Detecting Causality in the Presence of Byzantine Processes: The Synchronous Systems Case

Anshuman Misra
University of Illinois at Chicago, IL, USA

Ajay D. Kshemkalyani[1]
University of Illinois at Chicago, IL, USA

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. The positive result holds for unicast, multicast, as well as broadcast modes of communication. We prove the result by providing an algorithm. Our solution uses the Replicated State Machine (RSM) approach and vector clocks.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Networks → Network algorithms

Keywords and phrases Byzantine fault-tolerance, causality, happens before, distributed system, message-passing, synchronous system

Digital Object Identifier 10.4230/LIPIcs.TIME.2023.11

1 Introduction

1.1 Background

Causality is an important tool in understanding and reasoning about distributed executions [32]. Lamport formulated the "happens before" or the causality relation, denoted $\rightarrow$, between events in a distributed system [20]. 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 [18].

The causality relation between events can be captured by tracking causality graphs [7], scalar clocks [20], vector clocks [5, 8, 22], and several other variants of logical clocks such as hierarchical clocks [35], plausible clocks [34], dotted version vectors [30], interval tree clocks [1], logical physical clocks [19], Bloom clocks [16, 23], incremental clocks [33], and resettable prime clocks [17, 29]. 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 [32], the search for the holy grail of the ideal causality tracking mechanism is on. These above works in the literature assume that processes are correct (non-faulty). The causality detection problem for a system with Byzantine processes was recently introduced and studied in [25].

[1] Corresponding author

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 m' should not be delivered before m at all the common destinations of m and m'. Under the Byzantine failure model, causal ordering has recently been studied in [2] for broadcast communication and in [24, 26, 27] for unicast, multicast, as well as broadcast communication.

1.2 Contributions

It was recently 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 [25]. 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 [25] 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. 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.
2. We prove that causality detection between events can be solved in the presence of Byzantine processes in a synchronous system. We provide an algorithm that solves the causality detection problem. The positive result holds for unicasts, multicasts, as well as broadcasts. Our solution uses the Replicated State Machine (RSM) approach [31], which works only in synchronous systems, in conjunction with vector clocks.
3. This is the first paper to establish this result. The paper uses a simple combination of RSMs and vector clocks and is yet significant, similar to results in [8, 22, 32], because it establishes a fundamental possibility result about causality detection in the presence of Byzantine processes in a synchronous system.
4. The results for multicasts, unicasts, and broadcasts are summarized in Table 1. 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.

Roadmap. Section 2 gives the system model. Section 3 formulates the problem of detecting causality in the presence of Byzantine processes. Section 4 proves the results outlined under "Contributions" above. Section 5 gives a discussion and concludes.

Table 1 Detecting causality 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.

Mode of communication	Detecting "happens before" in asynchronous systems	Detecting "happens before" in synchronous systems
Multicasts	Impossible [25] FP, FN	Possible, Theorem 11 $\overline{FP}, \overline{FN}$
Unicasts	Impossible [25] FP, FN	Possible, Corollary 12 $\overline{FP}, \overline{FN}$
Broadcasts	Impossible [25] $\overline{FP}, FN$	Possible, Corollary 13 $\overline{FP}, \overline{FN}$

2 System Model

This paper deals with a distributed system having Byzantine processes which are processes that can misbehave [21, 28]. A correct process behaves exactly as specified by the algorithm whereas a Byzantine process may exhibit arbitrary behaviour including crashing at any point during the execution. A Byzantine process cannot impersonate another process or spawn new processes.

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 [6]. 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 [6]. A synchronous system guarantees that the relative speeds of non-faulty processors and messages is bounded, and this is equivalent to assuming that the system has synchronized real-time clocks [31].

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 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* [20] 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 $\{e' \in T(E) \,|\, e' \rightarrow e\}$.*

3 Problem Formulation

The problem formulation is done similar to the way in [25]. 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 and let $F = \bigcup_i \{F_i\}$. F thus denotes the execution history of the system as perceived and collected by the algorithm. 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. 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 $T(E) \supseteq CP(e_i^*)$.

