# Causal Mutual Byzantine Broadcast

Mathieu Féry, Vincent Kowalski, Florian Monsion, Achour Mostefaoui,
Samuel Pénault, Matthieu Perrin, Guillaume Poignant

# Causal Mutual Byzantine Broadcast

M. Féry, V. Kowalski, F. Monsion, A. Mostefaoui,
S. Pénault, M. Perrin and G. Poignant
LS2N, Nantes Université

**Abstract**

Recently, a new communication abstraction called Mutual Broadcast has been proposed for message-passing distributed systems where some processes may fail by crashing. It is a one-to-all broadcast abstraction providing an ordering property that allows it to be computationally equivalent to atomic registers. This paper proposes an adaptation of this abstraction, *Causal Mutual Byzantine Broadcast* (in short CMB-Broadcast) for message-passing systems where some processes may experience Byzantine faults. Byzantine faults are a more severe failure model compared to crash failures. A Byzantine process can behave arbitrarily. After defining this new communication abstraction, we show how it can be used to emulate atomic registers and also how it can be implemented using quorums and the famous Byzantine reliable broadcast abstraction of Bracha. We also prove a necessary condition on the size of the quorums.

**keywords:** Atomic register, Byzantine process, Communication abstraction, Message-passing system, Quorums.

## 1 Introduction

**Context: The Mutual Broadcast Abstraction** Distributed systems consist of a collection of processes that communicate either through shared memory or by sending and receiving messages to solve a common problem. When physical shared memory is not available, one approach is to emulate shared memory algorithms atop a message-passing system by implementing shared registers in this model [1]. Such algorithms rely on quorums to overcome partitions: each time a process reads or writes to a shared register, it broadcasts a message and waits for acknowledgment from a majority of the system's processes. However, designing quorum-based algorithms can be challenging due to their heavy reliance on the specific model employed.

Recently, Mutual Broadcast has been identified as a broadcast abstraction computationally equivalent to atomic registers [6, 5]. This equivalence implies that atomic registers can be emulated in a distributed system where Mutual Broadcast is available, and conversely, Mutual Broadcast can be implemented in systems where processes communicate using atomic registers. In other words, providing Mutual Broadcast as a basic building block of a message-passing system could replace quorums in solving many cooperation problems.

Although Mutual Broadcast seems to be the right abstraction to deal with many safety issues in distributed systems, those systems may also be prone to security issues. Byzantine failures, first introduced by Pease, Shostak and Lamport [12], occur when a process does not execute as expected. This can happen when for example malicious users are actively trying to attack the system by altering the code of their client application, or simply when they run outdated or corrupted versions of the application. The goal of this paper is to study Mutual Broadcast in distributed systems prone to Byzantine failures.

**Byzantine Atomic Registers** Many variants of shared registers have been studied in the literature, encompassing both crash-prone environments [9] and Byzantine contexts. Three types of single-writer multi-reader (SWMR) registers were compared in [8]: the classical read/write register [4], the read/write-increment register [10], and the read/append register [7]. It has been demonstrated that, although these three types of registers are computationally equivalent in crash-prone environments, they differ in Byzantine contexts.

Specifically, while read/append registers can implement the other two types of registers, the reverse is not true. Consequently, this paper focuses on the implementation of the strongest variant, namely the atomic SWMR read/append register, which tracks the sequence of all values written by the writer (whether Byzantine or not). This sequence is perceived identically by all non-Byzantine processes (read/append register). If the writer is Byzantine, the sequence depends on its behavior; in extreme cases, the history might be reduced to its initial state (an empty sequence).

**Contribution 1: The Causal Mutual Broadcast Abstraction** This paper investigates the adaptation of the mutual broadcast abstraction to Byzantine contexts, specifically, a broadcast abstraction that effectively captures the computing requirements for implementing an atomic SWMR read/append register in Byzantine-prone systems. The first major contribution is the definition of Causal Byzantine Mutual Broadcast (CMB-broadcast). In addition to the traditional properties of mutual broadcast, CMB-broadcast incorporates the causal ordering of message deliveries [2]. Subsequently, the paper presents an algorithm that utilizes the CMB-broadcast abstraction for implementing a SWMR read/append register, demonstrating resilience against any number of Byzantine failures.

**Contribution 2: Implementation of CMB-broadcast** We then illustrate how to implement the CMB-Broadcast abstraction atop a message-passing system, under the condition that at most $t < n/3$ processes may exhibit Byzantine behavior, where $n$ is the total number of processes in the system. Specifically, our implementation is based on FIFO Byzantine reliable broadcast, a variant of Bracha's Byzantine reliable broadcast. This variant ensures FIFO ordering of messages, maintaining this order consistency regardless of whether the sender is correct or Byzantine.

**Contribution 3: The Computational Power of FIFO-Broadcast** Interestingly, the reduction of CMB-Broadcast to FIFO Byzantine reliable broadcast requires the assumption that a majority of the processes are correct, specifically $t < n/2$. This is the same bound as in the crash-prone context. On the other hand, we establish that if the FIFO property in the FIFO Byzantine reliable broadcast is limited only to correct senders, then it becomes necessary that $t < n/3$ to implement CMB-Broadcast. This stresses, for the first time, the role of the FIFO property in enforcing the power of reliable broadcast in a Byzantine context.

**Roadmap** This paper is composed of 7 sections. First, Section 2 presents the underlying computing model, then Section 3 introduces the high-level CMB-Broadcast communication abstractions. The two following sections propose two implementations. Section 4 details an implementation of an atomic register using the proposed CMB-Broadcast communication abstraction and Section 5 gives an implementation of the CMB-Broadcast communication abstractions using the well-known Byzantine reliable broadcast abstraction of Bracha. Section 6 proves an impossibility result on the necessary condition to upgrade from the Byzantine reliable broadcast to CMB-Broadcast. Finally Section 7 concludes the paper. For the paper to be self-contained, we have added the implementation of a Byzantine reliable FIFO broadcast based on Bracha's algorithm in Appendix A, the proof of Algorithm 1 in Appendix B, and parts of the proof of Algorithm 2 in Appendix C.

