The Journal of Supercomputing (2024) 80:18728–18760
https://doi.org/10.1007/s11227-024-06187-6

Timestamp system for causal broadcast communication

Isabel Muñoz-Fernández[1] · Sergio Arévalo-Viñuales[2] · Pedro de-las-Heras-Quirós[3]

Accepted: 3 May 2024 / Published online: 22 May 2024

Abstract

In unreliable asynchronous distributed systems with failures, achieving a causal view of the system across all processes is a challenging task. The Causal Reliable Broadcast (CRB) abstraction is used to solve this task. When CRB is implemented with algorithms that use logical vector clocks to timestamp broadcast events, the causal relationships between broadcast events can be detected with maximal accuracy. However, this timestamping mechanism used by CRB might not be useful for systems that need to reason about the causal relationships among both broadcast and delivery events. To address this challenge, the paper proposes a Causal Timestamp System (CTS) based on vector clocks that timestamps broadcast and delivery events capturing with maximal accuracy the causal relationships among those events. CTS simplifies the formal verification and testing of implementations of CRB algorithms based on CTS. Additionally, a new Global State Monitoring (GSM) algorithm is proposed, tailored to a distributed system that uses CRB with CTS. GSM enables finer-grained assessment of global states and application-dependent predicates of that system. We clarify these concepts with an IoT example.

Keywords Causal broadcast · Distributed global states · Vector clock timestamps

✉ Isabel Muñoz-Fernández
mi.munozf@alumnos.urjc.es

Sergio Arévalo-Viñuales
sergio.arevalo@upm.es

Pedro de-las-Heras-Quirós
pedro.delasheras@urjc.es

[1] Universidad Rey Juan Carlos, Campus de Móstoles, 28933 Móstoles, Madrid, Spain

[2] Escuela Técnica Superior de Ingeniería de Sistemas Informáticos, Universidad Politécnica de Madrid, Campus Sur, 28031 Madrid, Spain

[3] Escuela de Ingeniería de Fuenlabrada, Universidad Rey Juan Carlos, Campus de Fuenlabrada, Fuenlabrada, 28942 Madrid, Spain

1 Introduction

In unreliable asynchronous distributed systems with failures, getting a causal view of the system in all processes is a complex task. These systems are prone to perturbations such as process crashes, loss, duplicity or disordering of messages, or delays in processing and communication. The Causal Reliable Broadcast (*CRB*) abstraction [1] serves as a critical component for obtaining in each process a coherent causal view of the system, where all processes observe cause events that *happened–before* their corresponding effect events [2].

The *CRB* abstraction reliably disseminates messages in a group of processes and delivers them in causal order in every process of the group. *CRB* offers respectively the *cBroadcast*(m) and the *cDeliver*(m) events to broadcast and deliver in causal order a message m. If the *cBroadcast* of m' was caused by the *cBroadcast* of m, we say that m causally precedes m'. Also, the causal order delivery ensures that m' is *cDeliver*ed after m in every process. However, if the *cBroadcast* of m' is concurrent with the *cBroadcast* of m, we say that m is concurrent with m'. Then m and m' can be *cDeliver*ed in any order. We call *CRB* algorithm any algorithm that implements the *CRB* abstraction. *CRB* algorithms that implement the *CRB* abstraction differ in the degree of accuracy to which they capture the causal relation among *cBroadcast* messages.

We say that a *CRB* algorithm has maximal accuracy with respect to a given set of events if it entirely reflects the partial order defined by the causality relation among those events. That is, it reflects not only if events are causally related, but also if events are concurrent.

The maximal causal accuracy is valuable for applications that use the *CRB* abstraction such as geo-replicated data stores. In [3], update operations on replicas are implemented using an underlying *CRB* service for causally ordering the updates along with scalar timestamps to totally order concurrent updates. Some authors use an implementation of the *CRB* abstraction that makes visible to the application layer the maximal accuracy information in order to identify and totally order concurrent messages as it is proposed in [4].

The causal history of a message m contains all messages that causally precede m. *CRB* algorithms with maximal causal accuracy can piggyback in a message m the causal history of m with either an unbounded list of all messages that causally precede m or a bounded and compact representation of the causal history of m [1].

A compact representation of the causal history of a message can be obtained with logical clocks defined by Lamport [2]. A logical clock is a software artifact that assigns timestamps to process events. These timestamps capture the *happened–before* relation among events, implementing in that way a logical clock that is global to all processes. Vector clocks are logical clocks that timestamp events using vectors [5, 6]. In the *CRB* algorithm by Birman et al. [7], a global clock is implemented defining in every process a local vector clock that ticks just before a *cBroadcast* event happens. A message is *cBroadcast* piggybacking the process current local clock value as a timestamp.

The causal relation among *cBroadcast* events is captured with the maximal accuracy when these events are timestamped with logical vector clocks that have at least the size of the group [8]. These logical clocks with maximal accuracy are said to characterize causality [5, 6]. The drawback of piggybacking timestamps is the communication overhead that is not negligible for large groups. Indeed, *CRB* algorithms based on vector clocks that characterize causality are not scalable.

As far as we know, there are two families of techniques that could be used to improve the scalability of *CRB* algorithms based on vector clocks. The first one is the compression of causal information to be piggybacked in a message: incremental changes in vector timestamps [7, 9, 10], interval tree clocks [11], causal barriers [12] and resettable encoded vector clocks [13]. Regardless of the technique employed, causality characterization requires the challenge of managing either the linear growth of causal information within messages or within process storage. In this family, we also consider the clocks of constant size that are scalable with respect to the number of processes, such as plausible clocks [14] and the probabilistic vector clocks known as Bloom clocks [15, 16]. Bloom clocks exhibit the best performance among vector clocks of any size for determining causality. However, both types of clocks may predict false causal relations among messages.

The second family of techniques takes advantage of overlay network topologies: process groups connected with FIFO channels [17, 18] and flooding in FIFO overlay network topologies [19, 20]. Unfortunately, none of these overlay network topology techniques characterize causality.

As a result, we have decided to use plain vector clocks to track causality, because they provide maximal accuracy. This solution is good enough in some distributed domains such as in geo-replicated, domotic or cloud systems that might not need scalability.

We have identified that *CRB* algorithms based on vector clocks [7, 21, 22] only capture with the maximal causal accuracy the causal relation among *cBroadcast* events and their corresponding messages. In this paper we propose to extend the causal tracking to *cDeliver* events. The *CRB* algorithm by Birman et al. [7] is the best-known *CRB* algorithm with maximal accuracy that uses vector clocks to timestamp globally *cBroadcast* events but not *cDeliver* events. Consequently, we claim that with Birman's *CRB* algorithm it is not possible to determine the *happened–before* relation among *cBroadcast* and *cDeliver* events using its timestamp system.

Note that we do not claim that Birman's *CRB* needs to associate additional timestamps to *cDeliver* events in order to ensure causal delivery but for other significant purposes, external and complementary to the distributed application, such as testing and verification. For example, as it was pointed out by [23], the lack of global timestamping of *cDeliver* events in Birman's *CRB* algorithm makes it very difficult to systematically test and verify the properties of the *CRB* abstraction.

Regarding with testing difficulties, the correctness of a run of a library that implements the Birman's *CRB* algorithm cannot be tested dynamically if the test is done with an external passive monitor [24]. The reason is because the monitor cannot obtain consistent global states from the causal ordering of *cBroadcast* and *cDeliver* events timestamps due to the lack of global timestamping of *cDeliver* events.

Regarding to formal verification difficulties in Birman's *CRB* libraries, the lack of *cDeliver* global timestamping can yield to implementations that differ in the interpretation of *cDeliver* events. This fact causes that the verification of *CRB* properties [10] in these libraries has to depend on the implementation rather than on the definition of the algorithm. For example, these libraries need to define in each process when a self-*cDeliver* or *cDeliver* of a message happens, and check locally the causal delivery.

To sum up, although there are techniques for scaling vector timestamp mechanisms, not all of them preserve the maximal causal accuracy. Moreover, *CRB* algorithms based on vector clocks only capture the causal ordering of *cBroadcast* events. Not capturing with the maximal accuracy *cDeliver* events makes it more difficult the verification of *CRB* algorithms and the implementation of external passive monitors for testing properties of runs of those algorithms. Finally, we want to emphasize that implementing the *CRB* abstraction with techniques based on clocks for timestamping *cDeliver* events and implementing an external passive monitor is not trivial, as we will show in this paper.

Contributions In this paper we propose and formally define a new Causal Timestamp System (*CTS*) based on vector clocks that timestamps not only *cBroadcast* but also *cDeliver* events of the *CRB* abstraction. *CTS* simplifies the formal verification and testing of the *CRB* implementations. We also propose a new passive monitor system for testing those *CRB* implementations based on *CTS*. In particular, the contributions of this paper are as follows:

- We define a new *CRB* algorithm and its corresponding timestamp system called *CTS* that timestamps *cBroadcast* and *cDeliver* events.
- We formally specify the *CTS* properties according to the timestamp framework of [14] and prove that the *CTS* respects the properties of a timestamp system and characterizes causality with respect to *cBroadcast* and *cDeliver* events. We also prove that *CTS* extends the maximal accuracy with *cDeliver* events.
- We present and prove the correctness of a new passive Global State Monitoring (*GSM*) algorithm for monitoring distributed applications that use our *CRB* with *CTS*. A *GSM* algorithm is suitable for making global predicate evaluations required by applications such as detecting deadlocks and termination, or for testing and debugging. This new *GSM* monitors *cBroadcast* and *cDeliver* events, hence it can generate a finer grained evaluation of consistent global states and predicates than other passive monitors that use Birman's *CRB* algorithm [24]. This is because our *CRB* algorithm timestamps *cDeliver* events.
- We show a domotic application example that uses both our *GSM* and *CRB* with *CTS*, in order to show how they work and in order to compare them with a passive monitor that uses Birman's *CRB*. Note that with the *GSM* algorithm and this example we show the usefulness of the *CRB* implemented with *CTS* due to its timestamping of *cDeliver*s.

Roadmap

Section 2 reviews in detail the Birman's algorithm that implements the *CRB* abstraction and describes the motivation of our work. Section 3 details the system model and presents some definitions. Section 4 describes formally the primitives and properties of the *CRB* abstraction. Section 5 introduces a new Causal timestamp system (*CTS*) for the implementation of the *CRB* abstraction with vector clocks. Section 6 introduces a novel Global State Monitoring (*GSM*) algorithm for applications that communicate through *CRB* implementations based on our *CTS*. Section 7 introduces an example of a distributed application implemented in two ways: (1) with our *CRB* with *CTS* and monitored by our *GSM*, and (2) with Birman's *CRB* monitored by a global state monitor. Finally, this section compares the degree of event global resolution of both monitoring approaches. Section 8 reviews the related work. We conclude in Sect. 9.

2 Problem statement

This section reviews in detail the Birman's algorithm [7] that implements the *CRB* abstraction using logical vector clocks that timestamp *cBroadcast* events. We also describe the problems caused for not timestamping *cDeliver* events.

