Causal Ordering in the Presence of Byzantine Processes

Anshuman Misra
University of Illinois at Chicago, USA
amisra7@uic.edu

Ajay D. Kshemkalyani
University of Illinois at Chicago, USA
ajay@uic.edu

***Abstract*—Causal ordering of messages in distributed systems is important for capturing application-level semantics. To the best of our knowledge, Byzantine fault-tolerant causal ordering has not been attempted for point-to-point communication in an asynchronous setting. In this paper, we first prove that it is impossible to causally order messages under point-to-point communication in an asynchronous system with one or more Byzantine processes. In the face of this impossibility, we then present an algorithm that can causally order messages under point-to-point communication in the face of Byzantine failures, assuming the network provides a known upper bound on the message latency. We also prove that it is impossible to causally order multicasts in an asynchronous setting with one or more Byzantine processes. We then give an extension of our algorithm for unicasts to provide Byzantine fault-tolerant causal ordering of multicasts under the assumption of a known upper bound on the message latency.**

***Index Terms*—Byzantine fault-tolerance, Causal Order, Causality, Asynchronous message-passing, Unicast, Multicast**

I. Introduction

Causality in distributed systems is important for capturing application-level semantics and is used to solve several problems. Causality is defined by the *happens before* [1] relation on the set of events. Logical clocks [2], [3], can be used to timestamp events (messages as well) in order to capture causality. If message $m1$ causally precedes $m2$ and both are sent to p_i, then $m1$ must be delivered before $m2$ at p_i to enforce causal order. Causal ordering ensures that causally related updates to data occur in a valid manner respecting that causal relation. Causal ordering is used in distributed data stores, fair resource allocation, and collaborative applications such as social networking, multiplayer online gaming, group editing of documents, event notification systems, and distributed virtual environments.

Recently, Byzantine-tolerant causal broadcasts have been considered in [4] and the works in [5]–[7] relied on broadcasts for Byzantine-tolerant shared memory and replicated databases. To the best of our knowledge, there has been no work on Byzantine-tolerant causal ordering of unicasts and multicasts besides our analysis in [8]. It is important to solve this problem under the Byzantine failure model because it mirrors the real world.

Contributions:

1) We prove that causal ordering of unicasts in an asynchronous system with even one Byzantine process is impossible because liveness cannot be guaranteed. The proof is based on the analysis in [8].
2) In view of the above impossibility result, we propose the Sender-Inhibition algorithm for Byzantine causal unicast under a stronger asynchrony model, which has a known upper bound on message latency. Such a system is synchronous. 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 the destination. The algorithm eliminates the $O(n^2)$ message space and time overhead of [9]–[13], where n is the number of processes in the system, and uses one control message of size $O(1)$ per application message sent.
3) We prove that it is impossible to provide causal ordering for multicasts in an asynchronous system with even a single Byzantine process because liveness cannot be guaranteed.
4) We then give an extension of the Sender-Inhibition algorithm to Byzantine fault-tolerant causal multicast, again assuming there is a known upper bound on message latency.

Paper Organization. Section II reviews previous work. Section III gives the system model. Section IV gives the impossibility result of being unable to provide liveness (while maintaining safety) for Byzantine causal unicasts in an asynchronous system. Sections V presents the Sender-Inhibition algorithm for solving Byzantine causal unicast in a synchronous system and its correctness proof. Section VI analyzes Byzantine causal multicast and proves that it is impossible to provide liveness (while maintaining safety) in an asynchronous system. Section VII gives the extension of the Sender-Inhibition algorithm for Byzantine causal multicast in a synchronous system and proves it correct. Section VIII gives a discussion.

II. Previous Work

Algorithms for causal ordering of point-to-point messages under a fault-free model are given in [12], [13]. These algorithms extend to implement causal multicasts in a failure-free setting [10], [11]. The RST algorithm presented in [12] is a canonical algorithm for causal ordering.

There has been significant work on causal broadcasts under various failure models. Causal ordering of broadcast messages under crash failures in asynchronous systems was introduced in [9]. This algorithm requires each message to carry the entire set of messages in its causal past as control information. The algorithm presented in [14] 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 in an asynchronous system with Byzantine failures is proposed in [4]. An analysis of the solvability of Byzantine causal ordering is given in [8]. There has been recent interest in applying the Byzantine fault model in implementing causal consistency in distributed shared memory and replicated databases [5]–[7]. In [6], Byzantine causal broadcast has been used to implement Byzantine eventual consistency. In [7], Byzantine reliable broadcast [15] is used to remove misinformation induced by the combination of asynchrony and Byzantine behaviour. In [5], PBFT (total order broadcast) [16] is used to achieve consensus among non-Byzantine servers regarding the order of client requests. To the best of our knowledge, no other paper has examined the feasibility of or solved causal ordering of unicasts and multicasts in a system with Byzantine failures.

III. System Model

The distributed system is modelled as an undirected graph $G = (P, C)$. Here P is the set of processes communicating asynchronously over a geographically dispersed network. Let n be $|P|$. C is the set of communication channels over which processes communicate by message passing. The channels are assumed to be FIFO. G is a complete graph. 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. We do not consider the use of digital signatures or cryptographic techniques in the system model because of their high cost as well as hidden/implicit assumptions such as bounds on message latency, as in [17], that are inapplicable to truly asynchronous systems.