# 2 Computing Model

In this paper, we consider the classical Byzantine-prone asynchronous message-passing computing model.

**Computing entities** The system is composed of a set of $n$ sequential processes, denoted $p_1, p_2, \ldots, p_n$. These processes are asynchronous in the sense that each process progresses at its own speed, which can be arbitrary and may vary along any execution, and remains always unknown to the other processes. Each process $p_i$ has access to its own identifier $i$ which can be used in the code.

**Failure model** Among the $n$ processes of the system, it is supposed that at most $t$ processes can exhibit a Byzantine behavior. A Byzantine behavior is characteristic of a process not following its algorithm and acting in an arbitrary way [12]: it may start in an arbitrary state, stop executing at any time (this behavior is called a crash), perform arbitrary state transitions, attempt to communicate arbitrary or different values

to different processes, etc. A Byzantine process is also called a *faulty* process, and a process that commits no failure (i.e., a non-Byzantine process) is called a *correct* process.

**Communication model** The different processes communicate by exchanging messages through bi-directional communication channels. These channels connect each pair of processes so that any process can identify the sender of a message and no process can impersonate another correct process. The sending of a message is asynchronous and reliable. "Asynchronous" means that there is no bound on message transfer delay, and "Reliable" means that channels do not create, duplicate, or modify information and that all messages sent by correct processes to correct processes will eventually be received.

**Notation** The acronym $\mathcal{BAMP}_{n,t}[\emptyset]$ is used to denote the previous Byzantine-prone Asynchronous Message-Passing model without additional computability power. $\mathcal{BAMP}_{n,t}[H]$ denotes $\mathcal{BAMP}_{n,t}[\emptyset]$ enriched with the additional computational power denoted by $H$. For example, $\mathcal{BAMP}_{n,t}[t < \frac{n}{3}]$ denotes the model in which at least two-thirds of the processes are correct, namely, $t < n/3$.

# 3 Causal Mutual Byzantine Broadcast

Causal Mutual Byzantine broadcast (CMB-Broadcast) is an extension, to the Byzantine context, of the Mutual broadcast abstraction proposed in the crash failure model in [6]. This communication abstraction allows a process to broadcast a message that will be delivered, at least, by all the correct processes, ensuring a certain ordering property among the delivered messages. It provides the processes with one operation denoted cmb_broadcast(), and one event denoted by cmb_deliver.

A process $p_i$ invokes the operation "cmb_broadcast(TYPE($m$))" to broadcast a message with type TYPE and content $m$. This action is referred to as $p_i$ cmb-broadcasting a message TYPE($m$). Subsequently, the event "cmb_deliver TYPE($m$) **from** $p_i$" might be triggered at some processes $p_j$, leading us to say that $p_j$ "cmb-delivers $m$ from $p_i$". It is assumed that, although messages may share type and content, each broadcast message is unique. The following properties define CMB-Broadcast.

**Validity.** If a correct process $p_i$ cmb-delivers a message $m$ from a correct process $p_j$, then $p_j$ previously invoked cmb_broadcast($m$).

**Integrity.** A correct process cmb-delivers a message $m$ at most once.

**Local progress.** If a correct process $p_i$ cmb-broadcasts a message $m$, then $m$ will eventually be cmb-delivered by $p_i$ from $p_i$.

**Consistency.** If a correct process cmb-delivers a message $m$ from some process $p_j$, then all correct processes will eventually cmb-deliver $m$ from $p_j$ ($p_j$ may be correct or Byzantine).

**Mutual ordering.** For any pair of correct processes $p$ and $p'$, if $p$ cmb-broadcasts a message $m$ and $p'$ cmb-broadcasts a message $m'$, it is not possible that $p$ cmb-delivers $m$ before $m'$ and $p'$ cmb-delivers $m'$ before $m$.

**CS-Causal ordering.** If a correct process $p_i$ cmb-delivers a message $m$, and then cmb-broadcasts a message $m'$, then no correct process cmb-delivers $m'$ before $m$.

**CS-Fifo ordering.** If a correct process $p_i$ cmb-broadcasts a message $m$ before a message $m'$, then no correct process cmb-delivers $m'$ before $m$.

**BS-Fifo ordering.** If a correct process $p_i$ cmb-delivers a message $m$ before a message $m'$, both from the same process $p_j$ ($p_j$ may be correct or not), then no correct process cmb-delivers $m'$ before $m$.

```
1 operation append(v) is:
2 |  synchro_cmb_broadcast(APPEND(v));
3 operation read() is:
4 |  synchro_cmb_broadcast(SYNCH());
5 |  let log_i ← replica_i.read();
6 |  synchro_cmb_broadcast(SYNCH());
7 |  return log_i;
8 when p_i cmb_delivers APPEND(v) from p_w do:
9 |  replica_i.append(v);
```

**Algorithm 1:** Implementation of an atomic SWMR read/append register using CMB-broadcast

The properties of Validity, Integrity, Local Progress, and Consistency are classical in defining Byzantine-tolerant broadcast abstractions and constitute the core of the Byzantine-reliable broadcast abstraction proposed by Bracha [3]. The Mutual Ordering property characterizes mutual broadcast. It stipulates that when two processes initiate broadcasts concurrently, at least one must cmb-deliver the other's message prior to its own. This condition prevents partitioning, thereby facilitating the implementation of an atomic register in crash-prone systems. The CS-Fifo Ordering and CS-Causal Ordering properties, where CS denotes "correct sender", mirror the conventional attributes of FIFO and causal broadcasts in crash-prone systems, as outlined in [2], but are confined to messages broadcast by correct processes. Given a Byzantine sender, referring to order in which it broadcasts and delivers messages is impossible. Instead, the BS-Fifo Ordering property states that correct processes must concur on a uniform delivery sequence for any broadcasting process, aligning with the emission order if the sender is indeed correct.

