Information and Computation 301 (2024) 105212

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

Detecting causality in the presence of Byzantine processes: The case of synchronous systems [☆]

Anshuman Misra, Ajay D. Kshemkalyani [*]

Department of Computer Science, University of Illinois at Chicago, Chicago, 60607, IL, USA

ARTICLE INFO

Article history:
Received 13 November 2023
Received in revised form 15 June 2024
Accepted 8 August 2024
Available online 12 August 2024

Keywords:
Byzantine fault-tolerance
Causality
"Happened before" relation
Synchronous system
Distributed algorithm
Synchronization
Message-passing

ABSTRACT

Detecting causality or the "happens before" relation between events in a distributed system is a fundamental building block for distributed applications. It was recently proved that this problem cannot be solved in an asynchronous distributed system in the presence of Byzantine processes, irrespective of whether the communication mechanism is via unicasts, multicasts, or broadcasts. In light of this impossibility result, we turn attention to synchronous systems and examine the possibility of solving the causality detection problem in such systems. In this paper, we prove that causality detection between events can be solved in the presence of Byzantine processes in a synchronous distributed system. We prove the result by providing two algorithms. The first algorithm uses the Replicated State Machine (RSM) approach and vector clocks. The second algorithm is round-based and uses matrix clocks. The RSM-based algorithm can also run deterministically in partially synchronous systems.

1. Introduction

1.1. Background

Causality is an important tool in understanding and reasoning about distributed executions [2]. Lamport formulated the "happens before" or the causality relation, denoted $\rightarrow$, between events in a distributed system [3]. Given two events e and e', the *causality detection* problem asks to determine whether $e \rightarrow e'$. There are many applications of causality detection including determining consistent recovery points in distributed databases, deadlock detection, termination detection, distributed predicate detection, distributed debugging and monitoring, and the detection of race conditions and other synchronization errors [4].

The causality relation between events can be captured by tracking causality graphs [5], scalar clocks [3], vector clocks [6–8], matrix clocks [9,10], and several other variants of logical clocks such as hierarchical clocks [11], plausible clocks [12], dotted version vectors [13], interval tree clocks [14], logical physical clocks [15], Bloom clocks [16,17], incremental clocks [18], and resettable prime clocks [19,20]. Some of these variants track causality accurately while others introduce approximations and inaccuracies as trade-offs in the interest of savings on the space and/or time and/or message complexity overheads. As stated by Schwarz and Mattern [2], the search for the holy grail of the ideal causality tracking mechanism is

[☆] An earlier version of a part of this paper appeared in TIME 2023 [1].
[*] Corresponding author.
E-mail address: ajay@uic.edu (A.D. Kshemkalyani).

https://doi.org/10.1016/j.ic.2024.105212
0890-5401/© 2024 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

on. These above works in the literature assume that processes are correct (non-faulty). It is important to solve this problem under the Byzantine failure model as opposed to a failure-free setting because it mirrors the real world. The causality detection problem for a system with Byzantine processes was recently introduced and studied in [21].

The related problem of causal ordering of messages asks that if the send event of message m happens before the send event of message m', then require that m' should not be delivered before m at all the common destinations of m and m' [22]. Under the Byzantine failure model, causal ordering has recently, and concurrently with [21], been studied in [23–26]. The framework is similar to that in [21] as causality detection is implicitly involved.

In the first formulation and study of causality detection under the Byzantine failure model, it was proved that the problem of detecting causality between a pair of events cannot be solved in an asynchronous system in the presence of Byzantine processes, irrespective of whether the communication is via unicasts, multicasts, or broadcasts [21]. In the multicast mode of communication, each send event sends a message to a group consisting of a subset of the set of processes in the system. Different send events can send to different subsets of processes. Communicating by unicasts and communicating by broadcasts are special cases of multicasting. It was shown in [21] that in asynchronous systems with even a single Byzantine process, the unicast and multicast modes of communication are susceptible to false positives and false negatives, whereas the broadcast mode of communication is susceptible to false negatives but no false positives. A false positive means that $e \not\rightarrow e'$ whereas $e \rightarrow e'$ is perceived/detected. A false negative means than $e \rightarrow e'$ whereas $e \not\rightarrow e'$ is perceived/detected.

1.2. Contributions

In light of the impossibility result for asynchronous systems, this paper examines the solvability of causality detection in synchronous systems in the presence of Byzantine processes. We prove that causality detection between events can be solved in the presence of Byzantine processes in a synchronous system by providing two algorithms. Our positive result holds for unicasts, multicasts, as well as broadcasts. This is the first paper to establish this result. The result is significant, similar to the results in [2,6,7], because it establishes a fundamental possibility result about causality detection in the presence of Byzantine processes in a synchronous system. The results for our two algorithms for multicasts, unicasts, and broadcasts are summarized in Table 1. The algorithms are presented for the multicast mode of communication – unicast and broadcast modes are special cases of multicast.

The first algorithm (Algorithm 1) uses the Replicated State Machine (RSM) approach [27], which works in synchronous systems, in conjunction with vector clocks [6,7]. In a system with n application processes, our RSM-based solution uses $3t+1$ process replicas per application process, where t is the maximum number of Byzantine processes that can be tolerated in a RSM. Thus, there can be at most nt Byzantine processes among a total of $(3t+1)n$ processes partitioned into n RSMs of $3t+1$ processes each, with each RSM having up to t Byzantine processes. By using $(3t+1)n$ processes and the RSM approach to represent n application processes, the malicious effects of Byzantine process behaviors are neutralized. There are neither false positives nor false negatives. Efficient implementations of RSMs also require the use of cryptography [28]. A RSM-to-RSM unicast message requires $(3t+1)^2$ messages between replicas whereas a message multicast by a RSM requires $n(3t+1)^2$ messages between replicas. Additionally, implementing an RSM typically requires worst-case $O((t+1)^2)$ messages and worst-case $O(t)$ time per message received from another RSM. Thus a message multicast costs overall $O(nt^2)$ messages and $O(t)$ time.

The second algorithm (Algorithm 2) provides an alternate solution for solving the causality detection problem that does not use expensive process replication. This algorithm uses threshold encryption in conjunction with matrix clocks [10] in a round-based synchronous system. For multicast communication in the presence of Byzantine processes, we use the primitives for Byzantine-tolerant Reliable Multicast (BRM) which under the covers invokes Byzantine-tolerant Reliable Broadcast (BRB) [29,30]. Further, the upper bound f on the number of Byzantine processes in the system needs to satisfy only $n > f+1$ as we use Dolev-Strong authenticated agreement [31] for BRB, and thus the algorithm has optimal fault-tolerance. Implementing a message multicast requires nf messages and $f+1$ rounds for the Dolev-Strong algorithm underlying BRB whereas the overall algorithm requires $O(n^2)$ messages and $f+2$ rounds.

The round-based Algorithm 2 also makes the assumption that processes will not perform any *out-of-band communication* (OOBC), which is communication not specified by the algorithm. This assumption prevents the establishment of the causality relation between a pair of events, that cannot be captured by this algorithm. Neither false positives nor false negatives can then arise. Two types of false negatives could potentially arise.

1. False negatives due to the non-observation of causal chains when a Byzantine process masks them by swapping the order of a local receive event followed by a local send event when it reports the local execution history. The algorithm prevents such false negatives by using threshold cryptography in conjunction with the BRB primitive.
2. False negatives due to the direct or *out-of-band* message-passing among the Byzantine processes causing causality chains that need to be accounted for but cannot be captured/observed by correct processes. The algorithm prevents such false negatives by making the *“No OOBC”* assumption. Without the *no OOBC* assumption, such false negatives can arise because OOBC bypasses BRB to establish causal dependencies; the BRB used by the algorithm is incapable of detecting such causal relationships. If OOBC is performed, OOBC events do count in the actual execution because they are real (a send

Table 1
Solvability of causality detection between events under different communication modes in asynchronous and synchronous systems. FP is false positive, FN is false negative. $\overline{FP}/\overline{FN}$ means no false positive/no false negative is possible. OOBC = *out-of-band communication.*

Communication mode	Asynchronous	Synchronous, Algorithm 1 (using RSM), [1]	Synchronous, Algorithm 2 (using BRB), No OOBC	Synchronous, Algorithm 2 (using BRB)
Multicasts	No [21] FP, FN	Yes [1], Theorem 2 $\overline{FP}, \overline{FN}$	Yes, Corollary 3 $\overline{FP}, \overline{FN}$	No, Theorem 6 $\overline{FP}, FN$
Unicasts	No [21] FP, FN	Yes [1], Corollary 1 $\overline{FP}, \overline{FN}$	Yes, Corollary 4 $\overline{FP}, \overline{FN}$	No, Corollary 6 $\overline{FP}, FN$
Broadcasts	No [21] $\overline{FP}, FN$	Yes [1], Corollary 2 $\overline{FP}, \overline{FN}$	Yes, Corollary 5 $\overline{FP}, \overline{FN}$	No, Corollary 7 $\overline{FP}, FN$

and receive did occur) and they establish causality relations which need to be correctly cognized in the algorithm's view of the actual execution.

The algorithm ensures that there are no false positives by leveraging the Byzantine Reliable Broadcast (BRB) primitive for each multicast in conjunction with the use of matrix clocks. A multicast is sent via BRB to ensure that all correct processes in the system see the same state of the system which is also the actual state of the system (subject to the *no OOBC* assumption). Even with OOBC, the algorithm cannot be misled into detecting false positives because OOBC does not remove any actual causal relations.

Algorithm 1 is immune to OOBC because each RSM uses $3t+1$ replicas and whatever OOBC is performed by the t Byzantine replicas is masked. For any event to get registered in a RSM, there should be at least $t+1$ identical reports of that event; as there are only t Byzantine replicas, their reports (based on OOBC) can never meet that quorum of $t+1$ and the corresponding OOBC events do not take place in the RSMs. The same logic explains why this RSM-based algorithm is immune to fake events reported by and actual events not reported by Byzantine replicas.

Table 1 summarizes the results and places them in perspective. An earlier version of the first algorithm appeared in [1].

Roadmap. Section 2 gives the system model. Section 3 formulates the problem of detecting causality in the presence of Byzantine processes. Section 4 proposes the RSM-based algorithm and proves its correctness. Section 5 proposes the round-based algorithm and proves its correctness. Section 6 gives a discussion and concludes.

2. System model

This paper deals with a distributed system having Byzantine processes which are processes that can misbehave [32,33]. A correct process behaves exactly as specified by the algorithm whereas a Byzantine process may exhibit arbitrary behavior including crashing at any point during the execution. A Byzantine process cannot impersonate another process or spawn new processes. If processes do not perform *out-of-band communication*, which is communication not specified by the application program, the system is said to satisfy the *no OOBC* assumption.

The distributed system is modelled as an undirected graph $G=(P,C)$. Here P is the set of processes communicating in the distributed system. Let $|P|=n$. C is the set of (logical) communication links over which processes communicate by message passing. The channels are assumed to be FIFO. G is a complete graph.

The distributed system is assumed to be *synchronous*, i.e., there is a known fixed upper bound δ on the message latency, and a known fixed upper bound ψ on the relative speeds of processors [34]. In contrast, an *asynchronous* system has been defined as one in which there is no upper bound on the message latency and on the relative speeds of processors [34]. A synchronous system guarantees that the relative speeds of non-faulty processors and message latencies are bounded, and this is equivalent to assuming that the system has synchronized real-time clocks [27]. A *partially synchronous* system behaves like an asynchronous system but with periods of synchrony [34].