We first assume an asynchronous system which is defined as one in which there is neither any known fixed upper bound δ on message latency nor any known fixed upper bound ψ on the relative speeds of processors/processes. In contrast, a synchronous system assumes both δ and ψ are known. We prove our impossibility results for Byzantine tolerant causal unicast and multicast in an asynchronous system. In light of the impossibility results, we give our algorithms 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 the algorithms can be said to run in a synchronous system. Alternate algorithms for the synchronous system with performance trade-offs are proposed in [18].

Let e_i^x, where $x \geq 0$, denote the x-th event executed by process p_i. In order to deliver messages in causal order, we require a framework that captures causality as a partial order on a distributed execution. The *happens before* [1] relation, denoted $\rightarrow$, is an irreflexive, asymmetric, and transitive partial order defined over events in a distributed execution that captures 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 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) *The set of messages delivered from any $p_i \in P$ by a process is totally ordered by $\rightarrow$.*
2) *If p_i sent or delivered message m before sending message m', then $m \rightarrow m'$.*
3) *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) *The set of messages delivered from any correct process $p_i \in P$ by any correct process is totally ordered by $\xrightarrow{B}$.*
2) *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 $m \xrightarrow{B} m'$.*
3) *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 $\xrightarrow{B}$.*

The correctness of a Byzantine causal order unicast/multicast/broadcast is specified on $(S, \xrightarrow{B})$ as follows.

Definition 6. *A causal ordering algorithm for unicast/multicast/broadcast messages must ensure the following:*

1) ***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.*

When $m \xrightarrow{B} m'$, then all processes that sent messages along the causal chain from m to m' are correct processes.

IV. Impossibility Result for Unicasts

All existing causal ordering algorithms for unicast messages in asynchronous systems use some form of *logical timestamps*. This principle is abstracted by the RST algorithm [12]. Each message m sent to p_i is piggybacked with a *logical timestamp* in the form of a matrix clock providing information about messages in the causal past of m. This is to ensure that all messages $m' \in CP(m)$ whose destination is p_i are delivered at p_i before m. The implementation is as follows:

1) Each process p_i locally stores (a) a vector $Delivered_i$ of size n, where $Delivered_i[j]$ is the number of messages sent by p_j and delivered by p_i, and (b) a matrix M_i of size $n \times n$, where $M_i[j,k]$ is the number of messages sent by p_j to p_k as known to p_i.
2) When p_i sends message m to p_j, m has a piggybacked matrix timestamp M^m, which is the value of M_i before the send event. Then $M_i[i,j] = M_i[i,j] + 1$.
3) When message m is received by p_i, it is delivered only after the following *delivery condition* is satisfied:
$$\forall k,\ M^m[k,i] \leq Delivered_i[k]$$
4) After delivering a message m, p_i merges the logical timestamp associated with m into its own matrix clock, as $\forall j,k,\ M_i[j,k] = \max(M_i[j,k], M^m[j,k])$.

In order to disrupt causal delivery of messages in asynchronous systems, a Byzantine process may fabricate values in the logical timestamps of its messages. In general, causal order of messages can be enforced by either: (a) performing appropriate actions at the receiver's end, or (b) taking appropriate actions at the sender's end.

To enforce causal ordering at the receiver's end, one needs to track causality, and some form of a logical clock is required to causally order messages. Traditionally, logical clocks use transitively collected control information attached to each incoming message for this purpose. The RST abstraction [12], described above is used. However, in case there is a single Byzantine process, it can cause a change in the values of the matrix timestamp piggybacked on a message it sends. Lemma 1 proves that transitively collected control information by a receiver can lead to liveness attacks in an asynchronous system with one or more Byzantine processes.

Next, we examine the possibility of appropriate action at the sender's end to ensure causal ordering. Such action is in the form of constraints on when the sending process can send messages in order to prevent causal violations. A sender process would need get an acknowledgement from the receiver before sending the next message. For increased concurrency and avoiding deadlocks, while waiting for an acknowledgment, each process would continue to receive and deliver messages. This can be implemented by using non-blocking synchronous sends, with the added constraint that all send events are *atomic* with respect to each other. Lemma 2 proves that this approach is also vulnerable to liveness attacks in the presence of one or more Byzantine processes. Theorem 1 combines these results and proves the impossibility of causally ordering unicast messages in asynchronous systems with Byzantine processes.

Lemma 1. *A single Byzantine process can execute a liveness attack when control information for causality tracking is transitively propagated and used by a receiving process for enforcing safety of causal ordering of unicasts.*

Proof. Transitively propagated control information for causality tracking, whether by explicitly maintaining the counts of the number of messages sent between each process pair, or by maintaining causal barriers, or by encoding the dependency information optimally or by any other mechanism, can be abstracted by the causal ordering abstraction [12], described earlier in this section. Each message m sent to p_k is accompanied with a *logical timestamp* in the form of a matrix clock providing an encoding of $CP(m)$. The encoding of $CP(m)$ effectively maintains an entry to count the number of messages sent by p_i to p_j, $\forall p_i, p_j \in P$. Such an encoding will consist of a total of n^2 entries, n entries per process. Therefore, in order to ensure that all messages $m' \in CP(m)$ whose destination is p_k are delivered at p_k before m, the matrix clock M whose definition and operation was reviewed earlier in this section is used to encode $CP(m)$.