Byzantine processes may collude 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.

▶ **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*

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

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).

Using the state-machine replication approach, 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.

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 [31]. 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 $2t + 1$ 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 [21, 28]; 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 [9]. 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 $smr_{j,w}$. A message is defined to be *stable* at $smr_{j,w}$ 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 $smr_{j,w}$. 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 [31]. 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 $p_{i,a}$, i.e., $smr_{i,a}$, 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 $smr_{i,a}$ *SR-delivers* such a message, then every other correct receiver replica $smr_{i,y}$ in that ensemble will also *SR-deliver* that same message M in exactly the order and sequence it was *SR-delivered* by $smr_{i,a}$. 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 $smr_{i,a}$ and that differ from the majority value of M received $t+1$ times by $smr_{i,a}$.

When $smr_{i,a}$ sends a message to p_j at the application level, it sends it to all replicas $smr_{j,b}$. When $smr_{i,a}$ *SR-delivers* a message, a receive event is said to have occurred at the application level. Henceforth, we also refer to $smr_{i,a}$ as $p_{i,a}$ and 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 $e_h^x \to e_i^*|_F$. Theorem 9 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 11 that using the output of the algorithm lines 22-26 function `test`, and Theorem 9, 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 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)
```

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 latest send event by p_k (as per/from the local F_k) at the point in time of the last send 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.
4. $p_{i,a}$ also maintains an auxiliary integer matrix $V[|T(F_i)|, n]$, where $V[s, k]$ is $maxeventID$-(F_k) in $F(e^s_{i,a})$, i.e., the highest sequence number in $F_k(\in F)$ when the sth local event $e^s_{i,a}$ 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 [8, 22] of event $e^w_{i,a}$ 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 $e^w_{i,a}$ 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 $e^x_h \to e^*_i$, viz., $V[*, h] \geq x$.

4.4 Correctness Proof

Events such as e^x_h 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 $e^x_{h,a}$ with two subscripts denote events at $smr_{h,a}$, i.e., $p_{h,a}$, the individual state machine 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 $\{e^x_h\}$ such that the events $e^x_{h,a}$ have occurred at at least* majority *$(= t + 1)$ number of processes $p_{h,a}$.*

▶ **Definition 5.** *The happens before relation $\to_{RSM}$ on events in E_RSM (which occur in ensembles of RSMs) consists of the following rules:*
1. ***Program Order**: For the sequence of events $\langle e^1_i, e^2_i, \ldots\rangle$ executed by RSM ensemble process p_i, $\forall\ x, y$ such that $x < y$ we have $e^x_i \to_{RSM} e^y_i$.*
2. ***Message Order**: If event e^y_j 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^x_i in RSM ensemble p_i (i.e., there are at least a* majority *events $e^x_{i,a}$ that are the corresponding message send events at processes $p_{i,a}$ to RSM ensemble p_j), we have $e^x_i \to_{RSM} e^y_j$.*
3. ***Transitive Order**: If $e \to_{RSM} e' \wedge e' \to_{RSM} e''$ then $e \to_{RSM} e''$.*

▶ **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 $\{e' \in E\_RSM \mid e' \to_{RSM} e\}$.*

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

▶ **Lemma 7.** *An event $e_h^x \in E\_RSM$ occurs at each correct process $p_{h,z}$ in the RSM ensemble p_h.*