It is important to note that if a process consistently waits for the local delivery of its previous messages before initiating a subsequent broadcast, then both CS-Causal Ordering and BS-Fifo Ordering inherently imply CS-Fifo Ordering. To facilitate this behavior, we introduce a blocking variant of cmb_broadcast, denoted as synchro_cmb_broadcast. This function is defined as:

cmb_broadcast $m$; wait until $m$ has been cmb-delivered locally.

# 4 Implementation of a SWMR atomic register

Algorithm 1 implements an atomic SWMR read/append register in the model $\mathcal{BAMP}_{n,t}$[cmb_broadcast]. It follows closely the algorithm implementing an atomic read/write register on top of mutual broadcast, in a crash-prone model [6]. In particular, this is a full-replication protocol. Hence, each correct process $p_i$ maintains a local variable, called $\mathsf{replica}_i$, whose sequence of states is the same as the sequence of abstract states of the shared read/append register. We will now describe how each safety property of CMB-broadcast serves a precise role in the algorithm.

When the writing process $p_w$ wants to append a value $v$ to its register, it invokes append($v$). Accordingly, it cmb-broadcasts a message APPEND($v$) (Line 2) to inform each other process $p_i$ there is new value to append to its local variable $\mathsf{replica}_i$ (Line 9). Thanks to the Validity, Integrity and BS-Fifo ordering properties of cmb-broadcast, all correct processes receive the same set of append($v$) messages, in the same order. Therefore, the different variables $\mathsf{replica}_i$ take the same sequence of values and remain consistent with each other. Note the importance of the BS-Fifo ordering property to ensure that values are appended in the same order, even if $p_w$ is Byzantine.

A correct process $p_i$ can read the shared register by invoking the read() operation. Process $p_i$ returns the value stored in its variable $\mathsf{replica}_i$ (Lines 5 and 7), but a memory barrier must be posed before and after the read in order to ensure the real-time ordering imposed by linearizability (Lines 4 and 6).

Line 4 ensures the read-after-write property which states that, if a read starts after an append has ended,

```
1  operation cmb_broadcast(m) is:
2      sending_i ← m; acked_i ← ∅;
3      fifo_broadcast MSG(m, i);
4      wait until m ∈ delivered_i;
5  when p_i fifo_delivers MSG(m, k) as the sn-th message from p_j do:
6      wait until next_i[j] = sn;
7      if m = sending_i then acked_i ← acked_i ∪ {j};
8      if j = k ∧ m ∉ delivered_i then
9          if i = k then wait until |acked_i| > n − t;
10         else fifo_broadcast MSG(m, k);
11         cmb_deliver m from p_k;
12         delivered_i ← delivered_i ∪ {m};
13     wait until m ∈ delivered_i;
14     next_i[j] ← sn + 1;
```

**Algorithm 2:** Implementation of CMB-broadcast

then the read must return a list of values containing the appended value. This is achieved thanks to the MUTUAL ORDERING property of CMB-broadcast: before returning from its append operation, $p_w$ waits until it has cmb-delivered its APPEND message locally (recall that synchro_cmb_broadcast $m$ is the blocking version of cmb_broadcast). Hence, $p_w$ cmb-delivers its own APPEND message before any SYNCH message cmb-broadcast by any other process $p_i$ after the end of the append$(v)$ operation. This ensures that the process $p_i$ must cmb-deliver these two messages in the same order. Hence, $p_i$ cmb-delivers APPEND$(v)$ before the end of Line 4, and replica$_i$ is up-to-date on Line 5 when it is read.

Line 6 ensures the read-after-read property which means that, if a read by a process $p_i$ starts after another read by $p_j$ has ended, the value returned by $p_i$ must be at least as recent as the value returned by $p_j$. In this case, the SYNCH message cmb-broadcast by $p_j$ at Line 6 serves the same synchronyzation purpose as $p_w$'s APPEND message: $p_i$ cmb-delivers $p_j$'s message before its own. Moreover, the CS-CAUSAL ORDERING property of cmb-broadcast ensures that $p_i$ cmb-delivers all APPEND messages cmb-delivered by $p_j$ before its read, which ensures an up-to-date read.

The proof of Algorithm 1 is in Appendix B. It follows the same framework as [6], which proposes a set of necessary and sufficient properties, that characterize a Byzantine-linearizable SWMR read/append register.

# 5 Implementation of CMB-broadcast

This section presents Algorithm 2 that implements CMB-broadcast. For the sake of simplicity, it is not built directly on a distributed message-passing system in the model $\mathcal{BAMP}_{n,t}[t < \frac{n}{3}]$, but is rather presented in a two-step design. We use another broadcast abstraction, called fifo-broadcast, as a building block. This abstraction ensures that the different broadcast messages are delivered in FIFO order. Specifically, fifo-broadcast imposes the following properties, defined in the same way as CMB-broadcast: VALIDITY, INTEGRITY, LOCAL PROGRESS, CONSISTENCY, CS-FIFO ORDERING and BS-FIFO ORDERING. Fifo-broadcast can be implemented by a variant of the Byzantine reliable broadcast of Bracha [3] enriched with sequence numbers. Its code is given in Appendix A.

Each process $p_i$ executing Algorithm 2 manages the following local variables:

- next$_i[1..n]$: This is an array of size $n$, where the entry next$_i[j]$, initialized to 1, gives the sequence number of the next message $p_i$ will fifo-deliver from $p_j$.
- sending$_i$: Variable used to save the last message sent by process $p_i$ to recognize later the answers of the other processes concerning this same message.