Let e_i^x, where $x \geq 1$, denote the x-th event executed by process p_i. An event may be an internal event, a message send event, or a message receive event. Let S_i, R_i, and I_i denote the set of send events, receive events, and internal events at process p_i. Let the state of p_i after e_i^x be denoted s_i^x, where $x \geq 1$, and let s_i^0 be the initial state. The execution at p_i is the sequence of alternating events and resulting states, as $\langle s_i^0, e_i^1, s_i^1, e_i^2, s_i^2 \ldots \rangle$. The sequence of events $\langle e_i^1, e_i^2, \ldots \rangle$ is called the execution history at p_i and denoted E_i. Let $E = \bigcup_i \{E_i\}$ and let $T(E)$ denote the set of all events in (the set of sequences) E. The *happens before* [3] relation, denoted $\rightarrow$, is an irreflexive, asymmetric, and transitive partial order defined over events in a distributed execution that is used to define causality.

Definition 1. The happens before relation $\rightarrow$ on events $T(E)$ consists of the following rules:

1. **Program Order**: For the sequence of events $\langle e_i^1, e_i^2, \ldots \rangle$ executed by process p_i, $\forall$ x, y such that $x<y$ we have $e_i^x \rightarrow e_i^y$.

2. **Message Order**: If event e_i^x is a message send event executed at process p_i and e_j^y is the corresponding message receive event at process p_j, then $e_i^x \rightarrow e_j^y$.
3. **Transitive Order**: If $e \rightarrow e' \wedge e' \rightarrow e''$ then $e \rightarrow e''$.

Definition 2. The *causal past* of an event e is denoted as $CP(e)$ and defined as the set of events $\{e' \in T(E) \,|\, e' \rightarrow e\}$.

The round-based algorithm explicitly assumes that the execution in the synchronous system proceeds in rounds, where messages sent in a round are received in the same round. The local sequence of events in a round contains send events, followed by receive events, followed by internal events.

3. Problem formulation

The problem formulation is done similar to the way in [21]. An algorithm to solve the causality detection problem collects the execution history of each process in the system and derives causal relations from it. E_i is the *actual* execution history at p_i. For any causality detection algorithm, let F_i be the execution history at p_i as perceived and collected by the algorithm (at a process, specified by the context) and let $F = \bigcup_i \{F_i\}$. F thus denotes the execution history of the system as perceived and collected by the algorithm. F_i and F can be thought of as local variables. Analogous to $T(E)$, let $T(F)$ denote the set of all events in F. Analogous to Definition 1, the *happens before* relation can be defined on $T(F)$ instead of on $T(E)$. With a slight relaxation of notation, let $T(E_i)$ and $T(F_i)$ denote the set of all events in E_i and F_i, respectively.

Let $e1 \rightarrow e2|_E$ and $e1 \rightarrow e2|_F$ be the evaluation (1 or 0) of $e1 \rightarrow e2$ using E and F, respectively. Byzantine processes may corrupt the collection of F to make it different from E as follows.

1. To delete e_h^x from F_h or in general, record F as any alteration of E such that $e_h^x \rightarrow e_i^*|_F = 0$, while $e_h^x \rightarrow e_i^*|_E = 1$, or
2. To add a fake event e_h^x in F_h or in general, record F as any alteration of E such that $e_h^x \rightarrow e_i^*|_F = 1$, while $e_h^x \rightarrow e_i^*|_E = 0$.

Without loss of generality, we have that $e_h^x \in T(E) \cup T(F)$. Note that e_h^x belongs to $T(F) \setminus T(E)$ when it is a fake event in F.

We assume that a correct process p_i needs to detect whether $e_h^x \rightarrow e_i^*$ holds and e_i^* is an event in $T(E)$. If $e_h^x \notin T(E)$ then $e_h^x \rightarrow e_i^*|_E$ evaluates to *false*. If $e_h^x \notin T(F)$ (or $e_i^* \notin T(F)$) then $e_h^x \rightarrow e_i^*|_F$ evaluates to *false*. We assume an oracle that is used for determining correctness of the causality detection algorithm; this oracle has access to E which can be any execution history such that $T(E) \supseteq CP(e_i^*)$.

Definition 3. The causality detection problem $CD(E, F, e_i^*)$ for any event $e_i^* \in T(E)$ at a correct process p_i is to devise an algorithm to collect the execution history E as F at p_i such that $valid(F) = 1$, where

$$valid(F) = \begin{cases} 1 & \text{if } \forall e_h^x, e_h^x \rightarrow e_i^*|_E = e_h^x \rightarrow e_i^*|_F \\ 0 & \text{otherwise} \end{cases}$$

When 1 is returned, the algorithm output matches the actual (God's) truth and solves *CD* correctly. Thus, returning 1 indicates that the problem has been solved correctly by the algorithm using F. 0 is returned if either

- $\exists e_h^x$ such that $e_h^x \rightarrow e_i^*|_E = 1 \wedge e_h^x \rightarrow e_i^*|_F = 0$ (denoting a false negative), or
- $\exists e_h^x$ such that $e_h^x \rightarrow e_i^*|_E = 0 \wedge e_h^x \rightarrow e_i^*|_F = 1$ (denoting a false positive).

Byzantine processes are an integral part of the system. The occurrence of an event at such a process, and its correct order with respect to other events locally, matters to correct processes because it can impact the causality relation among events at correct processes. Let p_{c1} and p_{c2} be correct processes and let p_b be a Byzantine process. Let message $m1$ sent at e_s_{c1} be received at e_r_b. Let message $m2$ sent at e_s_b be received at e_r_{c2}. Consider the following two scenarios.

1. In E, we have $e\_s_{c1} \rightarrow e\_r_b \rightarrow e\_s_b \rightarrow e\_r_{c2}$. If F at the correct processes does not match this (specifically, $e\_r_b \not\rightarrow e\_s_b$ due to p_b lying), a causality detection algorithm fails to recognize $e\_s_{c1} \rightarrow e\_r_{c2}$, resulting in a false negative.
2. In E, we have $e\_s_{c1} \rightarrow e\_r_b$, $e\_s_b \rightarrow e\_r_{c2}$, and $e\_s_b \rightarrow e\_r_b$. If F at the correct processes does not match this and reflects $e\_r_b \rightarrow e\_s_b$ (due to p_b lying), a causality detection algorithm wrongly detects $e\_s_{c1} \rightarrow e\_r_{c2}$, resulting in a false positive.

Therefore it not sufficient for the correct processes to agree mutually on a F that differs from E in what happened in E at the Byzantine processes; their F_j must also agree with E_j at all processes p_j.

For the algorithm using the state-machine replication approach (Algorithm 1), we show that F at a correct process can be made to exactly match E, hence there is no possibility of a false positive or of a false negative.

For the round-based algorithm (Algorithm 2), we solve a weaker version of CD where $e_h^x \in S_h \cup R_h$. This is because a Byzantine process p_h can always lie about $e_h^x \in I_h$ to others for the collection of F_h as e_h^x is completely local to that p_h. Thus we do not consider internal events in the CD problem solution. With this caveat and the "no-OOBC" assumption, we show that the set of events in the recorded F at any process exactly matches the set of events in E up to the same round in the synchronous execution.

4. Solution based on replicated state machines (RSMs)

4.1. Background on RSMs

The discussion in this section is based on the survey by Schneider [27]. A process execution is modelled as the actions of a finite state machine. Two basic requirements are: (O1: FIFO order) Messages issued by a client to a state machine are processed in the order issued, and (O2: Causal order) If a message $m1$ issued to a state machine *sm* by client c could have caused (i.e., causally preceded) a message $m2$ issued by client c' to *sm*, then *sm* processes $m1$ before $m2$.

A t-tolerant version of a state machine is implemented by replicating that state machine and running a *state machine replica smr* on different processors in an *ensemble*. If each replica run by a correct processor starts in the same initial state and executes the same requests in the same order, then each replica will execute the same step at each transition and produce the same output. Under Byzantine failures, an ensemble implementing a t tolerant RSM must have at least $3t+1$ replicas and the output of each (correct) replica in the ensemble is the output produced by $t+1$ replicas. To ensure that all replicas' actions and transitions are coordinated, all replicas in an ensemble must receive and process the same sequence of messages. This can be expressed as two requirements.

- Agreement: Every non-faulty replica receives every message.
- Total order: Every non-faulty replica processes the messages it receives in the same order.

Agreement requires that (IC1) for each message sent by a replica, all non-faulty replicas of the destination process agree on the contents of the message, and (IC2) if the transmitting replica is non-faulty, then all non-faulty replicas of the destination process use the transmitter's value as the one on which they agree. Any of the Byzantine agreement protocols in the literature can be used [33,32]; they all require that the total number of replicas (of the destination process) is at least $3t+1$. Furthermore, no deterministic algorithm can implement state machine replication, which requires agreement or consensus, in an asynchronous system [35]. So we assume a synchronous system.

Total order can be satisfied by assigning unique identifiers to messages sent and having the receiver's *smr*s process the messages as per a total order relation on these unique identifiers. For the RSM of application process p_j, its various $3t+1$ *smr*s are denoted $p_{j,w}$. A message is defined to be *stable* at $p_{j,w}$ once no message from a correct sender process replica (across all sender processes from various sender process ensembles) having a lower unique identifier can be subsequently delivered to $p_{j,w}$. Total order is implemented by requiring a replica process to next process the stable request with the smallest stable identifier. Mechanisms for generating unique identifiers satisfying FIFO and causal order are given by Schneider [27]. These mechanisms are based on synchronized real-time clocks (which guarantees O1 and causal order O2 implicitly), or based on receiver replica-generated unique identifiers; the latter approach also requires for maintaining FIFO order and causal order (O1 and O2) that once a transmitter replica starts disseminating a message, it performs no other communication until the current message has been delivered to every receiver replica that is a destination of the current message. In a system with Byzantine processes, the replica-generated unique identifiers approach along with using the assumptions on synchronized real-time clocks can satisfy the total order. But note here that the requirement of synchronized real-time clocks forces us to assume a synchronous system.

4.2. Adapting RSMs to our solution

In our system model having n application processes, each process p_i modelled as a RSM is replicated $3t+1$-way as $p_{i,1}, \ldots, p_{i,3t+1}$ and these processes form the ensemble p_i. Various RSM ensembles communicate in a peer-to-peer (P2P) manner with each other. When a RSM ensemble sends/receives a message, it is referred to as a sender/receiver RSM ensemble. Thus in a system having n application processes, there are $(3t+1)n$ processes (i.e., replicas) partitioned into n RSM ensembles and each ensemble can have at most t Byzantine processes. Each $p_{i,a}$ uses a sequence number denoted $seq_{i,a}$ that is incremented for each message that it sends/multicasts as a sender RSM replica. The $(3t+1)n$ processes can be viewed as running in an application layer that is above the RSM layer which provides Agreement and Total Order.

Using the implementation of RSMs described by Schneider or any of the subsequent implementations proposed since then, Agreement and Total Order are guaranteed. Furthermore, Total Order is guaranteed in a receiver RSM ensemble for messages from multiple sender RSM ensembles. In addition, when each replica in the sender RSM ensemble does a multicast, the following version of the Agreement property needs to be implemented.

- Agreement$-M$: Every non-faulty replica in every RSM ensemble that is included in the destination set of a multicast/broadcast receives the message multicast/broadcast.

Agreement-M requires that (IC1-M) for each message sent by a replica, all non-faulty replicas of the destination processes of a multicast/broadcast agree on the contents of the message, and (IC2-M) if the transmitting replica is non-faulty, then all non-faulty replicas of the destination processes of a multicast/broadcast use the transmitter's value as the one on which they agree.