CRB abstraction [1] is an important group communication tool for reliably delivering in causal order the same set of messages to all correct processes of a group. *CRB* abstraction is formally defined by the *cBroadcast* and *cDeliver* events and a set of properties [25]. *cBroadcast*(m) reliably sends a message m to all processes of the group including the sender, and *cDeliver*(m) delivers m in causal order to the process application layer. If the *cBroadcast* of m' could have been caused by the *cBroadcast* of m, we say *cBroadcast* of m potentially caused the *cBroadcast* of m' ($m \overset{c}{\rightarrow} m'$). The *CRB* causal order delivery property says that if the *cBroadcast* of a message m' was caused by the *cBroadcast* of a message m, m must be delivered before m' in all correct processes:

$$m \overset{c}{\rightarrow} m' \Rightarrow cDeliver(m) \rightarrow cDeliver(m') \tag{1}$$

Figure 1 shows an example of a run of a distributed application made of three processes that do not fail (correct processes) and that communicate using the *CRB* abstraction. Process p_0 *cBroadcast*s m_0 to all processes including itself in the event e_{00} and, according to the *CRB* validity property, self-*cDeliver*s m_0 in the event e_{01}. Process p_1 *cDeliver*s m_0 in the event e_{10} and then *cBroadcast*s m_1 in the event e_{11}. As m_0 is a potential cause of m_1, all processes including the sender, have to *cDeliver* m_0 before m_1, according to the causal delivery property (1). Due to the asynchrony of the system, p_2 receives m_1 before m_0, delaying the *cDeliver* of m_1 until the *cDeliver* of m_0. In process p_1, after the *cDeliver* of m_1, the messages m_2 and m_3 can be *cDeliver*ed in any order as the *cBroadcast* event e_{03} of m_2 and the *cBroadcast* event e_{22} of m_3 are concurrent events. Note that all processes *cDeliver* the same set of messages (m_0, m_1, m_2, m_3) according to the *CRB* agreement property, because there are not faulty processes.

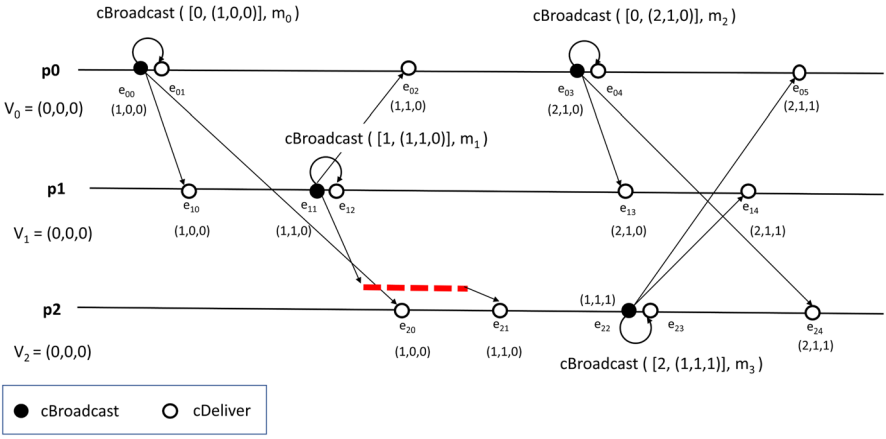

Fig. 1 Example of causal broadcast timestamping with Birman's algorithm

Algorithms that implement the *CRB* abstraction need to determine the *happened–before* relation among *cBroadcast* events of the messages they send in order to *cDeliver* their corresponding messages in causal order. *CRB* Birman's algorithm [7] is a well-known implementation of the *CRB* abstraction that offers the maximal accuracy with respect to *cBroadcast* events. It defines the causal delivery ordering for *cBroadcast* messages m and m' on a process p according to the following rule:

$$m \xrightarrow{c} m' \Rightarrow cDeliver(m) \xrightarrow{p} cDeliver(m') \tag{2}$$

The relation $\xrightarrow{p}$ is a local relation on process p that captures the event program order on p as follows: two events a and b are related as $(a \xrightarrow{p} b)$ if they happen on p and a occurs before b.

Birman's algorithm uses vector clocks of the size of the number of processes of the system [5, 6] to timestamp *cBroadcast* events. These timestamps are used to decide if a *cBroadcast* event occurs before another *cBroadcast* event. The mechanism of vector clocks assigns to an event a vector value called timestamp that respects the *happened–before* relation. As these vector clocks have one position for each process of the group, it is also possible to determine if two events are concurrent looking at their vector timestamps. Given two vector timestamps V and W in a system of P processes, the comparison of them is done according to the following rules:

$$\begin{aligned} V \leq W &\iff \forall i \in \{0, \ldots, |P|-1\} : V[i] \leq W[i] \\ V < W &\iff V \leq W \wedge \exists j \in \{0, \ldots, |P|-1\} \text{ such that } V[j] < W[j] \\ V \,||\, W &\iff V \not< W \wedge W \not< V \end{aligned}$$

Birman's *CRB* algorithm captures causality among *cBroadcast* events precisely. That is, given two messages m and m' and their corresponding timestamps $VT(m)$ and $VT(m')$ assigned by this *CRB* algorithm, it is true that:

$$m \xrightarrow{c} m' \iff VT(m) < VT(m') \tag{3}$$

Now, we describe in detail the Birman's *CRB* algorithm in a system of a set P of processes. Each process p_i maintains a local clock V_i that is a vector of non-negative integers with one entry per process. V_i is updated according to the following rules:

R0) Initialization:

$$i = \text{process identifier}$$
$$\forall j \in \{0, \ldots, |P| - 1\} : V_i[j] = 0$$

R1) Before the *cBroadcast* of a message m is generated at site i:

$$V_i[i] = V_i[i] + 1$$

Then, m is *cBroadcast* piggybacking $< i, VT(m) >$, where i is the identifier of the sender, and $VT(m)$ is the timestamp of the *cBroadcast* event of m taking the value of V_i.

When a process p_i receives a message m with timestamp $VT(m)$ from p_j, the process p_i delays the *cDeliver* of m until the following conditions are both fulfilled:

$$V_i[j] + 1 = VT(m)[j] \tag{4a}$$

$$V_i[k] \geq VT(m)[k], \forall k \neq j \tag{4b}$$

R2) Before a process p_i *cDeliver*s m with timestamp $VT(m)$, p_i updates its vector clock as:

$$\forall k \in \{0..|P| - 1\} : V_i[k] = max(V_i[k], VT(m)[k])$$

The vector timestamp $VT(m)$ assigned to the *cBroadcast* of message m counts the number of messages *cBroadcast* by each process that causally precede m.

Figure 1 shows an example of the execution of Birman's *CRB* algorithm. At the beginning of the execution, all processes initialize their local clocks according to rule $R0$. Then, process p_0 *cBroadcast*s the message m_0 in the event e_{00}, updating its vector clock and timestamping m_0 according to rule $R1$. Since the timestamp (1, 0, 0) of *cBroadcast* event e_{00} is less than the timestamp (1, 1, 0) of *cBroadcast* event e_{11} (3), it holds that *cBroadcast*(m_0) *happened–before cBroadcast*(m_1). Events e_{03} and e_{22} are concurrent because looking at their respective timestamps (2, 1, 0) and (1, 1, 1), e_{22} did not happen before e_{03} and vice versa. In p_2, after receiving m_1, the *cDeliver* event of m_1 is delayed until m_0 is *cDeliver*ed, according to conditions (4a) and (4b) of rule $R1$, because m_0 *happened–before* m_1. Finally, when a *cDeliver* event of a message happens in any process, its local clock is updated according to rule $R2$. For example, after *cDeliver* event e_{21} at process p_2, the vector clock of p_2 has the value (1, 1, 0).

As it has been described above, Birman's *CRB* algorithm ensures the causal delivery of broadcast messages by means of a vector timestamp system that

timestamps cBroadcast events. As the timestamping of Birman's *CRB* algorithm only defines how to timestamp *cBroadcast* events and how to compare them to determine if they are causally related (3), it is not possible to detect if a *cDeliver* event *happened-before* any other event (be it *cBroadcast* or *cDeliver*). In other words, Birman's *CRB* algorithm timestamping falls short in guaranteeing and identifying causal relationships among events of both *cBroadcast* and *cDeliver* types, such as $cBroadcast \rightarrow cDeliver$, $cDeliver \rightarrow cBroadcast$, and $cDeliver \rightarrow cDeliver$. As a consequence, causality in Birman's algorithm can only be established among *cBroadcast* events using timestamps.

Although Birman's timestamping does not pose any problem for the correct causal delivery of broadcast messages, it makes difficult to design systems implemented with Birman's *CRB* algorithm that need to reason about *cDeliver* events. For example, formal methods for verification and testing the correctness of Birman's *CRB* implementations cannot use the *cDeliver* timestamps and then they need to adopt ad hoc implementation-dependent solutions, as we show below.

Problems with formal verification techniques. Formal verification techniques can be used to establish the correctness of Birman's *CRB* libraries regarding to the formal properties of the *CRB* abstraction [10]. A Birman's *CRB* library is causally consistent if it verifies the *CRB* properties for all its possible executions. Examples of the difficulty of formal verification of Birman's *CRB* libraries can be seen in [26] and [27]. In both libraries, the causal delivery property needs to be verified locally in every process because of the lack of global *cDeliver* timestamping. In both cases they need to build local histories assigning ad hoc local timestamps to *cDeliver* events in each process p. In [27], the timestamp of *cDeliver* of m in p is the value of the local clock when this event happened in p, while in [26], the timestamp of *cDeliver* of m is the value of the timestamp of the *cBroadcast* of m.

Besides the causal delivery property problems identified in the previous paragraph, the validity property of *CRB* that says that a correct sender of a message m always self-*cDeliver*s m is also hard to prove. Because the *cDeliver* events are not timestamped, in [27] the *cBroadcast* and its corresponding self-*cDeliver* events are considered as only one atomic event. In [26] a *cBroadcast* primitive is proposed to broadcast a message to all processes except the sender, and hence their *cBroadcast* primitive does not generate a self-*cDeliver* event. As a consequence, the agreement *CRB* property, which says that all correct processes *cDeliver* the same set of messages, is also implementation dependent, as its correctness depends on the correctness of the validity property.

As a result of the lack of timestamping of *cDeliver* events, the use of formally verified Birman's *CRB* libraries requires to know how they interpret and implement the *CRB* properties. In contrast, our causal timestamp system *CTS* globally timestamps *cBroadcast* and *cDeliver* events, and thus the formal verification of *CRB* properties on libraries that implement our *CRB* algorithm with our *CTS* would not depend on particular interpretations.

Problems with testing techniques. Testing techniques applied to libraries implementing the Birman's *CRB* algorithm check whether a given library execution is correct regarding the *CRB* properties. One technique is to determine if *CRB* properties

applied to any global state of a run of the Birman's *CRB* library are being satisfied. A way to construct global states is to use an external monitor process executing concurrently with the system that uses the library. The monitor receives notifications of *cBroadcast* events that happen in a run, obtaining a global causal sequence of these events in a run. There are not *cDeliver* events in the causal sequence observed by the monitor. Thus, the monitor can only check the property of causal order delivery of a message m indirectly if it observes the *cBroadcast* of another message m' that causally depends on m. Note that this *cBroadcast* of m' could never happen as it depends on the logic of the application.

To sum up, the lack in Birman's *CRB* algorithm of detection of causality among *cBroadcast* and *cDeliver* events by means of timestamps makes not possible the global predicate evaluation of *CRB* properties using a passive monitor to obtain global states. Conversely, with our global state monitoring algorithm *GSM*, we can monitor runs of libraries that implement our *CRB* algorithm with our *CTS*. Our *GSM* can obtain a global causal sequence of both *cBroadcast* and *cDeliver* events to calculate global states and to evaluate global predicates, in addition to the evaluation of the properties of *CRB* abstraction.

As we show in this paper, the timestamping of *cDeliver* events, in addition to *cBroadcast* timestamping, improves the causal accuracy captured by the Birman's *CRB* algorithm, providing a finer grained detection of causality very useful for applications such as *GSM* and verification libraries.

3 System model and definitions

We consider an asynchronous fail-silent distributed system composed by a finite set P of processes that do not share neither memory nor clock. We assume crash-stop processes that may fail by crashing. A process is said to be correct if it does not fail during all execution of the system, otherwise it is said to be faulty. Each pair of processes are connected via asynchronous reliable point-to-point links [25]. We do not consider any order of delivery in the channels. Processes communicate through these reliable channels exchanging a set of M messages through a reliable broadcast communication service.