Let $m' \xrightarrow{B} m$, where m' and m are sent by p_i and p_j, respectively, to common destination p_k. The value $M_i[i,k]$ after sending m' propagates transitively along the causal chain of messages to p_j and then to p_k. But before p_j sends m to p_k, it has received a message m'' (transitively) from a Byzantine process p_x in which $M^{m''}[y,k]$ is artificially inflated (for a liveness attack using $M^{m''}[y,k]$). This inflated value propagates on m from p_j to p_k as $M^m[y,k]$. To enforce safety between m' and m, p_k implements the delivery condition in rule 3 of the RST abstraction, and will not be able to deliver m because of p_x's liveness attack wherein $M^m[y,k] \not\leq Delivered_k[y]$. p_k uniformly waits for messages from any process(es) that prevent the delivery condition from being satisfied and thus waits for $M^m[y,k] - Delivered_k[y]$ messages from p_y, which may never arrive if they were not sent. (If p_k is not to keep waiting for delivery of the arrived m, it might try to flush the channel from p_y to p_k by sending a *probe* to p_y and waiting for the *ack* from p_y. This approach can be seen to violate liveness, e.g., when p_x attacks p_k via p_i on $M^{m'}[j,k]$ and via p_j on $M^m[i,k]$. When p_x causes p_i to send $M^{m'}[j,k]$ to p_k with an inflated value, p_k will send a *probe* to p_j and wait for its *ack* before delivering m'. Similarly, when p_x causes p_j to send $M^m[i,k]$ to p_k with an inflated value, p_k will send a *probe* to p_i and wait for its *ack* before delivering m. As either *ack* may arrive first, neither m nor m' can be delivered; thus this mechanism cannot be used to provide liveness while guaranteeing safety. Morever, p_y may never reply with the *ack* if it is Byzantine, and p_k has no means of differentiating between a slow channel to/from a correct p_y and a Byzantine p_y that may never reply. So p_k waits indefinitely.) Therefore, the system is open to liveness attacks in the presence of a single Byzantine process. □

Lemma 2. *A single Byzantine process can execute a liveness attack in an asynchronous message passing system even if a sending process sends a message only when the receiving process is guaranteed not to be subject to a safety attack, i.e., only when it is safe to send the message and hence its delivery at the receiver will not violate safety, on causal order of unicasts.*

Proof. The only way that a sending process p_i can ensure safety of a message m it sends to p_j is to enforce that all messages m' such that $m \xrightarrow{B} m'$ and m' is sent to p_j will reach the (common) destination p_j after m reaches p_j. Assuming FIFO delivery at a process based on the order of arrival, m will be delivered before m'.

The only way the sender p_i can enforce that m' will arrive after m at p_j is not to send another message to any process p_k after sending m until p_i knows that m has arrived at p_j. p_i can know m has arrived at p_j only when p_j replies with an *ack* to p_i and p_i receives this *ack*. However, p_i cannot differentiate between a malicious p_j that never replies with the *ack* and a slow channel to/from a correct process p_j. Thus, p_i will wait indefinitely for the *ack* and not send any other message to any other process. This is a liveness attack by a Byzantine process p_j. □

Theorem 1. *It is impossible to guarantee liveness and safety while causally ordering point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. From Lemmas 1 and 2, no actions at a receiver or at a sender can prevent a liveness attack (while maintaining safety). The theorem follows. □

Lemma 1 considers receiver-initiated strategies, Lemma 2 considers sender-initiated strategies. These are the only two possible options. One cannot use a Byzantine fault-tolerant (BFT) causal broadcast protocol to implement point-to-point abstraction by adding recipient_id and filtering on arrival only those messages intended for the local process because the filtering mechanism at the local process can be voided/compromised if the local process is Byzantine. Furthermore, 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 process 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 not possible without the use of cryptographic primitives, which are not considered as mentioned in the system model.

Note that we cannot use a protocol that piggybacks on messages the entire causal history of messages because a unicast from p_i to p_j (or a multicast to group G) needs to be kept private to p_j (or to G).

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
$lck.release()$

V. Sender-Inhibition Algorithm

As a result of Theorem 1, we know that it is impossible to maintain both safety and liveness while trying to causally order messages in an asynchronous system with Byzantine faults. However, it is possible to extend the idea presented in Lemma 2 and develop a solution 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 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 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 can execute at a given point in time, multiple deliver and multiple receive events can occur concurrently with a single send event.

Theorem 2. *Under a network guarantee of delivering messages within δ time, Algorithm 1 ensures liveness while maintaining 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 safety. □

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.

VI. Impossibility Result for Byzantine Causal Multicasts

In a multicast, a send event sends a message to multiple destinations that form a subset of the process set P. Different send events by the same process can be addressed to different subsets of P. This models dynamically changing multicast groups and membership in multiple multicast groups. In the general case, there are $2^{|P|} - 1$ groups. Although there are several algorithms for causal ordering of messages under dynamic groups, such as [10], [11], none of them consider the Byzantine failure model.