When a RSM replica receives a message from the RSM layer satisfying Total Order and Agreement/Agreement-M, we say that the message is *TOA-delivered* to that RSM replica. Under Byzantine failures, an ensemble implementing a t tolerant RSM in a system model disallowing cryptography must have at least $3t+1$ replicas and the output of each (correct) replica in an ensemble is the output produced by a *majority* = $t+1$ replicas. Henceforth, we treat *majority* as having the value $t+1$. Since we are using RSMs for "clients" and "servers" in P2P mode, whenever a correct receiver replica is *TOA-delivered* (gets) $t+1$ identical messages M from the replicas of a sender ensemble, the (correct) receiver replica delivers the message to the layer above. We say that a message M is *SR-delivered* to a RSM replica if *majority* $= t+1$ identical copies of the message having the same $seq_{j,*}$ from the replicas of a sender ensemble j have been *TOA-delivered* to it. On *SR-delivery* of a message to a RSM replica, that replica makes the next transition according to the local state machine. The Agreement and Total Order properties guarantee that if $p_{i,a}$ *SR-delivers* such a message, then every other correct receiver replica $p_{i,y}$ in that ensemble will also *SR-deliver* that same message M in exactly the order and sequence it was *SR-delivered* by $p_{i,a}$. Note that there are at least $t+1$ votes for this message M from the sender replica ensemble and since there are at most t Byzantine processes in the sender replica ensemble, their state machines can send only up to t messages (for any particular sequence number $seq_{j,*}$ from the sender ensemble j) that are received by $p_{i,a}$ and that differ from the majority value of M received $t+1$ times by $p_{i,a}$.

When $p_{i,a}$ sends a message to p_j at the application level, it sends it to all replicas $p_{j,b}$. When $p_{i,a}$ *SR-delivers* a message, a receive event is said to have occurred at the application level. Henceforth, we also refer to RSM i as p_i.

4.3. Data structures and algorithm

Algorithm 1 is an online algorithm in which each correct replica $p_{i,a}$ records in $F = \bigcup_k \{F_k\}$ its view of the execution history of RSM p_k via lines 1-21. This recording of F in the local replica is done by piggybacking control information on the application messages; no extra messages are used. There is also a module in Algorithm 1 lines 22-26 that takes as input two events e_h^x and e_i^* and produces output from $\{true, false\}$ giving $e_h^x \rightarrow e_i^*|_F$. Theorem 1 shows that the set of events in E matches the set of events recorded in F, even though E is never recorded and is accessible only to an oracle. Next we show in Theorem 2 that using the output of the algorithm lines 22-26 function test, and Theorem 1, the causality detection problem is solved by Algorithm 1's recording of F and function test using this F, i.e., there are no false positives nor false negatives.

Algorithm 1 gives the processing of control information done at a RSM replica $p_{i,a}$. Each RSM replica maintains the following data structures.

1. An integer $seq_{i,a}$, initialized to 0, that gives the sequence number of the latest local event at $p_{i,a}$.
2. A local F that is a set of sequences F_k. F contains $p_{i,a}$'s view of the recorded execution history F_k of each RSM p_k.
3. An integer matrix $LASKALSJ[n,n]$, where $LASKALSJ[j,k]$ gives the sequence number of the <u>la</u>test <u>s</u>end event by p_k (as per/from the local F_k) <u>a</u>t the point in time of the <u>l</u>ast <u>s</u>end event to $p_{j,*}$.
   This data structure is for efficiently identifying to send to p_j only the *incremental updates* that have occurred to the local F_k at $p_{i,a}$ for each other process p_k, that need to be transmitted to the destinations p_j of a message send event since $p_{i,a}$'s last send to p_j. This matrix at $p_{i,a}$ is like a matrix clock but *without* the usual semantics that the $[j,k]$-th entry gives $p_{i,a}$'s knowledge of the latest scalar time at p_k as known to p_j.
4. $p_{i,a}$ also maintains an auxiliary integer matrix $V[|T(F_i)|, n]$, where $V[s,k]$ is $maxeventID(F_k)$ in $F(e_{i,a}^s)$, i.e., the highest sequence number in $F_k (\in F)$ when the sth local event $e_{i,a}^s$ was executed at $p_{i,a}$.

Lines 1-6 give the processing for sending a unicast. If multicast can be implemented as a set of independent unicasts, similar code (but with a single increment in line 2) can be executed for sending to each destination of the multicast group. Otherwise a multicast send processing can be implemented via lines 7-12. When a message along with the incremental update *inc_F* (containing the incremental updates for all p_k as per the sender) is *SR-delivered* to a RSM replica, it updates its F_k as shown in lines 13-18. A broadcast is a special case of multicast and is hence handled as a multicast. The test for the happens before relation using V is given in lines 22-26.

In the auxiliary matrix V at $p_{i,a}$, row $V[w]$ is the vector timestamp [6,7] of event $e_{i,a}^w$ and could be stored along with the event in F_i. $V[w,j]$ at $p_{i,a}$ identifies (gives the sequence number of) the event at the surface of the causal past cone of event $e_{i,a}^w$ at RSM p_j. At event $seq_{i,a}$ for each type of event (unicast send (line 3), multicast send (line 9), delivery (line 18), internal (line 21)), $V[seq_{i,a}, k]$ for all k is set to $maxeventID(F_k)$. V is used only to implement the test $e_h^x \rightarrow e_i^*$, viz., $V[*,h] \geq x$.

4.4. Correctness proof

Events such as e_h^x with a single subscript which denotes the application-level process ID of p_h, are at the application level or RSM-ensemble level. Events such as $e_{h,a}^x$ with two subscripts denote events at $p_{h,a}$, the individual state machine

Algorithm 1: Processing of control information and testing for $e_h^x \rightarrow e_i^*$. Code at process $p_{i,a}$.

```
Data: Each process $p_{i,a}$ maintains (i) an integer $seq_{i,a}$, (ii) $F$ which is the union of sequences $F_k$ (history of events at
      $p_k$) for all $k$, (iii) integer matrix $LASKALSJ[n,n]$, (iv) integer matrix $V[|T(F_i)|,n]$.
Input: $e_h^x, e_i^*$
Output: $e_h^x \rightarrow e_i^*|_F \in \{true, false\}$

1  when $p_{i,a}$ needs to send application message $M$ to $p_{j,*}$: ▷ Each other correct $p_{i,a'}$ state machine will
      execute likewise
2  $seq_{i,a} = seq_{i,a} + 1$
3  append current send event to $F_i$; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$
4  $(\forall k)$ include history from $F_k$ after event $LASKALSJ[j,k]$ in $inc\_F$
5  $(\forall k)$ $LASKALSJ[j,k] = maxeventID(F_k)$
6  send $(M, inc\_F, seq_{i,a}, j)$ to each $p_{j,*}$ via RSM layer (to satisfy RSM Total Order and Agreement for receiver ensemble
      $p_j$)

7  when $p_{i,a}$ needs to send application message $M$ to each $p_{j,*}$ for each $p_j \in G$: ▷ Each other correct $p_{i,a'}$
      state machine will execute likewise
8  $seq_{i,a} = seq_{i,a} + 1$
9  append current send event to $F_i$; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$
10 $(\forall k)$ include history from $F_k$ after event $\min_{p_j \in G}(LASKALSJ[j,k])$ in $inc\_F$
11 $(\forall p_j \in G)(\forall k)$ $LASKALSJ[j,k] = maxeventID(F_k)$
12 send $(M, inc\_F, seq_{i,a}, G)$ to each $p_{j,*}$ for each $p_j \in G$ via RSM layer (to satisfy RSM Total Order and Agreement$-M$
      for each receiver ensemble $p_j$)

13 when $(M, inc\_F, seq_j, i/G)$ is SR-delivered to $p_{i,a}$ from $p_j$: ▷ Happens when $t+1$ identical copies of
      $(M, inc\_F, seq_j, i/G)$ for $seq_j$ (which equals $seq_{j,*}$) are TOA-delivered from $p_{j,*}$
14 for all k do
15     if $maxeventID(F_k) < maxeventID(inc\_F_k)$ then
16         append history of events $\langle maxeventID(F_k)+1, \ldots, maxeventID(inc\_F_k)\rangle$ from $inc\_F_k$ to $F_k$

17 $seq_{i,a} = seq_{i,a} + 1$
18 append current receive event to $F_i$; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$

19 At internal event at $p_{i,a}$:
20 $seq_{i,a} = seq_{i,a} + 1$
21 append current internal event to $F_i$; $(\forall k) V[seq_{i,a}, k] = maxeventID(F_k)$

22 To determine $e_h^x \rightarrow e_i^*$ at correct state machine $p_{i,a}$ via call to test($e_h^x \rightarrow e_i^*$):
23 if $e_h^x$ is in $F_h$ and $* \leq maxeventID(F_i)$ then
24     return($e_h^x \rightarrow e_i^*|_F$) ▷ the test is whether $V[*,h] \geq x$
25 else
26     return(false)
```

sm of replica a of RSM p_h in its RSM ensemble. Next, we adapt the definitions of E, of the happens before relation, and of causal past to abstract away the RSM details.

Definition 4. Define E_RSM to be the set of all events $\{e_h^x\}$ such that the events $e_{h,a}^x$ have occurred at at least *majority* (= $t+1$) number of processes $p_{h,a}$.

Definition 5. The happens before relation $\rightarrow_{RSM}$ on events in E_RSM (which occur in ensembles of RSMs) consists of the following rules:

1. **Program Order**: For the sequence of events $\langle e_i^1, e_i^2, \ldots\rangle$ executed by RSM ensemble process p_i, $\forall$ x, y such that $x < y$ we have $e_i^x \rightarrow_{RSM} e_i^y$.
2. **Message Order**: If event e_j^y is a message receive event executed at RSM ensemble process p_j (i.e., at at least a *majority* of processes $p_{j,b}$) and there is a corresponding RSM send event e_i^x in RSM ensemble p_i (i.e., there are at least a *majority* events $e_{i,a}^x$ that are the corresponding message send events at processes $p_{i,a}$ to RSM ensemble p_j), we have $e_i^x \rightarrow_{RSM} e_j^y$.
3. **Transitive Order**: If $e \rightarrow_{RSM} e' \wedge e' \rightarrow_{RSM} e''$ then $e \rightarrow_{RSM} e''$.

Definition 6. The *RSM-causal past* of an event $e \in E\_RSM$ is denoted as $CP\_RSM(e)$ and defined as the set of events $\{e' \in E\_RSM \,|\, e' \rightarrow_{RSM} e\}$.

In the causality graph $(E\_RSM, \rightarrow_{RSM})$, there is a RSM-causal path from any event in $CP\_RSM(e)$ to e comprised of program order edges and message order edges.

The following two lemmas and Theorem 1 are now obvious; the reader is referred to [1] for the proofs if needed.

Lemma 1. *An event $e_h^x \in E\_RSM$ occurs at each correct process $p_{h,z}$ in the RSM ensemble p_h.*

Lemma 2. *An event $e_{h,z}^x$ that occurs at a correct process $p_{h,z}$ also occurs as event e_h^x in the RSM ensemble p_h.*

Theorem 1. *For an event e at a RSM p_i (e must occur at each correct process $p_{i,z}$ by Lemma 1), the set of events $T(F)$ when e is executed at each correct $p_{i,z}$ is $CP\_RSM(e)$. Thus,*

1. *If an event belongs to $CP\_RSM(e)$, the event must belong to $T(F)$ when event e is executed at correct process $p_{i,z}$.*
2. *If an event e' belongs to $T(F)$ when event e is executed at correct process $p_{i,z}$ (the event e must also occur at RSM ensemble process p_i by Lemma 2), the event e' must belong to $CP\_RSM(e)$.*

Next we adapt the definition of the *CD* problem to deal with the RSM approach. We assume an oracle that is used for determining correctness of the causality detection algorithm at $p_{i,z}^*$; this oracle has access to E_RSM which can be any downward-closed superset of $CP\_RSM(e_i^*)$. Also let $F(e_{i,z}^*)$ be the value of F at $p_{i,z}$ when $e_{i,z}^*$ is executed.