- $\mathsf{acked}_i$: The set of processes from which $p_i$ has fifo-delivered a message $\text{MSG}(\mathsf{sending}_i, i)$.
- $\mathsf{delivered}_i$: A set containing all the messages previously cmb-delivered by $p_i$.

When a process $p_i$ wants to cmb-broadcast a message $m$, it stores this message in the variable $\mathsf{sending}_i$ and sets the set of acknowledgments $\mathsf{acked}_i$ to $\emptyset$. Then, it fifo-broadcasts it and waits until this message has been cmb-delivered to itself (at Line 12). When a process $p_i$ fifo-delivers a message $\text{MSG}(m, k)$ from $p_j$, it first executes Line 6 that allows process $p_i$ to consider the messages it receives from the different processes in FIFO order. The fifo-delivered message $m$ may correspond to two different cases.

1. $k = j$: the message $m$ corresponds to a message process $p_j$ wants to cmb-broadcast and that process $p_i$ fifo-delivered. If moreover $j = i$, this means that $p_i$ fifo-delivered this message from itself.
2. $k = i \neq j$: the message corresponds to an acknowledgment sent by process $p_j$ to process $p_i$ for the message $m$ $p_i$ wants to cmb-broadcast.

The test in Line 7 means that $m$ is a message $p_i$ wants to cmb-broadcast ($i = k$) and $p_i$ considers the present message as an acknowledgment from $p_j$ for message $m$ it fifo-broadcast at Line 3 and updates the variable $\mathsf{acked}_i$ accordingly. Then if $k = j$, this means that $p_i$ fifo-delivered a message $m$ process $p_j$ wants to cmb-broadcast. Two cases are to be considered, if $i = k(= j)$, $p_i$ fifo-delivered its own message $m$, it consequently waits until it has received at least $n - t$ acknowledgments before cmb-delivering $m$ at Line 11. Otherwise, $i \neq k$ meaning that $p_i$ has to acknowledge message $m$ to $p_k(k = j)$. This is done at Line 11. This is done to ensure the MUTUAL ORDERING property. Line 13 is used to ensure the CAUSAL ORDERING property: if a correct process $p_j$ has cmb-delivered a message $m$ from some process $p_k$ before cmb-broadcasting a message $m'$, then $p_j$ has also fifo-broadcast a message $\text{MSG}(m, k)$ before its message $\text{MSG}(m', j)$, so Line 13 ensures that $p_i$ has cmb-delivered $m$ from $p_k$ before treating the message $\text{MSG}(m', j)$ and eventually cmb-delivering $m'$ from $p_j$.

**A remark on complexity.** Since the assumption $t < \frac{n}{3}$ is necessary to implement fifo-broadcast, practical uses for Algorithm 2 are to be considered in the model $\mathcal{BAMP}_{n,t}[t < \frac{n}{3}]$. In this model, the message on Line 10 can be sent to all processes, using fifo point-to-point communication channels, instead of relying on fifo-broadcast. The only difference in the correctness demonstration appears in the proof of the MUTUAL ORDERING property: $\mathsf{acked}_i$ eventually contains the identifiers of $n - t$ processes whose message $\text{MSG}(m_i, i)$ is fifo-delivered to $p_i$ before the message $\text{MSG}(m_j, j)$ related to a concurrent message $m_j$ from $p_j$. In Algorithm 2, $\mathsf{acked}_i \cap \mathsf{acked}_j$ must contain some process, which can be correct or Byzantine, so $t < n/2$ is sufficient. In the modified version, $\mathsf{acked}_i \cap \mathsf{acked}_j$ must contain a correct process, which is true when $t < n/3$.

**Correctness proof of Algorithm 2**

**Lemma 1.** *Let $S$ be the set of pairs $\langle \text{MSG}(m, k), j\rangle$, such that $\text{MSG}(m, k)$ is fifo-delivered by all correct processes, and $m$ is cmb-delivered from $p_k$ by some correct process. For all pairs $\langle \text{MSG}(m, k), j\rangle \in S$ and all correct processes $p_i$, $p_i$ eventually executes Line 14 in the code triggered by the fifo-delivery of $\text{MSG}(m, k)$ from $p_j$.(The proof is given in Appendix C.)*

**Theorem 1** (Correctness of Algorithm 2)**.** *Algorithm 2 implements CMB-broadcast in the model $\mathcal{BAMP}_{n,t}[t < \frac{n}{2}, \mathsf{fifo\_broadcast}]$.*

*Proof.* Our proof will go through all properties presented in Section 3 to characterize CMB-broadcast.

**VALIDITY.** Suppose a correct process $p_i$ cmb-delivers a message $m$ from a correct process $p_j$. This happens on Line 11, after $p_i$ fifo-delivered a message $\text{MSG}(m, j)$ from $p_j$. By the VALIDITY property of fifo-broadcast, $p_j$ fifo-broadcast $\text{MSG}(m, j)$ on Line 3 after invoking $\mathsf{cmb\_broadcast}(m)$.

**INTEGRITY.** Suppose a correct process $p_i$ cmb-delivers a message $m$ from a process $p_j$. This happens at most once on Line 11, since $m \notin \mathsf{delivered}_i$ before (Line 8) and $m \in \mathsf{delivered}_i$ afterward (Line 12).

**Local progress.** Suppose a correct process $p_i$ cmd-broadcasts a message $m$. Then, $p_i$ will fifo-broadcast $\text{MSG}(m, i)$ at Line 3 and will eventually fifo-deliver $\text{MSG}(m, i)$ from $p_i$ (itself). By Lemma 1, all correct processes have executed Line 14 for all the messages fifo-broadcast by $p_i$ prior to the message $\text{MSG}(m, i)$. Hence, all correct processes will treat the message $\text{MSG}(m, i)$ from $p_i$ and cmb-deliver $m$ from $p_i$. By Lemma 1 again, $\langle \text{MSG}(m, i), i \rangle \in S$ so $p_i$ eventually executes Line 14, after having cmb-delivered $m$ from $p_i$.

