica does not comprise this element, it executes the removal but the set remains unchanged. When Process p_3 finally receives the message stating that it should add $a_{1,1}$ to its replica, it integrates it. Replicas become divergent, for p_1 and p_2 ends up empty while p_3 has $a_{1,1}$. It breaks eventual consistency. A causal ordering mechanism would have avoided such situation. For instance, p_3 could have delayed the execution of the removal until the receipt and execution of the corresponding insertion.

Causal broadcast ensures causal order on message deliveries. Consequently, it releases applications from the burden of tracking causal relationship between operations. However, causality tracking has proven expensive in large and dynamic networks (REF).

First row of Table I shows the complexity of a vector-clock-based representative [7] of reactive approaches. Each broadcast message conveys control information the size of which increases linearly with the number of processes. Each process must check at each receipt if the message is ready for causal delivery which takes linear time in terms of number of processes and number of messages delayed. Consequently, even if its operation works in dynamic networks where membership changes over time, it eventually becomes expensive and inefficient.

TABLE I: Space and time complexity of causal broadcast protocols. N is the number of processes. W is the number of messages received but waiting to be delivered. P is the number of messages that are not yet purged. B is the size of a set of temporary buffers.

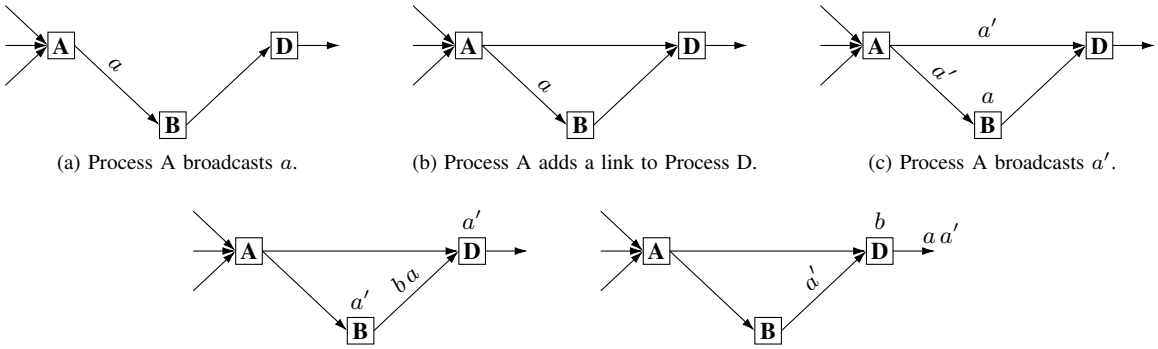
	dynamic	message overhead	local space	delivery time
reactive [7]	✓	$O(N)$	$O(N+W.N)$	$O(W.N)$
preventive [16]	✗	$O(1)$	$O(P)$	$O(1)$
this paper	✓	$\mathbf{O(1)}$	$\mathbf{O(N+B)}$	$\mathbf{O(1)}$

On the opposite, second row of Table I shows the complexity of a preventive approach [16]. It outperforms the reactive approach: message overhead is constant; it safely prunes its local structure; message delivery time is constant.

This preventive approach uses FIFO communication links to preserve causal order of messages. Intuitively, the dissemination pattern automatically makes sure that no path from a process to any other process carries messages out of causal order. Figure 2 shows an example of message dissemination. Process A broadcasts a then a'. All processes must receive a before a'. Process A sends it to the network using all its links. Process B and Process C necessarily receive a before a' for links are FIFO. They forward these messages in FIFO order using all their links too. Other processes necessarily receive a before a' for no subsequent path will ever carry a' before a. We see that in spite of concurrent messages broadcast at different time, Process F always receives a before a'. Concurrent messages also have causal relationship to ensure. Since Process B broadcasts b right after it received a, all processes must receive a before b. Since Process C broadcasts c after the receipt of a', all processes must receive c after a', hence after a. However, there is no constraint between message deliveries of c and b. Process F can receive them in any order.

However, ensuring causal delivery using FIFO links only holds for static networks where membership remains unchanged, and processes cannot add nor remove communication links. As soon as a process adds a communication link to another process, causal delivery may be violated.

Figure 3 shows an example of message dissemination in



(a) Process A broadcasts a.

(b) Process A adds a link to Process D.

(c) Process A broadcasts a'.

(d) Process D receives and delivers a' before a. This violates causal order.

(e) Process D propagates the causal order violation.

Fig. 3: Preventive causal broadcast may violate causal order in dynamic settings.