Definition 7. The causality detection problem $CD(E\_RSM, F(e_{i,z}^*), e_{i,z}^*)$ for any event $e_{i,z}^*$ at a correct process $p_{i,z}$ (where $e_i^* \in E\_RSM$) is to devise an algorithm to collect the execution history of events E_RSM as $F(e_{i,z}^*)$ at $p_{i,z}$ such that $valid(F) = 1$, where

$$valid(F) = \begin{cases} 1 & \text{if } \forall e_h^x, e_h^x \rightarrow e_{i,z}^*|_{E\_RSM} = e_h^x \rightarrow e_{i,z}^*|_F \\ 0 & \text{otherwise} \end{cases}$$

When 1 is returned, the algorithm output matches God's truth and solves *CD* correctly. Thus, returning 1 indicates that the problem has been solved correctly by the algorithm using F. 0 is returned if either

- $\exists e_h^x$ such that $e_h^x \rightarrow e_{i,z}^*|_{E\_RSM} = 1 \wedge e_h^x \rightarrow e_{i,z}^*|_F = 0$ (denoting a false negative and $(E\_RSM \cap CP\_RSM(e_{i,z}^*)) \setminus T(F(e_{i,z}^*)) \neq \emptyset$), or
- $\exists e_h^x$ such that $e_h^x \rightarrow e_{i,z}^*|_{E\_RSM} = 0 \wedge e_h^x \rightarrow e_{i,z}^*|_F = 1$ (denoting a false positive and $T(F(e_{i,z}^*)) \setminus (E\_RSM \cap CP\_RSM(e_{i,z}^*)) \neq \emptyset$).

Algorithm 1 produces the output of $e_h^x \rightarrow e_i^*|_F$ at $p_{i,a}$ (lines 22-26) via recording F (lines 1-21). Theorem 1 showed that the set of events in E_RSM, viz., $CP\_RSM(e_{i,z}^*)$, matched the set of events recorded in F, viz., $F(e_{i,z}^*)$, even though E_RSM is never recorded and is accessible only to an oracle. Next we show in Theorem 2 that using the output of the algorithm and Theorem 1, the causality detection problem $CD(E\_RSM, F(e_{i,z}^*), e_{i,z}^*)$ is solved, i.e., there are no false positives nor false negatives.

Theorem 2. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 1 for the multicast mode of communication in synchronous systems under the Byzantine failure model.*

Proof. This theorem has two parts – no false negatives and no false positives – and the proof leverages the two cases in Theorem 1 which cover the multicast mode of communication. Recall our assumption in Definition 7 that $p_{i,z}$ is a correct replica. By Lemma 2, event $e_{i,z}^*$ occurs as e_i^* in E_RSM. In what follows, we use $CP\_RSM(e_{i,z}^*)$ instead of $CP\_RSM(e_i^*)$ to emphasize that the reasoning is at $e_{i,z}^*$ at $p_{i,z}$.

1. $(E\_RSM \cap CP\_RSM(e_{i,z}^*)) \setminus T(F(e_{i,z}^*)) = \emptyset$. This follows from the first case of Theorem 1 because each event in $CP\_RSM(e_{i,z}^*)$ belongs to $T(F)$ at $e_{i,z}^*$. Let $e_h^x \in CP\_RSM(e_{i,z}^*)$. The causality test in lines 22-26 of Algorithm 1 will return *true* because $e_h^x \in T(F)$ at $e_{i,z}^*$ and $V[*, h] = maxeventID(F_h)$ (when e_h^x was added to $T(F)$ at $p_{i,z}$ at or before $e_{i,z}^*$ occurred) $\geq x$. Hence $\nexists e_h^x$ such that $e_h^x \rightarrow e_{i,z}^*|_{E\_RSM} = 1 \wedge e_h^x \rightarrow e_{i,z}^*|_F = 0$. Hence there are no false negatives.
2. $T(F(e_{i,z}^*)) \setminus (E\_RSM \cap CP\_RSM(e_{i,z}^*)) = \emptyset$. This follows from the second case of Theorem 1 because each event in $T(F(e_{i,z}^*))$ must also belong to $CP\_RSM(e_{i,z}^*)$ which is a subset of E_RSM by definition. For the causality test of $e_h^x \rightarrow e_i^*$ at $p_{i,z}$ in lines 22-26 of Algorithm 1, consider the two cases: e_h^x is in F_h and not in F_h. If e_h^x is not in F_h, then by case 1 of Theorem 1, $e_h^x \notin CP\_RSM(e_{i,z}^*)$ and the test correctly returns *false*. If e_h^x is in F_h, then by case 2 of Theorem 1,

$e_h^x \in CP\_RSM(e_{i,z}^*)$ and $V[*, h] = maxeventID(F_h)$ (when e_h^x was added to $T(F)$ at $p_{i,z}$ at or before $e_{i,z}^*$ occurred) $\geq x$. Hence the test correctly returns *true*. Hence $\nexists e_h^x$ such that $e_h^x \rightarrow e_{i,z}^*|_{E\_RSM} = 0 \wedge e_h^x \rightarrow e_{i,z}^*|_F = 1$. Hence there are no false positives.

The theorem follows. □

As unicast and broadcast are special cases of multicast, we have the following corollaries to Theorem 2.

Corollary 1. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 1 for the unicast mode of communication in synchronous systems under the Byzantine failure model.*

Corollary 2. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 1 for the broadcast mode of communication in synchronous systems under the Byzantine failure model.*

4.5. Complexity

For Algorithm 1, discounting F and V which store the history and thus require unbounded space, the space complexity at a replica is $O(n^2)$ and the time complexity (to update other data structures) is $O(n^2)$.

A RSM implementation requires $3t+1$-way process replication. Efficient implementations of RSMs also require the use of cryptography [28]. A RSM-to-RSM unicast message requires $(3t+1)^2$ messages between replicas whereas a message multicast by a RSM requires $|G|(3t+1)^2$=$O(nt^2)$ messages from sender RSM replicas to receiver RSM replicas. The size of each such message is the size of the incremental history which can be arbitrarily large. Additionally, the cost of implementing an RSM including Agreement and Total Order within the RSM needs to be accounted for. There are many solutions for BFT-RSMs offering trade-offs, as captured by the survey [28]. The typical deterministic solutions are leader-based, and are variants of the precursor PBFT [36]. Such a typical deterministic implementation of an RSM requires $O((3t+1)^2)$ message complexity and $O(t)$ time complexity. Thus a message multicast costs overall $O(nt^2) + O(n.t^2)$ = $O(nt^2)$ messages of potentially large size and $O(t)$ time.

We note that such deterministic RSMs can be implemented in partially synchronous systems [28] and Algorithm 1 also works in partially synchronous systems. Thus the causality detection problem is also solvable deterministically in partially synchronous systems, without any false positives or false negatives.

Algorithm 1 can be modified to allow any correct process p_j to detect $e_h^x \rightarrow e_i^*$. $(\forall k)F_k$ would store the vector timestamp of each event in F_k at every replica of every RSM. *inc_F* would now include the vector timestamps of the events in it, leading to a factor n increase in the already large sizes of the messages carrying *inc_F*. In order to detect $e_h^x \rightarrow e_i^*$ at e_j^z, both events would have to be in the causal past of e_j^z, which may never happen if there was no communication from those processes after the events. Therefore further changes may be needed. Whenever a RSM p_i adds a new local event with its vector timestamp to F_i (at a local replica), (a) each of the $3t+1$ replicas $p_{i,a}$ would need to do a broadcast of that information to each of the $3t+1$ replicas in each of the n RSMs system-wide (this requires an extra $O(nt^2)$ messages per event at one RSM), and (b) on receipt of such information, each replica in each recipient RSM would need to run a majority computation before inserting it in the local F. These changes would complicate the presentation of the algorithm and add to its message, time, and space costs.

5. A round-based algorithm in a synchronous system

5.1. Background

5.1.1. Some cryptographic basics

We utilize non-interactive threshold cryptography as a means to guarantee no false negatives of multicasts [37], as will be explained in Section 5.2.1. Threshold cryptography consists of an initialization function to generate keys, message encryption, sharing decrypted shares of the message and finally combining the decrypted shares to obtain the original message from ciphertext. The following functions are used in a threshold cryptographic scheme.

Definition 8. The dealer executes the generate() function to obtain the public key PK, verification key VK and the private keys SK_0, SK_1, ..., SK_{n-1}.

The dealer shares private key SK_i with each process p_i while PK and VK are publicly available.

Definition 9. When process p_i wants to send a message m to p_j, it executes $E(PK, m, L)$ to obtain C_m. Here C_m is the ciphertext corresponding to m, E is the encryption algorithm and L is a label to identify m. p_i then broadcasts C_m to the system of processes.

Definition 10. When process p_l receives ciphertext C_m, it executes $D(SK_l, C_m)$ to obtain σ_l^m where D is the decryption share generation algorithm and σ_l^m is p_l's decryption share for message m.

When process p_j receives a cipher message C_m intended for it, it has to wait for k decryption shares to arrive from the system to obtain m. The value of k depends on the security properties of the system. It derives the message from the ciphertext as follows.

Definition 11. When process p_j wants to generate the original message m from ciphertext C_m, it executes $C(VK, C_m, S)$ where S is a set of k decryption shares for m and C is the combining algorithm for the k decryption shares.

The following function verifies the authenticity of a decryption share.

Definition 12. When a decryption share σ is received for message m, the Share Verification Algorithm is used to ascertain whether σ is authentic: $V(VK, C_m, \sigma) = 1$ if σ is authentic, $V(VK, C_m, \sigma) = 0$ if σ is not authentic.

5.1.2. Reliable multicast via Byzantine reliable broadcast

In our solution to the causality detection problem, we consider multicast mode of communication (unicast mode and broadcast mode are special cases). To prevent Byzantine processes from causing false negatives and false positives, a process performs multicast via Byzantine-tolerant Reliable Broadcast (BRB) in conjunction with threshold cryptography. Specifically, the communication model does multicast message sends via Byzantine-tolerant Reliable Multicast (BRM). Under the covers, the multicast invokes broadcast in conjunction with threshold cryptography.

In a multicast, a message m is sent to a subset of processes forming a process group G. Different multicast send events can send to different process groups. Byzantine Reliable Multicast is invoked as BRM_multicast(m, G) and the message is delivered via BRM_deliver(m, G).

Definition 13. Byzantine Reliable Multicast (BRM) satisfies the following properties:

1. (BRM-Validity:) If a correct process p_i BRM_delivers message m from *sender*(m) to group G, then *sender*(m) must have BRM_multicast m to G and $p_i \in G$.
2. (BRM-Termination-1:) If a correct process BRM_multicasts a message m to G, then some correct process in G eventually BRM_delivers m.
3. (BRM-Agreement or BRM-Termination-2:) If a correct process BRM_delivers a message m from a possibly faulty process, then all correct processes in G will eventually BRM_deliver m.
4. (BRM-Integrity:) For any message m, every correct process p_i BRM_delivers m at most once.

The Byzantine-tolerant Reliable Broadcast (BRB) [29,30] is invoked by BRB_broadcast and its message is delivered by BRB_deliver, and satisfies the properties given below:

Definition 14. Byzantine-tolerant Reliable Broadcast (BRB) provides the following guarantees [29,30]:

1. (BRB-Validity:) If a correct process BRB_delivers a message m from *sender*(m), then *sender*(m) must have BRB_broadcast m.
2. (BRB-Termination-1:) If a correct process BRB_broadcasts a message m, then it eventually BRB_delivers m.
3. (BRB-Agreement or BRB-Termination-2:) If a correct process BRB_delivers a message m from a possibly faulty process, then all correct processes eventually BRB_deliver m.
4. (BRB-Integrity:) For any message m, every correct process BRB_delivers m at most once.

The solutions to BRB [29,30,38] in a synchronous system do not have any upper bound on the number of rounds for reaching BRB-Agreement and do not let all correct processes reach BRB-Agreement in the same round, as shown in [39,40]. Therefore, for BRB_broadcast we invoke Dolev-Strong's authenticated Byzantine Agreement algorithm in a synchronous system that overcomes these drawbacks [31].