**Consistency.** Suppose a correct process $p_i$ cmd-delivers a message $m$ from a process $p_k$. According to Line 5, $p_i$ has fifo-delivered $\text{MSG}(m, k)$ from $p_k$, so all correct processes fifo-deliver $\text{MSG}(m, k)$ from $p_k$ by the Consistency property of fifo-broadcast. Hence, $\langle \text{MSG}(m, k), k \rangle \in S$ and all correct processes eventually cmb-deliver $m$ from $p_k$ by Lemma 1.

**Mutual ordering.** Let $p_i$ and $p_j$ be two correct processes such that $p_i$ cmb-broadcasts a message $m_i$ and $p_j$ cmb-broadcasts a message $m_j$, and suppose, for contradiction, that $p_i$ cmb-delivers $m_i$ before $m_j$ and $p_j$ cmb-delivers $m_i$ before $m_j$. Let $S_i$ be the set $\mathsf{acked}_i$ when $p_i$ cmb-delivers $m_i$, and $S_j$ be the set $\mathsf{acked}_j$ when $p_j$ cmb-delivers $m_j$. We observe that, for all $k \in S_i$, $p_i$ fifo-delivered $\text{MSG}(m_i, i)$ from $p_k$ before $\text{MSG}(m_j, j)$ from $p_k$. Indeed, otherwise, $m_j \in \mathit{Delivered}_i$ between the two fifo-deliveries (thanks to Line 13), which contradicts the fact that $p_i$ cmb-delivered $m_i$ before $m_j$. Similarly, for all $k \in S_j$, $p_j$ fifo-delivered $\text{MSG}(m_j, j)$ from $p_k$ before $\text{MSG}(m_i, i)$ from $p_k$. Therefore, the CS-Fifo ordering property of fifo-broadcast implies that for all $k \in S_j$, $p_i$ fifo-delivered $\text{MSG}(m_j, j)$ from $p_k$ before $\text{MSG}(m_i, i)$ from $p_k$. In other words, $S_i$ and $S_j$ are disjoint. Therefore, by Line 9 we have $n \geq |S_i \cup S_j| = |S_i| + |S_j| > n$, which is a contradiction.

**CS-Fifo ordering.** Let $p_i$ and $p_j$ be two correct processes, and suppose that $p_i$ cmb-broadcasts two messages $m$ and then $m'$. By Line 3, $p_i$ fifo-broadcasts $\text{MSG}(m, i)$ and $\text{MSG}(m', i)$ in that order. By the CS-Fifo ordering property of fifo-broadcast, $p_j$ fifo-delivers $\text{MSG}(m, i)$ before $\text{MSG}(m', i)$ from $p_i$. Then, by Lines 14 and 6, $p_j$ cmb-delivers $m$ before $m'$.

**BS-Fifo ordering.** Suppose that two correct processes $p_i$ and $p_j$ cmb-deliver two messages $m$ and $m'$, both from the same process $p_k$. By Lines 8 and 11, the cmb-delivery of $m$ (resp. $m'$) happened after the fifo-delivery of a message $\text{MSG}(m, k)$ (resp. $\text{MSG}(m', k)$) from $p_k$. Moreover, by Lines 14 and 6, these fifo-deliveries happened in the same order as the cmb-deliveries, and in the same order for both $p_i$ and $p_j$, thanks to the BS-Fifo ordering property of fifo-broadcast. Hence $p_i$ and $p_j$ cmb-deliver $m$ and $m'$ in the same order.

**CS-Causal ordering.** Suppose a correct process $p_i$ cmb-delivers a message $m$ from $p_j$, and then cmb-broadcasts a message $m'$. Let $p_k$ be a correct process, and let us prove that $p_k$ does not cmb-deliver $m'$ before $m$. If $i = j$, the property is implied by the CS-Fifo ordering property proven above. Otherwise, the following events happen in this order at Process $p_i$: $p_i$ fifo-broadcast $\text{MSG}(m, j)$ (Line 10) before cmb-delivering $m$ from $p_j$ (Line 11), then $p_i$ cmb-broadcast $m'$ before fifo-broadcasting $\text{MSG}(m', i)$ (Lines 1 and 3). Hence, the CS-Fifo ordering property of fifo-broadcast implies that $p_k$ fifo-delivered $\text{MSG}(m, j)$ from $p_i$ before fifo-delivering $\text{MSG}(m', i)$ from $p_i$, and only later cmb-delivered $m'$ from $p_i$. By Lines 13, 14 and 6, $p_k$ cmb-delivered $m$ before cmb-delivering $m'$. □

# 6 An Impossibility Result

Let us recall that the assumption $t < n/3$ is necessary and sufficient to implement Bracha's Byzantine reliable broadcast abstraction, as well as its variant used in this paper. The difference between Bracha's abstraction and the one employed here lies in the latter's assurance of both CS-Fifo ordering and BS-Fifo ordering properties. These properties impose a FIFO order of all messages delivered from any process, irrespective of whether they are correct or Byzantine. Due to these properties, Algorithm 2 allows the upgrade from reliable FIFO broadcast to CMB-broadcast under the assumption $t < n/2$. Interestingly, $t < n/2$ is also the condition that allows an upgrade from reliable broadcast to mutual broadcast in the crash failure model. The theorem proved below states that if the underlying Byzantine reliable broadcast abstraction ensures

CS-FIFO ORDERING (FIFO order when the sender is correct) but not BS-FIFO ORDERING (an agreement on the delivery order when the sender is Byzantine) — an abstraction that we call weak-FIFO-broadcast — then the upgrade to CMB-broadcast becomes infeasible with $t < n/2$, necessitating $t < n/3$. This means that the maintenance of FIFO order, even when the sender is Byzantine, empowers the reliable broadcast abstraction to effectively handle and mitigate the impacts of Byzantine processes, thereby enabling the emulation of an atomic register or the CMB-broadcast abstraction.

