Byzantine-Tolerant Causal Ordering for Unicasts, Multicasts, and Broadcasts

Anshuman Misra and Ajay D. Kshemkalyani, *Senior Member, IEEE*

***Abstract*—Byzantine fault-tolerant causal ordering of messages is useful to many applications. Causal ordering requires a property that we term strong safety, and liveness. In this paper, we use execution histories to prove that it is impossible to solve causal ordering – strong safety and liveness – in a deterministic manner for unicasts, multicasts, and broadcasts in an asynchronous system with one or more Byzantine processes. We also define a weaker version of strong safety termed weak safety. We prove that it is impossible to solve causal ordering – weak safety and liveness – in a deterministic manner for unicasts and multicasts, in an asynchronous system with one or more Byzantine processes. In view of these impossibility results, we propose the Sender-Inhibition algorithm and the Channel Sync algorithm to provide causal ordering – weak safety and liveness – of unicasts under the Byzantine failure model in synchronous systems, which have a known upper bound on message latency. The algorithms operate under the synchronous system model, but are inherently asynchronous and offer a high degree of concurrency as lock-step communication is not assumed. The two algorithms provide different trade-offs. We also indicate how the algorithms can be extended to multicasts.**

***Index Terms*—Byzantine fault-tolerance, causal order, multicast, broadcast, asynchronous system.**

I. Introduction

CAUSALITY provides important application-level semantics to distributed programs. Causality is defined by the *happens before* [1] relation on the set of events, and by extension, on the set of messages. If message m_1 causally precedes m_2 and both are sent to p_i, then m_2 cannot be delivered before m_1 at p_i to enforce causal order [2]. This property is the *strong safety* property of causal ordering. An additional property is *liveness* which requires that a message from a correct process to another correct process is eventually delivered. Causal ordering ensures that causally related updates to data occur in a valid manner respecting that causal relation. Applications of causal ordering include distributed data stores, fair resource allocation, and collaborative applications such as multiplayer online gaming, social networks, event notification systems, group editing of documents, and distributed virtual environments.

It is important to solve causal ordering under the Byzantine failure model because it mirrors the real world. Byzantine-tolerant causal ordering of broadcasts was studied in [3]. Byzantine-tolerant causal ordering for unicasts or multicasts has not been considered besides the recent analysis in [4], [5], [6].

Contributions:

1) We prove using execution histories that causal ordering – strong safety and liveness – of unicasts, multicasts, and broadcasts using a deterministic algorithm in an asynchronous system with even one Byzantine process is impossible. We provide a weakening of strong safety that we term *weak safety*. Weak safety requires that if m_1 causally precedes m_2 and there is a causal path from the send event of m_1 to the send event of m_2 passing through only correct processes in the execution, then m_2 should not be delivered before m_1 at all common destinations of m_1 and m_2. We prove that it is impossible to provide causal ordering – weak safety and liveness – of unicasts and multicasts using a deterministic algorithm in an asynchronous system with even one Byzantine process. All the above results are disallowing cryptography.
We then prove that even with cryptography, it is impossible to provide strong safety and liveness in a deterministic manner for unicasts, multicasts, and broadcasts in a Byzantine failure prone system. However, we can provide weak safety and liveness in a deterministic manner with the help of cryptography.
2) The above results are tabulated in Table I. A main contribution in the results' proofs is to show a reduction from the consensus problem to the causal ordering problem, thus establishing the impossibility of solving causal ordering. In Section VII, we prove that causal ordering does not reduce to consensus, i.e., causal ordering cannot be solved even if consensus were solvable and hence causal ordering is harder than consensus.
3) In view of the above impossibility results for asynchronous systems, we show that a deterministic cryptography-free solution for weak safety and liveness for unicasts can be designed in a synchronous system. The strengthening is in the form of a known upper bound δ on message latency, and also a known upper bound ψ on the relative speeds of processors. Specifically, we propose two algorithms.
    a) We propose the Sender-Inhibition algorithm for weak safety and liveness of unicasts. The algorithm is simple to understand and implement. However send events at a process are blocking with respect to each other. This means that a process can initiate a message send only after the previous message it sent has been received at

Manuscript received 22 September 2023; revised 1 January 2024; accepted 17 February 2024. Date of publication 21 February 2024; date of current version 21 March 2024. Recommended for acceptance by J. Xu. *(Corresponding author: Ajay D. Kshemkalyani.)*

The authors are with the Department of Computer Science, University of Illinois Chicago, Chicago, IL 60607 USA (e-mail: amisra7@uic.edu; ajay@uic.edu).

Digital Object Identifier 10.1109/TPDS.2024.3368280

TABLE I
SOLVABILITY OF CAUSAL ORDERING USING DETERMINISTIC ALGORITHMS IN ASYNCHRONOUS SYSTEMS UNDER DIFFERENT COMMUNICATION MODES

Mode of communication	SS + L	SS + L with cryptography	WS + L	WS + L with cryptography
Unicasts	No, Theorem 1 $\overline{SS}, \overline{L}$	No, Theorem 8 $\overline{SS}, L$	No, Theorem 4 $WS, \overline{L}$	Yes, Corollary 1 WS, L
Broadcasts	No, Theorem 2 $\overline{SS}, L$	No, Theorem 9 $\overline{SS}, L$	Yes, Theorem 5 WS, L	Yes, Corollary 2 WS, L
Multicasts	No, Theorem 3 $\overline{SS}, \overline{L}$	No, Theorem 7 $\overline{SS}, L$	No, Theorem 6 $WS, \overline{L}$	Yes, Theorem 10 WS, L

SS = strong safety, WS = weak safety, L = liveness, $\overline{SS}$, $\overline{WS}$, $\overline{L}$ represent that strong safety, weak safety, liveness, respectively, cannot be guaranteed.

TABLE II
TABLE OF MAIN ACRONYMS AND NOTATIONS

Acronym/Notation	Full form or Explanation
$\rightarrow$ (Definitions 1, 2)	"Happens before" relation on events or messages
$CP(m)$ (Definition 3)	Causal past of message m
$\xrightarrow{B}$ (Definition 4)	"Byzantine happens before" relation on messages
$BCP(m)$ (Definition 5)	Byzantine causal past of message m
BRM (Definition 6)	Byzantine Reliable Multicast
BRU (Definition 7)	Byzantine Reliable Unicast
BRB (Definition 8)	Byzantine Reliable Broadcast
BCM	Byzantine Causal Multicast
BCU	Byzantine Causal Unicast
BCB	Byzantine Causal Broadcast
SS (Definition 9)	Strong safety
WS (Definition 10)	Weak safety
L (Definitions 9, 10)	Liveness
E_i	actual execution history at p_i
E	$\bigcup_i \{E_i\}$
F_i	execution history at p_i as recorded by the algorithm
F	$\bigcup_i \{F_i\}$
$M(E)$	messages sent and/or received in E
$M(F)$	messages sent and/or received in F
$CO(E, F, m_2)$ (Definition 11)	Causal Ordering problem for strong safety + liveness
$CO\_B(E, F, m_2)$ (Definition 13)	Causal Ordering problem for weak safety + liveness
$(M(E))_c$	messages of $M(E)$ sent by correct processes
G	multicast group
K_G	group key
C_m	ciphertext of message m
δ	fixed upper bound on the message latency
δ_r	timer wait time for receive control message
δ_s	timer wait time for send control message
cmr	control message for receive event
cms	control message for send event

the destination. The algorithm eliminates the $O(n^2)$ message space and time overhead of [2], [7], [8], [9], [10], where n is the number of processes in the system, and uses one control message of size $O(1)$ per application message sent.

b) We propose the Channel Sync algorithm for Byzantine-tolerant causal ordering of unicasts in a synchronous system. This algorithm uses $2(n-2)$ control messages of size $O(1)$ each, per application message. This algorithm allows complete concurrency in the execution. The implementation uses n queues per process. We prove the correctness of the algorithm and bound the time a message can spend in a queue, despite the presence of Byzantine processes in the system.

We also indicate how the Sender-Inhibition algorithm and the Channel Sync algorithm can be extended for multicasts.

Table II tabulates the main acronyms and notations used.

The Sender-Inhibition algorithm is based on [5], and the Channel Sync algorithm is based on [6]. Some portions of the solvability results are based on [6]. This paper has been greatly expanded, it has fully reworked proofs for the solvability results without cryptography, and the solvability results, theorems, and proofs using cryptography are entirely new. The result that causal ordering does not reduce to consensus is also entirely new.

Roadmap: Section II reviews related work. Section III gives the system model. Section IV gives the main results about the solvability of Byzantine causal unicast, multicast, and broadcast in a deterministic manner in an asynchronous system. We consider both cases – without cryptography and with cryptography. For a synchronous system where there is a known upper bound on the message latency, Sections V and VI present the Sender-Inhibition algorithm and the Channel Sync algorithm for solving Byzantine causal unicast – weak safety and liveness – in a deterministic cryptography-free manner. The correctness proofs are also given. Section VII gives a discussion and Section VIII concludes.

II. Related Work

Algorithms for causal ordering of unicast messages in an asynchronous setting under a fault-free model have been proposed, e.g., in [9], [11]. Algorithms for causal ordering of point-to-point messages in real-time applications have been proposed in [12], [13]. Algorithms for causal multicasts in a failure-free setting are given in [7], [8]. The above algorithms append control information to application messages. The algorithm in [14] for the same setting does broadcast via flooding on a overlay topology and no control information is used.

There has been some work on causal broadcasts under various failure models. Causal ordering of broadcast messages under crash failures in asynchronous systems was introduced in [2]. This algorithm required each message to carry the entire set of messages in its causal past as control information. The algorithm presented in [15] implements crash fault-tolerant causal broadcast in asynchronous systems with a focus on optimizing the amount of control information piggybacked on each message. An algorithm for causally ordering broadcast messages – providing only weak safety and liveness – in an asynchronous system with Byzantine failures is proposed in [3]. The feasibility of solving Byzantine causal order for unicasts, multicasts, and broadcasts was analyzed in [4]. Previously, a *probabilistic* algorithm based on atomic (total order) broadcast and cryptography for secure causal atomic broadcast (liveness and strong safety) in an asynchronous system was proposed [16]. This logic used acknowledgements and effectively processed the atomic broadcasts serially. More recently for the client-server configuration, two protocols for crash failures and a third for Byzantine failure of clients based on cryptography were proposed for secure causal atomic broadcast [17]. The third made assumptions on latency of messages, and hence works only in a synchronous system.