Definition 15. In Byzantine Agreement (BA), a single source has an initial value and the following must be satisfied [32,33]:

- (Agreement:) All non-faulty processes must agree on the same value.
- (Validity:) If the source process is non-faulty, then the agreed upon value by all the non-faulty processes must be the same as the initial value of the source.
- (Termination:) Each non-faulty process must eventually decide on a value.

In the Dolev-Strong authenticated BA algorithm, all correct processes reach agreement after exactly $f+1$ rounds, where f is the upper bound on the number of Byzantine processes and $f+1<n$, with nf messages [31]. They also proved $f+1$ rounds is a lower bound. When correct processes reach agreement on a non-null message, that message is BRB_delivered; if they reach agreement on a default null message, there are no BRB_deliver events for it. It can be seen that the Dolev-Strong authenticated Byzantine agreement algorithm straightforwardly satisfies BRB-Validity, BRB-Termination-1, BRB-Agreement, and BRB-Integrity.

5.2. Basic idea and algorithm

Algorithm 2 presents an online round-based algorithm for solving the CD problem under multicasts in the presence of Byzantine processes. A multicast is sent via Byzantine Reliable Broadcast (BRB) to ensure the BRM-Validity, BRM-Termination-1, BRM-Agreement, and BRM-Integrity properties. The ensuing BRM_multicast and BRM_deliver events at the various processes are carefully assigned timestamps by correct processes to ensure that no false positives result. Besides the no-OOBC assumption, threshold cryptography is used to ensure that there are no false negatives.

5.2.1. Need for threshold encryption

Let β denote the number of rounds of a BRB protocol. A BRB_broadcast is delivered after β number of rounds as BRB_deliver. A Byzantine process p_b may peek into the content of the message m_1 sent by any process at e_a^x during the (first of the) β rounds before the message is BRB_delivered, thereby creating a receive event e_r_b. It may then send message m_2 at send event e_s_b in the same round via BRB. We have $e_a^x \rightarrow e\_r_b \rightarrow e\_s_b$ (and furthermore there is a semantic information dependency of m_2 on m_1). Correct processes can get fooled into believing that the Byzantine process had the receive event in round β when m_1 gets BR_delivered to all, thereby concluding incorrectly that $e\_s_b \rightarrow e\_r_b$. Therefore $e\_r_b \rightarrow e\_s_b$ should be detected by all processes, or prevented from happening.

Consider an online auction. Here a Byzantine bidder can make a bid based on and after peeking into (reading) a correct bidder's bid and yet have its own bid be registered first and gain an unfair advantage. Or consider online gaming, where a Byzantine player could similarly win an unfair advantage.

Agreement between F at every good process is necessary but not sufficient. A trivial algorithm could have them agree on empty sequence F. So as a sufficient condition, there must be some correlation to E including what happened at Byzantine processes. A receive event (like peeking into, or even otherwise) e_r_b at a Byzantine process p_b that happens before a subsequent send event e_s_b at the Byzantine process must be recognized correctly (and $e\_r_b \rightarrow e\_s_b$ detectable or prevented) as they can form part of a longer causality chain between events at correct processes. The whole point is recognizing and agreeing on God's truth (i.e., what events actually occurred in the system and in what order). It is not all right if the correct processes agree that in F, at the Byzantine process, e_r_b (at the end of round β) happened after e_s_b (in round 1 of that same meta-round) because in reality e_r_b happened in round 1 before e_s_b and additionally introduced a causal and semantic dependency from e_r_b to e_s_b.

We solve this problem by encrypting each message using threshold encryption, which prevents a Byzantine process from peeking into messages before they are BRB_delivered. In round $\beta+1$, each process that has BRB_delivered the message sends its decryption share to the destinations of the multicast. A message m gets revealed to a process only at the end of round $\beta+1$ at which time the receive event occurs at the application; any message sent before the end of round $\beta+1$ cannot be causally or semantically dependent on this revealed message m, and the only messages that the process sends that are causally (or semantically) dependent on the above message m can get sent only in later rounds. The $\beta+1$ rounds $a\times(\beta+1)+0$, $a\times(\beta+1)+1$, $\ldots$ $a\times(\beta+1)+\beta-1$, $a\times(\beta+1)+\beta$ constitute a meta-round a for $a\geq 0$. Thus, rounds $r, r+1, \ldots r+\beta$, such that $r\, div\, (\beta+1)=a$ constitute meta-round a. The first β rounds are for BRB and the $(\beta+1)$th round is for sending the decryption shares to the destinations of the multicast.

Even with threshold encryption, a Byzantine process p_b can send message m_2 at e_s_b after learning in round 1 of a meta-round that some message m_1 has been sent; however, m_2 is not dependent on the content of m_1. Moreover, no receive event e_r_b at which m_1 was read happened at the Byzantine process before the send event e_s_b at it and there is no semantic dependency of m_2 on m_1. The knowledge that some m_1 was sent is not useful because the Byzantine process could always send a message in the first round of each meta-round anyway, but there would never be any semantic dependency on messages sent by others in that meta-round. In the example of online auction bidding, a Byzantine process p_b bidding in m_2 without knowing what the bid was in m_1 does not give any advantage to p_b. Similarly in online gaming, knowing that another player was making some move without knowing what move it is does not give the Byzantine player any advantage as it does not know what counter-move to make via m_2. If the above advantages of threshold encryption do not apply to some application, it can omit the use of threshold encryption and the round $\beta+1$.

5.2.2. Algorithm with description

Data Structures: The data structures at any process p_i are as follows.

1. Each process p_i has access to PK (global public key) and VK (global verification key). Each process has access to a local secret key SK_i. All processes in a multicast group G locally store the group key K_G.
2. Q: FIFO queue for incoming application messages.

Algorithm 2: Processing of control information for CD under multicasts. Code for p_i.

```
when $p_i$ invokes BRM_multicast($m$, $G_{id_m}$) in round $r$, $r \mod (\beta+1) = 0$:
$C'_m = Enc(K_{G_{id_m}}, m)$
$C_m = E(PK, C'_m, id_m)$
$q++$
BRB_broadcast($C_m, G_{id_m}, i, q$) in the next round $r$, $r \mod (\beta+1) = 0$

when all $\langle C_m, G_{id_m}, s, q_s \rangle$ are BRB_delivered at $p_i$ from $p_s$ in round $r$, $r \mod (\beta+1) = \beta - 1$:
$V\_previous = V$
for BRB_delivered messages in lexicographic order on $\langle s, q_s \rangle$ do
    $\sigma_i^m = D(SK_i, C_m)$
    if $p_i \in G_{id_m}$ then
        $Q.push(C_m)$
    for each $p_j \in G_{id_m}$ do
        enqueue in $Q_{sh}$ "send($\sigma_i^m$) to $p_j$"
    $V[s, s]++$
    enqueue in $Q_{sent}$ send event of $C_m$ from $p_s$ to $G_{id_m}$, along with $(T \leftarrow)V[s]$
for BRB_delivered messages in lexicographic order on $\langle s, q_s \rangle$ do
    for each $j \in G_{id_m}$ do
        $V[j, j]++$
        if $s \neq j$ then
            $V[j, s] = T[s]$ of the corresponding send event enqueued in $Q_{sent}$
        for each $a \in [1, n], a \neq s, j$ do
            $V[j, a] = \max(V[j, a], V\_previous[s, a])$
        enqueue in $Q_{rcvd}$ receive event of $C_m$ from $p_s$ to/at $p_j \in G_{id_m}$, along with $(T \leftarrow)V[j]$

when round $r$, $r \mod (\beta+1) = \beta$ begins:
while $Q_{sh}$ not empty do
    dequeue from $Q_{sh}$ "send($\sigma_i^m$) to $p_j$" and send($\sigma_i^m$) to $p_j$
while $Q_{sent}$ not empty do
    dequeue from $Q_{sent}$ send event $e$ of $C_m$ from $p_s$ to $G_{id_m}$, along with its vector timestamp $T$, and append to $F_s$
    if $s \neq i$ then
        $CPV[s] = T[s]$
while $Q_{rcvd}$ not empty do
    dequeue from $Q_{rcvd}$ receive event $e$ of $C_m$ from $p_s$ to/at $p_j \in G_{id_m}$, along with its vector timestamp $T$, and append
     to $F_j$
    if $j = i$ then
        $CPV[j] = T[j]$

when $p_i$ receives $(t+1)$ valid $\langle \sigma_x^m \rangle$ messages for $m$ in round $r$, $r \mod (\beta+1) = \beta$:
Store $(t+1)$ decryption shares in set $S_{id_m}$
$C'_m = C(VK, C_m, S_{id_m})$
replace $C_m$ in $Q$ with $C'_m$

when the application is ready to process a message at $p_i$:
if $Q.head()$ is decrypted then
    $C'_m = Q.pop()$
    $m = Dec(K_{G_{id_m}}, C'_m)$
    BRM_deliver($m$, $G_{id_m}$)
```

3. q: integer counter that is incremented for each local BRB_broadcast.
4. Q_{sh}: FIFO queue for decryption shares.
5. Q_{sent}: FIFO queue of system-wide send events with vector timestamps T assigned by p_i in this meta-round.
6. Q_{rcvd}: FIFO queue of system-wide deliver events with vector timestamps T assigned by p_i in this meta-round.

Algorithm 3: Test for CD between e_h^x and e_i^y at p_c.

```
1 test for $e_h^x \rightarrow e_i^y$ at the end of a meta-round at correct process $p_c$:
2 if $y \le CPV[i] \wedge x \le CPV[h]$ then
3     if $e_i^y.T[h] \ge x$ then
4         output true
5     else
6         output false
7 else
8     output false
```

7. $V[n,n]$: matrix clock that is an array of vector timestamps. $V[j,k]$ indicates p_i's knowledge of p_j's knowledge of p_k's count of local send (BRM_multicast) and deliver (BRM_deliver) events.
8. $V\_previous[n,n]$: array of vector timestamps reflecting the state (of V) at the end of the previous meta-round.
9. $CPV[n]$: array of timestamps. $CPV[j]$ ($j \neq i$) indicates p_i's knowledge of the local scalar timestamp of the latest send event at p_j. If $j = i$, it is p_i's knowledge of the local scalar timestamp of the latest event at p_i.
10. F: set of sequences F_k. F_k is the recorded history of events of p_k along with their vector timestamps assigned by p_i, denoted T.

Algorithm 2 requires that key generation and distribution has been accomplished by a trusted dealer prior to the start of execution. Therefore, all processes have access to a global PK (public key), VK (verification key) and have a local SK_i (secret key). In addition to this, all multicast groups share a unique symmetric key for encryption and decryption of messages intended for them. Algorithm 2 *double encrypts* each message, first with the group key ($K_{G_{id_m}}$) (line 2) and then with the public key (PK) (line 3) and invokes a Byzantine Reliable Broadcast (BRB_broadcast) on the resulting ciphertext (line 5). The BRB_broadcast has a latency of $\beta = f + 1$ rounds as it uses the Dolev-Strong authenticated Byzantine agreement under the covers [31], as explained in Section 5.1. This action of sending a message via BRB_broadcast can be executed only in round r, where $r \mod (\beta+1) = 0$. Upon receiving the ciphertext via BRB_deliver in round r', where $r' \mod (\beta+1) = \beta - 1$ (line 6), all processes compute their respective decryption shares (line 9), enqueue in a shares queue Q_{sh} (lines 12-13) so as to send their decryption share to all destinations of the multicast group in the next round r'' where $r'' \mod (\beta+1) = \beta$ (line 24-26) and that decryption share is received in that same round r'' (line 35). Meanwhile, the recipients of the multicast ciphertext message in round r' enqueue the ciphertext in their respective FIFO delivery queue Q (lines 10-11). Upon receiving the required number of valid and unique decryption shares in round r'' (line 35), the ciphertext is decrypted using the decryption shares of threshold cryptography to obtain the ciphertext encrypted with the group key in round r'' (lines 36-38). When this ciphertext reaches the head of the delivery queue in the same round r'', it is decrypted using the group key $K_{G_{id_m}}$ to obtain the original message and delivered to the application (lines 39-43). We assume the application is always ready to dequeue any plaintext message in the delivery queue Q. Note that the use of the group key ensures privacy because only members of group G_{id_m} should be allowed to see the plaintext. The number of Byzantine processes f that Algorithm 2 can tolerate satisfies $f + 1 < n$ as it invokes the Dolev-Strong authenticated agreement under the covers [31].