Proof. By definition, an event $e_h^x \in E\_RSM$ occurs at at least *majority* ($= t + 1$) processes $p_{h,a}$ in the RSM ensemble p_h. As at least one of these *majority* processes $p_{h,a'}$ must be correct and executes $e_{h,a'}^x$, and from the Agreement/Agreement-M and Total Order properties of the RSM, each correct *smr* $p_{h,z}$ will behave identically to $p_{h,a'}$ and will execute $e_{h,z}^x$. ◀

▶ **Lemma 8.** *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.*

Proof. As the RSM ensemble works in perfect unision, an event $e_{h,z}^x$ that occurs at a correct process $p_{h,z}$ also occurs at all the correct replicas in RSM ensemble p_h and thus at at least *majority* replicas in that ensemble. Then by Definition 4, the event is also said to occur as e_h^x in RSM ensemble p_h. ◀

▶ **Theorem 9.** *For an event e at a RSM p_i (e must occur at each correct process $p_{i,z}$ by Lemma 7), the set of events $T(F)$ when e is executed at each correct $p_{i,z}$ is $CP\_RSM(e)$.*

Proof. There are two parts to this theorem.

1. If an event belongs to $CP\_RSM(e)$, the event must belong to $T(F)$ when event e is executed at correct process $p_{i,z}$.
   If event $e_h^x \in CP\_RSM(e)$ then e_h^x must be genuine (not fake) and there is a "RSM-causal path" from e_h^x to e in $(E\_RSM, \rightarrow_{RSM})$. Such a RSM-causal path is comprised of program order edges and message order edges. For each message order edge under $\rightarrow_{RSM}$ corresponding to a message hop along such a causal path, there are at least *majority* ($= t + 1$) edges under $\rightarrow$ from each of the at least *majority* ($= t + 1$) correct processes in the sender RSM ensemble to the at least *majority* ($= t + 1$) correct processes in the receiver RSM ensemble. Furthermore, for each program order edge at p_j under $\rightarrow_{RSM}$ along the RSM-causal path, all the correct processes in the p_j ensemble preserve the content of F at the previous event along the corresponding program order edge under $\rightarrow$. Information about e_h^x gets propagated via inc_F_h and F_h along all such message order edges and program order edges through the processes p_j along the "RSM-causal path" from e_h^x to e and gets inserted in F_h at $p_{i,z}$, as can be seen from lines (1-6) (for unicasts) or (7-12) (for multicasts), and lines (13-18). Once an entry is inserted in F_h at a correct process, it is never deleted. Note, event $e \in E\_RSM$ and is specifically some event e_i^y that, by Lemma 7, must also occur as $e_{i,z}$ (or $e_{i,z}^y$) at each correct process $p_{i,z}$. Thus $e_h^x \in CP\_RSM(e)$ implies e_h^x is in F_h when e is executed at $p_{i,z}$.
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 8), the event e' must belong to $CP\_RSM(e)$.
   For event e_h^x in F_h when $e_{i,z}^*$ occurs at $p_{i,z}$, there are two cases: $h = i$ and $h \neq i$.
   a. First consider $h = i$. $e_{i,z}^x$ (as e_i^x) must have been inserted in F_i at $e_{i,z}^x$ only in line 3 (for unicast send event) or line 9 (for multicast send event) or line 18 (for receive event) or line 21 (for internal event), and is never deleted by the correct process $p_{i,z}$ as per the algorithm code. Clearly by Lemma 8, $e_{i,z}^x$ and $e_{i,z}^*$ and all events in between at $p_{i,z}$ must have occurred at RSM ensemble process p_i as $p_{i,z}$ is a correct process. As there exist program order edges under $\rightarrow_{RSM}$ from e_i^x to e_i^* and $p_{i,z}$ is a correct process, it must be that $e_{h=i}^x \in CP\_RSM(e_i^*)$.

b. Consider now $h \neq i$. Set y to $*$, i' to i, z' to z.