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 D. Message a is still traveling in the network. In particular, it did not reach Process D yet. In Figure 3c, Process A broadcasts a'. In this example, messages travel faster using the direct link from A to D than using B as intermediate. We see in Figure 3d that a' arrives at Process D before a. Figure 3e shows that not only it violates causal delivery but also propagates the violation to all processes downstream.

We would like a preventive causal broadcast that (i) ensures causal delivery at marginal cost and that (ii) handles both static and dynamic networks. Causal broadcast would finally become affordable and efficient in large and dynamic systems.

The causal broadcast presented in this paper achieve this. Last row of Table I shows the complexity we achieve. Similarly to the preventive approach, complexity stays constant in terms of message overhead and message delivery time. The local space complexity is linear in terms of number of processes. We suspect that bound to be minimal in dynamic settings. In addition, our algorithm employs another local structure to ensure causal order but we show that we can bound it in Algorithm 3.

Next section describes our preventive causal broadcast for large and dynamic networks. It details its operation, provides the proofs that it works in both static and dynamic settings, and shows its complexity analysis.

III. Causal broadcast for large and dynamic networks

In this section, we introduce a causal broadcast protocol that breaks scalability barriers for large and dynamic networks. To provide causal order, most state-of-the-art [19], [11], [13], [12], [14], [20], [15] approaches are reactive, for they check if message deliveries should be delayed to avoid causality violations. On the opposite our approach is preventive, for messages are immediately delivered on receipt without risk of causality violations. This difference not only removes most of control information piggybacked in broadcast messages, but also leads to constant delivery time. Protocols and applications can finally afford causal broadcast in large and dynamic networks without loss of efficiency.

A. Model

Definitions and theorems come from [1]. A network comprise processes. Processes can communicate with part of the network via messages. They may not have full knowledge of network membership, for maintenance costs become too expansive in large and dynamic networks. Instead, processes build overlay networks with local partial view the size of which is generally much smaller than the actual network size.

Definition 1 (Overlay network)**.** *Just as a network, an overlay network N comprises a set of processes P. Each Process runs a set of instructions sequentially.*
An overlay network N also comprises a set of links $E : P \times P$. p's neighborhood Q is the set of links departing from p. Processes can communicate with their neighbors using messages. Processes are faulty if they crash, otherwise they are correct. The set of correct processes is C. There are no byzantine processes.

For the rest of this paper, we will speak of networks and overlay networks indifferently.

Definition 2 (Static and dynamic networks)**.** *A network is static if both its set of processes and its set of edges are immutable. Otherwise, the network is dynamic.*

For the rest of the paper, we only consider networks without partitions.

Definition 3 (Network partition)**.** *A network has partitions if there exist two correct processes without any path between them, i.e., without a link or a sequence of links comprising correct processes only.*

We define time in a logical sense using Lamport's definition [6].

Definition 4 (Happen before)**.** *Happen before is a transitive, irreflexive, and antisymmetric relation that defines a strict partial orders of events. At process p, Event e happens before*

– or precedes – Event e' is noted $e_p \rightarrow e'_p$. The sending of a message $s_p(m)$ always precedes its receipt $r_q(m)$:
$\forall p,\, q \in P, s_p(m) \rightarrow r_q(m)$.

Processes communicate by sending messages to other processes. They can send messages to specific processes or all of them.

Definition 5 (Uniform reliable broadcast)**.** *A process p can broadcast a message $b_p(m)$, receive a message $r_p(m)$, and deliver a message $d_p(m)$. When a process p broadcasts a message m to all processes of the network, correct processes eventually receive it: $\forall p \in P,\, (b_p(m) \Leftrightarrow \forall q \in P,\, r_q(m))$.*
Uniform reliable broadcast guarantees 3 properties:
Validity: *If a correct process broadcasts a message, then it eventually delivers it: $\forall p \in C,\, b_p(m) \rightarrow d_p(m)$.*
Uniform Agreement: *If a process – correct or not – delivers a message, then all correct processes eventually deliver it:*
$\forall p \in P,\, (d_p(m) \implies \forall q \in C,\, d_q(m))$.
Uniform Integrity: *A process delivers a message at most once, and only if it was previously broadcast:*
$\forall p \in P, \neg(d_p(m) \rightarrow d_p(m)) \wedge$
$d_p(m) \implies \exists q \in P,\, b_q(m) \rightarrow d_p(m)$.