When a message from p_s to G_{id_m} is BRB_delivered to p_i, it infers that p_s must have sent m (from BRB-Validity of BRB_broadcast); and each (correct) $p_j \in G_{id_m}$ would have (and each Byzantine $p_j \in G_{id_m}$ could have) BRB_delivered m (from BRB-Agreement of BRB_broadcast). Further, p_j would get the required number of correct decryption shares in the last round of the meta-round. The vector timestamp T assigned to an event at p_α in Q_{sent}, Q_{recd}, and in F has to be carefully set to the current value of $V[\alpha]$ by p_i. The notation $e_\alpha^y.T_c[\gamma]$ denotes at process p_c, p_c's view of p_α's view of the count of the events at p_γ at the event e_α^y. p_i acts as follows.

1. p_i increments $V_i[s,s]$ for p_s's send event (line 14). p_i does so for all the messages that have been BRB_delivered.
2. Then for each of the BRB_delivered messages, considered in the same order as in the above bullet, for each $p_j \in G_{id_m}$, p_i does the following.
   (a) p_i increments $V_i[j,j]$ for the receive event at p_j (line 18).
   (b) Then if the sender $p_s \neq p_j$, p_i sets $V_i[j,s]$ to $T[s]$ (i.e., $V[s,s]$) of the corresponding send event (lines 19-20). The current value of $V[s,s]$ is not used because more send events could have occurred at p_s since the corresponding send event. (If $j = s$, p_i should not do this step as the corresponding send event may have an older value of $T[s]$ (i.e., $V[s,s]$) that would overwrite $V_i[j,j]$ updated in line 18.)
   (c) Further, for each p_j, for each $p_a (\neq p_s, p_j)$, p_i sets $V_i[j,a]$ to the maximum of its current value and $V\_previous[s,a]$ to update its local knowledge of $V_i[j,a]$ with the sender p_s's knowledge of $V_i[s,a]$ dated at the end of the previous meta-round (lines 21-22). That value reflects the most recent value of $V_i[s,a]$ as of the sending of the message C_m. The current value of $V_i[s,a]$ is not used because it may reflect updates due to receive events in the current meta-round.

Because of non-deterministic receives and because the BRB protocol running the Dolev-Strong authenticated agreement under the covers does not inherently maintain total order or even source order, two correct processes p_i and $p_{i'}$ may order the send event of the same message m differently (thus $V_i[s,s] \neq V_{i'}[s,s]$), and they may also order the deliver event of the same message at a particular process $p_j \in G_{id_m}$ differently (thus $V_i[j,j] \neq V_{i'}[j,j]$). Thus, p_i and $p_{i'}$ may have different values of $V[s,s]$ for the same multicast send event and different values for $V[j,j]$ for the receive event of the same message at p_j. To prevent this from happening, all messages $\langle C_m, G_{id_m}, s, q_s \rangle$ that are BRB_delivered in round $r, r \mod (\beta+1) = \beta - 1$, are processed in lexicographic order based on $\langle s, q_s \rangle$. In the lexicographic ordering, the q_s is a local counter value that is incremented by the sender of BRB_broadcast at each such broadcast. As the same set of messages have been BRB_delivered, all correct processes process the messages in the same order and assign the same timestamps to all send events and all deliver events. A Byzantine process may not follow this rule. Yet this does not matter because all BRB_broadcasts sent in a round are based on the *same* state of the sender which is the *state at the end of the previous meta-round*.

The algorithm processes all the BRB_broadcast send events before all the BRB_deliver events in a meta-round. This is achieved by first enqueuing all the send event timestamps in a queue Q_{sent}, and then enqueuing all the deliver event timestamps in a queue Q_{rcvd} in round $r, r \mod (\beta+1) = \beta - 1$, on BRB_delivery. In the next round, these events are dequeued from Q_{sent} and then from Q_{rcvd} and appended to the respective sender process p_s's F_s and receiver processes p_j's F_j, respectively. Q_{sh} is a temporary queue that enqueues the decryption shares of the messages on BRB_delivery and sends them to the corresponding destinations of the multicast in the next round, viz., round $r, r \mod (\beta+1) = \beta$.

Algorithm 3 allows any correct process p_c to determine whether $e_h^x \to e_i^y$. For this, firstly $e_h^x \in F_h$ and $e_i^y \in F_i$ at p_c. p_c adds a receive event at p_j in p_c's F_j and updates its V with the receive event's timestamp when the corresponding message is BRB_delivered to p_c. But p_j may be Byzantine or may have crashed before the BRM_deliver occurred at it. So there is a need to verify whether the event added to p_c's F_j actually occurred at p_j. Thus, p_c needs to check (verify) whether such events entered in its local F actually occurred in the causal past indicated by the V matrix. We adapt Definition 2 of the *causal past* for Algorithm 2.

Definition 16. The *causal past* at the end of a meta-round at a (correct) process p_c is defined as all the events $e_h^v \in T(E)$ such that $v \leq w$, where w is such that:

- if $c \neq h$, e_h^w is the latest BRM_multicast send event such that its ensuing BRB_deliver has occurred at p_c up to that meta-round, and
- if $c = h$, $w = V_c[h,h]$.

The *causal past* check in line 2 of Algorithm 3 is implemented by the CPV vector, where $CPV[j]$ $(j \neq c)$ at p_c gives the scalar timestamp of the latest send event at p_j witnessed by p_c. The jth entry is updated in lines 29-30 and 33-34 of Algorithm 2 after BRB_delivery processing.

5.3. Correctness proof

Algorithm 2 satisfies BRM-Validity, BRM-Termination-1, BRM-Agreement, and BRM-Integrity due to the corresponding properties of BRB. This gives the following theorem, see the Appendix for the proof.

Theorem 3. *Algorithm 2 satisfies the properties of Byzantine Reliable Multicast (Definition 13) as it performs multicast mode of communication.*

Definition 17. A send event of message m at p_s to G_{id_m} is said to have occurred in E if and only if m is received (after having been BRB_delivered and subsequently double-decrypted) at all correct processes in G_{id_m}.

Definition 18. A receive event of message m to G_{id_m} is said to have occurred at $p_j \in G_{id_m}$ in E if and only if m is received (after having been BRB_delivered and subsequently double-decrypted) at all correct processes in G_{id_m}.

We solve the following version of Definition 3 for the causality detection problem. This version differs in that it deals only with communication events but allows an arbitrary correct process p_c and not just p_i to detect $e_h^x \to e_i^y$; Algorithm 2 can achieve this without incurring any extra message, space, or time complexity due to (i) the use of BRB under the covers, and (ii) p_c assigning vector timestamps to e_h^x, e_i^*, and all events at other processes based on its local matrix clock without having the algorithm broadcast those vector timestamps.

Definition 19. For send or receive events e_h^x and e_i^y, the causality detection problem $CD(E, F, e_h^x, e_i^y)$ is to devise an algorithm to collect the execution history E as F at a correct process p_c such that $valid(F) = 1$, where

$$valid(F) = \begin{cases} 1 & \text{if } e_h^x \to e_i^y|_E = e_h^x \to e_i^y|_F \\ 0 & \text{otherwise} \end{cases}$$

Theorem 4. *The application-layer set of (multicast send and receive) events of the execution E up to a meta-round equals the set of events recorded in F at any correct process up to the same meta-round (with the* no out-of-band communication (no OOBC) *assumption).*

Proof. A message m sent by a correct process p_s via BRB_broadcast is necessarily BRB_delivered to all correct processes and the send event is said to have occurred in E. If m is sent by a Byzantine process p_s, it is either BRB_delivered to all correct processes or to none (follows from the BRB-Agreement property of BRB_broadcast) and only in the former case is the send event said to have occurred in E. If m is BRB_delivered to all correct processes in round r, where $r \mod (\beta+1) = \beta - 1$ (lines 6-23), then only all processes in G_{id_m} will receive the required number of decryption shares, decrypt the message (lines 35-38) and then double-decrypt using the group key $K_{G_{id_m}}$ (lines 39-43) in the next round and thus they will have the receive event for m. When m was BRB_delivered at correct processes (lines 6-23), then and only then do those processes add a send event in F_s (lines 15, 27-28) as well as add a receive event for each destination p_j in G_{id_m} in F_j (lines 23, 31-32).

Thus, if and only if m is BRB_delivered to a correct process does the correct process record the corresponding send event at p_s in its F and the corresponding receive event at each destination $p_j \in G_{id_m}$ in its F, and only then are the corresponding application-level send and receive events said to occur in E (as defined by Definitions 17 and 18, respectively). Note that the receive event for m at $p_j \in G_{id_m}$ may not actually have occurred if p_j has crashed or is Byzantine; however Algorithm 3 checks in the causality test whether the receive event indeed belongs to the *causal past*. □

Using Theorem 4 and reasoning with the updation of the V data structure and setting of the vector timestamp $e_k^{\gamma}.T_c$ associated with each event e_k^{γ} entered in the local F_k at a correct process p_c, the following theorem follows. The proof is given in the Appendix.

Theorem 5. *For (application-level) multicast send and receive events e_h^x and e_i^y, $e_h^x \to e_i^y|_E = 1$ if and only if $e_i^y.T_c[h] \geq e_h^x.T_c[h]$ at any correct process p_c in Algorithm 2 (with the* no out-of-band communication (no OOBC) *assumption).*

Therefore a causality relation $e_h^x \to e_i^y$ is inferred from F, viz., from the timestamps recorded by the algorithm, if and only if the relation actually exists in E.

Corollary 3. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 2 for the multicast mode of communication in synchronous systems under the Byzantine failure model (with the* no out-of-band communication (no OOBC) *assumption).*

As unicast and broadcast are special cases of multicast, we also have the following results.

Corollary 4. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 2 for the unicast mode of communication in synchronous systems under the Byzantine failure model (with the* no out-of-band communication (no OOBC) *assumption).*

Corollary 5. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 2 for the broadcast mode of communication in synchronous systems under the Byzantine failure model (with the* no out-of-band communication (no OOBC) *assumption).*

Theorem 6. *There are no false positives but false negatives may occur in solving causality detection as per Algorithm 2 for the multicast mode of communication in synchronous systems under the Byzantine failure model.*

Proof. In addition to the application-layer send and receive events, with OOBC there may be out-of-band communication events. Consider such a send-receive event pair (e_h^x, e_i^y). This dependency is not captured by our algorithm because this communication takes place without following the protocol, viz., without sending and receiving by using BRM_multicast and BRM_deliver via BRB_broadcast and BRB_deliver. Therefore false negatives may occur. In more detail, consider the following. Let $e_g^u \to e_h^{u'}$ due to a message sent via BRM_multicast following the algorithm, and let $e_i^{z'} \to e_j^z$ due to another message sent via BRM_multicast following the algorithm. Further, let $e_h^{u'} \to e_h^x$ and let $e_i^y \to e_i^{z'}$. Then we have $e_g^u \to e_h^{u'} \to e_h^x \to e_i^y \to e_i^{z'} \to e_j^z$. However $e_g^u \to e_j^z$ will not be detectable by the algorithm as $e_j^z.T_c[g] \not\geq e_g^u.T_c[g]$ because the OOBC ($e_h^x \to e_i^y$) did not allow $e_g^u.T[g]$ to propagate to $e_j^z.T_c[g]$. This results in a false negative.