Every process in the system is composed of a stack of asynchronous event-based service abstractions. From top to bottom: Application layer, Causal Reliable Broadcast abstraction layer, Reliable Broadcast abstraction layer and Reliable Point-to-point abstraction layer. Each service abstraction is modeled with a name and some properties, and it offers an API interface based on events it accepts (requests) and produces (indications), and it is implemented as a state machine, whose transitions are triggered by the reception of events.

Events of interest. Events generated in the layers are called internal events. Events generated outside of the abstraction stack are called external events (i.e., indication events of message delivery to the Point-to-point communication abstraction). Events are handled in mutual exclusion way in each stack. Internal events have priority over

external events and events with the same priority are handled in FIFO order. In a distributed execution, we consider as events of interest the *cBroadcast* and *cDeliver* events defined by the Causal Reliable Broadcast abstraction. We will denote by H the set of all events *cBroadcast* and *cDeliver* happened in a distributed execution. Now we define the properties of the reliable broadcast abstraction that are used by the causal reliable broadcast abstraction to offer reliability.

Definition 1 *(The Reliable Broadcast Abstraction)* The reliable broadcast abstraction reliably sends and delivers a message $m \in M$ to all processes $\in P$, including the sender process. This abstraction defines the request event *rBroadcast* and the indication event *rDeliver*. A process invokes the event *rBroadcast* to request the reliably broadcast of a message by this abstraction, and the reliable broadcast abstraction triggers the event *rDeliver* to deliver a message to the upper layer of a process.

This abstraction has the following properties [10]:

- **RB1. Integrity:** For any message m, every correct process *rDeliver*s m at most once and only if m was previously *rBroadcast* by sender of m.
- **RB2. Validity:** If a correct process p *rBroadcast*s a message m, then p eventually *rDeliver*s m.
- **RB3. Agreement:** If a correct process *rDeliver*s a message m, all correct processes in the system must eventually *rDeliver* m.

4 The causal reliable broadcast abstraction

Since the reliable broadcast abstraction delivers messages in any order, it is not possible to determine with this abstraction if a message could be the cause of broadcasting other messages. However, the causal broadcast abstraction captures this potential causality among messages, delivering first the *cause* messages and then their *effect* messages.

The causal broadcast abstraction generates *cBroadcast* and *cDeliver* events. The event $< p, cBroadcast \mid m >$ broadcasts a message $m \in M$ from p to all processes $\in P$. The event $< p, cDeliver \mid q, m >$ delivers in *Causal* order a message m to process p from q. The causal order delivery requires to know if two messages are causally related to deliver them in such order. We now define the *Causal* order relation among messages. This relation uses the Lamport's *happened–before* relation [2] particularized for capturing the potential causality among *cBroadcast* and *cDeliver* events $\in H$.

Definition 2 (*happened–before* Relation ($\rightarrow$)) Considering two events $a, b \in H$, this relation is defined as follows:

$$
\begin{aligned}
a \rightarrow b \iff & \exists p \in P{:}a, b \text{ are two events of } p \text{ and } a \text{ occurs before } b \\
& \vee \exists\, p, q \in P{:}(a =< p, cBroadcast|m >) \wedge (b =< q, cDeliver|p, m >) \\
& \vee \exists\, c \in H{:}a \rightarrow c \wedge c \rightarrow b
\end{aligned}
$$

An event a is concurrent with b and it is represented as $a \mid\mid b$ iff $a \nrightarrow b$ and $b \nrightarrow a$.

Definition 3 (Causal Order Relation $\overset{c}{\rightarrow}$)
Two messages $m, m' \in M$, are related by $\overset{c}{\rightarrow}$, if the *cBroadcast* of m' may have been potentially caused by the *cBroadcast* of m. More formally, $\forall a, b \in H$, $\forall p, q \in P$:

$$m \overset{c}{\rightarrow} m' \iff (a \rightarrow b) \wedge (a =< p, cBroadcast|m >) \wedge (b =< q, cBroadcast|m' >)$$

Messages m and m' are concurrent, represented as $m \mid\mid_c m'$, if $not(m \overset{c}{\rightarrow} m') \wedge not(m' \overset{c}{\rightarrow} m)$.

When $m \overset{c}{\rightarrow} m'$ we say that m is a causal predecessor of m' and m' is a causal successor of m. The causal history of m are made up of all predecessors of m.

The Causal Order delivery rule describes the delivery of messages in causal order.

Definition 4 *(Causal Order Delivery Rule)*
The Causal Order Delivery rule holds if for any pair of messages $m, m' \in M$ *cBroadcast* by *src* and *src'* respectively and related by $\overset{c}{\rightarrow}$, the *cDeliver* of m *happened–before* the *cDeliver* of m' in all processes of the system. Formally, $\forall p, src, src' \in P$:

$$m \overset{c}{\rightarrow} m' \Rightarrow < p, cDeliver|\ src, m > \rightarrow < p, cDeliver|src', m' >$$

The causal broadcast abstraction obeys the causal delivery rule but does not assure reliability guarantees. When the causal broadcast abstraction also respects the reliable broadcast abstraction guarantees (Def. 1), it is called causal reliable broadcast abstraction. This abstraction has the following properties:

- **CRB1-CBR3** are the properties of **Integrity**, **Validity** and **Agreement** described in the reliable broadcast abstraction (Def. 1).
- **CRB4. Causal delivery**: messages are delivered by all processes according to causal delivery rule. (Def. 4).

In the next section, we define a causal timestamp system valid for both causal broadcast and causal reliable broadcast abstractions. For the sake of simplicity, and from now on, we define the causal timestamp system only for the causal reliable broadcast abstraction.

5 Timestamping system for causal broadcast

In distributed systems, time can be considered a logical abstraction [2]. A timestamp system has a logical clock that stamps every event of interest [14]. A timestamp represents the instant in logical time in which an event occurs from the point of view

of some process in the system. The causal broadcast implementations based on vector clocks as shown in [21] and [7] define a timestamp framework for timestamping and reasoning about the *happened–before* relation of any pair of *cBroadcast* events $\in H$; however they do not define how to compare any pair of events by their timestamps. For example, it is not defined if a *cBroadcast* event of m *happened–before* the *cDeliver* event of any other message m' using timestamps. We propose a Causal Timestamp System (*CTS*) for the causal broadcast abstraction (Def. 3 and Def. 4) that timestamps all events in H. As *CTS* characterizes causality [14], it lets us reason about the *happened–before* relation of any pair of events, no matter if they are *cBroadcast* or *cDeliver* events. We have modeled *CTS* according to timestamp systems defined by Torres-Rojas et al. [14] but we have also included the Causal Order delivery condition of messages.

In a *CTS* for a system with P processes, each process p_i keeps a local clock in the pair (seq_i, V_i). The clock component seq_i counts the number of messages *cBroadcast* by p_i, while the clock component V_i is a vector of P entries where $V_i[j]$ counts the number of messages *cBroadcast* by p_j and *cDeliver*ed in p_i.

CTS is defined as a tuple $(H, S, CT.stamp, CDC, R_S, R_{CT})$, where:

1. H is the global history of causal broadcast (*cBroadcast*) and causal delivery (*cDeliver*) events of the system.
2. S is the set of timestamps. A timestamp is an identifier of the form (i, seq_i, V_i), where i is a non negative integer that identifies the process p_i, seq_i is the number of messages *cBroadcast* by p_i and V_i is a P-vector of the set $\mathbb{N}^{|P|}$ of integers, that represents the *cDeliver*ed messages in p_i.
3. $CT.stamp$ is a global identifier function that acts as a logical clock assigning to each event a timestamp of S. That is, $CT.stamp : H \rightarrow S$. If a is a *cBroadcast* or *cDeliver* event produced in a process p_i, the event has the timestamp $CT.stamp(a) = CT_i.stamp(a)$. For any process p_i, the function $CT_i.stamp$ is defined using the following rules:

RV0) Initialization:

$$i = \text{process identifier}$$
$$seq_i = 0$$
$$\forall j \in \{0, \ldots, |P| - 1\} : V_i[j] = 0$$

RV1) Before the *cBroadcast* event a of message m is generated at site i:

$$seq_i = seq_i + 1$$

Message m is broadcast with a timestamp $CT_i.stamp(a) = (i, seq_i, V_i)$.

RV2) Before a message m with the timestamp (j, seq_j, V_j), that fulfills the causal delivery condition CDC, is *cDeliver*ed at site i:

$$\forall k \in \{0, \ldots, |P| - 1\} : V_i[k] = max(V_i[k], V_j[k])$$
$$V_i[j] = V_i[j] + 1$$

In a timestamp $CT.stamp(a) = (i, seq_a, V_a)$, the value $V_a[k]$ counts the number of events of messages *cDeliver*ed in p_i from p_k, up to the event a. Similarly, seq_a counts the number of *cBroadcast* events happened in p_i up to the event a.

4. *CDC* is the Causal Delivery Condition to deliver messages in causal order. A message m with timestamp (j, seq_j, V_j) can be *cDeliver*ed in p_i, if the both following conditions hold:

$$seq_j = V_i[j] + 1 \tag{5a}$$

$$V_j[k] \leq V_i[k],\ \forall k \neq j \tag{5b}$$

That is, messages whose *cBroadcast* events *happened–before* the *cBroadcast* event of m either by the same sender (5a) or by different senders (5b) have to be *cDeliver*ed before m.

5. Set of relations $R_S = \{<_S, \overset{S}{=}, \overset{S}{||}\}$ over S. Let $t_1 = (i, seq_i, V_i)$ and $t_2 = (j, seq_j, V_j)$ be timestamps $\in S$, we define the relations as follows:

$$t_1 <_S t_2 \iff ((i = j) \wedge ((seq_i < seq_j) \vee (V_i < V_j))) \tag{6a}$$

$$\vee\ ((i \neq j) \wedge (V_i[i] < V_j[i]) \wedge (V_i \leq V_j)) \tag{6b}$$

$$t_1 \overset{S}{=} t_2 \iff (i = j)\ \wedge\ (seq_i = seq_j) \wedge\ (V_i = V_j) \tag{7}$$

$$t_1 \overset{S}{||} t_2 \iff not\ (t_1 <_S t_2)\ \wedge\ not\ (t_2 <_S t_1) \tag{8}$$

The pair $(S, <_S)$ is a strict partial ordered set.

Being the comparison of two vectors V and W as follows:

$$V = W \iff \forall i \in \{0, \ldots, |P| - 1\} : V[i] = W[i]$$
$$V \leq W \iff \forall i \in \{0, \ldots, |P| - 1\} : V[i] \leq W[i]$$
$$V < W \iff V \leq W \wedge \exists j \in \{0, \ldots, |P| - 1\} \text{ such that } V[j] < W[j]$$
$$V \,||\, W \iff V \not< W \wedge W \not< V$$

6. Set of relations $R_{CT} = \{\overset{CT}{\longrightarrow}, \overset{CT}{=}, \overset{CT}{||}\}$ over H. Let $a, b \in H$ with timestamps $CT.stamp(a)$ and $CT.stamp(b)$ over S, the relations of R_{CT} are defined as follows:

$$\begin{aligned} a \overset{CT}{\longrightarrow} b &\iff CT.stamp(a) <_S CT.stamp(b) \\ &\iff CTS \text{ "believes" that } a \text{ causally precedes } b. \end{aligned} \tag{9}$$

$$\begin{aligned} a \overset{CT}{=} b &\iff CT.stamp(a) \overset{S}{=} CT.stamp(b) \\ &\iff CTS \text{ "believes" that } a \text{ and } b \text{ are the same event.} \end{aligned} \tag{10}$$

$$\begin{aligned} a \overset{CT}{||} b &\iff CT.stamp(a) \overset{S}{||} CT.stamp(b) \\ &\iff CTS \text{ "believes" that } a \text{ and } b \text{ are concurrent.} \end{aligned} \tag{11}$$

As the *CT.stamp* is a bijective function that preserves the relations from H to S, the relations on the set R_{CT} are isomorphic to their respective relations on the set R_S.