Byzantine Reliable Multicast (BRM) [19], [20] has traditionally been defined based on Bracha's Byzantine Reliable Broadcast (BRB) [15], [21]. These algorithms require that in every multicast group G, less then $|G|/3$ processes are Byzantine. When a process does a multicast, it invokes `br_multicast` and when it is to deliver such a message, it executes `br_deliver`. In the discussion below, it is assumed that a message is uniquely identified by a (sender ID, seq_num) tuple. BRM satisfies the following properties.

- Validity: If a correct process `br_delivers` a message m from a correct process p_s, then p_s must have executed `br_multicast(m)`.
- Integrity: For any message m, a correct process executes `br_deliver` at most once.
- Self-delivery: If a correct process executes `br_multicast(m)`, then it eventually executes `br_deliver(m)`.
- Reliability (or Termination): If a correct process executes `br_deliver(m)`, then every other correct process in the multicast group G also (eventually) executes `br_deliver(m)`.

As causal multicast is an application layer property, it runs on top of the BRM layer. Byzantine Causal Multicast (BCM) is invoked as `bc_multicast`(m) which in turn invokes `br_multicast`(m') to the BRM layer. Here, m' is m plus some control information appended by the BCM layer. A `br_deliver`(m') from the BRM layer is given to the BCM layer which delivers the message m to the application via `bc_deliver(m)` after the processing in the BCM layer.

BCM needs to satisfy BC_Validity, BC_Integrity, BC_Self-Delivery, and BC_Reliability which are the counterparts of the above four properties with `br_multicast` and `br_deliver` replaced by `bc_multicast` and `bc_deliver`, respectively. In addition to these properties, BCM must satisfy safety and liveness as described in Section III. Observe that safety (+ liveness) needs to hold only for the $\xrightarrow{B}$ relation on messages, which are the messages sent by and received by only correct processes.

All the existing algorithms for causal multicast use transitively collected control information about causal dependencies in the past – they vary in the size of the control information, whether in the form of causal barriers as in [11], [22] or in the optimal encoding of the theoretically minimal control information as in [10], [23]. The RST algorithm still serves as a canonical algorithm for the causal ordering of multicasts in the BCM layer, and it can be seen that the same liveness attack described in Lemma 1 can be mounted on the causal multicast algorithms.

The intuitive reason for this is given below before proving the impossibility result for Byzantine Causal Multicast. A liveness attack is possible in the point-to-point model because a "future" message m from p_i to p_j can be advertised by a Byzantine process p_x, i.e., the dependency can be transitively propagated by p_x via $p_{x_1} \ldots p_{x_y}$ to p_j, without that message m actually having been sent (created). When the advertisement reaches p_j it waits indefinitely for m. Had a copy of m also been transitively propagated along with its advertisement, this liveness attack would not have been possible. But in point-to-point communication, m must be kept private to p_i and p_j and cannot be (transitively) propagated along with its advertisement. The same logic holds for multicasts – p_i can withold a multicast m to group G_x but advertise it on a later multicast m' to group G_y where, say $G_x \cap G_y = \{p_i\}$, even if using Byzantine Reliable Multicast (BRM) which guarantees all-or-none delivery to members of G_y. When a member of G_y receives m', it also receives the advertisement "m sent to $p_j(\in G_x)$", which may get transitively propagated to p_j which will wait indefinitely. Therefore, results for unicasts also hold for multicasts.

In contrast, in Byzantine causal broadcast [4], the underlying Byzantine Reliable Broadcast (BRB) layer which guarantees that a message is delivered to all or none of the (correct) processes ensures that the message m is not selectively withheld. This m propagates from p_i to p_j (directly, as well as indirectly/transitively in the form of (possibly a predecessor of) entries in the causal barriers) while simultaneously guaranteeing that m is actually eventually delivered from p_i to p_j by the BRB layer. Thus a liveness attack is averted in the broadcast model.

Lemma 3. *It is impossible to guarantee BC_Reliability and liveness when transitively propagated control information is used for ensuring safety while causally ordering multicast messages in an asynchronous message passing system with one or Byzantine processes.*

Proof. For this proof, we assume the existence of a Byzantine Reliable Multicast (BRM) primitive that ensures the conditions of validity, integrity, self-delivery, reliability (or termination). BRM is invoked by the Byzantine Causal Multicast (BCM) layer as `br_multicast(`m`)`, where m is the multicast message (along with any associated control information appended by the sender's BCM layer). The BCM layer then delivers the message based on the RST protocol abstraction.

Here, we prove that only four out of six of the essential properties of Byzantine Causal Multicast can be satisfied.

BC_Validity: Since the BRM layer guarantees validity, if the BCM layer at a correct process executes `bc_deliver(m)`, it means that the sender which is a correct process must have executed `bc_multicast(m)` that triggered `br_multicast(m)` at it.

BC_Integrity: At a correct process, the BRM layer delivers messages to the BCM layer. As the BRM layer executes `br_deliver(m)` at most once for any message m, the BCM layer executes `bc_deliver(m)` at most once since the BCM layer delivers messages as described by the RST abstraction protocol.