Algorithm 1: R-broadcast at Process p.

```
1  INITIALLY:
2      Q                                   // p's neighborhood
3      received ← ∅                        // To detect double
         receipts
4  DISSEMINATION:
5      function R-broadcast(m)                          // b_p(m)
6          received ← received ∪ m
7          foreach q ∈ Q do sendTo(q, m)
8          R-deliver(m)                                 // d_p(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)                             // d_p(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. This algorithm ensures validity, uniform agreement, and uniform integrity.

In addition to reliably conveying messages to all correct processes, broadcast protocols can ensure that messages are delivered in a specific order.

To order messages broadcast from one process, we define FIFO order.

Definition 6 (FIFO order)**.** *If a process broadcasts two messages, processes deliver the first before the second:*
$\forall p,\, q \in P,$
$b_p(m) \rightarrow b_p(m') \implies d_q(m) \rightarrow d_q(m')$.

To order messages broadcast by different processes, we define local order.

Definition 7 (Local order)**.** *If a process broadcasts a message after having delivered another message broadcast by another process, processes deliver the latter before the former:*
$\forall p,\, q,\, r,\, \in P,\, p \neq q,$
$b_p(m) \wedge d_q(m) \rightarrow b_q(m') \implies d_r(m) \rightarrow d_r(m')$.

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

Definition 8 (Causal order)**.** *The delivery order of messages follows the happen before relationships of the corresponding broadcasts:*
$\forall p,\, q,\, r \in P,$
$b_p(m) \rightarrow b_q(m') \implies d_r(m) \rightarrow d_r(m')$.

Theorem 1 (Causal order equivalence)**.** *FIFO order and local order is equivalent to causal order.*

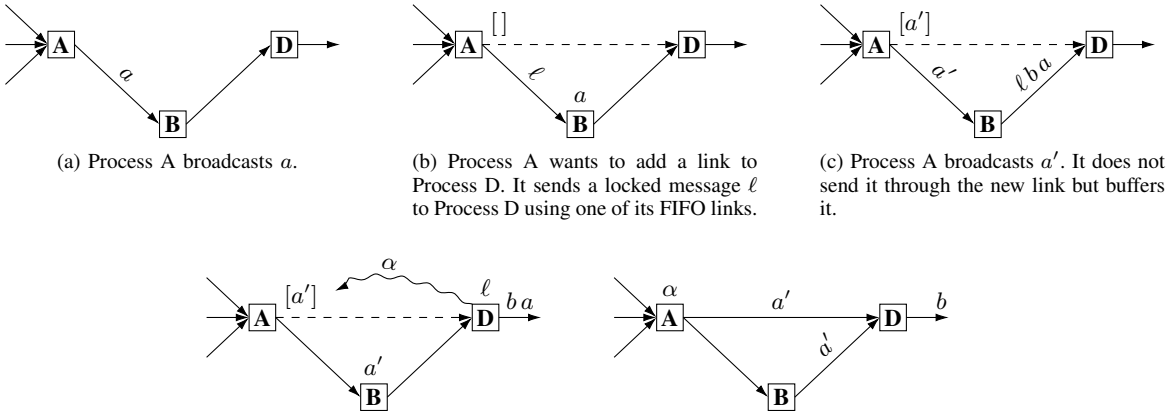
Definition 9 (Causal broadcast)**.** *Causal broadcast is a uniform reliable broadcast ensuring causal order.*

B. Operation

Algorithm 2: FBC-BROADCAST$^+$ at Process p.