However, for every causal relation for an application-layer send-receive event pair, as (e_h^x, e_i^y) sent and received using BRM_multicast and BRM_deliver, it is observed/detected by the algorithm at correct process p_c, the events (with their timestamps) would have been inserted in F_h and F_i, respectively at p_c on BRB_delivery of the message at p_c, and it is a real dependency. OOBC does not result in the removal of such dependencies. Hence there will not be any false positives. □

The same logic as in Theorem 6 holds for unicast and broadcast modes of communication. This leads to the following corollaries.

Corollary 6. *There are no false positives but false negatives may occur in solving causality detection as per Algorithm 2 for the unicast mode of communication in synchronous systems under the Byzantine failure model.*

Corollary 7. *There are no false positives but false negatives may occur in solving causality detection as per Algorithm 2 for the broadcast mode of communication in synchronous systems under the Byzantine failure model.*

5.4. Complexity

Barring the unbounded space complexity of the history F, Algorithm 2 requires $O(n^2)$ space for the arrays, and the queues should be large enough to hold the send and deliver events for messages sent in a meta-round. The message and time complexities are that of implementing BRB and of sending decryption shares. The message complexity is $n^2 + nf$ messages (nf messages for Dolev-Strong authenticated broadcast to implement BRB and n^2 for sending decryption shares) per multicast message. The time complexity is $f + 2$ rounds in a meta-round ($f + 1$ rounds for Dolev-Strong authenticated broadcast to implement BRB and one round for sending decryption shares), common to all messages multicast in that meta-round. The algorithm has optimal $n > f + 1$ fault-tolerance.

6. Discussion and conclusions

We proved that the causality detection problem is solvable deterministically in a synchronous message-passing distributed system subject to Byzantine failures by proposing two algorithms that provide different trade-offs. These trade-offs were presented in Section 1.2.

As deterministic RSM implementations also work in partially synchronous systems [28], Algorithm 1 works in partially synchronous systems. This gives the following result.

Theorem 7. *There are neither false negatives nor false positives in solving causality detection as per Algorithm 1 for the multicast, broadcast, and unicast modes of communication in partially synchronous systems under the Byzantine failure model.*

Detecting causality between a pair of events is a fundamental problem [2]. Other problems that use this problem as a building block include the following:

- detecting the interaction type between a pair of intervals at different processes [41],
- detecting the fine-grained modality of a distributed predicate [42,43], and data-stream based global event monitoring using pairwise interactions between processes [44],
- detecting causality relation between two “meta-events” [45–47], each of which spans multiple events across multiple processes [48].

It can be shown that these problems in Byzantine failure-prone synchronous systems are solvable because they are reducible to causality detection in the presence of Byzantine processes in synchronous systems.

CRediT authorship contribution statement

Anshuman Misra: Formal analysis, Investigation, Methodology, Writing – original draft. **Ajay D. Kshemkalyani:** Conceptualization, Formal analysis, Investigation, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Proofs of Theorem 3 and Theorem 5

Theorem 3. *Algorithm 2 satisfies the properties of Byzantine Reliable Multicast (Definition 13) as it performs multicast mode of communication.*

Proof. We show that BRM-Validity, BRM-Termination-1, BRM-Agreement, and BRM-Integrity are satisfied.

1. BRM-Validity: On BRM_deliver(m, G_{id_m}) in line 43, the message must have been decrypted with the group key (lines 41-42) after receiving $t+1$ valid decryption shares and having been decrypted with them (lines 35-38), after BRB_deliver-(C_m, G_{id_m}) in line 6, all in the same-meta-round. By BRB-Validity, BRB_broadcast(C_m, G_{id_m}) must have occurred. From the algorithm code, this is possible only after double-encryption via a BRM_multicast(m, G_{id_m}) invocation in that same meta-round. BRM-Validity follows.
2. BRM-Termination-1: On BRM_multicast(m, G_{id_m}) by a correct process, BRB_broadcast(C_m, G_{id_m}) occurs after double-encryption in round $r, r \mod (\beta+1) = 0$ (lines 2-5). BRB_delivery occurs and all correct processes p_j in G_{id_m} enqueue C_m in Q in round $r, r \mod (\beta+1) = \beta - 1$ (lines 8-11). At least all the correct processes send valid decryption shares to $p_j \in G_{id_m}$ in round $r, r \mod (\beta+1) = \beta$ (lines 12-13 and 25-26) and these are received by the correct processes $p_j \in G_{id_m}$, which double-decrypt (lines 35-38 and 41-42) and dequeue and execute BRM_deliver(m, G_{id_m}) (lines 39-43), all in the same round. This happens along with BRM_delivery of other messages sent in that meta-round. BRM-Termination-1 follows.
3. BRM-Agreement: On BRM_deliver(m, G_{id_m}) in line 43 at any correct process, the message must have been decrypted with the group key (lines 41-42) after receiving $t+1$ valid decryption shares and having been decrypted with them (lines 35-38), after BRB_deliver(C_m, G_{id_m}) in line 6, all in the same-meta-round. By BRB-Agreement, the BRB_delivery must have occurred at all correct processes. They would have sent their decryption shares to all $p_j \in G_{id_m}$ (lines 12-13 and 25-26). All these p_j (which would also have enqueued C_m in Q on BRB_delivery (lines 10-11)) would receive at least $t+1$ valid decryption shares, double-decrypt C_m and BRM_deliver(m, G_{id_m}) (lines 35-38, 39-43). BRM-Agreement follows.
4. BRM-Integrity: BRM_delivery occurs only if BRB_delivery of that message occurs at a correct process. By BRB-Integrity, a message is delivered at most once. Thus that BRM_delivery of that message occurs at most once at a correct process. BRM-Integrity follows. □

Theorem 5. *For (application-level) multicast send and receive events e_h^x and e_i^y, $e_h^x \to e_i^y|_E = 1$ if and only if $e_i^y.T_c[h] \geq e_h^x.T_c[h]$ at any correct process p_c in Algorithm 2 (with the* no out-of-band communication (no OOBC) *assumption).*

Proof. We prove each direction separately. For both directions, we first note from Theorem 4 that the set of send and receive events in E equals the set of send and receive events in F up to any meta-round mr. (Recall from the proof of that theorem that this is subject to the check of belonging in the *causal past* as implemented in line 2 of Algorithm 3.)

($\Longrightarrow$:) Let $e_h^x \to e_i^y|_E = 1$. We show that $e_i^y.T_c[h] \geq e_h^x.T_c[h]$.

If $h = i$, then in the causality graph there is a program order path from e_h^x to e_i^y where $e_h^x.mr \leq e_i^y.mr$. From Theorem 4, both events will be in F_h at any correct process p_c. Further, from Algorithm 2, e_h^x will precede e_i^y in $F_{h(=i)}$ at p_c as events are processed/appended to F_h in lexicographic order of $\langle s, q_s \rangle$ with send events being processed/appended before receive events, in a meta-round, and successive meta-round events are processed/appended serially. (Refer lines 27-28 for send and lines 31-32 for receive.) At each next event processed/appended, $V_c[h,h]$ is incremented by one (refer line 14 for send and lines 17-18 for receive). Hence $e_i^y.T_c[h] \geq e_h^x.T_c[h]$.

If $h \neq i$, then in the causality graph there are alternating program order paths and message order edges from e_h^x to e_i^y satisfying the following.

1. There exist messages $m_q, q \in [1,l]$ where m_q is sent by $p_{a_{q-1}}$ at $e'_{a_{q-1}}$ and received by p_{a_q} at e''_{a_q}, and $e'_{a_{q-1}}.mr = e''_{a_q}.mr$,
2. $a_0 = h$, and $e_h^x \to e'_{a_0}$ or $e'_{a_0} = e_h^x$,
3. $a_l = i$, and $e''_{a_l} \to e_i^y$ or $e''_{a_l} = e_i^y$.

From Algorithm 2, we have the following.

1. From the reasoning for the $h = i$ case above, if $e_h^x \to e'_{a_0}$ or $e_h^x = e'_{a_0}$ then $e_h^x.T_c[h] \leq e'_{a_0}.T_c[h]$.
2. In the causality graph, for the program order edge at p_{a_q} ($q \in [1, l-1]$) from e''_{a_q} to e'_{a_q}, we have $e''_{a_q}.mr < e'_{a_q}.mr$. Then $e''_{a_q}.T_c[h] \leq e'_{a_q}.T_c[h]$ as $V_c[a_q, h]$ is a non-decreasing function at any process.
   Observe that e''_{a_q} (receive event) $\to$ e'_{a_q} (send event) and as m_{q-1} is double-decrypted by p_{a_q} only in round β of meta-round $e''_{a_q}.mr$, a send event $\hat{e}'$ that depends on the decrypted content of m must be in a later meta-round as it could not have been sent in the first β rounds (i.e., rounds 0 to $\beta-1$) of $e''_{a_q}.mr$. If $\hat{e}'$ is sent by a Byzantine process in round β of the meta-round, it will not get processed by the correct processes as part of the BRB protocol and will not be BRB_delivered. This prevents false negatives.
3. In the causality graph, for the message order edge $(e'_{a_{q-1}}, e''_{a_q})$ ($q \in [1,l]$) where $e'_{a_{q-1}}.mr = e''_{a_q}.mr$, we have $e'_{a_{q-1}}.T_c[h] \leq e''_{a_q}.T_c[h]$ because of lines 18, 19-20, and 21-22 (where $j = a_q, a = h, s = a_{q-1}$) and line 23.
4. If $e''_{a_l} \to e_i^y$ or $e''_{a_l} = e_i^y$ then $e''_{a_q}.T_c[h] \leq e_i^y.T_c[h]$ because of lines 14 and/or 17-18. Refer to the reasoning for the $h = i$ case above.

Applying transitivity to the program order edges and message order edges from e_h^x to e_i^y and from the above relations, we have $e_i^y.T_c[h] \geq e_h^x.T_c[h]$ at a correct process p_c.

($\Longleftarrow$:) We first identify a sequence of processes having indices a_q from $a_0, a_1, \ldots a_{q_{max}}$, where $q_{max} \leq n-1$ and events e_{a_q} at these processes such that e_{a_q} is the earliest event at p_{a_q} having $e_{a_q}.T[h] \geq e_h^x.T[h]$. We then show that $e_h^x \to e_{a_{q_{max}}-1} \to e_{a_{q_{max}}-2} \to \ldots e_{a_0} \to e_i^y$.

Let $a_0 = i$ and let $e'_{a_0} = e_i^y$. Initialize $q = 0$. There are two cases for a_q.

1. Case $a_q = h$. Trace backwards in F_{a_q} to event e such that $e_{a_q}.T[h] = e_h^x.T[h]$. Then $e_h^x \overset{=}{\to} e'_{a_q}$. This is the base termination case of the proof.
2. Case $a_q \neq h$. Let e_{a_q} be the earliest event at p_{a_q} (by tracing backwards in F_{a_q}) such that $e_{a_q}.T[h] \geq e_h^x.T[h]$. Then $e_{a_q} \overset{=}{\to} e'_{a_q}$.
   From Algorithm 2, e_{a_q} must be a receive event and $e_{a_q}.T[h]$ *must have* been set in line 23 to the value set in line 20 (with $j = a_q$, $s = h$, where $V[a_q, h]$ was set to $T[h]$ of the corresponding send event by s) or line 22 (with $j = a_q$, $a = h$, where $V[a_q, h]$ was assigned $V[s, h]$). Set a_{q+1} to s, the sender of the message. Thus e_{a_q} is a receive event at p_{a_q} and the message was sent by $p_{a_{q+1}}$ at an event $e'_{a_{q+1}}$ such that $e'_{a_{q+1}}.T[h] = e_{a_q}.T[h] \geq e_h^x.T[h]$. We also have $e'_{a_{q+1}} \to e_{a_q}$.
   We also identified a_{q+1}. If $a_{q+1} = h$, then we apply case 1 to a_{q+1}, i.e., invoke case 1 setting q to $q+1$, and the logic terminates in that case. Otherwise either (a) $q+1 \leq q_{max}$, or (b) $q+1 > q_{max}$.
   (a) If $q+1 \leq q_{max}$, we set q to $q+1$. Now, as $a_q \neq h$, apply the logic of case 2 which is the applicable case. The case 2a will get invoked $q_{max} \leq n$ times after which case 1 must get invoked and the invocations terminate.
   (b) $q+1 > q_{max} = n-1$ can never occur because by the pigeonhole principle, there are n processes and each process p_{a_q} has its event e_{a_q} uniquely identified as the earliest event such that $e_{a_q}.T[h] \geq e_h^x.T[h]$. Further, $e_{a_{q+z}} \to e_{a_q}$ for $z > 0$. Therefore there cannot be a z, with $z > 0$, such that $p_{a_q} = p_{a_{q+z}}$.