BC_Self-Delivery: If a correct process p_i executes `bc_multicast(m)`, it will certainly execute `br_deliver(m)` as a result of the Self_delivery property of the BRM layer. Then, the BCM layer will eventually execute `bc_deliver(m)` because $\forall x, M^m[x,i] \leq Delivered_i[x]$.

BC_Reliability: Consider the following counter-example where only two messages are exchanged: a Byzantine process p_i sends the first multicast message that is delivered at the BCM layer at p_j with a boosted value of $M_i[i,x]$. Next p_j sends a multicast message m to group $G = \{p_x, p_y\}$. p_y executes `bc_deliver(m)` since $M^m[*,y] = 0$. Although p_x is guaranteed to execute `br_deliver(m)`, p_x will never execute `bc_deliver(m)` since $M^m[i,x] > 0$ and $M^m[i,x] > Delivered_x[i]$ as p_i has not sent any messages to p_x. Therefore, BC_Reliability is violated.

Safety: By definition, correct processes do not artificially reduce the values of their local matrix clocks. As a result of that, $m_1 \in BCP(m_2)$ ensures that $\forall x, y, M^{m_1}[x,y] \leq M^{m_2}[x,y]$ and $\exists x, y$ such that $M^{m_1}[x,y] < M^{m_2}[x,y]$. By enforcing the Delivery Condition of the RST abstraction protocol, safety is seen to be guaranteed.

Liveness: The control information piggybacked by the BCM layer on each message m is $CP(m)$ in the form of a matrix clock M^m prior to invoking `br_multicast(`m`)`. When a message m from the BRM layer is received by the BCM layer by executing `br_deliver(m)`, the BCM layer extracts M^m from m. The BCM layer (at a correct process p_k) is now susceptible to the liveness attack described in Lemma 1. □

Lemma 4. *A single Byzantine process can execute a liveness attack in an asynchronous system even if a sending process multicasts a message only when it is safe to do so.*

Proof. We analyze all six properties of Byzantine Causal Multicast.

Safety: The only way that a sending process p_i can ensure safety of a message m it sends to p_j without causality tracking control information is to enforce that (i) all messages m' such that $m \xrightarrow{B} m'$ and m' is sent to p_j will reach the (common) destination p_j after m reaches p_j, and (ii) before sending m, all messages m'' such that m'' has been locally queued for `bc_delivery` have also been locally queued at all recipients of that multicast. Here, messages delivered from the BRM layer are queued and delivered in FIFO order by the BCM layer. In order to enforce (i) and (ii), p_i will require acknowledgments $ack1$ confirming that m has been queued for delivery, from every process in m's multicast group G, and then provide these processes with an acknowledgment $ack2$ that "the multicast message has been queued for delivery at all members of G" upon receiving the required $ack1$s. This means that p_i will have to require each recipient p_x of its multicast to wait to get an acknowledgment $ack2$ from p_i that "the multicast has been queued for delivery at each recipient," before p_x issues its next multicast. And upon multicasting m, p_i will also have to wait for an acknowledgement $ack1$ from each process in the multicast group of m before executing its next multicast. Waiting for these acknowledgments ensures that (i) m arrives in the FIFO queue at each process before messages m' that it causally precedes, and (ii) m arrives in the FIFO queue at each process after messages m'' that causally precede it. As a result safety is guaranteed because m will be delivered before m' and after m''.

BC_Reliability: The above logic for safety also guarantees BC_Reliability since the BRM layer guarantees reliability, and all processes that execute `br_deliver(m)` will execute `bc_deliver(m)` once m reaches the head of the FIFO queue.

Liveness: However, p_i cannot differentiate between a malicious p_j in the multicast group that never replies with the *ack1* and a slow channel to/from a correct process p_j. Thus, p_i and all correct processes in the multicast group will wait indefinitely for the *ack1* and *ack2*, respectively, and not send any other message to any other process. This is a liveness attack (while maintaining safety) by a Byzantine process p_j.

BC_Validity: Since the BRM layer guarantees validity, if the BCM layer at a correct process executes `bc_deliver(m)`, it means that the sender which is a correct process must have executed `bc_multicast(m)` that triggered `br_multicast(m)` at it.

BC_Integrity: At a correct process, the BRM layer delivers messages to the BCM layer. As the BRM layer executes `br_deliver(m)` at most once for any message m, the BCM layer executes `bc_deliver(m)` at most once since the BCM layer enqueues the `br_deliver`ed message at most once for `bc_delivery`.

BC_Self-Delivery: If a correct process p_i executes `bc_multicast(m)`, that will invoke `br_multicast(m)` and then it will certainly execute `br_deliver(m)` as a result of the Self_delivery property of the BRM layer. Then, the BCM layer will enqueue the message for `bc_delivery` and the process will eventually execute `bc_deliver(m)`.

From the above, it follows that liveness cannot be satisfied. □

Theorem 3. *It is impossible to guarantee liveness and safety while causally ordering multicast messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. From Lemmas 3 and 4, no actions at a receiver or at a sender can prevent a liveness attack while maintaining safety for causally ordering multicast messages. The theorem follows. □

VII. Sender-Inhibition Algorithm for Byzantine Causal Multicast

The idea of the Sender-Inhibition algorithm can be extended to design an algorithm for Byzantine Causal Multicast, also in a system that provides a known upper bound δ on the message latency.