Recently there has been some work on the related problem of implementing Byzantine-tolerant causal consistency in distributed shared memory and replicated databases [18], [19], [20]; these approaches relied on broadcast communication. In [18], Byzantine Reliable Broadcast (BRB) [21] is used to remove misinformation induced by the combination of asynchrony and Byzantine behaviour. In [19], PBFT (total order broadcast) [22] is used to achieve consensus among non-Byzantine servers regarding the order of client requests. In [20], Byzantine causal broadcast has been used to implement Byzantine eventual consistency.

To the best of our knowledge, no paper has considered, analyzed, or attempted to solve causal ordering of unicasts and multicasts in a deterministic manner in an asynchronous system with Byzantine failures, besides our analysis of solvability [4].

III. System Model

This paper deals with a distributed system having Byzantine processes which are processes that can misbehave [23], [24]. 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 asynchronously in the distributed system. Let n be $|P|$. C is the set of FIFO (logical) communication links over which processes communicate by message passing. The communication links are reliable implying messages cannot get lost or be duplicated, and communication is authenticated. G is a complete graph.

While stating and proving our solvability results, the system is assumed to be asynchronous, i.e., there is no fixed upper bound δ on the message latency, nor any fixed upper bound ψ on the relative speeds of processors [25]. In contrast, a synchronous system has been defined as one in which both δ and ψ exist and are known [25]. As solving causal ordering is impossible in most circumstances (summarized in Table I), we give two deterministic cryptography-free algorithms for weak safety and liveness for a system where δ is known and used by the algorithms; the algorithms rely on timeouts which can use knowledge of ψ for accuracy. Thus, it can be said that the algorithms assume a synchronous system.

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 *execution history* at p_i is the finite execution at p_i up to the current or most recent or specified local state. The *happens before* [1] 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 on events 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, $\forall\ j, k$ such that $j < k$ we have $e_i^j \rightarrow e_i^k$.
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 $e \rightarrow e''$.

Next, we define the partial order happens before relation $\rightarrow$ on the set of all application-level messages R.

Definition 2: The happens before relation $\rightarrow$ on messages in R consists of the following rules:

1) If p_i sent or delivered message m before sending message m', then $m \rightarrow m'$.
2) If $m \rightarrow m'$ and $m' \rightarrow m''$, then $m \rightarrow m''$.

Definition 3: The *causal past* of message m is denoted as $CP(m)$ and defined as the set of messages in R that causally precede message m under $\rightarrow$.

We require an extension of the happens before relation on messages to accommodate the possibility of Byzantine behaviour. We present a partial order on messages called *Byzantine happens before*, denoted as $\xrightarrow{B}$, defined on S, the set of all application-level messages that are both sent by and delivered at correct processes in P.

Definition 4: The Byzantine happens before relation $\xrightarrow{B}$ on messages in S consists of the following rules:

1) If p_i is a correct process and p_i sent or delivered message m (to/from another correct process) before sending message m' to a correct process, then $m \xrightarrow{B} m'$.
2) If $m \xrightarrow{B} m'$ and $m' \xrightarrow{B} m''$, then $m \xrightarrow{B} m''$.

The Byzantine causal past of a message is defined as follows:

Definition 5: The *Byzantine causal past* of message m, denoted as $BCP(m)$, is defined as the set of messages in S that causally precede message m under $\xrightarrow{B}$.

We consider three possible modes of communication: multicast, unicast, and broadcast. Though well-understood, we define these next in the context of Byzantine fault-tolerance. In a multicast, a message is sent to a subset of processes forming a process group G. Different multicast send events can send to different process groups. In unicast, the process group consists of a single destination process. In broadcast, G is the set of all processes. In a multicast/unicast/broadcast, a message m is sent at a send event using send(m, G), send($m, \{p_i\}$), send(m, P), respectively, and is delivered at a receive event via deliver(m).

Definition 6: Byzantine Reliable Multicast (BRM) to group G satisfies the following properties:

1) (Validity:) If a correct process p_i delivers message m from $sender(m)$ sent to group G, then $sender(m)$ must have executed send(m, G) and $p_i \in G$.
2) (Self-delivery:) If a correct process executes send(m, G), then it eventually delivers m.
3) (Agreement:) If a correct process delivers a message m from a possibly faulty process, then all correct processes in G will eventually deliver m.
4) (Integrity:) For any message m, a correct process p_i delivers m at most once.
5) (No Information Leakage:) No process outside the group G sees the content of m.

Definition 7: Byzantine Reliable Unicast (BRU) to p_i satisfies the following properties:

1) (Validity:) If a correct process p_i delivers message m from $sender(m)$, then $sender(m)$ must have executed send($m, \{p_i\}$).
2) (Delivery:) If a correct process executes send($m, \{p_i\}$) and p_i is a correct process, then p_i eventually executes deliver(m).
3) (Integrity:) For any message m, a correct process p_i delivers m at most once.
4) (No Information Leakage:) No process besides the sender and receiver of m sees the content of m.

Definition 8: Byzantine-tolerant Reliable Broadcast (BRB) provides the following guarantees [21], [26]:

1) (Validity:) If a correct process delivers a message m from $sender(m)$, then $sender(m)$ must have executed send(m, P).
2) (Self-delivery:) If a correct process executes send(m, P), then it eventually delivers m.
3) (Agreement:) If a correct process delivers a message m from a possibly faulty process, then all correct processes eventually deliver m.
4) (Integrity:) For any message m, a correct process delivers m at most once.

It can be seen from the above three definitions that BRU and BRB are special cases of BRM. As a unicast has a single destination, the Agreement property of BRM goes away in BRU. As the destination set of a broadcast is the set of all processes, the No Information Leakage property of BRM goes away in BRB.

As Byzantine causal multicast/unicast/broadcast is an application layer property, it runs on top of the Byzantine Reliable Multicast (BRM)/ Unicast (BRU) /Broadcast (BRB) communication layer below. Byzantine Causal Multicast (BCM)/ Unicast (BCU)/ Broadcast (BCB) is invoked as send(m, G)/ send($m, \{p_i\}$)/ send(m, P) which in turn invokes send(m', G)/ send($m', \{p_i\}$)/ send(m', P) to the BRM/ BRU/ BRB layer below. Here, m' is m plus some control information appended by the BCM/ BCU/ BCB layer. A deliver(m') from the BRM/ BRU/ BRB layer below is given to the BCM/ BCU/ BCB layer which delivers the message m to the application via deliver(m) after the processing in that layer.

The correctness of Byzantine causal order unicast/multicast/ broadcast is specified on $(R, \rightarrow)$ and $(S, \xrightarrow{B})$. BCM/ BCU/ BCB needs to satisfy properties that are the counterparts of the properties in Definitions 6, 7, 8, respectively. In addition to these properties safety and liveness need to be satisfied as follows.

Definition 9: A causal ordering algorithm for unicast/multicast/broadcast messages must ensure the following:

1) *Strong Safety:* $\forall m' \in CP(m)$ such that m' and m are sent to the same (correct) process, no correct process delivers m before m'.
2) *Liveness:* Each message sent by a correct process to another correct process will be eventually delivered.

Definition 10: A causal ordering algorithm for unicast/multicast/broadcast messages must ensure the following:

1) *Weak Safety:* $\forall m' \in BCP(m)$ such that m' and m are sent to the same (correct) process, no correct process delivers m before m'.
2) *Liveness:* Each message sent by a correct process to another correct process will be eventually delivered.

The goal is to satisfy both properties in the above definitions. A trivial but unacceptable solution to satisfy strong safety, but no liveness, is to never deliver a message. Likewise, a trivial but unacceptable solution to satisfy liveness, but no strong safety, is to simply deliver any message that is received. Ruling out such trivial but unacceptable solutions, in the sequel when we say that neither safety nor liveness can be guaranteed, or say that one of the two properties but not the other can be guaranteed, we mean using a non-trivial algorithm that attempts to satisfy both properties.

While we have given the formal definitions of BRM, BRU, BRB, and BCM, BCU, BCB in terms of the same send and deliver primitives, in the sequel the context will be well-identified. Further, in our solvability results for BCM/ BCU/ BCB, we consider only the strong safety or weak safety, and liveness properties. The Validity, Self-delivery, Agreement, Integrity, No Information Leakage properties at the application layer follow straightforwardly from these properties at the communication layer below and are orthogonal to our study.

When $m \xrightarrow{B} m'$, then all processes that sent messages along the causal chain from m to m' are correct processes. This definition is different from $m \rightarrow_M m'$ [3], where M was defined as the set of all application-level messages delivered at correct processes, and $MCP(m')$ could be defined as the set of messages in M that causally precede m'. When $m \rightarrow_M m'$, then all processes, *except the first*, that sent messages along the causal chain from m to m' are correct processes. Our definition of $\xrightarrow{B}$ (Definition 4) allows for the purest notion of safety – weak safety (Definition 10) – that can be guaranteed to hold under unicasts and multicasts. The equivalent safety definition, that could be defined on MCP instead of BCP, would not be guaranteed under unicasts and multicasts, but is satisfied under broadcasts [3]. Our definition of $\xrightarrow{B}$ and $\rightarrow_M$ [3] both make the assumption that from the second to the last process that send messages along the causal chain from m to m', are correct processes.