```
1  INITIALLY:
2      Q          // p's neighborhood, FIFO links
3      B ← ∅          // link → buffered messages
4      counter ← 0              // Message identifier
5  SAFETY:
6      upon open(q)
7          if |Q| > 1 then
8              counter ← counter + 1
9              Q ← Q \ q
10             B[q] ← ∅
11             sendLocked(p, q, counter)
12     upon receiveLocked(from, to, id)              // to = p
13         sendAck(from, to, id)
14     upon receiveAck(from, to, id)                 // from = p
15         if to ∈ B then
16             foreach m ∈ B[to] do sendTo(to, m)
17             B ← B \ to
18             Q ← Q ∪ to
19     upon close(q)
20         B ← B \ q
21 DISSEMINATION:
22     function FBC-broadcast+(m)                    // b_p(m)
23         foreach q ∈ B do B[q] ← B[q] ∪ m   // Buffers
24         R-broadcast(m)
25     upon R-deliver(m)
26         foreach q ∈ B do B[q] ← B[q] ∪ m   // Buffers
27         FBC-deliver+(m)                           // d_p(m)
```

Algorithm 2 shows the instructions of our preventive causal broadcast. Its two operations broadcast and deliver rely on reliable broadcast (see Algorithm 1). In fact, without additions nor removals of links, our protocol only executes instructions of reliable broadcast (see Line 24, 25).



(a) Process A broadcasts a.

(b) Process A wants to add a link to Process D. It sends a locked message ℓ to Process D 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 D receives ℓ and acknowledges it to A. The acknowledgment message α can travel by any communication mean.

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

Fig. 4: Preventive causal broadcast does not violate causal order in dynamic networks anymore.

Theorem 2 (FBC-BROADCAST$^+$ is causal in static networks)**.** FBC-BROADCAST$^+$ *is a causal broadcast in static networks.*

Proof. In static networks, FBC-BROADCAST$^+$ only executes the instructions of reliable broadcast. Reliable broadcast along with FIFO links ensures causal broadcast. The proof can be found in Paper [16]. □

The removal of links and the departure of processes are not an issue, for it does not reorder messages traveling through the links[1]. Algorithm 2 does not provide specific instructions for such cases.

Lemma 1 (FBC-BROADCAST$^+$ is causal in dynamic networks subject to removals)**.** FBC-BROADCAST$^+$ *is a causal broadcast in dynamic networks where processes can leave the network or links can be removed.*

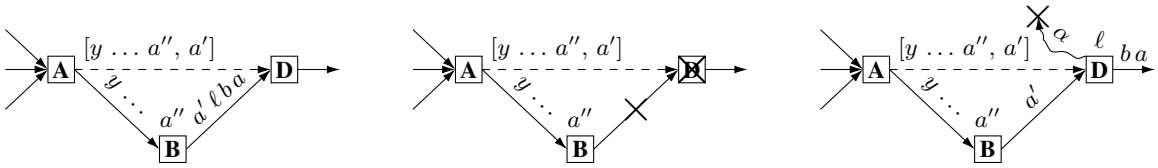
Proof. Removing a process from the network and removing all the incoming and outgoing links of this process is equivalent. We assume that removals do not create network partitions. Removing a link does not change the delivery order of causally related messages. Identically to the proof of Theorem 2, our causal broadcast only executes instructions of reliable broadcast in such cases. The proof is identical to that of Paper [16]. □

Our causal broadcast becomes more sophisticated when the process adds links. Figure 3 shows the issue with link additions. New links may act as shortcut for messages. First messages that travel through new links may arrive before preceding messages that took longer paths. New links create additional diffusion paths that potentially disorder messages. Solving this issue requires that even new links convey all messages in the right order. Sending all broadcast messages since the beginning would be too costly. Instead, a process p creating a link to a process q needs to know the messages received by q to send potentially missing messages in the right order using this link. While obtaining q's acknowledgment would be costly in general settings, using already created FIFO links keeps it cheap. Once missing messages have been sent through this new link, p starts to use it normally. New links do not create diffusion paths that could break causal order.