Example of CTS timestamping

The example of Fig. 2 depicts a group of three processes communicating with a *CRB* that employs the *CTS* timestamp system defined above. In the example, we do not only show how *CRB* broadcasts messages and delivers them in causal order but also showcase how *CTS* captures the causal relationship between *cBroadcast* and *cDeliver* events. Let us observe this capturing in action in Fig. 2, analyzing the following type of events:

cBroadcast and self-cDeliver events. Consider the process p_0 that *cBroadcast*s the message m_0 in the event e_{00}. The rule *RV*1 of *CTS* assigns to the event e_{00} the timestamp t_{00} with the value (0, 1, (0, 0, 0)), as m_0 is the first message *cBroadcast* by p_0 and p_0 has not yet *cDeliver*ed any messages. The message m_0 piggybacks the timestamp t_{00}. When the message m_0 is received by *CRB* in p_0, m_0 can be delivered in causal order according to the *CDC* of *CTS*, so the message m_0 is self-*cDeliver*ed in p_0 in the event e_{01}. According to the rule *RV*2 of *CTS*, the timestamp of the event e_{01} is $t_{01} = (0, 1, (1, 0, 0))$. From the point of view of *CTS*, the timestamp t_{00} is earlier than the timestamp t_{01} because $t_{00} <_S t_{01}$, as seen in (6a). As a result, from the point of view of *CTS*, the event e_{00} happened before the event e_{01}, $(e_{00} \overset{CT}{\longrightarrow} e_{01})$ as seen in (9).

cBroadcast and cDeliver events. *CTS* captures the causal relation between the *cBroadcast* event e_{00} of the message m_0 in p_0 and the *cDeliver* event e_{10} of the message m_0 in p_1. The event e_{00} is timestamped as $t_{00} = (0, 1, (0, 0, 0))$ and e_{10} is timestamped as $t_{10} = (1, 0, (1, 0, 0))$. As these two events happened in different processes p_0 and p_1, *CTS* determines that the timestamp t_{00} is earlier than the

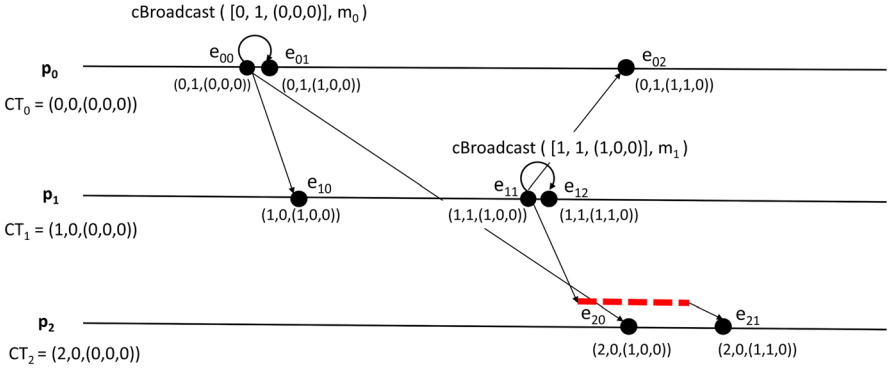

Fig. 2 Example of causal broadcast and causal deliver event timestamping with *CTS*. All timestamps can be compared to determine if they are causally related or if they are concurrent

timestamp t_{10}. This is true because $t_{00} <_S t_{10}$, according to (6b) of the relation $<_S$. As a result, from the point of view of the *CTS*, the *cBroadcast* event e_{00} occurred before the *cDeliver* event e_{10}, that is, $e_{00} \overset{CT}{\longrightarrow} e_{10}$ as seen in (9).

cBroadcast events. The process p_0 *cBroadcast*s the message m_0 in the event e_{00}. The message m_0 piggybacks the timestamp $t_{00} = (0, 1, (0, 0, 0))$. In the event e_{11}, p_1 *cBroadcast*s m_1. The message m_1 piggybacks the timestamp $t_{11} = (1, 1, (1, 0, 0))$ of event e_{11}. According to (6b), it is true that $t_{00} <_S t_{11}$. So from the point of view of *CTS* in (9), the event e_{00} happened before the event e_{11}, $(e_{00} \overset{CT}{\longrightarrow} e_{11})$. Hence, by the causal delivery rule (Def. 4) implemented by the *CDC* of *CTS*, m_0 is *cDeliver*ed before m_1 in every process. As it can be seen in p_2, the *cDeliver*y of m_1 is delayed until the *cDeliver*y of m_0.

Concurrent events. With *CTS* it is also possible to determine if any pair of events are concurrent comparing their timestamps. For example, any process having access to the timestamps can determine the causal relation between the *cDeliver* event e_{01} in p_0 and the *cDeliver* event e_{12} in p_1. The timestamps of the events e_{01} and e_{12} are $t_{01} = (0, 1, (1, 0, 0))$ and $t_{12} = (1, 1, (1, 1, 0))$ respectively. From the point of view of *CTS* these timestamps are concurrent as seen in (8). So, from the point of view of *CTS*, e_{01} and e_{12} are concurrent events according to (11).

Finally, as we will prove in the theorem 1, the *CTS* characterizes causality, so *CTS* captures correctly and with the maximal accuracy the causal relationship between any pair of events, whether they are *cBroadcast* or *cDeliver* events.

Definition 5 $(m \overset{CT}{\longrightarrow} m')$ The *cBroadcast* of a message m' may have been potentially caused by the *cBroadcast* of another message m, denoted as $m \overset{CT}{\longrightarrow} m'$, if from the point of view of *CTS* the *cBroadcast* event a of m *happened–before* the *cBroadcast* event b of m'. More formally:

$$m \overset{CT}{\longrightarrow} m' \iff a \overset{CT}{\longrightarrow} b$$
$$\wedge\ (a =< p, cBroadcast|m >)$$
$$\wedge\ (b =< q, cBroadcast|\ m' >)$$

From now on, we use $CT(a)$ instead of $CT.stamp(a)$ when no confusion arises.

Theorem 1 *CTS characterizes causality* [14]. *That is*, $\forall a, b \in H$:

1. $a = b \iff a \overset{CT}{=} b$
2. $a \rightarrow b \iff a \overset{CT}{\longrightarrow} b$
3. $a \,||\, b \iff a \overset{CT}{||} b$

The first condition asserts that the *CTS* gives only one timestamp to each event, and all events have different timestamps. The second condition asserts that the *CTS* always detects the causal relation among events. The third condition asserts that when two events are concurrent, the *CTS* detects that these events are concurrent. In order to prove these properties, let $CT(a) = (i, seq_a, V_a)$ and

$CT(b) = (j, seq_b, V_b)$ be the timestamps of events a and b that happened in p_i and p_j, respectively.

Proof (1). Suppose that $a = b$ holds. We want to prove that *CTS* clock only ticks once for each *cBroadcast* or *cDeliver* event happened. According to clock rules *RV*1 and *RV*2, an event only has one timestamp assigned by the *CTS* in the course of a run. Hence $a = b \Rightarrow CT(a) \overset{S}{=} CT(b)$. By definition of $a \overset{CT}{=} b$, it follows $a = b \Rightarrow a \overset{CT}{=} b$.

Conversely, now suppose $a \overset{CT}{=} b$. We want to prove that *CTS* clock never assigns the same timestamp to different events in a run. As *CTS* clock is a strictly monotonic increasing function, it never assigns in rules *RV*1 and *RV*2 the same timestamp to events a and b if they are different. Therefore, it follows that $a \overset{CT}{=} b \Rightarrow a = b$. □

Proof (2). First, we want to prove that *CTS* always detects the causal relation among any pair of events. That is, if $a \rightarrow b \Rightarrow a \overset{CT}{\longrightarrow} b$. We prove by induction on the number n of events that happened after a and *happened–before* b. That is: $P(n)$: if $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n$, where $a = e_0 \wedge b = e_n$, then $a \overset{CT}{\longrightarrow} b$.

Base case. The statement holds for $n = 1$, i.e., a and b are consecutive events. We consider two cases.

Case 1. The event a happened before b in the same process p_i. We have the following sub-cases:

a) a can be any type of event and b is a *cBroadcast* event. Then, by rule *RV*1, $seq_a < seq_b$, so by (6a) $CT(a) <_S CT(b)$.
b) a can be any type of event and b is a *cDeliver* event of a message m from p_k. Then, by rule *RV*2, $(V_a[k] < V_b[k]) \wedge (V_a[r] \leq V_b[r]), \forall r \neq k$, so by (6a) $CT(a) <_S CT(b)$.

As in both cases a) and b) it is true that $CT(a) <_S CT(b)$, then by (9) it is true that $a \overset{CT}{\longrightarrow} b$.

Case 2. The events a and b happened in different processes $i \neq j$. Let a be the *cBroadcast* event in p_i of a message m and b be the *cDeliver* of m in p_j. Then, by rule *RV*2 and condition *CDR*, it is true $(V_a[i] < V_b[i]) \wedge (V_a[k] \leq V_b[k]), \forall k \neq i$. Therefore by (6b) $CT(a) <_S CT(b)$, so by (9) it is true that $a \overset{CT}{\longrightarrow} b$.

Inductive step. For any $k \geq 1$, if $P(k)$ holds, then $P(k + 1)$ also holds. $P(k + 1)$: if $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_k \rightarrow e_{k+1}$, where $a = e_0$ and $b = e_{k+1}$, then $a \overset{CT}{\longrightarrow} b$.

By the induction hypothesis, we assume that $n = k$ holds for a given k, meaning $P(k)$ is true: if $e_0 \rightarrow e_1 \ldots \rightarrow e_k$ where $a = e_0$, then $a \overset{CT}{\longrightarrow} e_k$. It follows that events e_k and $b = e_{k+1}$ are consecutive events. By the base case, $e_k \overset{CT}{\longrightarrow} b$ so it is true by (9) that $CT(e_k) <_S CT(b)$. By the induction hypothesis $a \overset{CT}{\longrightarrow} e_k$, so it is true by (9) that $CT(a) <_S CT(e_k)$. Then, as $CT(a) <_S CT(e_k) <_S CT(b)$, it follows that $CT(a) <_S CT(b)$ and by (9) it is true that $a \overset{CT}{\longrightarrow} b$.

Since both the base case and the inductive step have been proved as true, it is true, for any events $a, b \in H$, that if $a \rightarrow b$ then $a \overset{CT}{\longrightarrow} b$.

Second, we want to prove that if *CTS* "believes" that a and b are causally related, then a *happened-before* b. That is, if $a \overset{CT}{\longrightarrow} b \Rightarrow a \rightarrow b$. We use a proof by contradiction. We have to prove two cases: (1) *CTS* "believes" that a and b are causally related but a and b are concurrent and (2) *CTS* "believes" that a and b are causally related but $b \rightarrow a$.