IV. Solvability Results

A. Strong Safety and Liveness without Cryptography

An algorithm to solve causal ordering collects the execution history of each process in the system and derives causal relations from it. Let E_i denote the (actual) execution history at p_i and let $E = \bigcup_i \{E_i\}$. For any causal ordering algorithm, let F_i be the execution history at p_i as collected by the algorithm and let $F = \bigcup_i \{F_i\}$. F thus denotes the execution history as collected by the algorithm. Let $M(E)$ and $M(F)$ denote the messages sent and/or received in E and sent and/or received in F, respectively. p_r is a correct process which receives $m_2 \in M(E)$. $m_1 \in M(E) \cup M(F)$ is a message sent to p_r; because m_1 need not have reached p_r yet, it may belong to $M(F) \setminus M(E)$. Let $m_1 \rightarrow m_2|_E$ and $m_1 \rightarrow m_2|_F$ be the evaluation (1 or 0) of $m_1 \rightarrow m_2$ using E and F, respectively.

When correct process p_r receives m_2, it needs to correctly determine whether to deliver m_2 before a message m_1 or to wait for m_1 before delivery of m_2. To formulate this, we rephrase the causal ordering problem (Definition 9) as *CO(*E, F, m_2*)* as follows.

Definition 11: The causal ordering problem $CO(E, F, m_2)$ for a message m_2 received by a correct process p_r is to devise an algorithm to collect the execution history E as F at p_r such that $CO\_Deliv(m_2) = 1$, where

$$CO\_Deliv(m_2) = \begin{cases} 1 & \text{if } \forall m_1, m_1 \rightarrow m_2|_E = m_1 \rightarrow m_2|_F \\ 0 & \text{otherwise} \end{cases}$$

$CO\_Deliv(m_2)$ returns 1 iff $\forall m_1, m_1 \rightarrow m_2|_E = m_1 \rightarrow m_2|_F$. When 1 is returned, the algorithm output matches the actual truth and solves *CO* correctly. Thus, returning 1 indicates that the problem has been solved correctly by the algorithm using F. 0 is returned if either

1) $\exists m_1$ such that $m_1 \rightarrow m_2|_E = 1$ and $m_1 \rightarrow m_2|_F = 0$, denoting a strong safety violation because p_r will not wait for m_1 before delivery of m_2, or
2) $\exists m_1$ such that $m_1 \rightarrow m_2|_E = 0$ and $m_1 \rightarrow m_2|_F = 1$, denoting a liveness violation because p_r may continue waiting indefinitely for a fake m_1 to arrive before delivering the arrived m_2.

To determine whether *CO* is solved correctly, we have to evaluate $\forall m_1, m_1 \rightarrow m_2|_E = m_1 \rightarrow m_2|_F$ even if $m_1 \in (M(E) \cup M(F)) \setminus M(E)$ because such an m_1 is recorded by the algorithm as part of F. The key observation we make is that in *CO*, a single Byzantine process p_b can cause F (as recorded by the algorithm) to be different from E. This is not just a mismatch between E_b and F_b at p_b but also at other processes, and also a mismatch between other E_a and F_a at processes p_c, by contaminating F_b and/or F_a via direct and transitive message passing (across different messages) originated at or passing through p_b.

Our results relate causal ordering to the *Consensus* problem [23], [24], defined as follows.

Definition 12: In the *Consensus* problem, each process has an initial value and all correct processes must agree on a single value. The solution needs to satisfy the following three conditions.

1) Agreement: All non-faulty processes must agree on the same single value.
2) Validity: If all non-faulty processes have the same initial value, then the agreed-on value by all the non-faulty processes must be that same value.
3) Termination: Each non-faulty process must eventually decide on a value.

Theorem 1: It is impossible to solve causal ordering (Definition 9) as specified by *CO(*E, F, m_2*)* of unicast messages in an asynchronous message passing system with one or more Byzantine processes as neither strong safety nor liveness is guaranteed.

Proof: We prove the impossibility of solving the *CO* problem by showing:

1) a reduction (denoted $\preceq$) from *Black_Box* to *CO*, where *Black_Box* is defined below,
2) a reduction from the *Consensus* problem (which by the FLP result [27] is unsolvable in the presence of a single Byzantine process) to the *Black_Box* problem.

Specifically, we show how *Consensus* can be solved by solving *Black_Box*, and how *Black_Box* can be solved by solving *CO*. If *CO* were solvable, *Black_Box* would be solvable, and then *Consensus* would also be solvable. That contradicts the unsolvability of *Consensus*. Hence, there cannot exist any algorithm to solve *CO*.

Black_Box($\overline{V}, E, F, m_2$*)* takes as input a vector $\overline{V}$ of initial boolean values, one per process, E, F, and message m_2 sent to a correct (non-Byzantine) process p_r and m_2 is received by p_r. *Black_Box* acts as follows. The correct process p_r broadcasts the value w where:

$$w = \begin{cases} 0 & \text{if each correct } p_i \text{ has } V[i] = 0 \\ 1 & \text{if each correct } p_i \text{ has } V[i] = 1 \\ \bigwedge_{m_1} (m_1 \to m_2|_E = & \\ m_1 \to m_2|_F) & \text{otherwise} \end{cases}$$

Black_Box is solvable if *CO* at p_r is solvable correctly because solving *CO* requires using the execution histories of potentially Byzantine processes as recorded by the algorithm in F besides identifying the Byzantine processes. In order for any algorithm to correctly solve *CO*, it must ensure that F matches E. For this, the following must hold.

- A Byzantine process may attempt to insert a fake entry in F_x about sending a message from p_x to p_y and contaminate the reporting of histories in F, leading to a liveness violation and $M(F) \setminus M(E) \neq \emptyset$. Therefore, either contamination of F has to be prevented or malicious entries have to be filtered out from F in bounded time. But due to unicasting, a message from a potentially Byzantine p_x to p_y in F_x, cannot be verified in bounded time by other processes while collecting the reported execution history as the message itself cannot be broadcast or communicated to any process other than p_y to keep it private. Therefore, identification of Byzantine processes, their actual execution histories, and causal chains from and through them is required.
- Let there be a message m sent by p_x in E_x. During the collection of E_x for reporting F_x, Byzantine processes may delete information about m from F_x, leading to a strong safety violation and $M(E) \setminus M(F) \neq \emptyset$. Therefore, either deletion of information from E in F has to be prevented, or such deletions from E when presented with F have to be recognized in bounded time. This requires identification of the Byzantine processes, their actual execution histories, and causal chains from and through them.

If there were an algorithm to make F match E, it *requires identifying whether each of the processes that input their execution histories is correct or Byzantine, and tracing and dealing with/resolving the impact of contamination via message passing by the Byzantine processes from/through those Byzantine sources on the execution histories of processes at other processes*. Thus, *Black_Box* $\preceq$ *CO*.

When *Consensus*($\overline{V}$) is to be solved, it invokes the black box for *Black_Box*($\overline{V}, E, F, m_2$). Each correct process outputs as its consensus value the value that it receives from p_r and terminates. Agreement, Validity, and Termination clauses of *Consensus* can be seen to be satisfied. So *Consensus* $\preceq$ *Black_Box*.

If *CO* is (correctly) solvable, it returns 1 for $\forall m_1, m_1 \to m_2|_E = m_1 \to m_2|_F$, (and implicitly for all m_2). We now have

$$Consensus \preceq Black\_Box \preceq CO$$

This implies that if the *CO* problem is solvable, then *Consensus* is also solvable. That contradicts the FLP impossibility result for a Byzantine process system, hence *CO* is not solvable. □

Remark: Observe that under the crash-failure model, even though *Consensus* $\preceq$ *Black_Box*, we have that *Black_Box* $\not\preceq$ *CO*. This latter relation $\not\preceq$ is because solving *CO* does not require identifying the crashed processes; their (correct) execution histories can be faithfully transmitted to other processes (transitively) via the execution messages sent in the execution history itself as it grows and be present at the other (correct) processes' execution histories and in in-transit messages. As m_1 and m_2 must have been sent, the execution histories of their senders can transitively propagate to other non-crashed processes. In other words, the execution history of any prefix can be represented by that execution. Therefore, $M(E) = M(F)$. Hence, it suffices to consider the execution histories E_i of non-crashed processes (that include p_r) to determine $m_1 \to m_2$ without having to identify the crashed processes.

We now consider the broadcast communication mode. The proof analyzing the *CO* problem uses Byzantine Reliable Broadcast (BRB) [21], [26] as a layer beneath the broadcast invocation. Without loss of generality, this proof considers the strongest form of broadcast that gives the highest resilience to Byzantine behavior, namely BRB (Definition 8).

Theorem 2: It is impossible to solve causal ordering (Definition 9) as specified by *CO*(E, F, m_2) of broadcast messages in an asynchronous message passing system with one or more Byzantine processes as only liveness but not strong safety is guaranteed.

Proof: We outline the logic that *CO* (Definition 9) cannot be solved for Byzantine causal broadcast. For Byzantine causal broadcast, F cannot be made to match E.

- A process sends a broadcast via BRB. By running the causal ordering layer above the Byzantine Reliable Broadcast (BRB) [21], [26] layer, liveness violation can be prevented by ensuring $M(F) \setminus M(E) = \emptyset$. If a Byzantine process p_b attempts to insert a fake entry about broadcast of m by p_x in F_x ($x = b$ or $x \neq b$) at a correct process p_y, p_y can verify whether or not this insertion is valid as based on the Agreement property of BRB, m must be delivered by the BRB layer at all correct processes including p_y. Therefore, no message from a correct process to another correct process will wait indefinitely for causal delivery.
- However, a Byzantine process p_x can delete from F_x that it discloses to the rest of the system, information about a broadcast of m_1 by p_k that it has received, where p_k may be a correct process, despite running the causal ordering layer above the BRB layer. A message m_2 then broadcast, where $m_1 \to m_2$ and the message chain passes through a message broadcast by p_x (after receiving m_1), can be delivered by a correct process p_r before m_1 is, if p_r is not to wait indefinitely. This is because p_r does not learn of $m_1 \to m_2$ and m_1 could be any message. Thus, $M(E) \setminus M(F) \neq \emptyset$ and strong safety violations may occur.

Thus, to solve *CO*, it is necessary to identify Byzantine processes, their actual execution histories, and causal chains from and through them. Then *Black_Box* $\preceq$ *CO* and, as *Consensus* $\preceq$ *Black_Box*, hence *Consensus* $\preceq$ *CO*. □

In a multicast, a message is sent to a subset of processes and different send events can send to different multicast groups. We have the following result.