Compared to reliable broadcast, Algorithm 2 adds a structure associating each new link with a buffer of messages. Figure 4 shows how it solves causal order violations. In Figure 4a, Process A broadcasts a. In Figure 4b, it wants to add a link to Process D. It sends a locked message ℓ to process D (see Line 11) and awaits for the latter's acknowledgment. We leave aside the implementation of this send function (e.g. broadcast or routing). Although, using already established FIFO links constitutes a cheap way to achieve it. While awaiting, Process A keeps its normal functioning and maintain a buffer of messages associated with the new link (see Line 23, 26). In Figure 4c, Process A broadcasts another message a'. It sends it normally to Process B but does not send it to Process D directly. Instead, it buffers it. In Figure 4d, Process D receives Process A's locked message ℓ. Since links are FIFO, it implicitly means that Process D also received a. Process D sends an acknowledgment α to Process A (see Line 13). α can travel through any communication mean. In Figure 4e, Process A receives α. Consequently, Process A knows that Process D received and delivered at least a and all preceding messages. It empties the buffer of messages to Process D (see Line 16). Afterwards, Process A uses the new link normally.

The acknowledgment phase ensures that Process D received all messages preceding a and a itself. The buffering phase ensures that Process D receives all messages between the sending of the locked message ℓ and the receipt of Process D's acknowledgment α. Afterwards, Process D will receive Pro-

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



(a) Slow locked message ℓ. (b) Process D crashes. (c) Acknowledgment α fails.

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

cess A's broadcast or forwarded messages from this new direct link or transitively through Process B.

Lemma 2 (FBC-BROADCAST$^+$ is causal in dynamic networks subject to additions)**.** FBC-BROADCAST$^+$ *is a causal broadcast in dynamic networks where processes can join the network or links can be added.*

Proof. To prove that FBC-BROADCAST$^+$ is a causal broadcast, we must show that it ensures validity, uniform agreement, uniform integrity, and causal order – that is FIFO order and local order.

Validity, uniform agreement, uniform integrity: FBC-BROADCAST$^+$ is an R-BROADCAST.

FIFO: Suppose a process p broadcasts m before m'. Consider that a correct process q delivers m'. We must show that q delivers m before m'. Since adding a FIFO link from a Process r to any other Process s is like adding potential FIFO paths from p to q, we summarize this as a link from p to q with arbitrary settings but still FIFO. Should this be a proof? Maybe a proof of equivalent model if kept.

Since Process q delivers m' it either received m' from already well established links, or from the new one. In the former case, the proof is identical to that of Theorem 2. In the latter case, Process q delivers m' means that Process p broadcasts m then m' either (i) m' during buffering and m before buffering; (ii) m' during buffering and m during buffering; (iii) m' after acknowledgment and m before buffering; (iv) m' after acknowledgment and m during buffering; (v) m' after acknowledgment and m after acknowledgment. We must show that either (1) Process q already received m from another link before receiving m' from the new link, (2) or that this new link conveyed m before m'.