A. *Sender-Inhibition Algorithm for Multicast over BRM Layer*

Let δ_{BRM} denote the maximum time for the BRM protocol. ($\delta_{BRM} = 3\delta$ when BRM is based on Bracha's algorithm [15], [21].) If the four specifications corresponding to those of BRM are required, the adaptation of the Sender-Inhibition algorithm (Algorithm 1), as given in Algorithm 2, can be used. The timer $timer\_s$ in line (14) is set to $\delta_{BRM} + \delta$: δ_{BRM} (=3δ for the three phases of Bracha-based BRM) and δ for the ack_r to arrive. The timer $timer\_r$ in line (8) is set to $\delta_{BRM} + 2\delta$, because that is the maximum time it can take for a ack_s to arrive from the sender from the time of sending the multicast. The main modifications to Algorithm 1 are: (a) the sender waits for all the recipients in the multicast group of message m it multicast to receive m and add it to their delivery queues before initiating the next multicast, and (b) a receiver waits to know from the sender that the message m has been added to the delivery queue of each recipient of that multicast, before that receiver initiates its next multicast. Note that the acknowledgement messages are not sent via the BRM layer.

Algorithm 2: Byzantine Causal Multicast based on the Sender-Inhibition Algorithm, on top of BRM layer.

Data: Each p_i maintains a FIFO queue Q and a lock lck

1 **when** application is ready to process a message:
  ▷ `Deliver event`
2 $m = Q.pop()$
3 **if** $m \neq \phi$ **then**
4   `bc_deliver(`m`)`

5 **when** message m arrives from p_j via `br_deliver(m,G)`:
  ▷ `Receive event`
6 $Q.push(m)$
7 $send(ack\_r, j)$ to p_j
8 start $timer\_r$

9 **when** message m is ready to be sent to G via
  `bc_multicast(m,G)`: ▷ `Send event`
10 $lck.acquire()$ ▷ `Executes atomically`
11 **while** $\exists m'$ *such that* m' *has been* `bc_delivered` *prior to this* `bc_multicast` *being issued* $\wedge$ *(*ack_s *for* m' *has not been received* $\wedge$ $timer\_r$ *for* m' *has not timed out)* **do**
12   wait in a nonblocking manner
13 `br_multicast(m,G)`
14 start $timer\_s$
15 **while** ack_r *for message* m *not arrived from each* $p_j \in G$ $\wedge$ $timer\_s$ *not timed out* **do**
16   wait in a nonblocking manner
17 **if** ack_r *has arrived from each* $p_j \in G$ **then**
18   send ack_s to each $p_j \in G$
19 $lck.release()$

Theorem 4. *Under a network guarantee of delivering messages within δ time, Algorithm 2 ensures BC_Validity, BC_Integrity, BC_Self-delivery, BC_Reliability, safety and liveness.*

Proof. Algorithm 2 utilizes the `br_multicast` and `br_deliver` primitives implementing BRM as the underlying layer. Algorithm 2 guarantees BC_Validity, BC_Integrity, BC_Self-delivery, BC_Reliability by utilizing the corresponding guarantees provided by the BRM layer as follows.

BC_Validity: If a correct process executes `br_deliver(m)` from a correct process p_s, it will also execute `bc_deliver(m)` (lines 5-6, 1-4), and since the BRM layer guarantees validity, the sender (correct) process p_s must have executed `br_multicast(m)`. The sender correct process p_s must then have executed `bc_multicast(m)` (from lines 9-13).

BC_Integrity: At a correct process, the BRM layer delivers messages to the BCM layer by pushing messages into a FIFO queue as seen in lines 5-6. Since the BRM layer executes `br_deliver(m)` at most once for any message m, each message is placed in the queue at most once. Each message in the queue is delivered by the BCM layer only once as seen in lines 1-4.

BC_Self-Delivery: If a correct process p_i executes `bc_multicast(m)`, and it is present in the multicast group G, it will `br_multicast` m to G (lines 9, 13), then it will `br_deliver` message m into the FIFO queue in lines 5-6 and eventually `bc_deliver` m from the queue in lines 1-4.

BC_Reliability: When a correct process executes `bc_deliver(m)` for any message m, it means that it must have executed `br_deliver(m)` as seen in lines 1-8. By the reliability property provided by the BRM layer, all correct processes in the group G will eventually execute `br_deliver(m)` (lines 5-8) and place m in their FIFO delivery queues. This means that they will eventually execute `bc_deliver(m)` as seen in lines 1-4.

The remainder of this proof accounts for safety and liveness.