Theorem 3: It is impossible to solve causal ordering (Definition 9) as specified by *CO*(E, F, m_2) of multicast messages in an asynchronous message passing system with one or more

Byzantine processes as neither liveness nor strong safety is guaranteed.

Proof: Unicast mode of communication is a special case of multicast mode of communication. As the problem is impossible to solve for unicasts (Theorem 1), it is impossible to solve for multicasts. □

B. Weak Safety and Liveness without Cryptography

We now show a similar result to Theorem 1 with strong safety (Definition 9) defined in terms of the $\rightarrow$ relation replaced by weak safety (Definition 10) defined in terms of the $\xrightarrow{B}$ relation in the correctness criteria for causal ordering.

Similar to Definition 11, we rephrase the causal ordering problem (Definition 10) as *CO_B(*E, F, m_2*)* as follows.

Definition 13: The causal ordering problem $CO\_B(E, F, m_2)$ for a message m_2 received by a correct process p_r is to devise an algorithm to collect the execution history E as F at p_r such that $CO\_B\_Deliv(m_2) = 1$, where

$$CO\_B\_Deliv(m_2) = \begin{cases} 1 & \text{if } \forall m_1, \\ & m_1 \xrightarrow{B} m_2|_E = m_1 \xrightarrow{B} m_2|_F \\ 0 & \text{otherwise} \end{cases}$$

Observe, $m_1 \xrightarrow{B} m_2$ is equivalent to: ($m_1 \rightarrow m_2 \wedge$ *there is a causal path from send event of* m_1 *to send event of* m_2 *going through correct processes in the execution*). We define $m_1 \xrightarrow{B} m_2|_F$ as ($m_1 \rightarrow m_2|_F \wedge$ *there is a causal path from send event of* m_1 *to send event of* m_2 *going through correct processes in the execution)*. (Likewise for $m_1 \xrightarrow{B} m_2|_E$.) The algorithm to solve *CO_B* does not have to determine whether the path through correct processes exists.

Theorem 4: It is impossible to solve causal ordering (Definition 10) as specified by *CO_B(*E, F, m_2*)* of unicast messages in an asynchronous message passing system with one or more Byzantine processes as liveness cannot be guaranteed even though weak safety can be guaranteed.

Proof: Note that m_2 is necessarily sent by a correct process when $m_1 \xrightarrow{B} m_2$ holds. The proof of Theorem 1 carries identically, subject to the following changes. In the specification of *Black_Box*, the definition $\bigwedge_{m_1} (m_1 \xrightarrow{B} m_2|_E = m_1 \xrightarrow{B} m_2|_F)$ instead of $\bigwedge_{m_1} (m_1 \rightarrow m_2|_E = m_1 \rightarrow m_2|_F)$ is used.

That *Consensus* $\preceq$ *Black_Box* still holds is self-evident. *Black_Box* $\preceq$ *CO_B* still holds because solving *CO_B* correctly still requires using the execution histories of Byzantine processes as recorded by the algorithm in F besides identifying the Byzantine processes, similar to the proof for Theorem 1. In order for any algorithm to correctly solve *CO_B*, it must ensure that F matches E. For this, the following must hold.

- Due to unicasting, a message m from a potentially Byzantine p_x to p_y in F_x, cannot be verified in bounded time by other processes while collecting the reported execution history as the message itself cannot be broadcast or communicated to any process other than p_y to keep it private. Thus, a fake entry may be inserted in F_x by a Byzantine process, even if there exists some causal path through correct processes from send event of m_1 to send event of m_2, leading to a liveness violation and $M(F) \setminus M(E) \neq \emptyset$. (Note, liveness of m_2 is not with respect to a m_1 sent by a correct process but all m_1.) Therefore, either contamination of F has to be prevented or malicious entries have to be filtered out from F in bounded time. This requires identifying Byzantine processes, their actual execution histories, and causal chains from and through them.
- Let there be a message m_1 sent by correct process p_x in E_x. During the collection of E_x for reporting F_x, if there are no Byzantine processes along some causal path from send event of m_1 at p_x to send event of m_2 at p_k, (hence p_k must be a correct process), it is possible to ensure that no Byzantine processes can cause deletion of information about m_1 from F_x, thus $(M(E))_c \setminus M(F) = \emptyset$, where $(M(E))_c$ is the messages of $M(E)$ sent by correct processes. Thus, weak safety violation of m_2 (with respect to m_1 sent by correct processes) can be prevented.

If there were an algorithm to make F match E, it *still requires identifying whether each of the processes that input their execution histories is correct or Byzantine, and tracing and dealing with/resolving the impact of contamination via message passing by the Byzantine processes from/through those Byzantine sources on the execution histories of processes at other processes*. Hence *Black_Box* $\preceq$ *CO_B*. The theorem follows. □

Theorem 5: It is possible to solve causal ordering (Definition 10) as specified by *CO_B(*E, F, m_2) of broadcast messages in an asynchronous message passing system with one or more Byzantine processes as both liveness and weak safety can be guaranteed.

Proof: We outline the logic that *CO_B* (Definition 10) can be solved for Byzantine causal broadcast. For Byzantine causal broadcast, F can be made to match E. A broadcast is sent via BRB as in the proof of Theorem 2.

- $M(F) \setminus M(E) = \emptyset$, hence liveness violations cannot occur. Same reasoning as in first bullet in Theorem 2.
- If there is a path through correct processes along $m_1 \xrightarrow{B} m_2$, processes along that path can faithfully propagate information about the causal chain of messages through those correct processes to other processes. When m_2 arrives at p_r it will wait for m_1 which must arrive at p_r because of the Agreement property of the BRB layer over which the causal ordering layer is run. Thus $(M(E))_c \setminus M(F) = \emptyset$ and weak safety violations cannot occur. (This holds even if broadcaster of m_1 is Byzantine.)

Thus to solve *CO_B* for broadcasts under Definition 10, it is not necessary to identify whether each process is Byzantine, hence *Black_Box* $\npreceq$ *CO_B* and hence *Consensus* $\npreceq$ *CO_B*. □

Theorem 6: It is impossible to solve causal ordering (Definition 10) as specified by *CO_B(*E, F, m_2) of multicast messages in an asynchronous message passing system with one or more Byzantine processes as liveness cannot be guaranteed even though weak safety is guaranteed.

Proof: Unicast mode of communication is a special case of multicast mode of communication. As the problem is impossible to solve for unicasts (Theorem 4), it is impossible to solve for multicasts. □

C. Results for Cryptography

1) Strong Safety and Liveness Using Cryptography: In order for a correct process to be able to verify a message has indeed been sent, messages need to be sent via broadcast. But to maintain confidentiality, the message needs to be encrypted. When p_j has to multicast a message m to group G, it creates the ciphertext C_m by encrypting m with the group key K_G and does a Byzantine Reliable Broadcast (BRB) of (C_m, G) so that other processes can verify that the message was indeed sent. It is thus assumed that each multicast group shares a unique symmetric key for encryption and decryption of messages sent to that group. When a correct process p_c receives (C_m, G) and $p_c \in G$, and decrypts and delivers m, it includes (C_m, G) as control information on the next message m' to G' it sends (via BRB) to convey $m \rightarrow m'$ to others. Other processes p_d can verify whether $m \rightarrow m'$ as follows. When p_d receives (C'_m, G') with $\{(C_{m_1}, G_1), (C_{m_2}, G_2), \dots (C_{m_k}, G_k)\}$ as control information, for each $x, x \in [1, k]$, p_d waits to receive (C_{m_x}, G_x) directly from the sender via BRB and "deliver" it before "delivering (C'_m, G')", in order to verify C_{m_x} sent to G_x. Here, "deliver" is in a logical sense; actual delivery happens after decryption only if $p_d \in G_x$ and $p_d \in G'$, respectively. It follows that m' will not be delivered unless the causally preceding m_x is also delivered (and transitively so for messages in the control information of (C_{m_x}, G_x)) and messages sent previously by the sender of m' have been delivered. Only when (C_{m_x}, G_x) $(\forall x)$ arrive directly and are delivered is $m_x \rightarrow m'$ true. (At this stage the deliver events of $m_1 \dots m_k$ and the send event of m' can be added to F_c locally at p_d.) With this protocol, if a Byzantine process inserts a fake entry (C_{m_x}, G_x) in the control information on $(C_{m'}, G')$, its message will never be delivered at other correct processes – this is a strong disincentive to insert fake information.

Theorem 7: It is impossible to solve causal ordering (Definition 9) as specified by *CO*(E, F, m_2) of multicast messages in an asynchronous message passing system with one or more Byzantine processes even with the use of cryptography as strong safety cannot be guaranteed even though liveness can be guaranteed.

Proof: We outline the logic that *CO* (Definition 9) cannot be solved for the multicast mode of communication by showing that F cannot be made to match E.

- By running the causal ordering layer above the Byzantine Reliable Broadcast (BRB) [21], [26] layer, liveness violation can be prevented by ensuring $M(F) \setminus M(E) = \emptyset$. If a Byzantine process p_b attempts to insert a fake entry about sending of (C_m, G) by p_x to $p_b \in G$ that p_b has decrypted and delivered, in F_x at a correct process p_y via control information on message $(C_{m'}, G')$, p_y can verify whether or not this insertion is valid as based on the Agreement property of BRB, (C_m, G) must be delivered by the BRB layer at all correct processes including p_y and p_b must be a destination within G. If the message from p_b to p_y does not pass this verification, that message is not considered delivered. If p_b were a correct process, (C_m, G) is guaranteed to arrive at p_y. Therefore, no message from a correct process to another correct process will wait indefinitely for causal delivery.
- However, a Byzantine process p_x can delete from F_x that it discloses to the rest of the system, information about a message m_1, i.e., (C_{m_1}, G_{m_1}), sent by p_k that it has received, decrypted and delivered, where p_k may be a correct process, despite running the causal ordering layer above the BRB layer. A message m_2 then multicast, where $m_1 \rightarrow m_2$ and the message chain passes through a message multicast by p_x subsequent to the local delivery of m_1, can be delivered by a correct process p_r before m_1 is, if p_r is not to wait indefinitely. Thus, $M(E) \setminus M(F) \neq \emptyset$ and strong safety violations may occur.