(i) Process p put m' in the buffer. Process p empties the buffer containing m' after p received an acknowledgment meaning that at least m has been received at q. This shows (1).
(ii) Process p put m then m' in the buffer. When the acknowledgment is received, it sends m then m' to Process q. This shows (2).
(iii) Identical to case (i) without the need of buffering.
(iv) Since Process p empties the buffer containing m when it receives the acknowledgment, and since Process p broadcasts m' afterwards using the new link, it shows (2).
(v) It directly shows (2).

Local: Since broadcast and forward have identical instructions, the proof is identical to that of FIFO order.

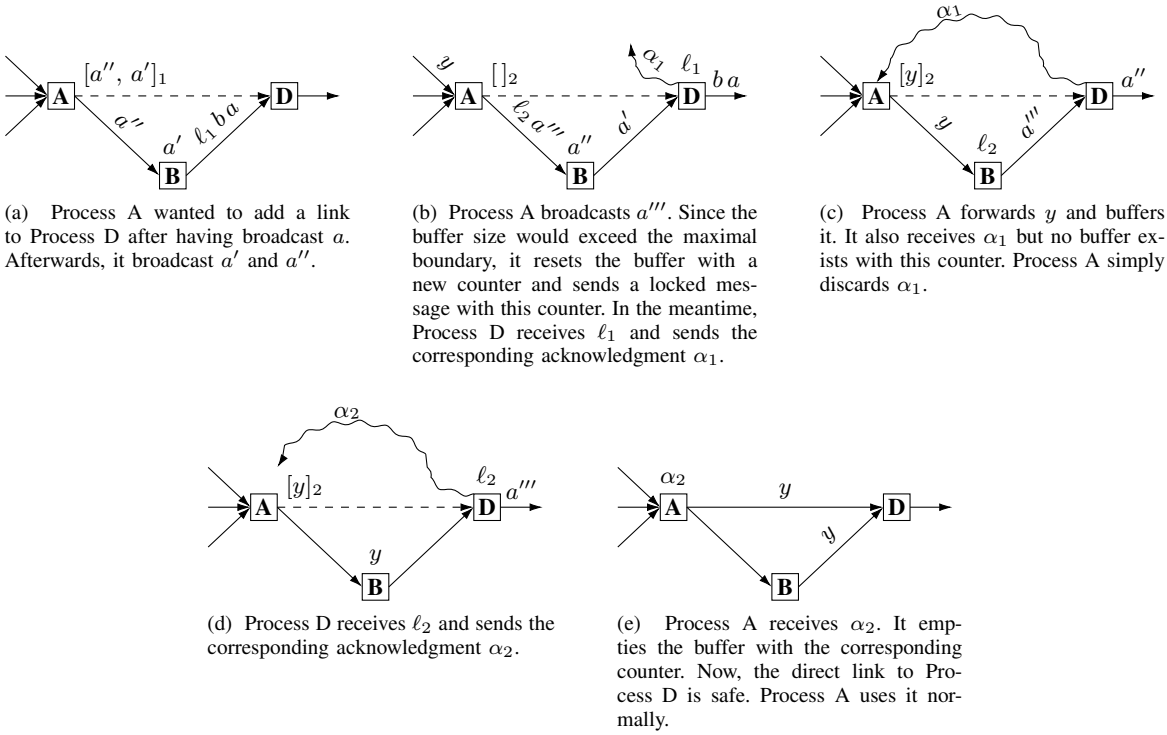
Causal: From Theorem 1, since FBC-BROADCAST$^+$ ensures both FIFO order and local order, it ensures causal order. □

Theorem 3 (FBC-BROADCAST$^+$ is a causal broadcast)**.** FBC-BROADCAST$^+$ *is a causal broadcast in both static and dynamic network settings.*

Proof. For static networks, it comes from Theorem 2. For dynamic networks, it comes from Lemmas 1 and 2. □

Algorithm 2 ensures causal delivery of messages even with dynamic network settings. Compared to preventive causal broadcast for static networks, it uses an additional local structure: buffers of messages. It associates a buffer to each new links not yet acknowledged. We assumed that the size of these buffer stays small in general, for it depends of the time between the sending of locked message and the receipt of its acknowledgment 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 D compared to the number of messages to broadcast and forward. The locked message ℓ did not reach Process D yet that the buffer contains a lot of messages. In Figure 5b, the issue comes from the departure of Process D. Depending on network settings, Process A may not be able to detect Process D's departure. The former will never receive the awaited acknowledgment and the buffer will grow forever. In Figure 5c, the acknowledgment α 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 D 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 D so it sent a locked message; then broadcast a' and a'' so it buffered them. We see that the locked message ℓ_1 carries a counter. The new buffer is identified by the same counter. In Figure 6b, Process A broadcasts another message a'''. 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 a''' to the buffer would exceed the bound of 2 elements. It is first failure, so Process A simply restarts the acknowledgment phase: it resets the buffer and sends another locked message ℓ_2 (see Line 24). The counter of the reset buffer is the one



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

(b) Process A broadcasts a'''. Since the buffer size would exceed the maximal boundary, it resets the buffer with a new counter and sends a locked message with this counter. In the meantime, Process D receives ℓ_1 and sends the corresponding acknowledgment α_1.

(c) Process A forwards y and buffers it. It also receives α_1 but no buffer exists with this counter. Process A simply discards α_1.

(d) Process D receives ℓ_2 and sends the corresponding acknowledgment α_2.

(e) Process A receives α_2. It empties the buffer with the corresponding counter. Now, the direct link to Process D is safe. Process A uses it normally.

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

Algorithm 3: Bounding the size of buffers and handling network failures.