Safety: Consider a message m multicast by a correct process p_j in the FIFO delivery queue of a correct process p_i. Consider $m' \in BCP(m)$ such that m' has been received by p_i and is in the FIFO delivery queue. m' may be one of the following:

1) m' was multicast before m by p_j. p_j waits for all (correct) processes to acknowledge that they have received m' or for $timer\_s$ for m' to time out before multicasting m. The correct processes (including p_i) in the multicast group acknowledge the reception of m' by sending ack_r to p_j. All ack_r will arrive at p_j within the timeout period due to the network guarantee of delivering messages within a bounded time period or $timer\_s$ at p_j times out. In either case, m' arrives before m in the delivery queue at p_i.
2) m' was multicast by a correct process p_k and delivered by p_j before p_j multicast m. After multicasting m', p_k waits for all (correct) processes in the multicast set to acknowledge that they have received m' (by sending ack_r to p_k) or until its $timer\_s$ times out. In either case, correct process p_i's ack_r would have reached p_k. If $timer\_s$ has not timed out, p_k informs p_j that the multicast is complete by sending ack_s to p_j. Once p_j delivers m', it waits for ack_s to arrive from p_k or its $timer\_r$ to time out before initiating a multicast for message m. In either case, m' will arrive at the delivery queue at p_i before m.
3) $m'(= m_0) \xrightarrow{D} m_1 \xrightarrow{D} \ldots \xrightarrow{D} m_{t-1} \xrightarrow{D} (m_t =)m$, where $\xrightarrow{D}$ is a subset of $\xrightarrow{B}$ and $m_x \xrightarrow{D} m_y$ is defined as m_x was either delivered at or multicast by a process and m_y was the next message multicast by the same process. We now have the following:
   a) As $m'(= m_0) \xrightarrow{D} m_1$, one of the following holds for m_1:
      i) m_1 is multicast by p_k after multicasting m_0. In this case, by Observation 1, m_1 will arrive at all its destinations after m_0 has arrived at its destinations.
      ii) m_1 is multicast by some (correct) process p_h after delivering m_0. In this case, by Observation 2, m_1 will arrive at its destinations after m_0 has arrived at its destinations.
   b) Using logic similar to (a) above, $\forall l \in [1, t]$, m_l will arrive at its destinations after m_{l-1} arrives at its destinations. Transitively, this implies that $m_t(= m)$ will arrive at p_i after $m_0(= m')$ arrives at the common destination p_i.

   Since m necessarily arrives after m' in the FIFO delivery queue at p_i, it will get delivered after m' ensuring safety under $\xrightarrow{B}$.

In all the cases, safety is seen to be satisfied.

Liveness: A Byzantine process may try to attack the liveness of the system by either:

1) not sending ack_r to the multicast sender process after receiving a multicast, or
2) not sending ack_s to one or more receiver processes of a multicast that it has sent out.

In both of these cases, a correct process waiting for the acknowledgement message will stop waiting after the timeout period ($timer\_s$ or $timer\_r$, respectively) expires and will be free to initiate its next multicast, because messages to/from Byzantine processes are not part of $\xrightarrow{B}$. In other words, if a sender process does not receive ack_r from a receiver process, it does not have to worry about a safety violation at/involving the receiver because messages sent by a Byzantine process are not part of $\xrightarrow{B}$. Similarly, if a receiver process does not receive ack_s from the sender process of a multicast (because the sender or one of the receivers is Byzantine), it does not have to worry about a safety violation because messages sent by a Byzantine process are not part of $\xrightarrow{B}$ and all correct receivers are guaranteed to have enqueued the multicast message before the timeout of $timer\_r$.

Therefore, Algorithm 2 guarantees liveness while maintaining safety for multicast messages. $\square$

B. Sender-Inhibition Algorithm for Multicast without BRM Layer

The four specifications of BCM may not strictly be required because (i) they are expensive to implement, requiring $O(|G|^2)$ messages in the BRM layer and a latency of δ_{BRM} to implement BRM, (ii) the application is interested only in thwarting Byzantine attacks on the safety and liveness of the multicasts, and/or (iii) the constraint that less than $|G|/3$ processes in each group may be Byzantine cannot be satisfied. If so, the Sender-Inhibition Algorithm can implement causal order of multicasts without using the Byzantine Reliable Multicast primitive. Intuitively, each multicast message m with $|G| = k$ ($k > 1$), can be considered as k unicast messages with the same contents as m each directed to one of the k receivers. The changes to the unicast algorithm are: (i) the sender waits for k acks from the k receivers before sending acks to the k

receivers and the next multicast, and (ii) each receiver waits for an ack from the sender before it can multicast its next message. This is to avoid the following causality violation. Let p_i multicast m_1 to processes p_j and p_k. When p_j delivers m_1 and multicasts message m_2 to p_k (p_k is part of the multicast group), we have $m_1 \xrightarrow{B} m_2$. If m_2 arrives at p_k before m_1, it will get enqueued prior to m_1 and therefore will get delivered before m_1 resulting in a causality violation. Therefore, p_j must wait for an acknowledgement from p_i informing it that p_k has enqueued m_1 before sending its next multicast m_2 to avoid this potential causality violation.

The following changes will need be made to Algorithm 2. (i) The `br_multicast(m,G)` in line (13) would be replaced by just "send (m, G) to each $x \in G$" and the `br_deliver(m,G)` in line (5) would be replaced by arrival of a message (m, G). (ii) $timer\_s$ would reduce from $\delta_{BRM} + \delta$ to 2δ and $timer\_r$ would reduce from $\delta_{BRM} + 2\delta$ to 3δ.