Thus, to solve *CO*, it is necessary to identify Byzantine processes, their actual execution histories, and causal chains from and through them. Then *Black_Box* $\preceq$ *CO* and, as *Consensus* $\preceq$ *Black_Box*, hence *Consensus* $\preceq$ *CO*. □

Theorem 8: It is impossible to solve causal ordering (Definition 9) as specified by *CO*(E, F, m_2) of unicast messages in an asynchronous message passing system with one or more Byzantine processes even with the use of cryptography as strong safety cannot be guaranteed even though liveness can be guaranteed.

Proof: The proof of Theorem 7 applies almost identically to unicasts with the observation that each multicast group is of size two – the sender and the receiver. □

Theorem 9: It is impossible to solve causal ordering (Definition 9) as specified by *CO*(E, F, m_2) of broadcast messages in an asynchronous message passing system with one or more Byzantine processes even with the use of cryptography as strong safety cannot be guaranteed even though liveness can be guaranteed.

Proof: The proof of Theorem 2 which is for broadcast-based communication without allowing cryptography also applies with the following observations.

- Liveness can be guaranteed even without cryptography.
- Strong safety cannot be guaranteed because the proof applies even if cryptography is used, i.e., the supression of information of E in F as described in the second bullet of Theorem 7 can occur even with the use of cryptography. □

2) Weak Safety and Liveness Using Cryptography

Theorem 10: It is possible to solve causal ordering (Definition 10) as specified by *CO_B*(E, F, m_2) of multicast messages in an asynchronous message passing system with one or more Byzantine processes with the use of cryptography as weak safety and liveness can be guaranteed.

Proof: Liveness can be guaranteed as shown in the proof of Theorem 7. Weak safety can be guaranteed as shown in the proof of Theorem 4 (for unicasts) – the guarantee of weak safety holds even for multicasts. □

As unicasts and broadcasts are special cases of multicast, we have the following two results.

Corollary 1: It is possible to solve causal ordering (Definition 10) as specified by *CO_B*(E, F, m_2) of unicast messages in an asynchronous message passing system with one or more Byzantine processes with the use of cryptography as weak safety and liveness can be guaranteed.

Corollary 2: It is possible to solve causal ordering (Definition 10) as specified by *CO_B*(E, F, m_2) of broadcast messages in an asynchronous message passing system with one or more

Byzantine processes with the use of cryptography as weak safety and liveness can be guaranteed.

D. Analysis of Strong Safety Violation

Our results show that it is impossible to satisfy strong safety in a deterministic manner for unicasts, multicasts or broadcasts, whether without or even with the use of cryptography. This is because if $m_1 \to m_2$ and there is no causal path from the send event of m_1 to the send event of m_2 going through only correct processes, and where both messages are sent to the same correct process p_r, then a Byzantine process p_b along any causal path from send of m_1 to send of m_2 first receives/delivers a message from its predecessor along the causal path and then sends a message to its successor along the causal path. Both events are local to the Byzantine process. p_b can choose to supress the receive event from what it discloses to the rest of the system, or swap the order of the receive event and the send event in what it discloses to the rest of the system. Both options have the effect of breaking the causality chain in what is disclosed to the rest of the system and projecting $m_1 \not\to m_2$ even though in reality $m_1 \to m_2$. Thus the rest of the system can see $m_1 \not\to m_2$ and p_r is not obligated to deliver m_1 before m_2, leading to a possible strong causality violation. No deterministic protocol even using cryptography can exist to counter p_b's action in an asynchronous system.

Examples of strong safety violations in real-world applications are as follows.

1) Social media posts: Correct processes may see $post\_b$ by a Byzantine process, whose contents depend on $post\_a$, before they see $post\_a$.
2) Multiplayer gaming: A Byzantine process can cause strong safety violations to gain an advantage over correct processes in winning the game.

V. Sender-Inhibition Algorithm

As a result of Theorems 1 and 4, we know that it is impossible to maintain both (strong as well as weak) safety and liveness while trying to causally order messages in an asynchronous system with Byzantine faults. Here, we develop a solution for causal order of unicasts based on timeouts under a synchronous system model. Under the assumption of a network guarantee of an upper bound δ on message latency, we prevent the Byzantine processes from making non-faulty processes wait indefinitely resulting in a liveness attack. This prevents a correct process from being unable to send messages because it is waiting for an acknowledgment from a Byzantine process. This solution can maintain both weak safety and liveness.

The solution is as follows. Each process maintains a FIFO queue, Q and pushes messages as they arrive into Q. Whenever the application is ready to process a message, the algorithm pops a message from Q and delivers it to the application. After pushing message m into Q, each process sends an acknowledgement message to the sending process. Whenever process p_i sends a message to process p_j, it waits for an acknowledgement to arrive from p_j before sending another message. While waiting for p_j's acknowledgement to arrive, p_i can continue to receive and

Algorithm 1: Sender-Inhibition Algorithm.

```
Data: Each $p_i$ maintains a FIFO queue $Q$ and a lock $lck$
when application is ready to process a message: ▷ Deliver event
    $m = Q.pop()$
    if $m \neq \phi$ then
        deliver $m$
when message $m$ arrives from $p_j$: ▷ Receive event
    $Q.push(m)$
    $send(ack, j)$ to $p_j$
when message $m$ is ready to be sent to $p_j$: ▷ Send event
    $lck.acquire()$ ▷ Executes atomically
    $send(m, j)$ to $p_j$
    start $timer$
    while (ack for m not arrived from $p_j$ $\wedge$ no timeout) do
        wait in a nonblocking manner
    $lck.release()$
```

deliver messages. If p_i does not receive p_j's acknowledgement within time $2 * \delta$ (timeout period), it is certain that p_j is faulty and p_i can execute its next send event without violating $\xrightarrow{B}$.

Algorithm 1 consists of three *when* blocks. The *when* blocks execute asynchronously with respect to each other. This means that either the algorithm switches between the blocks in a fair manner or executes instances of the blocks concurrently via multithreading. In case a block has not completed executing and the process switches to another block, its context is saved and reloaded the next time it is scheduled for execution. If multithreading is used, each instance of a *when* block spawns a unique thread. This maximizes the concurrency of the execution. Algorithm 1 ensures that while only one send event at a process can execute at a given point in time, multiple deliver and multiple receive events can occur concurrently with a single send event.

Theorem 11: Under a network guarantee of delivering messages within δ time, Algorithm 1 ensures liveness while maintaining weak safety.

Proof: The send event in Algorithm 1 is implemented by the *when* block in Lines 8–14. A send event is initiated only after the previous send has released the lock, which happens when the sender p_i (a) has received an *ack* from the receiver p_j, or (b) times out.

1) In case (a), the sender learns that p_j has queued its message m in the delivery queue, and the sender can safely send other messages. Any message m' such that $m \xrightarrow{B} m'$ and m' is sent to p_j will necessarily be queued after m in p_j's delivery queue. Due to FIFO withdrawal from the delivery queue, m is delivered before m' at p_j and safety is guaranteed. As p_i receives the *ack* before the timeout, progress occurs at p_i. There is no blocking condition for m at p_j and hence progress occurs at p_j.

2) In case (b) where a timeout occurs, the lock is released at p_i and there is progress at p_i. It is left up to the application to decide how to proceed at p_i. This prevents a Byzantine process from executing a liveness attack by making a correct process wait indefinitely for the *ack*. It can be assumed that p_j is a Byzantine process and so safety of delivery at p_j does not matter under the $\xrightarrow{B}$ relation.

Therefore, Algorithm 1 ensures liveness while maintaining weak safety. □

Complexity: In the Sender-Inhibition algorithm, the sender waits for at most $2 * \delta$ time for the ack to arrive from the receiver before sending its next message. The timeout period is fixed at $2 * \delta$ because this is the maximum time an ack can take to arrive from the point of sending the application message.

The algorithm is simple to understand and implement. However, send events at a process are blocking with respect to each other. The algorithm eliminates the $O(n^2)$ message space and time overhead of [2], [7], [8], [9], [10] and uses one control message of size $O(1)$ per application message sent.

An extension of the Sender-Inhibition algorithm to provide weak safety and liveness for multicasts is given in [5].

VI. Channel Sync Algorithm

As a result of Theorems 1, 4 we know that it is impossible to maintain both (strong and weak) safety and liveness while trying to causally order unicast messages in an asynchronous system with Byzantine faults. In Section V, we presented an algorithm for weak safety and liveness in the synchronous system model. In this section, we present another solution satisfying weak safety and liveness based on timeouts in the synchronous system model. Under the assumption of a network guarantee of an upper bound δ on message latency, we prevent the Byzantine nodes from making non-faulty nodes wait indefinitely resulting in a liveness attack.

Algorithm 2 presents a solution that assumes that the underlying network guarantees that all messages are delivered within δ time. As long as this assumption holds, Algorithm 2 can guarantee both weak safety and liveness. Each process maintains FIFO queues for each other process where it stores incoming messages from the concerned process. Application messages are delivered immediately after getting popped from the queue. However, control messages are not processed immediately; the algorithm checks to make sure that it is safe to deliver the next message in the queue before completing processing. Whenever a process sends a message it informs every other process about the send event via a control message. Whenever a process delivers a message, it also informs every other process via a control message. Whenever process p_i receives a control or application message from process p_j, it pushes it into Q_j. All control messages have timers associated with them to time them out in case of Byzantine behaviour of the sender and/or receiver. When p_i pops a *receive control message* from any queue Q_x it waits for either the corresponding *send control message* to reach the head of its queue (be dequeued), or the receive control message gets timed out in case the send control message does not arrive. This ensures that causality is not violated at p_i, while ensuring progress. We also need to ensure that in case of non-Byzantine behaviour on part of both the sender and receiver, both the send control message and receive control message do not time out before the other one arrives. In order to achieve this, the timer for receive control messages has to be set to at least δ as shown in Lemma 1 while the timer for send control messages can be varied (see discussion below). The timer for send control messages can be reduced (it can be set to 0 without compromising weak safety) to implement different behaviours in the system, but the timer for receive control message has to be at least δ, and increasing it will only result in sub-optimal behaviour. Therefore, the timer for receive control messages should always be δ.