Case 1. Assume for the sake of contradiction that *CTS* "believes" that a and b are causally related but a and b are concurrent. That is, $a \overset{CT}{\longrightarrow} b \Rightarrow a \,||\, b$. If a and b are concurrent, these events necessarily happened in different processes p_i and p_j respectively. In addition, the event a cannot be a *cBroadcast* event of a message m in p_i and b cannot be a *cDeliver* event of m in p_j. Also there do not exist two events c and d, where c is the *cBroadcast* in p_i of a message m and d is the *cDeliver* of m in p_j, such that $a \rightarrow c$ and $d \rightarrow b$. However, from the point of view of *CTS* a and b are causally related, so by (9) $CT(a) <_S CT(b)$. In addition, as a and b happened in p_i and p_j respectively, by (6b), $V_a[i] < V_b[i] \ \wedge \ V_a \leq V_b$. So, the local clock of p_j had to *cDeliver*ed a message m from p_i according to rule RV2) of *CTS* in a event that happened before b or in the same event b. This fact contradicts the fact that a and b are concurrent. Thus, if *CTS* "believes" a and b are causally related it is not possible that a and b are concurrent events.

Case 2. Assume for the sake of contradiction that *CTS* "believes" that a and b are causally related but $b \rightarrow a$. If from the point of view of *CTS* a and b are causally related, then $CT(a) <_S CT(b)$ by (9). Every time an event happens, the *CTS* clock increments its value according to clock rules *RV*1 and *RV*2, so if a were to happen after b, this would imply that $CT(a)$ could not be less that $CT(b)$ by (6a) and (6b) contradicting the fact that $CT(b) <_S CT(a)$. Thus, if *CTS* "believes" a and b are causally related it is not possible that $b \rightarrow a$.

Since both cases have been proved by contradiction, it is true, for any events $a, b \in H$, that if $a \overset{CT}{\longrightarrow} b$ then $a \rightarrow b$. □

Proof (3). Suppose that $a||b$. Def. 2 says that $a||b \iff not\ (a \rightarrow b) \ \wedge \ not\ (b \rightarrow a)$. Considering the property (2) of theorem 1, it is true that

$a||b \Rightarrow not\ (a \overset{CT}{\longrightarrow} b) \wedge not\ (b \overset{CT}{\longrightarrow} a)$. Thus, it follows by (11) that $a \,||\, b \Rightarrow a \overset{CT}{||} b$.

Conversely, suppose $a \overset{CT}{||} b$. Considering the property (2) and the definition of concurrent events in the relation *CT*, it is true that

$a \overset{CT}{||} b \Rightarrow not\ (a \rightarrow b) \wedge not\ (b \rightarrow a)$. Hence $a \overset{CT}{||} b \Rightarrow a \,||\, b$. □

Theorem 2 $m \overset{c}{\rightarrow} m' \iff m \overset{CT}{\longrightarrow} m'$

Proof Let a and b be the *cBroadcast* events of messages m and m' respectively, where $a \rightarrow b$. As *CTS* characterizes causality, it is true that $a \rightarrow b \iff a \overset{CT}{\longrightarrow} b$. Therefore, by Def. 3 and Def. 5, $m \overset{c}{\rightarrow} m' \iff m \overset{CT}{\longrightarrow} m'$. □

Theorem 3 *CTS timestamps cDeliver events respect the causal order Delivery rule* (*Def.* 4).

Proof Let a and b be the *cBroadcast* events of messages m and m' respectively. By hypothesis, message m is *cBroadcast* before the *cBroadcast* of m'. First, consider that m and m' are *cBroadcast* by the same sender p. According to condition (5a) of *CTS*, messages from the same sender are *cDeliver*ed in FIFO order. Therefore m is causally delivered in any process before m'. Second, consider that m and m' were *cBroadcast* by p and q with timestamps (p, seq_m, V_m) and $(q, seq_{m'}, V_{m'})$ respectively. Then, it is true that $V_m[p] < V_{m'}[p]$. According to condition (5b) of *CDC* and theorem 2, in any process r, message m' cannot be *cDeliver*ed unless $V_{m'}[p] \leq V_r[p]$. Hence, m has already been *cDeliver*ed in process r. □

The important property of *CTS* of capturing with the maximal accuracy the *happened–before* relation among *cBroadcast* and *cDeliver* events improves the quality of the passive monitoring of distributed applications. In the next section we define a passive monitor for the global predicate evaluation of distributed applications that use a *CRB* with *CTS* and illustrate the monitoring with an example.

6 Consistent global state monitoring algorithm for CTS

In this section we propose a consistent Global State Monitoring algorithm (*GSM*) for monitoring passively the *CTS* timestamp events in a distributed application, in order to construct consistent global states. The distributed application uses a causal reliable broadcast service that implements causal delivery with the Causal Timestamp System (*CTS*) described in Sect. 5. For short we refer to *GSM* for *CTS* as *GSM*.

This *GSM* algorithm is a variation of the algorithm from [24], where our monitor receives *CTS*-timestamps not only of *cBroadcast* events but also of *cDeliver* events. This increase in the number of correctly timestamped events will allow our monitor to detect more fine-grained global state transitions, and hence to get better global predicate evaluations. This monitor schema can be generalized as a stream processing system of *cBroadcast* and *cDeliver* events that can be causally ordered according to their timestamps and then analyzed either with an on-line monitor or off-line from a database [28].

GSM algorithm assumes the model described in Sect. 3, but with some failure model variations described in [24]. It assumes that during the monitoring executions, application processes $\in P$ do not fail. It also assumes that the monitor is a correct process. Application processes communicate with the monitor using reliable point-to-point links. Application processes use the reliable broadcast communication layer and, on top of this layer, all processes use the causal broadcast service. In order to let application processes to access event timestamp information, *GSM* assumes that the causal broadcast service delivers each timestamp of every event (*cBroadcast* and *cDeliver*) to the application process as soon as they happen. With all of these assumptions, the monitoring algorithm can ensure that if an application

process *cBroadcast* a message m, it will be *cDeliver*ed in any application process (reliable broadcast properties of Def. 1) and the notifications of the corresponding *CTS* timestamps of *cBroadcast* and *cDeliver* events will be sent reliably from the application processes to the monitor process.

These timestamp notifications must causally order by the monitor. For this purpose, the monitor uses a clock that is implemented by two vectors for counting the application *cBroadcast* event timestamps causally delivered by *GSM* and the application *cDeliver* event timestamps causally delivered by *GSM* respectively. The reason that the *GSM* uses two vectors instead of the one used by *CTS* is that the monitor algorithm needs to causally order timestamps of *cBroadcast* and *cDeliver* notifications, while the application processes only need to causally order timestamps of *cBroadcast* messages.

6.1 The global state monitoring algorithm

The global state monitoring algorithm has two parts: the distributed application A that runs P processes that communicate among them with the Causal Reliable Broadcast abstraction, and a passive Global State Monitor process *GSM* that communicates with each application process with a reliable point to point channel. Each application process sends messages to *GSM* containing *CTS*-timestamps of causal broadcast (*cBroadcast*) and causal deliver (*cDeliver*) events as soon as the events happen in the process, and *GSM* delivers them in causal order. The event $< p, send \mid GSM, m_e >$ is used by the process p for sending a message m_e with the timestamp of an event e to *GSM*, while the event $< GSM, deliver \mid m_e >$ is used by *GSM* to deliver a timestamp message m_e.

Let a and b be any pair of events $\in H$ where a *happened–before* b, and m_a and m_b be the messages containing the *CTS*-timestamps of a and b respectively. The *GSM* process implements the following causal monitoring delivery rule:

$a \rightarrow b \Rightarrow$

$< GSM, Deliver \mid m_a > \rightarrow < GSM, Deliver \mid m_b >$

This rule is implemented in *GSM* using a local logical clock called Monitoring Clock (*MC*) and the following delivery conditions:

- BDC is the *cBroadcast* Delivery Condition. *GSM* causally delivers a timestamp m_e of *cBroadcast* event e that happened in p_j when the *MC* clock time is greater than the timestamps of all *cBroadcast* and *cDeliver* events in p_j that *happened–before* e.
- DDC is the *cDeliver* Delivery Condition. *GSM* causally delivers a timestamp m_e of *cDeliver* event e in p_j when the *MC* clock time is greater than all timestamps of events in p_j that *happened–before* e, and for each of these events that are *cDeliver* events, the *MC* clock time is also greater than their corresponding *cBroadcast* event timestamps, no matter the sender.

When a *cBroadcast* or *cDeliver* timestamp message m_e fulfills the condition *BDC* or *DDC* respectively, *MC* clock ticks just before *GSM* causally delivers m_e.

The clock *MC* is implemented by two vectors: Broadcast Vector (*BV*) and Delivery Vector (*DV*). *BV* counts the application *cBroadcast* event timestamps causally delivered by *GSM*. *DV* counts the application *cDeliver* event timestamps causally delivered by *GSM*.

More formally, the monitor *GSM* is defined as a tuple (*H*, *S*, *MC*, *BDC*, *DDC*) where:

1. *H* is the global history of *cBroadcast* and *cDeliver* events of the application *A*.
2. *S* is the set of timestamps. A timestamp is an event identifier of the form (j, seq_j, V_j), where j represents the process p_j where the event happened, seq_j is the number of messages *cBroadcast* by p_j and V_j is a P-vector of the set $\mathbb{N}^{|P|}$ of integers, that represents the *cDeliver*ed messages in p_j.
3. *MC* is the Monitor Clock. The monitor *GSM* causally delivers a message m_e with the timestamp (j, seq_j, V_j) of a *cBroadcast* or *cDeliver* event e that happened in p_j. This causal delivery is expressed according to the following rules for updating *MC*:

   RM0) Initialization. $\forall j \in \{0, \ldots, |P| - 1\}$:
   $BV[j] = 0$
   $DV[j] = 0$

   RM1) Let m_e be a message received by *GSM* that is the timestamp of a *cBroadcast* event e in p_j that fulfills the *BDC* condition. Before m_e is *Deliver*ed by *GSM*:
   $BV[j] = BV[j] + 1$

   RM2) Let m_e be a message received by *GSM* that is the timestamp of a *cDeliver* event e in p_j that fulfills the *DDC* condition. Before m_e is *Deliver*ed by *GSM*:
   $DV[j] = DV[j] + 1$

4. *BDC* is the delivery condition for *cBroadcast* events. A message m_e with timestamp (j, seq_j, V_j) of a *cBroadcast* event e in p_j can be causally delivered by *GSM* if the following conditions hold:

   BDC1) $BV[j] + 1 = seq_j$
   (i.e., *GSM* knows all *cBroadcast* events in p_j that *happened–before* the event e)

   BDC2) $DV[j] = \sum_k V_j[k]$
   (i. e. *GSM* knows all *cDeliver* events in p_j that *happened–before* the event e)

5. *DDC* is the delivery condition for *cDeliver* events. A message m_e with the timestamp (j, seq_j, V_j) of a *cDeliver* event e in p_j can be causally delivered by *GSM* if the following conditions hold:

   DDC1) $BV[j] = seq_j$

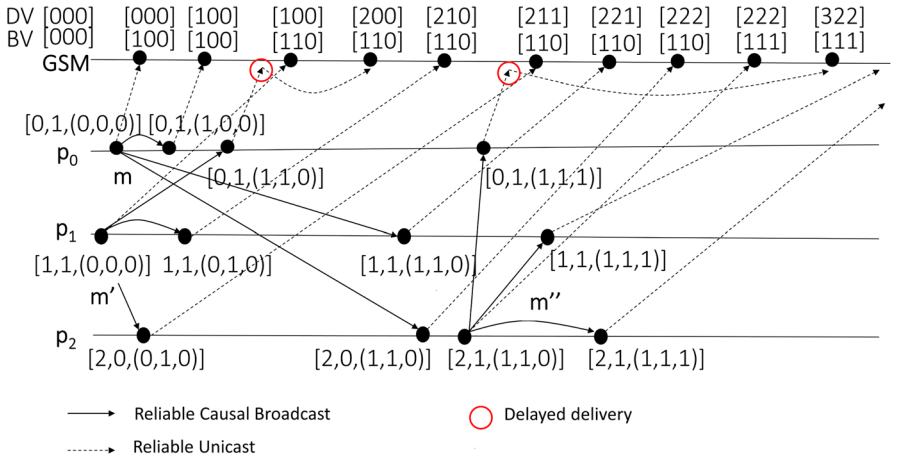

Fig. 3 Example of the global state monitoring algorithm using *CTS* timestamps, where *GSM* is the monitor process and p_i are the application processes that communicate using the Causal Reliable Broadcast service with our *CTS*