We then have $e_h^x \to e_i^y$ as $e_h^x \overset{=}{\to} e'_{a_{q_{max}}} \to e_{a_{q_{max}}-1} \to e'_{a_{q_{max}}-1} \to e_{a_{q_{max}}-2} \to e'_{a_{q_{max}}-2} \to \ldots e'_{a_1} \to e_{a_0} \overset{=}{\to} e_i^y$. □

References

[1] A. Misra, A.D. Kshemkalyani, Detecting causality in the presence of Byzantine processes: the synchronous systems case, in: 30th International Symposium on Temporal Representation and Reasoning, (TIME), in: LIPIcs, vol. 278, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 11:1–11:14, https://doi.org/10.4230/LIPICS.TIME.2023.11.

[2] R. Schwarz, F. Mattern, Detecting causal relationships in distributed computations: in search of the holy grail, Distrib. Comput. 7 (3) (1994) 149–174, https://doi.org/10.1007/BF02277859.

[3] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565, https://doi.org/10.1145/359545.359563.

[4] A.D. Kshemkalyani, M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, 2011, https://doi.org/10.1017/CBO9780511805318.

[5] E. Elnozahy, Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication, Phd thesis, Tech. Rep., Tech. Report 93-212, Computer Science Department, Rice University, 1993.

[6] C.J. Fidge, Logical time in distributed computing systems, IEEE Comput. 24 (8) (1991) 28–33, https://doi.org/10.1109/2.84874.

[7] F. Mattern, Virtual time and global states of distributed systems, in: Parallel and Distributed Algorithms, North-Holland, 1988, pp. 215–226.

[8] B. Charron-Bost, Concerning the size of logical clocks in distributed systems, Inf. Process. Lett. 39 (1) (1991) 11–16, https://doi.org/10.1016/0020-0190(91)90055-M.

[9] G.T.J. Wuu, A.J. Bernstein, Efficient solutions to the replicated log and dictionary problems, in: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, ACM, 1984, pp. 233–242, https://doi.org/10.1145/800222.806750.

[10] A.D. Kshemkalyani, The power of logical clock abstractions, Distrib. Comput. 17 (2) (2004) 131–150, https://doi.org/10.1007/s00446-003-0105-9.

[11] P.A.S. Ward, D.J. Taylor, A hierarchical cluster algorithm for dynamic, centralized timestamps, in: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001), 2001, pp. 585–593, https://doi.org/10.1109/ICDSC.2001.918989.

[12] F.J. Torres-Rojas, M. Ahamad, Plausible clocks: constant size logical clocks for distributed systems, Distrib. Comput. 12 (4) (1999) 179–195, https://doi.org/10.1007/s004460050065.

[13] N.M. Preguiça, C. Baquero, P.S. Almeida, V. Fonte, R. Gonçalves, Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors, in: ACM Symposium on Principles of Distributed Computing, PODC, 2012, pp. 335–336, https://doi.org/10.1145/2332432.2332497.

[14] P.S. Almeida, C. Baquero, V. Fonte, Interval tree clocks, in: Proc. 12th International Conference on Principles of Distributed Systems, OPODIS, 2008, pp. 259–274, https://doi.org/10.1007/978-3-540-92221-6_18.

[15] S.S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, M. Leone, Logical physical clocks, in: Proc. 18th International Conference on Principles of Distributed Systems, OPODIS, 2014, pp. 17–32, https://doi.org/10.1007/978-3-319-14472-6_2.

[16] A. Misra, A.D. Kshemkalyani, The bloom clock for causality testing, in: D. Goswami, T.A. Hoang (Eds.), Proc. 17th International Conference on Distributed Computing and Internet Technology, in: Lecture Notes in Computer Science, vol. 12582, Springer, 2021, pp. 3–23, https://doi.org/10.1007/978-3-030-65621-8_1.

[17] A.D. Kshemkalyani, A. Misra, The bloom clock to characterize causality in distributed systems, in: The 23rd International Conference on Network-Based Information Systems, NBiS 2020, in: Advances in Intelligent Systems and Computing, vol. 1264, Springer, 2020, pp. 269–279, https://doi.org/10.1007/978-3-030-57811-4_25.

[18] M. Singhal, A.D. Kshemkalyani, An efficient implementation of vector clocks, Inf. Process. Lett. 43 (1) (1992) 47–52, https://doi.org/10.1016/0020-0190(92)90028-T.

[19] A.D. Kshemkalyani, M. Shen, B. Voleti, Prime clock: encoded vector clock to characterize causality in distributed systems, J. Parallel Distrib. Comput. 140 (2020) 37–51, https://doi.org/10.1016/j.jpdc.2020.02.008.

[20] T. Pozzetti, A.D. Kshemkalyani, Resettable encoded vector clock for causality analysis with an application to dynamic race detection, IEEE Trans. Parallel Distrib. Syst. 32 (4) (2021) 772–785, https://doi.org/10.1109/TPDS.2020.3032293.

[21] A. Misra, A.D. Kshemkalyani, Detecting causality in the presence of Byzantine processes: there is no holy grail, in: 21st IEEE International Symposium on Network Computing and Applications (NCA), 2022, pp. 73–80, https://doi.org/10.1109/NCA57778.2022.10013644.
[22] K.P. Birman, T.A. Joseph, Reliable communication in the presence of failures, ACM Trans. Comput. Syst. 5 (1) (1987) 47–76, https://doi.org/10.1145/7351.7478.
[23] A. Misra, A.D. Kshemkalyani, Solvability of Byzantine fault-tolerant causal ordering problems, in: M.-A. Koulali, M. Mezini (Eds.), Networked Systems, Springer International Publishing, Cham, 2022, pp. 87–103, https://doi.org/10.1007/978-3-031-17436-0_7.
[24] A. Misra, A.D. Kshemkalyani, Byzantine fault-tolerant causal ordering, in: 24th International Conference on Distributed Computing and Networking (ICDCN), 2023, pp. 100–109, https://doi.org/10.1145/3571306.3571395.
[25] A. Misra, A.D. Kshemkalyani, Causal ordering in the presence of Byzantine processes, in: 28th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2022, pp. 130–138, https://doi.org/10.1109/ICPADS56603.2022.00025.
[26] A. Misra, A.D. Kshemkalyani, Byzantine-tolerant causal ordering for unicasts, multicasts, and broadcasts, IEEE Trans. Parallel Distrib. Syst. 35 (5) (2024) 814–828, https://doi.org/10.1109/TPDS.2024.3368280.
[27] F.B. Schneider, Implementing fault-tolerant services using the state machine approach: a tutorial, ACM Comput. Surv. 22 (4) (1990) 299–319, https://doi.org/10.1145/98163.98167.
[28] T. Freitas, J. Soares, M.E. Correia, R. Martins, Deterministic or probabilistic? - A survey on Byzantine fault tolerant state machine replication, Comput. Secur. 129 (2023) 103200, https://doi.org/10.1016/J.COSE.2023.103200.
[29] G. Bracha, S. Toueg, Asynchronous consensus and broadcast protocols, J. ACM 32 (4) (1985) 824–840, https://doi.org/10.1145/4221.214134.
[30] G. Bracha, Asynchronous Byzantine agreement protocols, Inf. Comput. 75 (2) (1987) 130–143, https://doi.org/10.1016/0890-5401(87)90054-X.
[31] D. Dolev, H.R. Strong, Authenticated algorithms for Byzantine agreement, SIAM J. Comput. 12 (4) (1983) 656–666, https://doi.org/10.1137/0212045.
[32] M.C. Pease, R.E. Shostak, L. Lamport, Reaching agreement in the presence of faults, J. ACM 27 (2) (1980) 228–234, https://doi.org/10.1145/322186.322188.
[33] L. Lamport, R.E. Shostak, M.C. Pease, The Byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401, https://doi.org/10.1145/357172.357176.
[34] C. Dwork, N.A. Lynch, L.J. Stockmeyer, Consensus in the presence of partial synchrony, J. ACM 35 (2) (1988) 288–323, https://doi.org/10.1145/42282.42283.
[35] M.J. Fischer, N.A. Lynch, M.S. Paterson, Impossibility of distributed consensus with one faulty process, J. ACM 32 (2) (1985) 374–382, https://doi.org/10.1145/3149.214121.
[36] M. Castro, B. Liskov, Practical Byzantine fault tolerance, in: M.I. Seltzer, P.J. Leach (Eds.), Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), 1999, pp. 173–186, https://dl.acm.org/citation.cfm?id=296824.
[37] V. Shoup, R. Gennaro, Securing threshold cryptosystems against chosen ciphertext attack, J. Cryptol. 15 (2) (2002) 75–96, https://doi.org/10.1007/s00145-001-0020-9.
[38] D. Imbs, M. Raynal, Trading off t-resilience for efficiency in asynchronous Byzantine reliable broadcast, Parallel Process. Lett. 26 (04) (2016) 1650017, https://doi.org/10.1142/S0129626416500171.
[39] A. Misra, A.D. Kshemkalyani, Solvability of Byzantine fault-tolerant causal ordering: synchronous systems case, in: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, ACM, 2024, pp. 251–256, https://doi.org/10.1145/3605098.3636063.
[40] A. Misra, A.D. Kshemkalyani, Towards stronger blockchains: security against front-running attacks, in: Networked Systems, Springer International Publishing, Cham, 2024, https://doi.org/10.48550/arXiv.2311.10253.
[41] A.D. Kshemkalyani, Temporal interactions of intervals in distributed systems, J. Comput. Syst. Sci. 52 (2) (1996) 287–298, https://doi.org/10.1006/jcss.1996.0022.
[42] A.D. Kshemkalyani, A fine-grained modality classification for global predicates, IEEE Trans. Parallel Distrib. Syst. 14 (8) (2003) 807–816, https://doi.org/10.1109/TPDS.2003.1225059.
[43] P. Chandra, A.D. Kshemkalyani, Causality-based predicate detection across space and time, IEEE Trans. Comput. 54 (11) (2005) 1438–1453, https://doi.org/10.1109/TC.2005.176.
[44] P. Chandra, A.D. Kshemkalyani, Data-stream-based global event monitoring using pairwise interactions, J. Parallel Distrib. Comput. 68 (6) (2008) 729–751, https://doi.org/10.1016/j.jpdc.2008.01.006.
[45] A.D. Kshemkalyani, A framework for viewing atomic events in distributed computations, Theor. Comput. Sci. 196 (1–2) (1998) 45–70, https://doi.org/10.1016/S0304-3975(97)00195-3.
[46] A.D. Kshemkalyani, Reasoning about causality between distributed nonatomic events, Artif. Intell. 92 (1–2) (1997) 301–315, https://doi.org/10.1016/S0004-3702(97)00004-0.
[47] A.D. Kshemkalyani, R. Kamath, Orthogonal relations for reasoning about posets, Int. J. Intell. Syst. 17 (12) (2002) 1101–1110, https://doi.org/10.1002/int.10062.
[48] A.D. Kshemkalyani, Causality and atomicity in distributed computations, Distrib. Comput. 11 (4) (1998) 169–189, https://doi.org/10.1007/s004460050048.