```
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 sendLocked(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 FBC-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)
```

of the new locked message. In the meantime, Process D receives ℓ_1 and sends the corresponding acknowledgment α_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 acknowledgment α_1 but discards it, for no buffer has such counter. In Figure 6d, Process D receives ℓ_2 and sends the corresponding acknowledgment α_2 to Process A. In Figure 6e, Process A receives α_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 acknowledgment 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 are safely acknowledged, 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 locked message. The time taken between the sending and the receipt of the acknowledgment would increase when failures occur. However, it would take less time than resetting the buffering phase.

C. Complexity

We review and discuss about the complexity of FBC-BROADCAST$^+$. We distinguish the complexity brought by (i) causal ordering, (ii) reliable broadcast, (iii) and the overlay network.

Causal ordering. Regarding message overhead, broadcast messages convey two types of control information: data to ensure FIFO order links, and data to globally identify the message. Thus, message overhead is constant. Regarding number of messages, processes must send each message to all their neighborhood exactly once. It creates as many copies as neighbors. This number of copies may constitute an issue when the size of messages is large. To solve this issue, processes can send identifiers instead of large messages then send these messages on demand. It introduces additional delays in communications but greatly reduces generated traffic [21]. Regarding local space consumption, our protocol maintains one buffer per link during its acknowledgment time. We assume that this time is short so the number of buffered messages stays small. As shown in Section III, network conditions can make this assumption false. Algorithm 3 allows to bound the size of each buffer and handle network failures.

Reliable broadcast. Algorithm 1 shows the instructions of reliable broadcast. Even in presence of message duplicates it avoids multiple deliveries of a same message. To achieve this, the most straightforward structure is a set saving all new received messages. However, it increases linearly with the number of delivered messages (REF). Assigning a unique identifier $\langle p, counter \rangle$ to each message changes the complexity. It becomes a vector that increases linearly with the number of processes that ever broadcast a message [13]. Using interval tree clocks [19] slightly overloads messages with identifiers but it improves the space complexity: the local structure increases linearly with the number of processes that are currently involved in broadcasting.

Overlay network. Values such as the number of messages sent by each process, or the number of hops for a message to reach all processes, are interdependent values brought by the overlay network. For instance, random peer-sampling protocols [22] build network overlays with properties similar to those of random graphs [23]. They provide each process with a random subset of neighbors the size of which is logarithmically scaling with the network size. The number of messages sent by each process for each broadcast message is logarithmic. Since random peer-sampling protocols build topologies close to random graphs, messages take a logarithmic number of hops to reach all processes. In addition, random peer-sampling protocols such as SPRAY [24] or CYCLON [25] create links using only neighbor-to-neighbor interactions, i.e., they establish links only two hops apart. It takes only 4 hops for new links to get acknowledged. The overhead brought by these messages is negligible. The acknowledgment time of buffers, hence the size of buffers, remains small.

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

IV. EXPERIMENTATION

Define what we want. We measure the time between the broadcast of a message and its delivery by all processes in the network. We expect that our measurements slowly increase as the network becomes more dynamic.

V. RELATED WORK

This section reviews the related work going from piggybacking approaches to vector approaches and their following compaction approaches.

Piggybacking approaches [11], [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 newly broadcast message. Similarly to our approach, these broadcast protocols can deliver messages as soon as they arrive. However, even by piggybacking the identifiers of messages instead of messages themselves, the broadcast message size may increase quickly depending on the application. In our approach, a similar behavior arises during buffering. However, as discussed in Section III-C, we can assume that links are quickly checked so the buffer size stays small, and we can easily set a threshold on the buffer size (see Algorithm 3).

Vector clock approaches [13], [14]. 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. These approaches are blocking while our approach remains non-blocking. In addition, 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 for each process of the network. To accurately track causality, processes cannot share their entry. To safely track causality, processes cannot reclaim entries. Hence, even with **compaction approaches** [15], the vectors grow linearly in terms of number of processes that ever broadcast a message. Paper [19] reduces this complexity to depend of the actual number of processes in the network. Still, these approaches do not scale, particularly in dynamic networks subject to churn and failures.

Probabilistic approaches [20] sacrifices on causality tracking accuracy: messages may be delivered out of order under a computable boundary. The size of control information in messages depends of the desired boundary. In comparison to these vector-based approaches, our broadcast cannot compare any pair of messages. Nevertheless, accurate causal delivery is a feature provided by default by the propagation scheme. It frees up messages from the need to piggyback any control information [16].

Topological approaches. A trivial way to ensure causal delivery of messages without message overhead consists in building an overlay network shaped as a ring: each broadcast message loops once in FIFO order (REF). Unfortunately, maintaining a specific topology can prove costly in dynamic networks subject to churn. Our approach relies on a peer-sampling protocol (Relies on R-broadcast that relies on such psp) but we make

no assumption on the built topology. One can choose the most suitable topology depending on network settings. For instance, random peer-sampling protocols [22] have a low upkeep and guarantee a connected network even in dynamic settings.

Explain super-peer approaches.

Inter-group broadcast is a kind of topological approach.

Inter-group broadcast [26], [27]. Inter-group broadcast allows processes to broadcast messages to members of several inter-connected networks. Paper [27] states that inter-group causal broadcast is ensured when groups that internally ensure causal broadcast are linked together by communication channels that ensure FIFO ordering of messages. One can employ these approaches to cluster the network in subsystems. Control information in messages depends of the size of each small subsystem. Our approach is an extreme case where each process is a subsystem.