**Theorem 2.** *It is impossible to implement CMB-broadcast in the model* $\mathcal{BAMP}_{n,t}[t < \frac{n}{2}, \mathsf{weak\_fifo\_broadcast}]$, *when* $n \geq 3$ *and* $t \geq \frac{n}{3}$.

*Proof.* Let us assume that there exists an algorithm, $A$, which implements CMB-broadcast in the model $\mathcal{BAMP}_{n,t}[\mathsf{weak\_fifo\_broadcast}]$. even when $t \geq \frac{n}{3}$. We are going to show (proof by contradiction) that there exists an execution allowed by $A$ and does not respect the specification of CMB-broadcast.

Let us consider a system made up of $n \geq 3$ processes, and let $t \geq \frac{n}{3}$. We can partition the set of processes into three non-empty sub-sets $P$, $Q$ and $R$, whose size is at most $t$. Let us pick two processes $p \in P$ and $q \in Q$, and let us consider three executions of $A$.

**$S_1$:** In the first scenario, the processes in $Q$ are Byzantine (or simply slow) and do not take any step during the execution. All other processes are correct. Process $p$ cmb-broadcasts a message $m_p$, and, by the LOCAL PROGRESS property of CMB-broadcast, $p_i$ eventually cmb-delivers $m_p$ from itself.

**$S_2$:** The second scenario is similar, except that the roles of the processes in $P$ and $Q$ are exchanged. This time, the processes in $P$ are Byzantine and do not take any step during the execution, Process $q$ cmb-broadcasts a message $m_q$ and eventually cmb-delivers $m_q$ from itself.

**$S_3$:** In the third scenario, the processes in $R$ are Byzantine, and the other processes are correct. In the first stage of the execution, $p$ cmb-broadcasts $m_p$, and $q$ cmb-broadcasts $m_q$. In the second stage of the execution, all messages between $P$ and $Q$ are delayed, and processes in $R$ weak-fifo-broadcast the same messages as in $S_1$ and $S_2$. Since the processes in $R$ are Byzantine, there is no restriction on the order in which these messages are weak-fifo-delivered by the other processes. Hence, during this second phase, all processes in $P$ weak-fifo-deliver the same messages as in $S_1$, and all processes in $Q$ weak-fifo-deliver the same messages as in $S_2$. Since $p$ receives the same messages, in the same order, in $S_1$ and $S_3$, $p$ cmb-delivers $m_p$ without cmb-delivering $m_q$. similarly, $S_2$ and $S_3$ are indistinguishable from $q$, so $q$ cmb-delivers $m_q$ without cmb-delivering $m_p$. This violates the MUTUAL ORDERING property of CMB-broadcast, hence a contradiction. □

# 7 Conclusion

This article proposes a communication abstraction called CMB-broadcast (for Causal Byzantine Mutual Broadcast) whose computability is equivalent to that of SWMR atomic registers in a distributed system where some processes can be Byzantine. It is an adaptation of the mutual broadcast proposed for crash-prone systems. We first demonstrate how it can be used to implement a SWMR atomic read/append register. Additionally, we show how to implement CMB-broadcast over a system where the number of Byzantine processes is $t < n/3$. This implementation was structured in two steps to facilitate understanding. It was built on top of another abstraction, which is Bracha's Byzantine reliable broadcast in a FIFO version.

Interestingly, we observed that, in reality, when CMB-broadcast is implemented over reliable FIFO broadcast, it is sufficient to assume only $t < n/2$. We then demonstrated that this is possible only if the FIFO property is guaranteed regardless of whether the message sender is Byzantine or not. If this property is restricted only to correct processes, then $t < n/3$ is necessary to implement CMB-broadcast over reliable broadcast, highlighting for the first time the role of strong FIFO property in enforcing the power of reliable broadcast in a Byzantine context.

# References


[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[2] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[3] Gabriel Bracha. Asynchronous Byzantine Agreement Protocols. *Information and Computation*, 75(2):130–143, 1987.

[4] Shir Cohen and Idit Keidar. Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer. In *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209, 2021.

[5] Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Brief Announcement: The MBroadcast Abstraction. In *Proc. of the 2023 ACM Symposium on Principles of Distributed Computing, PODC, Orlando, FL, USA*, pages 282–285, 2023.

[6] Mathilde Déprés, Achour Mostéfaoui, Matthieu Perrin, and Michel Raynal. Send/Receive Patterns Versus Read/Write Patterns in Crash-Prone Asynchronous Distributed Systems. In *37th International Symposium on Distributed Computing, DISC 2023, L'Aquila, Italy*, volume 281 of *LIPIcs*, pages 16:1–16:24, 2023.

[7] Damien Imbs, Sergio Rajsbaum, Michel Raynal, and Julien Stainer. Read/write shared memory abstraction on top of asynchronous Byzantine message-passing systems. *J. Parallel Distributed Comput.*, 93-94:1–9, 2016.

[8] Vincent Kowalski, Achour Mostéfaoui, and Matthieu Perrin. Atomic Register Abstractions for Byzantine-Prone Distributed Systems. In *27th International Conference on Principles of Distributed Systems, OPODIS, Tokyo, Japan*, volume 286 of *LIPIcs*, 2023.

[9] Leslie Lamport. On Interprocess Communication. Part I: Basic Formalism. *Distributed Comput.*, 1(2):77–85, 1986.

[10] Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic Read/Write Memory in Signature-Free Byzantine Asynchronous Message-Passing Systems. *Theory Comput. Syst.*, 60(4):677–694, 2017.