As event e_h^x is in F_h when $e_{i',z'}^y$ occurs at correct process $p_{i',z'}$, then e_h^x could have been inserted only in line 16 on the *SR-delivery* of a message m at an event $e1_{i',z'}$ in the causal past of $e_{i',z'}^y$ (along program order edges under $\rightarrow$ or under $\rightarrow_{RSM}$) that resulted from the *TOA-delivery* from some (at least *majority*) of processes $p_{j,a}$ that sent at event e_j^w. As at least *majority* ($= t+1$) processes in the preceding sender RSM ensemble reported the same inc_F in the same message for the message to have been *SR-delivered* to $p_{i,z}$, the impact of Byzantine processes in the ensemble is filtered out. By Lemma 8 note that receive event $e1_{i'}$ occurs at RSM ensemble $p_{i'}$ because $e1_{i',z'}$ occurs at a correct process. Also, send event e_j^w at RSM p_j must have occurred as at least a *majority* of processes $p_{j,a}$ sent m, and so e_j^w must have occurred at all correct processes $p_{j,c}$ in the ensemble p_j. The message m corresponds to a message order edge under $\rightarrow_{RSM}$. Moreover the send event e_j^w at RSM p_j belongs to $CP\_RSM(e_{i'}^y)$ (and hence to $CP\_RSM(e_{i,z}^* = e)$ by transitivity because $e_{i'}^y \in CP\_RSM(e)$ as per the previous invocation (if any) of this case 2b). e_h^x must have existed in F_h at the time these correct $p_{j,c}$ sent m at $e_{j,c}^w$. There are two subcases: $j \neq h$ and $j = h$.

  i. $j \neq h$. Invoke case 2b but with y set to w, i' set to j, z' set to c. This case gets invoked at most $n-1$ times as there are $n-1$ processes (RSMs) p_j ($j \neq h$) in the system and e_h^x gets added to F_h at correct replica processes c of any particular p_j at most once.

  ii. $j = h$: e_h^x must have been inserted in F_h at $e_{h(=j),c}^x$ in line 3 (for unicast send event) or line 9 (for multicast send event) or in line 18/21 at a receive/internal event, at p_h, i.e., at each correct $p_{j,c}$. Clearly by Lemma 8, $e_{j,c}^x$ and $e_{j,c}^w$ and all events in between at $p_{j,c}$ must have occurred at RSM ensemble process $p_{j(=h)}$ as $p_{j,c}$ is a correct process and hence there exist program order edges under $\rightarrow_{RSM}$ from $e_{j(=h)}^x$ to e_j^w. Moreover such an event e' (or e_h^x) must belong to $CP\_RSM(e_j^w)$ (and hence to $CP\_RSM(e_{i,z}^* = e)$ by transitivity because $e_j^w \in CP\_RSM(e)$ as shown above for case 2b of this invocation).

Combining transitively the above case invocations, it follows that event e_h^x is in F_h when $e = e_{i,z}^*$ is executed at $p_{i,z}$ implies $e_h^x \in CP\_RSM(e)$ and e_h^x cannot be fake.

Both parts of the theorem thus stand proved. ◀

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 10.** *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 $T(F(e_{i,z}^*)) \setminus E\_RSM \neq \emptyset$).

Algorithm 1 produces the output of $e_h^x \to e_i^*|_F$ at $p_{i,a}$ (lines 22-26) via recording F (lines 1-21). Theorem 9 showed that the set of events in E_RSM matched the set of events recorded in F, even though E_RSM is never recorded and is accessible only to an oracle. Next we show in Theorem 11 that using the output of the algorithm and Theorem 9, 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 11.** *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.*

Proof. This theorem has two parts – no false negatives and no false positives – and the proof leverages the two cases in the proof of Theorem 9 which cover the multicast mode of communication. Recall our assumption in Definition 10 that $p_{i,z}$ is a correct replica. By Lemma 8, 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 9 proof 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 $e_h^x \to e_{i,z}^*|_{E\_RSM} = 1 \wedge e_h^x \to e_{i,z}^*|_F = 0$. Hence there are no false negatives.
2. $T(F(e_{i,z}^*)) \setminus E\_RSM = \emptyset$. This follows from the second case of Theorem 9 proof 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 $e_h^x \to e_i^*$ 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 9 proof, $e_h^x \notin CP\_RSM(e_{i,z}^*)$ and the test correctly returns *false*. If e_h^x is in F_h, then by case 2 of Theorem 9 proof, $e_h^x \in CP\_RSM(e_{i,z}^*)$ 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 $e_h^x \to e_{i,z}^*|_{E\_RSM} = 0 \wedge e_h^x \to e_{i,z}^*|_F = 1$. Hence there are no false positives.

The theorem follows. ◀

As unicast and broadcast are special cases of multicast, the prevention of false positives and of false negatives for multicasts implies the prevention of false positives and of false negatives for unicasts and for broadcasts also. Thus we have the following corollaries to Theorem 11.

▶ **Corollary 12.** *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.*