Lemma 1: Under the assumption of a network guarantee of delivering messages within a finite time period δ, no receive control message with a timer greater than or equal to δ can get processed before the matching send control message at any process when both the sender and receiver processes are correct, during the execution of Algorithm 2.

Proof: Without any loss of generality, we take $\delta_r = \delta$ and $\delta_s = 0$. Here δ_r and δ_s are timer wait times for receive control and send control messages, respectively. Whenever, a send control message arrives in Algorithm 2, it stops the timer of the matching receive control message (if already present) to make sure that the receive control message waits for the send control message to get processed. If the send control message gets popped from the queue and the receive control message has not arrived, it simply gets processed. Now whenever the receive control message arrives, it waits for the timeout period and gets timed out without impacting weak safety because the send control message has already been processed.

In order to ensure that a receive control message waits for a send control message to get processed, we need to ensure that the send control message arrives before the receive control message times out. The maximum amount of time the send control message can take to arrive at any process p_i is δ and the minimum amount of time the matching receive control message can take to arrive at p_i is 0. This means that in the worst-case scenario, the send control message will arrive in time δ after the arrival of the receive control message. Therefore, since the send control message arrives before the receive control message times out, the receive control message will have to wait for the matching send control message to get processed. (Note: the sender and receiver are non-Byzantine. If either of them is Byzantine, the receive control message, if present, may still time out at correct process p_i but, as we will show in Corollary 3 and Theorem 13, correctness of causal ordering is not impacted under $\xrightarrow{B}$.) □

From Lemma 1, the timer for send control messages can be set as low as 0 without impacting weak safety. The timer for send control messages can be tweaked based on the desired system performance. For instance, setting $\delta_s = 0$ would result in reduced latency for all send control messages at the expense of some receive control messages waiting out their entire waiting period of δ in the queue. If $\delta_s > 0$ a send control message waits after being popped until timeout. If in this interval any receive control message arrives, the receive control message gets deleted

Algorithm 2: Channel Sync Algorithm.

```
    Data: Each $p_i$ maintains a FIFO queue $Q_j$ for every
          process $p_j$
1   when the application is ready to send message $m$ to $p_j$:
2   $send(m, j, app)$ to $p_j$
3   for all $x \neq i, j$ do
4       $send(\langle i, j, sent\rangle, x, control)$ to $p_x$

5   when $\langle m, i, type\rangle$ arrives from $p_j$:
6   $Q_j.push(\langle m, i, type\rangle)$
7   if $type = control$ then
8       start $timer$ for message $m$
9       if $m[2] = sent$ then
10          if matching receive control message is in $Q_{m[1]}$ or popped then
11              stop timers of send control message and
                matching receive control message
12      if $m[2] = delivered$ then
13          if matching send control message is in $Q_{m[1]}$ or popped then
14              stop timers of receive control message and
                matching send control message

15  when the application is ready to process a message
        from $p_j$ and $|Q_j| \neq 0$:  ▷ Only one instance
        of this block is executed at a time
        for a particular $Q_x$
16  $\langle m, *, type\rangle = Q_j.pop()$
17  if $type = control \wedge m[2] = delivered$ then
18      while timeout period not exceeded $\wedge$ timer not stopped do
19          wait in a non-blocking manner
20      if $timer$ stopped then
21          while matching send control message not reached head of $Q_{m[1]}$ do
22              wait in non-blocking manner
23  else if $type = control \wedge m[2] = sent$ then
24      while timeout period not exceeded $\wedge$ timer not stopped do
25          wait in a non-blocking manner
26      if timer stopped then
27          delete the matching receive control message
            (popped/in $Q_{m[1]}$ if present)
28  else if $type = app$ then
29      deliver $m$
30      for all $x \neq i, j$ do
31          $send(\langle i, j, delivered\rangle, x, control)$ to $p_x$
```

(Lines 12–14 and 26–27) and does not have to wait after being popped and until its timeout. So although the wait of a send control message increases, that of a receive control message decreases. It would be interesting to simulate the effect on overall system latency by varying δ_s from 0 upwards while keeping δ_r fixed at δ as per Lemma 1.

If $\delta_s = 0$ (effectively, no timer for send control messages), then in Algorithm 2, stopping the send control message timer (Lines 11 and 14) and testing if it was stopped (Lines 24 and 26) can be replaced by setting and testing a boolean $flag\_timer\_stopped$.

A send event and a receive event are referred to as s and r, respectively. The control messages we use for send and receive events are denoted cms and cmr, respectively.

Theorem 12: Under the assumption of a network guarantee of delivering messages within a finite time period δ, queued messages in Algorithm 2 will be dequeued in at most $\delta_r + \max(\delta_r, \delta_s)$ time.

Proof: As a simplifying assumption, the time taken to pop a message from a queue is considered to be 0. The time each message spends in the queue is only because of latency induced by control messages. Let m be an application message inserted in Q_{i_0} at process p_j at time 0 (as a reference instant). The waiting time in the queue can be analyzed as follows.

1) There may be no control messages in front of m in Q_{i_0}. Since the latency induced by application messages that may be in front of m is 0, m will be popped and delivered immediately. The waiting time in the queue for m is 0.
2) There may be one or more send control messages before m in Q_{i_0}. Each of the control messages will take at most δ_s time to get processed. Since the timers for all of those control messages are ticking concurrently, m will have to wait for at most δ_s time.
3) There may be one receive control message cmr_{i_0} in front of m in Q_{i_0}. cmr_{i_0} is for application message m_1 sent from i_1 (before time 0) to i_0 (received before time 0). Note, if there are multiple receive control messages ahead, the analysis can be independently made for each of them.
    a) cms_{i_1} does not arrive in δ_r. cmr_{i_0} times out at δ_r. So total delay is δ_r.
    b) Otherwise cms_{i_1} is inserted in Q_{i_1} in time δ_r from time 0.
        i) It may be blocked by cms'_{i_1}. This times out in δ_s time. Total delay is therefore $\delta_r + \delta_s$.
        ii) It may be blocked by cmr_{i_1} for application message m_2 from i_2 sent before time 0 to i_1 received before time 0, ahead in Q_{i_1}. Therefore cmr_{i_1} arrived within time δ_r from time 0. It waits for cms_{i_2}.
4) Reasoning for the delay introduced by wait for cms_{i_2}, corresponding to application message m_2, in Q_{i_2} is as follows.
    a) cms_{i_2} does not arrive in δ_r. cmr_{i_1} times out in δ_r after its arrival which was latest at δ_r from time 0. Total delay is therefore $\delta_r + \delta_r$.
    b) Otherwise cms_{i_2} arrived within δ_r from time 0 because m_2 was sent before time 0 due to transitive chain $m_2 \rightarrow m_1$ and m_1 was received before time 0. Therefore cms_{i_2} is inserted in Q_{i_2} in δ_r from time 0.
        i) It may be blocked by cms'_{i_2}. This times out in δ_s time. Total delay is therefore $\delta_r + \delta_s$.
        ii) It may be blocked by cmr_{i_2} for application message m_3 from i_3 sent before time 0 to i_2 received before time 0, ahead in Q_{i_2}. Therefore cmr_{i_2} arrived within time δ_r from time 0. It waits for cms_{i_3}.

5) The reasoning for the delay introduced by wait for cms_{i_3} in Q_{i_3} is identical to the reasoning for the wait introduced by cms_{i_2} in the previous item. In particular, cms_{i_3} was inserted in Q_{i_3} within δ_r from time 0.

We generalize the above analysis as follows. Define $\leftarrow$ as the "waits for" or "succeeds in time" relation on control messages in the queues at p_j. Then, there exists a chain of control messages

$$cmr_{i_0} \leftarrow cms_{i_1} \leftarrow cmr_{i_1} \leftarrow cms_{i_2} \leftarrow cmr_{i_2} \leftarrow \ldots \leftarrow cms_{i_k}$$

each of which must have arrived in the corresponding Q_{i_α} within time δ_r from time 0 (see $(*)$ below). This chain corresponds to the following chain of application messages:

$$m_k \rightarrow m_{k-1} \rightarrow \ldots m_2 \rightarrow m_1$$

We prove that "$(*)$ $cmr_{i_{a-1}}$ is inserted in $Q_{i_{a-1}}$ within time δ_r from time 0, cms_{i_a} was inserted in Q_{i_a} within time δ_r from time 0." We use induction. The base case, being for $a = 2$, was shown above. Assume the induction hypothesis is true for $x, x \geq 2$. We show the result $(*)$ for $x + 1$. As cmr_{i_x} arrives in Q_{i_x} before cms_{i_x}, from the induction hypothesis for x, cmr_{i_x} is inserted in Q_{i_x} within δ_r from time 0. It waits for $cms_{i_{x+1}}$. $cms_{i_{x+1}}$ arrived within δ_r from time 0, because m_{x+1} was sent before time 0 due to transitive chain $m_{x+1} \rightarrow m_x \rightarrow \ldots m_1$ and m_1 was received before time 0 (because cmr_{i_0} was received in Q_{i_0} before time 0). Therefore $cms_{i_{x+1}}$ is inserted in $Q_{i_{x+1}}$ within δ_r from time 0. (end of proof of $(*)$)

We also claim k is finite and bounded because the corresponding control messages existed in the queues at p_j at time 0 or later and were therefore added to the queues at the earliest at $-\max(\delta_r, \delta_s)$; this implies the corresponding application messages were therefore sent after $-\delta - \max(\delta_r, \delta_s)$.

The chain of control messages terminates at cms_{i_k}, for $k > 0$. The queues contribute delays as analyzed by the following cases.