(i.e., *GSM* knows all *cBroadcast* events in p_j up to the event e)

DDC2) $BV[k] \geq V_j[k],\ \forall k \neq j$

(i.e., *GSM* knows at least $V_j[k]$ *cBroadcast* events from p_k)

DDC3) $DV[j] + 1 = \sum_k V_j[k]$

(i.e., *GSM* knows all *cDeliver* events that *happened–before* the event e in p_j)

Figure 3 shows how the algorithm delivers the event messages in the monitor process. It shows how the monitor delays the delivery of the message from p_0 with timestamp [0, 1, (1, 1, 1)]. This message is a *cDeliver* event of a message *cBroadcast* by process p_2 with timestamp [2, 1, (1, 1, 0)]. Hence the monitor must delay the deliver of the *cDeliver* event at least until it has delivered its corresponding *cBroadcast* event. Of course, it also must delay the delivery of *cDeliver* event until it has delivered all the other causal preceding events of this one such as the events with timestamps [0, 1, (0, 0, 0)], [2, 0, (1, 1, 0)], [1, 1, (0, 0, 0)], and [2, 0, (0, 1, 0)]

Now we will show that *GSM* algorithm satisfies the safety an liveness properties. First, we prove that the causal order delivery property is never violated (safety). Assume that *GSM* receives messages m and m' from p_j and p_k respectively. Assume that m has the timestamp $t = (j, seq_e, V_e)$ corresponding to an event e in p_j and m' has the timestamp $t' = (k, seq_{e'}, V_{e'})$ corresponding to an event e' in p_k. Let e *happened–before* $e'(e \rightarrow e')$. Next we will prove that as *CTS* characterizes causality (theorem 1), $e \rightarrow e' \Leftrightarrow t <_S t'$ and thus *GSM* must *deliver* m before m'.

Theorem 4 *Safety property. The Monitoring Algorithm delivers messages in causal order. I. e. for any events* $e, e' \in H$, *if* $e \rightarrow e'$ *then GSM delivers* m *before* m'.

Proof We give a proof by induction on the number n of events that happened after e and *happened–before* e'. That is: $P(n)$: if $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n$, where $e = e_0 \wedge e' = e_n$ then *GSM delivers* m before m'.

Base case. The statement holds for $n = 1$, i.e., e and e' are consecutive events. We consider two cases.

Case 1. m and m' are both transmitted by the same process p_j. We have the following sub-cases:

a) e can be any type of event and e' is a *cBroadcast* event, both in p_j. Then by rule *RV*1 of Sect. 5, $seq_e < seq_{e'}$ and $V_e \leq V_{e'}$. Hence, under the *BDC*1 condition of Sect. 6, $BV[j] + 1 = seq_{e'}$ (*GSM* must have *Delivered* all the *cBroadcasts* from p_j previous to the *cBroadcast* of m) and, under the *BDC*2 condition of Sect. 6, $DV[j] = \sum_k V_{e'}[k]$ (*GSM* must have *Delivered* all *cDeliver* events that *happened–before* m'). Then *GSM* must *Deliver* m before m'.

b) e can be any type of event and e' is a *cDeliver* event in p_j. Then $seq_e \leq seq_{e'}$ and $V_e < V_{e'}$. Hence, under the rule *DDC*1 in Sect. 6, $BV[j] = seq_{e'}$ (*GSM* knows all *cBroadcast* events of j up to event e') and, under the *DDC*3 condition in Sect. 6, $DV[j] + 1 = \sum_k V_{e'}[k]$ (*GSM* knows all *cDeliveries* in p_j that *happened–before* e'), *GSM* must *Deliver* m before m'.

Case 2. m and m' are transmitted by two distinct processes p_j and p_k. Message m corresponds to the *cBroadcast* event e of a message x in p_j and m' corresponds to the *cDeliver* event e' of this message x in p_k. Then, by rule *RV*2 of Sect. 5, $seq_e = V_{e'}[j], V_e[j] < V_{e'}[j]$, and $V_e[i] \leq V_{e'}[i] \forall i \neq j$. Hence, under the condition *DDC*2 in Sect. 6, $BV[j] \geq V_{e'}[j]$ (*GSM* knows at least $V_{e'}[j]$ *cBroadcast* events from p_j), *GSM* must *Deliver* m before m'.

Inductive step. Show that for any $k \geq 1$, if $P(k)$ holds, then $P(k + 1)$ also holds. That is: $P(k + 1)$: if $e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_k \rightarrow e_{k+1}$ then *GSM Delivers* m before m'. Assume by the induction hypothesis that for a particular k, the single case $n = k$ holds, meaning $P(k)$ is true: if $e_0 \rightarrow e_1 \ldots \rightarrow e_k$ then *GSM Delivers* m before m_k. It follows that: Events e_k and e_{k+1} are consecutive events that can happen in the same process or in different processes. Then, it is true by the base case that *GSM Delivers* m_k before m_{k+1}. Also, by the induction hypothesis *GSM Delivers* m before m_k, then it follows that *GSM Delivers* m before m_{k+1}. That is, the statement $P(k + 1)$ also holds true, establishing the inductive step. Since both the base case and the inductive step have been proved as true, by mathematical induction it follows for any events $e, e' \in H$, if $e \rightarrow e'$ then *GSM Delivers* m before m'. □

Now we prove that any message received by the monitor process will be eventually *Delivered* (liveness).

Theorem 5 *Liveness property. The Monitoring algorithm never delays the Delivery of a message indefinitely.*

Proof To prove liveness, we will suppose that there exists an event e_k (*cBroadcast* or *cDeliver*) of which *GSM* has notice after receiving a message m from p_k informing of e_k, and that this message is never *Deliver*ed at *GSM*. Then we will show that we end up with a contradiction. Message m comes with the timestamp (p_k, seq_k, V_k). $\forall i \neq k, V_k[i]$ counts the number of *cBroadcast* events in p_i that causally precede e_k, and seq_k counts the number of *cBroadcast* events in p_k that causally precede e_k. Also, either e_k is *cBroadcast* event and $V_k[k]$ counts the number of *cDeliver* events in p_k that causally precede e_k, or e_k is a *cDeliver* event and $V_k[k] - 1$ counts the number of *cDeliver* events in p_k that causally precede e_k. Finally, $\forall j \neq k$, if e_j is the *cBroadcast* that corresponds with the $V_k[j]$-th and last *cBroadcast* from p_j that causally precedes event e_k, and e_j has the timestamp (p_j, seq_j, V_j), then $V_j[j]$ counts the number of *cDeliver* events in p_j that causally precede e_k. The number of events that causally precede e_k is thus bounded. In absence of failure, and after a finite time, messages of all these events will have arrived to *GSM*, and event e_k will be deliverable (see the causal delivery rules). Note that, by the delivery rules, a message of a *cDeliver* event e_r cannot be *Deliver*ed in *GSM* if its corresponding *cBroadcast* event message has not been *Deliver*ed in *GSM* and also that a *cBroadcast* event e_s of a process s cannot be *Deliver*ed in *GSM* if the *Deliver* events in s that causally precede e_s have not been delivered in *GSM*.

So, if e_k cannot be delivered, there exists an event e_i, such that $e_i \rightarrow e_k$, which is never *Deliver*ed. The same reasoning applied to e_k can again be applied to e_i, and so on. As the number of events causally preceding e_k is finite, we end up with an event e_n which has no event causally preceding it. So e_n will be *Deliver*ed enabling the delivery of e_i and e_k, which shows the contradiction. □

7 Example: monitored domotic application using CTS

This section describes an example of a domotic application that allows comparing two passive monitoring systems, the *GSM* monitoring system as described in Sect. 6 and the Babaoğlu monitoring for Birman's *CRB* as described in [24].

The domotic application is made up of an alarm keypad to arm and disarm the surveillance system, sensors for detecting intrusions and actuators to ring alarms, and a passive monitor. This application is distributed, with processes controlling the keypad, sensors and actuators respectively. When we refer to the monitor for Birman's *CRB*, it is assumed that processes communicate using the Birman's *CRB* algorithm and its timestamping described in Sect. 2. When we refer to the *GSM* monitor, processes communicate using a *CRB* with *CTS* timestamping, as it is described in 5.

Each monitoring system receives notifications from processes of the domotic application. *GSM* monitor receives notifications of *cBroadcast* and *cDeliver* events. The monitor for Birman's *CRB* receives only notifications of *cBroadcast* events. Based on these notifications, these monitors can compute consistent global states and evaluate predicates.

Figure 4 shows on its lower right side a state diagram that models the behavior of the alarm system. It shows the states and the meaningful transitions of the state

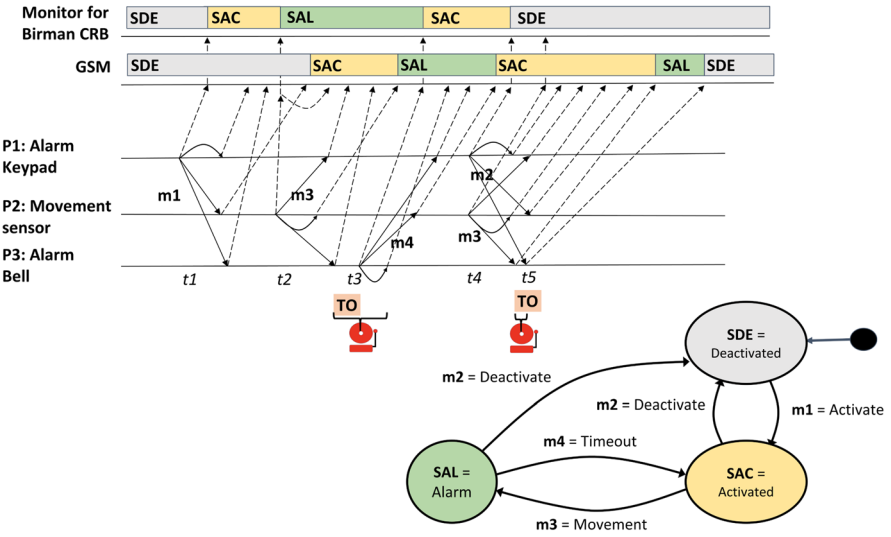

Fig. 4 Domotic example of causal reliable broadcast with two monitoring systems: Monitor for Birman's *CRB* and *GSM* for *CTS*. The state transition diagram in the bottom right shows the meaning of both the states and the transitions. The space-time diagram shows how the alarm system is activated with an *activate cBroadcast* and how an alarm rings with a *movement cBroadcast*. Also shows how notifications of all the events are sent and delivered to the monitor process

machine implemented by the domotic application. Both, processes controlling sensors and actuators and the monitor processes implement this same state machine. However, they are not replicated state machines. Each application process transitions through a different sequence of states depending on the order of *cDeliver* of the received messages, which can be different due to concurrent events. On the other hand, the *GSM* monitor transitions when it receives notifications of *cBroadcast* and *cDeliver* events. However, the monitor for Birman's *CRB* transitions when it receives *cBroadcast* events.

While the system is in the *Deactivated* initial state, the user can arm the alarm system through the keypad, what triggers the *Activate* transition to the *Activated* state. While the system is in the *Activated* state, the event of a sensor detecting movement triggers the *Movement* transition to the *Alarm* state, with the associated action of ringing the alarm. While the system is in the *Alarm* state, either (1) a timeout event stops the bell action and triggers the *Timeout* transition to the *Activated* state, or (2) the event of a user deactivating the alarm in the keypad stops the bell and triggers the *Deactivate* transition to the *Deactivated* state.