▶ **Corollary 13.** *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.*

5 Discussion and Conclusions

We proposed a RSM-based algorithm for solving the causality determination problem CD in synchronous systems that can have at most nt Byzantine processes among a total of $(3t+1)n$ processes partitioned into n ensembles of $3t+1$ processes each with each ensemble 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. This is true irrespective of whether the communication mode is by unicasting, multicasting, or broadcasting. The RSM approach works only in synchronous systems. This result is in contrast to the impossibility result for solving the *CD* problem in asynchronous systems in the presence of even a single Byzantine process [25]. It would be interesting to determine whether the *CD* problem can be solved in synchronous systems in the presence of Byzantine processes using a direct approach without using RSMs.

Detecting causality between a pair of events is a fundamental problem [32]. 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 [10],
- detecting the fine-grained modality of a distributed predicate [3, 14], and data-stream based global event monitoring using pairwise interactions between processes [4],
- detecting causality relation between two "meta-events" [11, 13, 15], each of which spans multiple events across multiple processes [12].

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.

Byzantine-tolerant causal ordering of messages under unicast mode or multicast mode of communication has been proved to be unsolvable in asynchronous systems [26, 27]. Two forms of safety – *strong safety* (or unconditional safety) and a weaker form of safety called *weak safety* were defined [26], and it was also shown that Byzantine-tolerant causal ordering under broadcast mode of communication in asynchronous systems cannot satisfy strong safety [26] (in a system model in which cryptographic techniques are not allowed). Neither can the algorithm given in [2] for the broadcast mode of communication satisfy strong safety. Algorithms to provide weak safety and liveness of Byzantine-tolerant causal ordering were provided for synchronous systems in [24, 27] (implicitly for unicast mode, multicast mode, and broadcast mode). The use of the RSM approach can be seen to implicitly provide strong safety and liveness of Byzantine-tolerant causal ordering (of unicast mode, multicast mode, and broadcast mode of communication) in synchronous systems as order requirement O2 (Causal order) of the RSM specification is satisfied.

References

1 Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In *Proc. 12th International Conference on Principles of Distributed Systems, OPODIS*, pages 259–274, 2008. doi:10.1007/978-3-540-92221-6_18.

2 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Byzantine-tolerant causal broadcast. *Theoretical Computer Science*, 885:55–68, 2021.

3 Punit Chandra and Ajay D. Kshemkalyani. Causality-based predicate detection across space and time. *IEEE Trans. Computers*, 54(11):1438–1453, 2005. doi:10.1109/TC.2005.176.

4 Punit Chandra and Ajay D. Kshemkalyani. Data-stream-based global event monitoring using pairwise interactions. *J. Parallel Distributed Comput.*, 68(6):729–751, 2008. doi:10.1016/j.jpdc.2008.01.006.

5 Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991. doi:10.1016/0020-0190(91)90055-M.

6 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

7 E.N. Elnozahy. Manetho: Fault tolerance in distributed systems using rollback-recovery and process replication, phd thesis. Technical report, Tech. Report 93-212, Computer Science Department, Rice University, 1993.

8 Colin J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991. `doi:10.1109/2.84874`.

9 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

10 Ajay D. Kshemkalyani. Temporal interactions of intervals in distributed systems. *J. Comput. Syst. Sci.*, 52(2):287–298, 1996. `doi:10.1006/jcss.1996.0022`.

11 Ajay D. Kshemkalyani. Reasoning about causality between distributed nonatomic events. *Artif. Intell.*, 92(1-2):301–315, 1997. `doi:10.1016/S0004-3702(97)00004-0`.