1) There is no receive control message ahead of cms_{i_k} in Q_{i_k}. Total delay this queue contributes is $\delta_r + \delta_s$.
2) Total overall delay contributed by queues Q_{i_z}, $z = [1, k-1]$ combined is considered next. Send control messages ahead of and including cms_{i_z} on timing out contribute up to δ_s combined delay. Receive control messages ahead of cms_{i_z} on timing out contribute up to δ_r combined delay. The combined contribution of such send and receive control messages is up to $\max(\delta_r, \delta_s)$. Plus the up to δ_r delay contributed by cms_{i_z} to get enqueued in Q_{i_z} as seen above in $(*)$ gives a combined delay bound of $\delta_r + \max(\delta_r, \delta_s)$. This is also the combined delay contributed by queues Q_{i_1} through $Q_{i_{k-1}}$.
3) Send (or receive) control messages ahead of m in Q_{i_0} contribute a delay of $\max(\delta_s, \delta_r)$.

Total overall delay contributed by all queues Q_{i_0} to Q_{i_k} is thus $\max(\delta_r + \delta_s, \delta_r + \max(\delta_r, \delta_s), \max(\delta_r, \delta_s)) = \delta_r + \max(\delta_r, \delta_s)$.

If $k = 0$, there is no receive control message ahead of m in Q_{i_0}, and as shown at the start of the proof, total delay is bounded by δ_s.

Combining $k = 0$ and $k > 0$ cases, the total overall delay of m is bounded by $\max(\delta_s, \delta_r + \max(\delta_r, \delta_s)) = \delta_r + \max(\delta_r, \delta_s)$. □

Since the amount of time each message spends in the message queue is bounded by a finite quantity, every application message will eventually be delivered. Therefore liveness is maintained by Algorithm 2.

Corollary 3: Algorithm 2 guarantees liveness.

Theorem 13: Under the assumption of a network guarantee of delivering messages within a finite time period δ, Algorithm 2 can guarantee weak safety by setting timers for control messages as a function of δ.

Proof: In order to ensure weak safety, prior to delivering any message m' at process p_j, we need to ensure that if $\exists m \in BCP(m')$ such that m is sent to p_j, then m is delivered before m' at p_j.

Algorithm 2 ensures weak safety at any process as follows:

- *Program Order:* Since we assume FIFO channels, messages from p_i to p_j get enqueued in Q_i in program order and get delivered in program order.
- *Transitive Order:* Let m be sent by p_i to p_j at send event s_i^x. Consider a causal chain of b messages starting at s_i^y from $i = i_0$ and ending at $j = i_b$ through correct processes and having these events:

$$\langle s_i^y = s_{i_0} \rightarrow r_{i_1} \rightarrow s_{i_1} \rightarrow r_{i_2} \rightarrow \ldots . \rightarrow r_{i_{b-1}} \rightarrow s_{i_{b-1}} \rightarrow r_{i_b} \rangle$$

Let $s_i^x \xrightarrow{B} s_i^y$ and $m' = \langle s_{i_{b-1}} \rightarrow r_{i_b} \rangle$ be the last message of the causal chain. This implies that $m \in BCP(m')$ by transitivity. The control messages for all the events in the causal chain above will reach p_j.

We make the following observations at p_j.

1) In Q_{i_0}, cms_{i_0} (control message for s_{i_0}) waits for m (sent at $s_{i_0}^x$) to get delivered.
2) From Lemma 1, in Q_{i_α} $(1 \leq \alpha \leq (b-1))$, cmr_{i_α} waits for $cms_{i_{\alpha-1}}$ in $Q_{i_{\alpha-1}}$ to be processed.
3) In Q_{i_α} $(1 \leq \alpha \leq (b-2))$, cms_{i_α} waits for cmr_{i_α} to be processed.
4) In $Q_{i_{b-1}}$, m' (sent at $s_{i_{b-1}}$) waits for $cmr_{i_{b-1}}$ to be processed.

Hence, message m' waits for message m to get delivered.

Algorithm 2 therefore ensures weak safety: "that $\forall m \in BCP(m')$ sent to the same p_j, m gets delivered before m' at p_j," under a network guarantee of delivering messages within a fixed time. □

Complexity for Unicasts: The Channel Sync algorithm uses $2(n-2)$ control messages of size $O(1)$ each per application message and does not inhibit concurrency (beyond what is necessary to enforce causal order). Any delay up to the maximum in Theorem 12 is essential for causal order in the face of Byzantine processes. The algorithm has a very high degree of concurrency but each process has to manage n queues and a timer per control message.

Note that in contrast to the Channel Sync algorithm, the algorithm in [3] for causal ordering of broadcasts under weak safety requires $O(n)$ broadcasts (control message broadcasts) of size $O(n)$ each per application message broadcast. It also has an added latency equivalent to 3δ due to the underlying Bracha's BRB protocol [21].

An extension of the Channel Sync algorithm to provide weak safety and liveness for multicasts is given in [6].

VII. Discussion

A main contribution in the result proofs was to show a reduction from the *Consensus* problem to the CO problem, thus establishing the impossibility of solving CO in an asynchronous system with a Byzantine process. We now show that CO does not reduce to *Consensus*, i.e., CO cannot be solved even if *Consensus* were solvable and hence CO is harder than *Consensus*.

Theorem 14: In an asynchronous message passing system with Byzantine processes, $CO \not\preceq Consensus$.

Proof: To solve *Consensus*, assume an oracle that identifies each process as being Byzantine or crash-prone or as being correct. The oracle is accessible to each process. The correct processes thus know the identity of all other correct processes. They execute a simple broadcast of their initial value and wait for the corresponding broadcasts from all other correct processes. Thus knowing the initial values of all correct processes, a correct process simply runs a local algorithm to decide on the consensus value – agree on a default value if initial values include both 0s and 1s, otherwise agree on the single value that is the initial value of all correct processes. It is straightforward to observe that Agreement, Validity, and Termination of the *Consensus* problem are satisfied.

However, knowing the identities of the Byzantine/faulty/crashed processes does not help to solve CO. To see this consider the following scenario. p_i and p_j sent m_1 and m_2 to p_d and $send(m_1) \rightarrow send(m_2)$. The causal path from $send(m_1)$ to $send(m_2)$ goes through a Byzantine process p_b which received m_r and then sent m_s along this causal path. If p_b chooses not to disclose to the system that $receive(m_r) \rightarrow send(m_s)$ then no other process will be able to detect $send(m_1) \rightarrow send(m_2)$ and thus CO can get violated at p_d. Observe that when p_d receives m_2 before m_1, it has no way to distinguish between (a) $m_1 \rightarrow m_2$, (b) m_1 was never sent at all, and (c) $m_2 \rightarrow m_1$. If not to wait indefinitely and avoid a liveness violation in case (b), p_d will deliver m_2, thus risking a safety violation if case (a) were true. To avoid a safety violation in case (a), p_d will wait for m_1 to arrive and be delivered before delivering m_2, thus risking a liveness violation if case (b) were true and risking a safety violation if case (c) were true. This above reasoning is true for multicasts, unicasts, and broadcasts. Knowing that p_b is Byzantine does not help in any way.

Thus to solve *Consensus*, it is sufficient to identify the Byzantine/faulty processes. But to solve *CO*, it is necessary to also identify execution histories at Byzantine processes and causal paths passing through or originating from the Byzantine processes. □

To solve CO, it is not sufficient that the non-faulty processes construct a local ordered sequence of messages intended only for them by using *Consensus*. The non-faulty processes need to also determine whether the send events of any two such messages were causally ordered with respect to each other and that cannot be achieved by solving *Consensus*, as proved above. However, note that *Consensus* is a harder problem than *CO* in the crash failure setting.

Other problems that are impossible to solve in the presence of one faulty process are related to the impossibility of solving *Consensus* in a similar setting in [27] [28].

Deterministic, cryptography-free Byzantine causal broadcast under weak safety + liveness is solvable in an asynchronous system [3], [4] as per Theorem 5. Deterministic cryptography-free Byzantine causal unicast or multicast under weak safety + liveness are not (Theorems 4 and 6). One cannot use a Byzantine fault-tolerant (BFT) causal broadcast protocol to implement point-to-point or multicast abstraction by adding recipient-ID and filtering on arrival only those messages intended for the local node because the filtering mechanism at the local node can be voided/compromised if the local node is Byzantine. Recall from Definitions 6 and 7 that the No Information Leakage property has to be satisfied. Here the BFT causal broadcast execution which is at a lower layer on top of which the application runs can be peeped into by the local Byzantine node and it can read a message not intended for it. A p_i to p_j unicast must be kept private to the two. This is possible in a deterministic manner with the use of cryptographic primitives for weak safety + liveness (Cor. 1, Th. 10), and is impossible in a deterministic manner for strong safety + liveness even with using cryptographic primitives (Theorems 8, 7).

We rule out full-information protocols (FIP) [29] for providing weak safety and liveness, where the entire transitively collected message history is used as control information because a FIP obviates the need for causal ordering. The proof structure for the solvability results is similar to that for the analysis of detection of the causality relation between events [30].

Synchronization mechanism in the algorithms: In view of the impossibility results, the algorithms we presented are in a synchronous system model. Here, processes are not required to execute in lock-step rounds. In a step of lock-step execution, a process first sends messages and then receives messages sent by others in that very step. After receiving a message in a step, it has to wait for the start of the next step to send messages. (Lock-step execution can be provided by synchronizers [31] in an asynchronous system, and is useful when the application program is synchronous, i.e., written assuming lock-step execution. It is not possible to design synchronizers under Byzantine failures.) Our algorithms are designed for asynchronous applications that do not necessarily use lock-step in their code (see list of applications listed in Section I, e.g., social networking). If lock-step were emulated or simulated in a synchronous system, an additional delay of at least the time needed to emulate a step, which would be at least δ, would be incurred besides the message latency and wait time for a send event before the start of the next step, in addition to the other costs of emulation. In the Channel Sync algorithm, 2δ is an *upper bound* on the delay when there is Byzantine behavior whereas the total delay can be as low as 0.