We assume that our domotic application is implemented using three processes that perform different actions: process p_1 controls the keypad, process p_2 controls the movement sensors, and process p_3 rings the alarm. These processes *cBroadcast* notifications that, when *cDeliver*ed, cause the corresponding transitions in the receiving processes.

We also assume that each monitoring system is implemented by a monitor process having two layers: (1) a monitor's communication layer that delivers in

causal order the notification messages it receives informing of the corresponding events in p_1, p_2, p_3, and (2) the monitor's application layer that detects state transitions of the domotic application and evaluates predicates.

Figure 4 shows a space-time diagram of an execution scenario, with the domotic application processes p_1, p_2, p_3, and the monitor processes. The diagram does not show the vector clocks neither in the domotic application nor in the monitors. The domotic application processes deliver their messages in causal order and the monitors deliver the application event notifications in causal order too.

At t_1 the three processes are in the *Deactivated* state. The user arms the alarm in the keypad and so process p_1 broadcasts $m1$ informing of the *Activate* transition. Upon delivery of $m1$ different actions occur in each process:

- p_1: it transitions locally to the *Activated* state.
- p_2: it transitions locally to the *Activated* state and turns on the movement sensors.
- p_3: it transitions locally to the *Activated* state.

At t_2 the three processes are in the *Activated* state. Movement is detected by a sensor and thus process p_2 *cBroadcast*s m_3 informing of the *Movement* transition. Upon *cDeliver* of m_3 different actions occur in each process:

- p_1: it transitions locally to the *Alarm* state. Although not shown in the figure, note that after t_2, if p_1 does not either self-*cDeliver* a message m_2 informing of the *Deactivate* transition, or *cDeliver* a message $m4$ informing of the *Timeout* transition sent by p_3, then it could detect that p_3 is failing and no user is deactivating the alarm. In this case p_1 could send, for example, an SMS message to the user cellphone.
- p_2: it transitions locally to the *Alarm* state.
- p_3: it transitions locally to the *Alarm* state, activating the ringing of the bell through a local actuator.

At t_3 the three processes are in the *Alarm* state. The bell is still ringing, and a timeout event *TO* occurs at p_3, which broadcasts $m4$ informing of the *Timeout* transition. Upon *cDelivery* of $m4$ different actions occur in each process:

- p_1: it transitions locally to the *Activated* state.
- p_2: it transitions locally to the *Activated* state.
- p_3: it stops the bell and transitions locally to the *Activated* state.

At t_4 the three processes are in the *Activated* state. Movement is detected again by a sensor and thus process p_2 *cBroadcast*s m_3 informing of the *Movement* transition. With respect to the *cDeliver*ies of message m_3, we have already shown above what are their effects in the different processes. In particular, p_3 starts ringing the bell. Approximately at t_4, in parallel the user disarms the alarm in the keypad and so p_1

*cBroadcast*s m_2 informing of the *Deactivate* transition. Upon *cDeliver*y of m_2 different actions occur in each process:

- p_1: it transitions locally to the *Deactivated* state.
- p_2: it transitions locally to the *Deactivated* state and turns off the movement sensors.
- p_3: it transitions locally to the *Deactivated* state, it cancels the timer and before the timeout *TO* is triggered it stops the bell.

In the space-time diagram we see that the *cDeliver*y order of these last two concurrent messages m_2 and m_3 is different depending on the process.

Note, although the effect of *cDeliver* of concurrent messages causes different sequences of transitions in the local state machines of different processes, all process state machines eventually converge to the same state. For example, in Fig. 4 we see that p_1 does not transition to the *Alarm* state after *cDeliver*ing m_3 because it *cDeliver*s m_2 before, and so it is in the *Deactivated* state when it *cDeliver*s m_3. But both, p_2 and p_3 *cDeliver* first m_3, entering the *Alarm* state before they *cDeliver* m_2. Eventually, the three processes reach the *Deactivated* state after they have *cDeliver*ed both messages.

7.1 States observed in the GSM using the CTS

The process labeled as Global States Monitor in the top of Fig. 4 shows the states observed by the *GSM*. The communication subsystem of the *GSM* causally *cDeliver*s the notifications of the *cBroadcast*s and *cDeliver* events that happen in p_1, p_2 and p_3. For example, in Fig. 4 the notification of *cBroadcast* of m_1 event in p_1 and its corresponding *cDeliver* events in all processes p_1, p_2 and p_3 are *cDeliver*ed in causal order in *GSM*. The same happens with all other messages.

The application part of the *GSM* computes the state transitions of the alarm system based on the notifications delivered. As a design decision (other valid design decisions are possible), *GSM* considers that a transition occurs when it has received notifications of a *cBroadcast* event of a given message and all of the *cDeliver* events of that message in p_1, p_2, p_3. Global states are shown with labels and colors corresponding with the names of the states in the state machine. Note that the *GSM* receives more information about the functioning of the domotic application than the processes themselves because the *GSM* receives notifications about the *cDeliver* events occurring in all processes, while processes p_1, p_2, p_3 only know about its own *cDeliver* events.

The sequence of transitions observed by the *GSM* is computed using the same state machine that is used by the rest of processes.

As we explained above, the three state machines of the application processes may transit through different sequences of states due to the order of concurrent *cBroadcast* and *cDeliver* events. Note that the state machine of *GSM* may also differ in the sequence of states it observes with respect to the sequences observed by the

application processes. The *GSM* and application processes eventually converge to the same state.

In the example, *GSM* computes first the notifications of the *cBroadcast* and *cDeliver* events of message m_2 sent by p_1 at t_4. Later it computes the notifications of the *cBroadcast* and *cDeliver* events of message m_3 sent by p_2. Thus the *GSM* observes the same state transitions that p_2 and p_3, which is different from the one observed at p_1.

7.2 Monitoring using Birman's causal broadcast

In order to assess *GSM* algorithm that uses *CTS*, we are going to compare it now with the monitoring system proposed by Babaoğlu et al. [24] that uses a *CRB* algorithm similar to Birman's.

In this monitoring system the processes of the domotic application are implemented using the Birman's *CRB* algorithm and its timestamps for *CRB* communication, instead of using *CTS* like in our *GSM*. This monitoring system now works as follows: every time an application process *cBroadcast*s a message *m*, it also sends the monitor for Birman's *CRB* a notification message with the vector timestamp of the *cBroadcast* of *m*. In every instant, the monitor for Birman's *CRB* can construct a consistent observation of the global state by causal ordering only *cBroadcast* timestamps and evaluate global predicates over this global state.

In the topmost of Fig. 4, labeled as monitor for Birman's *CRB*, we show the sequence of notifications of the *cBroadcast* events that this monitor receives from the domotic application, the causal order delivery of those notifications and finally, the application state transitions that the monitor for Birman's *CRB* process calculates.

7.3 Discussion

As we have seen above in previous Sect. 7.2 (monitoring using Birman's Broadcast), the monitoring system that uses Birman's *CRB* algorithm does not notify *cDeliver* events to the monitor. Thus, the monitor for Birman's *CRB* cannot know about the existence of a *cDeliver* of a message *m* that happened on a process *p* until it receives a notification of a ulterior *cBroadcast* event of a message m' in *p* that depends causally of *m*. Furthermore, the monitor will never know the program ordering of *cDeliver* events in *p* occurred between two consecutive *cBroadcast* events in *p*. Therefore, the monitor for Birman's *CRB* might not be able to reason, among other global state predicates, about the causal delivery predicate in every process.

On the other hand, our Global State Monitoring (*GSM*) that uses *CTS* has more information about the monitored system because it receives notifications of the *cDeliver* events, and not only of *cBroadcast* events. With these notifications, our *GSM* monitor makes more accurate global predicate evaluations over a run compared with the monitoring system with Birman's *CRB* timestamp. For example, our *GSM* evaluates systematically global predicates such as the properties of the *CRB* abstraction.

Hence, our *GSM* gives a more structured tool for solving the problem of the ad-hoc verification techniques described in Sect. 1.

Hence, when monitoring with Birman's *CRB* algorithm, the number of global states of the system and the number of global predicates that the monitor can evaluate are both less than the number of states and predicates that our *GSM* can evaluate. Also, the evaluation of predicates over chained global states to detect bad patterns can be made more accurate with our *GSM* than with Birman's *CRB* monitoring [23].

As an example, it can be seen in Fig. 4 that the *GSM* does not consider the alarm system is activated (transition to *SAC* state) until it has received the notifications of both the *cBroadcast* and all the *cDeliver* events of m_1. To get the same global state information with the monitoring system using Birman's *CRB* algorithm, the monitor would have to wait the reception of the *cBroadcast* notifications of messages that have m_1 as their causal predecessor from each application processes to ensure that all of them have *cDelivered* m_1.

Another example of use of our *GSM* that evaluates predicates online is the one that detects communication failures in the application, i.e., if it *cDeliver*s the notification of *cBroadcast* of m_1 but it does not *cDeliver* all the notifications of *cDeliver*s of m_1 in the application processes. The offline evaluation of predicates, also called *postmortem analysis*, is easier with our *GSM*, as it is explained in the next section.

7.4 Postmortem analysis of the domotic example

A postmortem analysis of events of an execution of the domotic application is useful among others for (1) the discovery of the possible causes of an unexpected incident detected by the application, (2) the discovery of common or unusual patterns of events in executions, and (3) detection of bugs in the implementation of the *CRB* abstraction used in the domotic application.

We show here that the use of our *CTS* simplifies the implementation of this kind of postmortem analysis process. The postmortem analysis is simpler than the global state monitoring because application processes can store locally the events with their timestamps, and the postmortem analysis process just needs to retrieve the whole set of events after the end of an execution. Previously to any analysis, the events must be causally sorted. To do this sorting, the postmortem analysis process just needs to apply the set of relations R_S of our *CTS*, defined in Sect. 5. With those *CTS* relations all *cBroadcast* and *cDeliver* events can be successfully causally ordered to get a consistent analysis. Then, the partial order can be extended to a causal total order by using the process unique identifiers of the events as in [2].

Note that with the timestamp system of the Birman's *CRB* algorithm it is possible to make the kind of postmortem analysis explained above but using only *cBroadcast* events. In this case, the analysis will be of worse resolution due to a lower number of events processed.

8 Related work

In this paper we have formally defined a causal timestamp system (*CTS*) based on logical vector clocks to implement the Causal Reliable Broadcast (*CRB*) abstraction with the maximal accuracy among *cBroadcast* and *cDeliver* events. The formalization of our *CTS* follows the timestamp framework defined by Torres et al. [14].

CRB algorithm by Birman et al. [7] is certainly the most well-known and widest used algorithm of the vector clock *CRB* algorithms [21, 22, 25, 29] that captures with the maximal accuracy the causal relation among *cBroadcast* events. However, Birman's *CRB* algorithm (and, as far as we know, all vector-based *CRB* algorithms) only timestamps globally *cBroadcast* but not *cDeliver* events. Therefore with Birman's *CRB* algorithm tracking causality among *cBroadcast* and *cDeliver* events with timestamps is not possible.

As it was explained in Sect. 1 there are two techniques to address the scalability problems of vector clocks. The first one compresses the causal information to be transmitted in different ways. The compression proposals in [7, 9–12] are not scalable, because in a group of P processes, the complexity of message overhead is $O(P)$, which it is not acceptable for a large value of P.