[11] Achour Mostéfaoui and Michel Raynal. Intrusion-Tolerant Broadcast and Agreement Abstractions in the Presence of Byzantine Processes. *IEEE Trans. Parallel Distributed Syst.*, 27(4):1085–1098, 2016.

[12] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching Agreement in the Presence of Faults. *Journal of the ACM*, 27(2):228–234, 1980.


# Appendix A: Bracha's Reliable Broadcast Algorithm

The broadcast algorithm presented in Figure 3 is Bracha's algorithm [3] enriched with sequence numbers. On the one hand, it enables the broadcasting of a sequence of messages rather than a single one. On the other hand, it ensures that the different broadcast messages are delivered in FIFO order. Specifically, each correct process delivers the same sequence of messages from each given process (possibly Byzantine). For that, each process $p_i$ manages a local array $next_i[1..n]$, where $next_i[j]$ is the sequence number of the next application message that will be fifo-delivered, by process $p_i$, from process $p_j$. Initially, for all $i, j$, $next_i[j] = 1$.

When a process $p_i$ invokes fifo_broadcast$(m)$ 1, it sends the message INIT$(m, current)$ (Line 3) to all processes, where *current* is its next sequence number and then waits until this message is fifo-delivered to $p_i$ itself (lines 18 and 19). The procedure fifo_broadcast is thus blocking.

```
1  operation fifo_broadcast(m) is:
2      let current ← next_i[i];
3      send INIT(m, current) to all processes;
4      wait until next_i[i] = current + 1;
5  when one of the following conditions holds for the first time, for each pair (j, sn):
6          • p_i has received INIT(m, sn) from p_j
7          • p_i has received ECHO(m, j, sn) from more than (n+t)/2 different processes
8          • p_i has received READY(m, j, sn) from more than t different processes
9  do: send ECHO(m, j, sn) to all processes;
10 when one of the following conditions holds for the first time, for each pair (j, sn):
11         • p_i has received ECHO(m, j, sn) from more than (n+t)/2 different processes
12         • p_i has received READY(m, j, sn) from at least t + 1 different processes
13 do: send READY(m, j, sn) to all processes;
14 when the following condition holds for the first time, for each pair (j, sn):
15         • p_i has received READY(m, j, sn) from at least 2t + 1 different processes
16 do:
17     wait until next_i[j] = sn;
18     fifo_deliver m from p_j;
19     next_i[j] ← sn + 1;
```

**Algorithm 3:** Implementation of FIFO-broadcast

- When a process $p_i$ receives a message INIT$(m, current)$ from a process, it echoes it by sending a message ECHO$(m, j, sn)$ to all processes (Line 9). This is done to inform the other processes it received the application message $m$. This echo is sent only once for each sequence number. This is because if $p_j$ is Byzantine it can send different messages with the same sequence number.
- Then, when $p_i$ has received the same message ECHO$(m, j, sn)$ from "enough" processes (where "enough" means here "more than $(n+t)/2$ different processes"), and has not yet broadcast a message READY$(m, j, sn)$, $p_i$ does it in Line 10.

  The aim of (a) the messages ECHO$(m, j, sn)$, and (b) the cardinality "greater than $(n+t)/2$ processes", is to ensure that no two correct processes can fifo-deliver distinct messages at Line 18 from $p_j$ (even if $p_j$ is Byzantine). The aim of the messages READY$(m, j, sn)$ is related to the liveness of the algorithm. More precisely, it aims to allow the fifo-delivery by the correct processes of the very same message $m$ from $p_j$, and this must always occur if $p_j$ is correct. It is nevertheless possible that a message fifo_broadcast by a Byzantine process $p_j$ is never fifo-delivered by the correct processes.
- Finally, when $p_i$ has received the message READY$(m, j, sn)$ from $(t+1)$ different processes, it broadcasts the same message READY$(m, j, sn)$ , if not yet done. This is required to ensure the termination property. If $p_i$ has received "enough" messages READY$(m, j, sn)$ ("enough" means here "from at least $(2t + 1)$ different processes"), it fifo-delivers the message $m$ fifo-broadcast by $p_j$.

More explanations and proofs that this algorithm satisfies the properties defining the reliable broadcast abstraction can be found in [3, 11].

# Appendix B: Proof of Algorithm 1

**Lemma 2** (Validity)**.** *If a* `read` *operation performed by a correct process returns log, and if the writing process is correct, then for all* $s \in \{1, ..., |log|\}$, $log[s-1]$ *is the* $s^{th}$ *value written.*

*Proof.* Suppose that the writing process $p_w$ is correct, and that a `read` operation performed by a correct process $p_i$ returns $log_i$. By the VALIDITY, INTEGRITY and FIFO ORDERING properties of CMB-broadcast,

the sequence of cmb-deliveries of APPEND(_) messages by $p_i$ from $p_w$ is the same as the sequence of cmb-broadcasts of APPEND(_) messages by $p_w$. This sequence is also the same as both the sequence of write operations (Line 2), and the sequence of values appended in $\mathsf{replica}_i$ (Line 9), whose $log_i$ is a prefix (Line 5). □

**Lemma 3** (Read after write)**.** *If a read done by a correct process starts after the $s^{th}$ write of a correct process completes, then the read cannot return a sequence containing less than $s$ values.*

*Proof.* Suppose a read done by a correct process $p_i$ starts after the $s^{\text{th}}$ write of a correct process $p_w$ completes. Then $p_w$ has cmb-delivered at least $s$ of its own APPEND(_) messages before the SYNCH message cmb-broadcast on Line 4. By the MUTUAL ORDERING property of CMB-broadcast, $p_i$ must also cmb-deliver at least $s$ APPEND(_) messages before its own SYNCH message. Hence, $\mathsf{replica}_i$ contains at least $s$ values on Line 5. □