TABLE III
SOLVABILITY OF CAUSAL ORDERING USING DETERMINISTIC ALGORITHMS IN SYNCHRONOUS SYSTEMS UNDER DIFFERENT COMMUNICATION MODES

Mode of communication	SS + L	SS + L with cryptography	WS + L	WS + L with cryptography
Unicasts	No, [32] $\overline{SS}, L$	Yes, [33] SS, L	Yes, Algorithms 1, 2 WS, L	Yes, asynchronous WS, L
Broadcasts	No, [32] $\overline{SS}, L$	Yes, [16], [32] SS, L	Yes, asynchronous WS, L	Yes, asynchronous WS, L
Multicasts	No, [32] $\overline{SS}, L$	Yes, [32], [33] SS, L	Yes, [5], [6] WS, L	Yes, asynchronous WS, L

SS = strong safety, WS = weak safety, L = liveness, $\overline{SS}$, $\overline{WS}$, $\overline{L}$ represent that strong safety, weak safety, liveness, respectively, cannot be guaranteed.

To ensure Byzantine-tolerant causal order, the Channel Sync algorithm synchronizes on a per message basis ($2(n-2)$ control messages of size $O(1)$ each) and all concurrent messages are synchronized independently but concurrently. This minimizes the delay experienced by a message from the time of sending to the time of Byzantine-tolerant causal delivery, while factoring out the effects of Byzantine processes and allowing the application program to be asynchronous (in the synchronous system) without the restricting paradigm of rounds.

VIII. CONCLUSION

This paper analyzed the solvability of Byzantine-tolerant causal order under strong and weak safety and under liveness in a deterministic manner in asynchronous systems for unicasts, broadcasts, and multicasts, without and with cryptography. The results are summarized in Table I. In particular, the results showed that it is impossible to implement Byzantine-tolerant causal order – strong safety or weak safety, and liveness – of unicasts and multicasts in a deterministic manner without using cryptography in an asynchronous system. In view of these negative results, the paper proposed the Sender-Inhibition algorithm and the Channel Sync algorithm for providing weak safety and liveness of Byzantine-tolerant causal order in synchronous systems where there is a network guarantee of an upper bound on message latency. The Sender-Inhibition algorithm uses one control message per application message whereas the Channel Sync algorithm uses $2(n-2)$ control messages per application message. In both algorithms, the control message size is $O(1)$. The Sender-Inhibition algorithm is easy to understand and implement, but has reduced concurrency in the sense that a process cannot have multiple sends outstanding. The Channel Sync algorithm has a non-trivial cost of implementation but provides a very high degree of concurrency. These algorithms can be extended to implement causal multicast, as shown in [5] and [6], respectively.

Table III summarizes the solvability of causal ordering using a deterministic algorithm in synchronous systems and includes some recent results.

REFERENCES

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[2] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 47–76, 1987.
[3] A. Auvolat, D. Frey, M. Raynal, and F. Taïani, "Byzantine-tolerant causal broadcast," *Theor. Comput. Sci.*, vol. 885, pp. 55–68, 2021.
[4] A. Misra and A. D. Kshemkalyani, "Solvability of byzantine fault-tolerant causal ordering problems," in *Proc. Int. Conf. Netw. Syst.*, Cham, Springer International Publishing, 2022, pp. 87–103.
[5] A. Misra and A. D. Kshemkalyani, "Causal ordering in the presence of byzantine processes," in *Proc. 28th IEEE Int. Conf. Parallel Distrib. Syst.*, Nanjing, China, 2022, pp. 130–138. [Online]. Available: https://doi.org/10.1109/ICPADS56603.2022.00025
[6] A. Misra and A. D. Kshemkalyani, "Byzantine fault-tolerant causal ordering," in *Proc. 24th Int. Conf. Distrib. Comput. Netw.*, Kharagpur, India, 2023, pp. 100–109. [Online]. Available: https://doi.org/10.1145/3571306.3571395
[7] A. D. Kshemkalyani and M. Singhal, "Necessary and sufficient conditions on information for causal message ordering and their optimal implementation," *Distrib. Comput.*, vol. 11, no. 2, pp. 91–111, 1998. [Online]. Available: https://doi.org/10.1007/s004460050044
[8] R. Prakash, M. Raynal, and M. Singhal, "An adaptive causal ordering algorithm suited to mobile computing environments," *J. Parallel Distrib. Comput.*, vol. 41, no. 2, pp. 190–204, 1997. [Online]. Available: https://doi.org/10.1006/jpdc.1996.1300
[9] M. Raynal, A. Schiper, and S. Toueg, "The causal ordering abstraction and a simple way to implement it," *Inf. Process. Lett.*, vol. 39, no. 6, pp. 343–350, 1991.
[10] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," in *Proc. Int. Workshop Distrib. Algorithms*, Springer, 1989, pp. 219–232.
[11] F. Mattern and S. Fünfrocken, "A non-blocking lightweight implementation of causal order message delivery," in *Proc. Theory Pract. Distrib. Syst., Int. Workshop*, Springer, 1994, pp. 197–213. [Online]. Available: https://doi.org/10.1007/3-540-60042-6_14
[12] R. Baldoni, A. Mostéfaoui, and M. Raynal, "Causal delivery of messages with real-time data in unreliable networks," *Real Time Syst.*, vol. 10, no. 3, pp. 245–262, 1996. [Online]. Available: https://doi.org/10.1007/BF00383387
[13] F. Adelstein and M. Singhal, "Real-time causal message ordering in multimedia systems," in *Proc. 15th Int. Conf. Distrib. Comput. Syst.*, 1995, pp. 36–43.
[14] R. Friedman and S. Manor, "Causal ordering in deterministic overlay networks," Computer Science Department, Technion, Tech. Rep. CS-2004–04, 2004.
[15] A. Mostefaoui, M. Perrin, M. Raynal, and J. Cao, "Crash-tolerant causal broadcast in O (n) messages," *Inf. Process. Lett.*, vol. 151, 2019, Art. no. 105837.
[16] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," *IACR Cryptol. ePrint Arch.*, vol. 2001, 2001, Art. no. 6. [Online]. Available: http://eprint.iacr.org/2001/006
[17] S. Duan, M. K. Reiter, and H. Zhang, "Secure causal atomic broadcast, revisited," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2017, pp. 61–72.
[18] L. Tseng, Z. Wang, Y. Zhao, and H. Pan, "Distributed causal memory in the presence of byzantine servers," in *Proc. IEEE 18th Int. Symp. Netw. Comput. Appl.*, 2019, pp. 1–8.
[19] K. Huang, H. Wei, Y. Huang, H. Li, and A. Pan, "Byz-gentlerain: An efficient byzantine-tolerant causal consistency protocol," 2021, *arXiv:2109.14189*.
[20] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," 2020, *arXiv: 2012.00472*.

[21] G. Bracha, "Asynchronous byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, pp. 130–143, 1987.
[22] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. 3rd USENIX Symp. Operating Syst. Des. Implementation*, 1999, pp. 173–186.
[23] M. C. Pease, R. E. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980. [Online]. Available: http://doi.acm.org/10.1145/322186.322188
[24] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Prog. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982. [Online]. Available: http://doi.acm.org/10.1145/357172.357176
[25] C. Dwork, N. A. Lynch, and L. J. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988. [Online]. Available: http://doi.acm.org/10.1145/42282.42283
[26] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, pp. 824–840, Oct. 1985.
[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.
[28] S. Moran and Y. Wolfstahl, "Extended impossibility results for asynchronous complete networks," *Inf. Process. Lett.*, vol. 26, no. 3, pp. 145–151, 1987. [Online]. Available: https://doi.org/10.1016/0020-0190(87)90052-4
[29] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning About Knowledge*. Cambridge, MA, USA: MIT Press, 1995. [Online]. Available: https://doi.org/10.7551/mitpress/5803.001.0001
[30] A. Misra and A. D. Kshemkalyani, "Detecting causality in the presence of byzantine processes: There is no holy grail," in *Proc. 21st IEEE Int. Symp. Netw. Comput. Appl.*, 2022, pp. 73–80. [Online]. Available: https://doi.org/10.1109/NCA57778.2022.10013644
[31] B. Awerbuch, "Complexity of network synchronization," *J. ACM*, vol. 32, no. 4, pp. 804–823, 1985.
[32] A. Misra and A. D. Kshemkalyani, "Solvabiity of byzantine fault-tolerant causal ordering: Synchronous systems case," in *Proc. 39th ACM Symp. Appl. Comput.*, 2024.
[33] A. Misra and A. D. Kshemkalyani, "Byzantine fault-tolerant causal order satisfying strong safety," in *Proc. 25th Int. Symp. Stabilization, Saf., Secur. Distrib. Syst.*, 2023, pp. 111–125. [Online]. Available: https://doi.org/10.1007/978-3-031-44274-2_10

Anshuman Misra received the BS degree in computer science from the Vellore Institute of Technology. He is currently working toward the PhD degree with the Department of Computer Science, University of Illinois at Chicago. His research interests are in distributed systems, fault-tolerance, and blockchain.

Ajay D. Kshemkalyani (Senior Member, IEEE) received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1987, and the MS and PhD degrees in computer and information science from the Ohio State University, in 1988 and 1991, respectively. He spent six years with IBM Research Triangle Park working on various aspects of computer networks, before joining academia. He is currently a professor with the Department of Computer Science, University of Illinois at Chicago. His research interests are in distributed computing, distributed algorithms, computer networks, and concurrent systems, and he has published more than 100 articles in top-quality journals and conferences in these areas. In 1999, he received the National Science Foundation Career Award. He has served in various positions (such as general chair, program co-chair, steering committee member, or program committee member) for international conferences such as IEEE ICDCS, IEEE SRDS, ACM PODC, and ICDCN. He has served on the editorial board of the *Elsevier* journal, *Computer Networks* and the *IEEE Transactions on Parallel and Distributed Systems*. He has co-authored a book entitled Distributed Computing: Principles, Algorithms, and Systems (Cambridge University Press, 2008). He is a distinguished scientist of the ACM.