The proposal in [30] introduces Encoded Vector Clocks (EVCs) using prime numbers, where each event timestamp is represented by a single scalar number, characterizing causality regardless of the dynamic number of processes of the system. However, the drawback lies in the exponential storage growth and operational complexity required by EVCs. Conversely, in [13], Resettable Encoded Vector Clocks (REVCs) are built on EVCs to solve the problem of the exponential storage growth of EVCs. A REVC performs a clock reset operation whenever its value overflows a predefined number of bits. However, REVCs suffer from overheads in computation time and unbounded linear storage growth with respect of the number of messages. To address these challenges, REVCs can operate within an upper bound on storage requirements, which may entail sacrificing some degree of causal precision.

In this family, we also consider the clocks of a constant size M that are scalable with respect to the number of processes P, as $M < P$. In plausible clocks [14] each process maintains a plausible vector clock of size M where each vector clock entry might track events generated by several processes, so plausible clocks do not characterize causality. Bloom clocks [15] based on the Bloom filter [31] are used to probabilistically assess causality among events in a system. In [16], a Bloom clock is proposed for a system with P processes. Each process p_i maintains a Bloom clock B_i, which is a vector of M integers and k random hash functions. Each hash function maps to one of the m indices in B_i. Every time an event occurs in p_i, all k hash functions are applied to their corresponding positions in B_i, and then these corresponding positions are incremented. Bloom clocks exhibit the best performance among vector clocks of any size for determining causality. However, the drawback of these clocks is that the probability of detecting events as causally related when they are not related is greater than zero. In [32], recommendations for setting the Bloom clock parameters M and k to improve the precision of causal detection are provided.

The second technique to address scalability is based on network overlay topologies and FIFO channels [17–20], but these solutions have the problem of not characterizing causality.

Recently [12] has proposed a causal broadcast system that timestamps *cBroadcast* events in a similar way as we do in our *CST* system, although they do not explicitly identify and study a formal timestamp system as we have done with *CTS* in this paper.

In [4] a *CRB* communication system is proposed where the timestamps of *cBroadcast* message events are made available to the application layer so that it can order those timestamps to implement replicated databases. Other applications, like passive monitoring systems, could find beneficial to obtain not only the timestamps of *cBroadcast* events but also the *cDeliver* timestamp information provided by our *CTS* system.

Related with the global predicate evaluation over global states of a distributed system, in [24] it was proposed a monitoring system for applications that use *CRB* algorithms based on vector clocks. In a run, an external monitor passively records *cBroadcast* event timestamps generated by the application. Like the Global State Monitoring algorithm, [33, 34] also record global snapshots assuming that the underlying system supports causal message *cDelivery*, although unicast, not multicast. In both of them the monitor process must broadcast a token to every process including itself each time it wants to record a new global snapshot. In [30], it is also shown how to determine global states of distributed systems that use EVCs. In [13], a practical use of REVC for dynamic race detection is shown, with significant performance improvements over traditional logical clock implementations in tracking event causality. In [35], a kind of monitor algorithm is used to demonstrate impossibility and possibility results. This algorithm demonstrates that it is possible to detect the causal relation of events of an execution of a distributed application, where processes communicate with broadcast or unicast assuming a synchronous and byzantine model.

9 Conclusion

The proposed Causal Timestamp System (*CTS*) based on vector clocks provides an efficient and effective solution for timestamping *cBroadcast* and *cDeliver* events in the Causal Reliable Broadcast (*CRB*) abstraction. *CTS* simplifies the formal verification and testing of *CRB* implementations, allowing for improved accuracy and causality characterization.

The formal specification of *CTS* and the verification of its properties that we have done follow a well known formal timestamp framework. Furthermore, the proofs establish that *CTS* extends the maximal accuracy to *cDeliver* events while maintaining a comparable cost in terms of piggybacked information to the Birman's *CRB* algorithm.

We have developed a new passive Global State Monitoring (*GSM*) algorithm tailored to our *CRB* with *CTS* that enables a finer-grained assessment of consistent global states and predicates. This advancement is particularly beneficial for

distributed applications requiring global predicate evaluations, such as deadlock and termination detection, testing, and debugging.

The domotic example we presented showcases the practical application and comparative analysis of the *GSM* and *CRB* with *CTS*, offering insights into their functionalities and advantages over a passive monitor utilizing the Birman's *CRB* algorithm. The inclusion of *CTS* timestamping of *cDeliver* events further highlights the usefulness and potential of our *CRB* with *CTS* algorithm.

Overall, this paper presents a novel timestamp system *CTS* for implementing the *CRB* abstraction, demonstrates its properties and benefits, and shows a new passive monitoring algorithm *GSM*. These findings open up possibilities for improved formal verification, online and offline testing, and debugging techniques in distributed systems based on *CRB* algorithms that use vector clocks, with potential applications in various domains.

Acknowledgements We want to thank the reviewers for their comments on our paper. This research has been partially funded by the Community of Madrid, under grant EDGEDATA-CM (P2018/TCS-4499), the Spanish Research Council, under grant QoSData (PID2020-119461GB-I00) and by URJC, under grant OpenSSDWMN 5 G (URJC 2022/00004/017).

Author Contributions All the authors have contributed to all sections of this study.

Declarations

Conflict of interest The authors have no conflict of interest that may have affected the content of this work.

Ethics approval Not applicable.

References

1. Birman KP, Joseph TA (1987) Reliable communication in the presence of failures. ACM Trans Comput Syst 5(1):47–76. https://doi.org/10.1145/7351.7478
2. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565. https://doi.org/10.1145/359545.359563
3. Lloyd W, Freedman MJ, Kaminsky M, Andersen DG (2011) Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. SOSP '11, pp. 401–416. ACM, New York. https://doi.org/10.1145/2043556.2043593
4. Baquero C, Almeida PS, Shoker A (2017) Pure operation-based replicated data types. http://arxiv.org/abs/1710.04469
5. Fidge CJ (1988) Timestamps in message-passing systems that preserve the partial ordering. In: Proceedings 11th Australasian Computer Science Conference, vol. 10, pp. 56–66
6. Mattern F (1989) Virtual time and global states of distributed systems. In: Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms, vol. 1. Amsterdam, the Netherlands, pp. 215–226
7. Birman K, Schiper A, Stephenson P (1991) Lightweight causal and atomic group multicast. ACM Trans Comput Syst 9(3):272–314. https://doi.org/10.1145/128738.128742
8. Charron-Bost B (1991) Concerning the size of logical clocks in distributed systems. Inf Process Lett 39(1):11–16. https://doi.org/10.1016/0020-0190(91)90055-M
9. Singhal M, Kshemkalyani A (1992) An efficient implementation of vector clocks. Inf Process Lett 43(1):47–52. https://doi.org/10.1016/0020-0190(92)90028-T

10. Hadzilacos V, Toueg S (1994) A modular approach to fault-tolerant broadcasts and related problems. https://api.semanticscholar.org/CorpusID:13974342
11. Almeida PS, Baquero C, Fonte V (2008) Interval tree clocks. In: Baker, T.P., Bui, A., Tixeuil, S. (eds.) Principles of distributed systems, pp. 259–274. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-92221-6_18
12. Auvolat A, Frey D, Raynal M, Taiani F (2021) Byzantine-tolerant causal broadcast. Theor Comput Sci 885, 55–68. https://doi.org/10.1016/j.tcs.2021.06.021
13. Pozzetti T, Kshemkalyani AD (2020) Resettable encoded vector clock for causality analysis with an application to dynamic race detection. IEEE Trans Parallel Distrib Syst 32(4):772–785. https://doi.org/10.1109/TPDS.2020.3032293
14. Torres-Rojas FJ, Ahamad M (1999) Plausible clocks: constant size logical clocks for distributed systems. Distrib Comput 12(4):179–195
15. Ramabaja L (2019) The bloom clock. arXiv preprint arXiv:1905.13064
16. Kshemkalyani AD, Misra A (2021) The bloom clock to characterize causality in distributed systems. In: Advances in networked-based information systems: the 23rd international conference on network-based information systems (NBiS-2020) 23, pp. 269–279. https://doi.org/10.1007/978-3-030-57811-4_25 . Springer
17. Rodrigues L, Veríssimo P (1995) Causal separators for large-scale multicast communication. In: of 15th International Conference on Distributed Computing Systems (ICDCS), pp. 83–91. IEEE
18. Baldoni R, Friedman R, Renesse R (1997) The hierarchical daisy architecture for causal delivery. In: Proceedings of 17th International Conference on Distributed Computing Systems, pp. 570–577.https://doi.org/10.1109/ICDCS.1997.603422
19. Friedman R, Manor S (2004) Causal ordering in deterministic overlay networks. Technical report, Israel Institute of Technology, Haifa, Israel
20. Nédelec B, Molli P, Mostéfaoui A (2018) Breaking the scalability barrier of causal broadcast for large and dynamic systems. In: 2018 IEEE 37th symposium on reliable distributed systems (SRDS), pp. 51–60. https://doi.org/10.1109/SRDS.2018.00016
21. Attiya H, Welch J (2004) Distributed computing: fundamentals, simulations and advanced topics, 2nd Ed., pp. 175–177. Wiley, Inc., Hoboken
22. Raynal M (2013) Distributed algorithms for message-passing systems, sec.7.2, 7.3, 12.1 and 12.3. Springer, Berlin, Heidelberg
23. Bouajjani A, Enea C, Guerraoui R, Hamza J (2017) On verifying causal consistency. In: Proceedings of the 44th ACM SIGPLAN symposium on principles of programming languages. POPL '17, pp. 626–638. ACM, New York. https://doi.org/10.1145/3009837.3009888
24. Babaoğlu O, Marzullo K (1993) Consistent global states of distributed systems: fundamental concepts and mechanisms, pp. 55–96. ACM Press/Addison-Wesley Publishing Co.
25. Cachin C, Guerraoui R, Rodrigues L (2011) Introduction to reliable and secure distributed programming, 2nd Ed. Springer, Berlin, Heidelberg
26. Nieto A, Gondelman L, Reynaud A, Timany A, Birkedal L (2022) Modular verification of op-based CRDTs in separation logic. Proc ACM Program Lang. 6(OOPSLA2). https://doi.org/10.1145/3563351
27. Redmond P, Shen G, Vazou N, Kuper L (2023) Verified causal broadcast with liquid haskell. In: Proceedings of the 34th symposium on implementation and application of functional languages. IFL '22. ACM, New York. https://doi.org/10.1145/3587216.3587222
28. Cugola G, Margara A (2012) Processing flows of information: From data stream to complex event processing. ACM Compt Surv 44(3):1–62. https://doi.org/10.1145/2187671.2187677
29. Raynal M (2018) Fault-tolerant message-passing distributed systems: an algorithmic approach. Springer, Berlin
30. Kshemkalyani AD, Shen M, Voleti B (2020) Prime clock: encoded vector clock to characterize causality in distributed systems. J Parallel Distrib Comput 140, 37–51. https://doi.org/10.1016/J.JPDC.2020.02.008
31. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
32. Misra A, Kshemkalyani AD (2021) The bloom clock for causality testing. In: Distributed computing and internet technology: 17th international conference, ICDCIT 2021, Bhubaneswar, India, 2021, Proceedings 17, pp. 3–23. https://doi.org/10.1007/978-3-030-65621-8_1. Springer
33. Acharya A, Badrinath B (1992) Recording distributed snopshots based on causal order of message delivery. Inf Process Lett 44(6):317–321. https://doi.org/10.1016/0020-0190(92)90107-7

34. Alagar S, Venkatesan S (1994) An optimal algorithm for distributed snapshots with causal message ordering. Inf Process Lett 50(6):311–316. https://doi.org/10.1016/0020-0190(94)00055-7
35. Misra A, Kshemkalyani AD (2022) Detecting causality in the presence of byzantine processes: There is no holy grail. In: 2022 IEEE 21st international symposium on network computing and applications (NCA), vol. 21, pp. 73–80. https://doi.org/10.1109/NCA57778.2022.10013644