**Lemma 4** (Inclusion)**.** *Let $r_i$ and $r_j$ be two* `read` *operations, done by correct processes, that return respectively $log_i$ and $log_j$. Then $log_i$ is a prefix of $log_j$, or $log_j$ is a prefix of $log_i$.*

*Proof.* By the INTEGRITY and FIFO ORDERING properties of CMB-broadcast, the sequence of cmb-deliveries of APPEND(_) messages by any pair of correct processes is the same. Since $log_i$ and $log_j$ are two prefixes of this common sequence, one must be a prefix of the other, following Levi's lemma. □

**Lemma 5** (Read after read)**.** *Let $r_i$ and $r_j$ be two* `read` *operations, done by correct processes, that return respectively $log_i$ and $log_j$. If $r_i$ completes before $r_j$ starts, then $log_i$ is a prefix of $log_j$.*

*Proof.* Let $m_i$ be the SYNCH message $p_i$ cmb-broadcasts on Line 6, and $m_j$ be the SYNCH message $p_j$ cmb-broadcasts on Line 4.

Suppose $r_i$ completes before $r_j$ starts. Then $p_i$ cmb-delivers $m_i$ before $m_j$. By the MUTUAL ORDERING property of CMB-broadcast, $p_j$ must also cmb-deliver $m_i$ before $m_j$. Therefore, by the CS-CAUSAL ORDERING property of cmb-broadcast, before cmb-delivering $m_j$, $p_j$ must also cmb-deliver all the APPEND(_) messages that $p_i$ cmb-delivered before cmb-broadcasting $m_i$. Hence, $log_j$ contains at least all the values of $log_i$, so by Lemma 4, $log_i$ is a prefix of $log_j$. □

**Theorem 3** (Correctness of Algorithm 1)**.** *Algorithm 1 implements a wait-free and Byzantine linearizable SWMR read/append register.*

*Proof.* Let $H$ be a distributed history of Algorithm 1. By lemmas 2-5, $H$ verifies the four hypotheses of Proposition 2, stated in [8]. Hence, $H$ is Byzantine linearizable. Moreover, Algorithm 1 does not contain any loop or recursion, and all its waiting times terminate thanks to the LOCAL PROGRESS property of CMB-broadcast. Hence, Algorithm 1 is wait-free. □

# Appendix C: Proof of Algorithm 2

**Lemma 1.** *Let $S$ be the set of pairs $\langle \text{MSG}(m,k), j\rangle$, such that $\text{MSG}(m,k)$ is fifo-delivered by all correct processes, and $m$ is cmb-delivered from $p_k$ by some correct process. For all pairs $\langle \text{MSG}(m,k), j\rangle \in S$ and all correct processes $p_i$, then $p_i$ eventually execute Line 14 in the code triggered by the fifo-delivery of $\text{MSG}(m,k)$ from $p_j$.*

*Proof.* We define the binary relation $\leq$ on elements of $S$ as: $\langle \text{MSG}(m,k), j\rangle \leq \langle \text{MSG}(m',k'), j'\rangle$ if one of the following conditions hold:

- $j = j'$ and all correct processes fifo-deliver $\text{MSG}(m,k)$ from $p_j$ before $\text{MSG}(m',k')$ from $p_j$,
- $k = k' = j$ and $m = m'$,
- there exists $s \in S$ such that $\langle \text{MSG}(m,k), j\rangle \leq s \leq \langle \text{MSG}(m',k'), j'\rangle$.

We first remark that $\leq$ is a antisymmetric: this is because, by Lines 6, 13 and 14, a correct process cannot cmb-deliver $m'$ before $m$ if $\langle \text{MSG}(m,k), j\rangle \leq \langle \text{MSG}(m',k'), j'\rangle$, so pairs $\langle \text{MSG}(m,k), j\rangle$ participating in a cycle would not match the definition of $S$. Hence, $\leq$ is an order relation. Moreover, it is well-founded because executions have a beginning in time.

We now prove, by Noetherian induction on $S$, that for all $\langle \text{MSG}(m,k), j\rangle \in S$ and all correct processes $p_i$, $p_i$ eventually execute Line 14 in the code triggered by the fifo-delivery of $\text{MSG}(m,k)$ from $p_j$. Let $\langle \text{MSG}(m,k), j\rangle \in S$ and $p_i$ be a correct process. We suppose the result is true for all $s \in S$ such that $s < \langle \text{MSG}(m,k), j\rangle$. It is sufficient to prove that $p_i$ cannot wait forever at Line 6, 9 or 13 after its fifo-delivery of $\text{MSG}(m,k)$ from $p_j$.

**Line 6** By induction, the property holds for all messages fifo-broadcast by $p_j$ prior to $\text{MSG}(m,k)$, so $p_j$ eventually executes Line 14 $sn-1$ times, and $p_i$ is not blocked on Line 6.

**Line 9** By induction, all correct processes have executed Line 14 for all the messages fifo-broadcast by $p_i$ prior to the message $\text{MSG}(m,i)$. Hence, all correct processes (at least $n-t$) will eventually fifo-broadcast $\text{MSG}(m,i)$ after fifo-delivering $\text{MSG}(m,i)$ from $p_i$. Therefore, $p_i$ cannot block at Line 9.

**Line 13** By the definition of $S$, a correct process has cmb-delivered $m$ from $p_k$. By Lines 8 and 11, this process has fifo-delivered a message $\text{MSG}(m,k)$ from $p_k$, so all correct processes do so by the CONSISTENCY property of fifo-broadcast. Hence, $\langle \text{MSG}(m,k), k\rangle \in S$, and $\langle \text{MSG}(m,k), k\rangle < \langle \text{MSG}(m,k), j\rangle$. By induction, $p_i$ has executed Line 14 upon fifo-delivery of $\text{MSG}(m,k)$ from $p_k$, after cmb-delivering $m$ from $p_k$. Hence, $p_i$ is not blocked on Line 13.

□