VI. Conclusion

In this paper, we described a uniform reliable causal broadcast protocol that breaks scalability barriers for large and dynamic networks. It extends preventive causal broadcast complexity to dynamic networks. Among others, message overhead and message delivery time remain constant. This result means that causal broadcast finally becomes affordable and efficient in large and dynamic systems.

As future work, we plan to investigate on reducing the space complexity of reliable broadcast. Section III-C reviews structures with linearly increasing space consumption. We can reduce this complexity in static networks. 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. Unfortunately, this does not hold in dynamic networks. We would like to investigate on a way to prune the structure in such settings. This would make causal broadcast fully scalable as well on generated traffic as on space consumption.

We also plan to investigate on retrieving partial order of events. Section V 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. Preventive approaches 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 network settings where processes have partial knowledge of the membership. We would like to analyze its minimal cost.

Acknowledgments

Morbi tincidunt posuere arcu. Cras venenatis est vitae dolor. Vivamus scelerisque semper mi. Donec ipsum arcu, consequat scelerisque, viverra id, dictum at, metus. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut pede sem, tempus ut, porttitor bibendum, molestie eu, elit. Suspendisse potenti. Sed id lectus sit amet purus faucibus vehicula. Praesent sed sem non dui pharetra interdum. Nam viverra ultrices magna.

References

[1] V. Hadzilacos and S. Toueg, “A modular approach to fault-tolerant broadcasts and related problems,” Ithaca, NY, USA, Tech. Rep., 1994.

[2] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” 03 2009.

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “A comprehensive study of Convergent and Commutative Replicated Data Types,” INRIA, Rapport de recherche RR-7506, Jan. 2011. [Online]. Available: http://hal.inria.fr/inria-00555588

[4] B. Nédelec, P. Molli, and A. Mostefaoui, “Crate: Writing stories together with our browsers,” in *Proceedings of the 25th International Conference Companion on World Wide Web*, ser. WWW ’16 Companion. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2016, pp. 231–234. [Online]. Available: http://dx.doi.org/10.1145/2872518.2890539

[5] A. Mostéfaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. Abbadi, “From static distributed systems to dynamic systems,” in *24th IEEE Symposium on Reliable Distributed Systems, 2005. SRDS 2005.* IEEE, 2005, pp. 109–118.

[6] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[7] R. Schwarz and F. Mattern, “Detecting causal relationships in distributed computations: In search of the holy grail,” *Distributed Computing*, vol. 7, no. 3, pp. 149–174, Mar 1994. [Online]. Available: https://doi.org/10.1007/BF02277859

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” *Stabilization, Safety, and Security of Distributed Systems*, pp. 386–400, 2011.

[9] P. Bailis and A. Ghodsi, “Eventual consistency today: Limitations, extensions, and beyond,” *Queue*, vol. 11, no. 3, pp. 20:20–20:32, Mar. 2013. [Online]. Available: http://doi.acm.org/10.1145/2460276.2462076

[10] B. Charron-Bost, “Concerning the size of logical clocks in distributed systems,” *Information Processing Letters*, vol. 39, no. 1, pp. 11–16, Jul. 1991. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(91)90055-M

[11] K. P. Birman and T. A. Joseph, “Reliable communication in the presence of failures,” *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 47–76, Jan. 1987. [Online]. Available: http://doi.acm.org/10.1145/7351.7478

[12] V. Hadzilacos and S. Toueg, “Distributed systems (2nd ed.),” S. Mullender, Ed. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1993, ch. Fault-tolerant Broadcasts and Related Problems, pp. 97–145. [Online]. Available: http://dl.acm.org/citation.cfm?id=302430.302435

[13] C. J. Fidge, “Timestamps in message-passing systems that preserve partial ordering,” vol. 10, pp. 56–66, 02 1988.

[14] F. Mattern, “Virtual time and global states of distributed systems,” *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[15] M. Singhal and A. Kshemkalyani, “An efficient implementation of vector clocks,” *Information Processing Letters*, vol. 43, no. 1, pp. 47–52, Aug. 1992. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(92)90028-T

[16] R. Friedman and S. Manor, “Causal ordering in deterministic overlay networks,” *Israel Institute of Technology: Haifa, Israel*, 2004.

[17] M. Mukund, G. Shenoy R., and S. Suresh, “Optimized or-sets without ordering constraints,” in *Distributed Computing and Networking*, ser. Lecture Notes in Computer Science, M. Chatterjee, J.-n. Cao, K. Kothapalli, and S. Rajsbaum, Eds. Springer Berlin Heidelberg, 2014, vol. 8314, pp. 227–241. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-45249-9_15

[18] S. Weiss, P. Urso, and P. Molli, “Logoot: a scalable optimistic replication algorithm for collaborative editing on p2p networks,” in *ICDCS’09. 29th IEEE International Conference on Distributed Computing Systems.* IEEE, 2009, pp. 404–412.

[19] P. S. Almeida, C. Baquero, and V. Fonte, “Interval tree clocks,” in *Proceedings of the 12th International Conference on Principles of Distributed Systems*, ser. OPODIS ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 259–274. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-92221-6_18

[20] A. Mostéfaoui and S. Weiss, “Probabilistic causal message ordering,” in *Proceedings of Parallel Computing Technologies: 14th International Conference, PaCT 2017, Nizhny Novgorod, Russia, September 4-8, 2017.* Cham: Springer International Publishing, 2017, pp. 315–326. [Online]. Available: https://doi.org/10.1007/978-3-319-62932-2_31

[21] D. Frey, R. Guerraoui, A.-M. Kermarrec, and M. Monod, "Live Streaming with Gossip," Inria Rennes Bretagne Atlantique ; RR-9039, Research Report, Mar. 2017. [Online]. Available: https://hal.inria.fr/hal-01479885

[22] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8, 2007.

[23] P. Erdős and A. Rényi, "On random graphs i." *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.

[24] B. Nédelec, J. Tanke, D. Frey, P. Molli, and A. Mostéfaoui, "An adaptive peer-sampling protocol for building networks of browsers," *World Wide Web*, Aug 2017. [Online]. Available: https://doi.org/10.1007/s11280-017-0478-5

[25] S. Voulgaris, D. Gavidia, and M. van Steen, "Cyclon: Inexpensive membership management for unstructured p2p overlays," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 197–217, 2005. [Online]. Available: http://dx.doi.org/10.1007/s10922-005-4441-x

[26] S. Johnson and F. Jahanian, "Scalable group composition with end-to-end delivery semantics," Tech. Rep., 1998.

[27] S. Johnson, F. Jahanian, and J. Shah, "The inter-group router approach to scalable group composition," in *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, 1999, pp. 4–14.