VIII. Discussion

Complexity: The Sender-Inhibition algorithm for unicasts uses one control message per application message sent. Each sender has to wait to know that its message has been received before sending the next message. The Sender-Inhibition algorithm thus requires up to a round-trip message transmission delay between two consecutive send events at a process. However it is very easy to implement. For multicasts, we do not count the message and space overheads of the BRM layer. The Sender-Inhibition algorithm for multicasts uses $2|G|$ point-to-point control messages per multicast to G, a delay up to $\delta_{BRM} + \delta$ at the sender, and a delay up to $\delta_{BRM} + 2\delta$ at each receiver. The algorithms for both unicast and multicast eliminate the $O(n^2)$ message space and processing time overhead of RST and other algorithms [9]–[13], and instead use very small $O(1)$ control messages.

Synchronization mechanism in the algorithms. In view of the impossibility results, the algorithms we presented are in a synchronous system model without the requirement of lock-step execution. 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 [24] in a fault-free asynchronous system. It is not possible to design synchronizers under Byzantine failures. Our algorithms are designed for asynchronous applications that do not use lock-step in their code. If lock-step were used, 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 our Sender-Inhibition Algorithm for unicasts, 2δ is the *upper bound* on the waiting time for the sender in case of a Byzantine receiver. This delay can be as low as 0. Additionally, processes (including senders waiting for a response) can receive messages without a waiting period leading to increased concurrency.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM 21, 7*, pp. 558–565, 1978.

[2] C. J. Fidge, "Logical time in distributed computing systems," *IEEE Computer*, vol. 24, no. 8, pp. 28–33, 1991.

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

[4] A. Auvolat, D. Frey, M. Raynal, and F. Taïani, "Byzantine-tolerant causal broadcast," *Theoretical Computer Science*, vol. 885, pp. 55–68, 2021.

[5] K. Huang, H. Wei, Y. Huang, H. Li, and A. Pan, "Byz-gentlerain: An efficient byzantine-tolerant causal consistency protocol," *arXiv preprint arXiv:2109.14189*, 2021.

[6] M. Kleppmann and H. Howard, "Byzantine eventual consistency and the fundamental limits of peer-to-peer databases," *arXiv preprint arXiv:2012.00472*, 2020.

[7] L. Tseng, Z. Wang, Y. Zhao, and H. Pan, "Distributed causal memory in the presence of byzantine servers," in *IEEE 18th International Symposium on Network Computing and Applications (NCA)*, 2019, pp. 1–8.

[8] A. Misra and A. D. Kshemkalyani, "Solvability of byzantine fault-tolerant causal ordering problems," in *Networked Systems*, M.-A. Koulali and M. Mezini, Eds. Cham: Springer International Publishing, 2022, pp. 87–103. [Online]. Available: https://doi.org/10.1007/978-3-031-17436-0_7

[9] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, 1987.

[10] A. D. Kshemkalyani and M. Singhal, "Necessary and sufficient conditions on information for causal message ordering and their optimal implementation," *Distributed Comput.*, vol. 11, no. 2, pp. 91–111, 1998. [Online]. Available: https://doi.org/10.1007/s004460050044

[11] R. Prakash, M. Raynal, and M. Singhal, "An adaptive causal ordering algorithm suited to mobile computing environments," *J. Parallel Distributed Comput.*, vol. 41, no. 2, pp. 190–204, 1997.

[12] M. Raynal, A. Schiper, and S. Toueg, "The causal ordering abstraction and a simple way to implement it," *Information processing letters*, vol. 39, no. 6, pp. 343–350, 1991.

[13] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," in *International Workshop on Distributed Algorithms*. Springer, 1989, pp. 219–232.

[14] A. Mostefaoui, M. Perrin, M. Raynal, and J. Cao, "Crash-tolerant causal broadcast in o (n) messages," *Information Processing Letters*, vol. 151, p. 105837, 2019.

[15] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[16] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, M. I. Seltzer and P. J. Leach, Eds., pp. 173–186.

[17] S. Duan, M. K. Reiter, and H. Zhang, "Secure causal atomic broadcast, revisited," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2017, pp. 61–72.

[18] A. Misra and A. D. Kshemkalyani, "Byzantine fault-tolerant causal ordering," in *Proceedings of the 24th International Conference on Distributed Computing and Networking (ICDCN)*, 2023, to appear.

[19] D. Malkhi, M. Merritt, and O. Rodeh, "Secure reliable multicast protocols in a WAN," in *Proceedings of the 17th International Conference on Distributed Computing Systems*, 1997, pp. 87–94.

[20] D. Malkhi and M. K. Reiter, "A high-throughput secure reliable multicast protocol," *J. Comput. Secur.*, vol. 5, no. 2, pp. 113–128, 1997.

[21] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, p. 824–840, Oct. 1985.

[22] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," *Tech Report 94-1425, Cornell University*, p. 83 pages, 1994.

[23] A. D. Kshemkalyani and M. Singhal, "An optimal algorithm for generalized causal message ordering (abstract)," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, J. E. Burns and Y. Moses, Eds. ACM, 1996, p. 87. [Online]. Available: https://doi.org/10.1145/248052.248064

[24] B. Awerbuch, "Complexity of network synchronization," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 804–823, 1985.