12 Ajay D. Kshemkalyani. Causality and atomicity in distributed computations. *Distributed Comput.*, 11(4):169–189, 1998. `doi:10.1007/s004460050048`.

13 Ajay D. Kshemkalyani. A framework for viewing atomic events in distributed computations. *Theor. Comput. Sci.*, 196(1-2):45–70, 1998. `doi:10.1016/S0304-3975(97)00195-3`.

14 Ajay D. Kshemkalyani. A fine-grained modality classification for global predicates. *IEEE Trans. Parallel Distributed Syst.*, 14(8):807–816, 2003. `doi:10.1109/TPDS.2003.1225059`.

15 Ajay D. Kshemkalyani and Roshan Kamath. Orthogonal relations for reasoning about posets. *Int. J. Intell. Syst.*, 17(12):1101–1110, 2002. `doi:10.1002/int.10062`.

16 Ajay D. Kshemkalyani and Anshuman Misra. The bloom clock to characterize causality in distributed systems. In *The 23rd International Conference on Network-Based Information Systems, NBiS 2020*, volume 1264 of *Advances in Intelligent Systems and Computing*, pages 269–279. Springer, 2020. `doi:10.1007/978-3-030-57811-4_25`.

17 Ajay D. Kshemkalyani, Min Shen, and Bhargav Voleti. Prime clock: Encoded vector clock to characterize causality in distributed systems. *J. Parallel Distributed Comput.*, 140:37–51, 2020. `doi:10.1016/j.jpdc.2020.02.008`.

18 Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems.* Cambridge University Press, 2011. `doi:10.1017/CBO9780511805318`.

19 Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *Proc. 18th International Conference on Principles of Distributed Systems, OPODIS*, pages 17–32, 2014. `doi:10.1007/978-3-319-14472-6_2`.

20 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21, 7*, pages 558–565, 1978.

21 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. `doi:10.1145/357172.357176`.

22 Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1988.

23 Anshuman Misra and Ajay D. Kshemkalyani. The bloom clock for causality testing. In Diganta Goswami and Truong Anh Hoang, editors, *Proc. 17th International Conference on Distributed Computing and Internet Technology*, volume 12582 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2021. `doi:10.1007/978-3-030-65621-8_1`.

24 Anshuman Misra and Ajay D. Kshemkalyani. Causal ordering in the presence of byzantine processes. In *28th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2022. `doi:10.1109/ICPADS56603.2022.00025`.

25 Anshuman Misra and Ajay 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)*, pages 73–80, 2022. `doi:10.1109/NCA57778.2022.10013644`.

26 Anshuman Misra and Ajay D. Kshemkalyani. Solvability of byzantine fault-tolerant causal ordering problems. In Mohammed-Amine Koulali and Mira Mezini, editors, *Networked Systems*, pages 87–103, Cham, 2022. Springer International Publishing. `doi:10.1007/978-3-031-17436-0_7`.

27 Anshuman Misra and Ajay D. Kshemkalyani. Byzantine fault-tolerant causal ordering. In *24th International Conference on Distributed Computing and Networking (ICDCN)*, pages 100–109, 2023. doi:10.1145/3571306.3571395.

28 Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

29 Tommaso Pozzetti and Ajay D. Kshemkalyani. Resettable encoded vector clock for causality analysis with an application to dynamic race detection. *IEEE Trans. Parallel Distributed Syst.*, 32(4):772–785, 2021. doi:10.1109/TPDS.2020.3032293.

30 Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors. In *ACM Symposium on Principles of Distributed Computing, PODC*, pages 335–336, 2012. doi:10.1145/2332432.2332497.

31 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. doi:10.1145/98163.98167.

32 Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Comput.*, 7(3):149–174, 1994. doi:10.1007/BF02277859.

33 Mukesh Singhal and Ajay D. Kshemkalyani. An efficient implementation of vector clocks. *Inf. Process. Lett.*, 43(1):47–52, 1992. doi:10.1016/0020-0190(92)90028-T.

34 Francisco J. Torres-Rojas and Mustaque Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–195, 1999. doi:10.1007/s004460050065.

35 Paul A. S. Ward and David J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS 2001)*, pages 585–593, 2001. doi:10.1109/ICDSC.2001.918989.