Solvability of Byzantine Fault-Tolerant Causal Ordering Problems

Anshuman Misra and Ajay D. Kshemkalyani(✉)

University of Illinois at Chicago, Chicago, IL 60607, USA
{amisra7,ajay}@uic.edu

Abstract. Causal ordering in an asynchronous setting is a fundamental paradigm for collaborative software systems. Previous work in the area concentrates on ordering messages in a faultless setting and on ordering broadcasts under various fault models. To the best of our knowledge, Byzantine fault-tolerant causal ordering has not been studied for unicasts and multicasts in an asynchronous setting. In this paper we first show that protocols presented in previous work fail for unicasts and multicasts under Byzantine faults in an asynchronous setting. Then we analyze, propose, and prove results on the solvability of the related problems of causal unicasts, multicasts, and broadcasts in an asynchronous system with one or more Byzantine failures.

Keywords: Byzantine fault-tolerance · Causal order · Broadcast · Causality · Asynchronous · Message-passing

1 Introduction

Causality is an important tool in reasoning about distributed systems [15]. Theoretically causality is defined by the *happens before* [16] relation on the set of events. In practice, logical clocks [17] are 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 [2]. 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 social networks, multiplayer online gaming, group editing of documents, event notification systems, and distributed virtual environments.

The only work on causal ordering under the Byzantine failure model is the recent result by Auvolat et al. [1] which considered Byzantine-tolerant causal broadcasts, and the work in [11,12,25] which relied on broadcasts. To our knowledge, there has been no work on Byzantine-tolerant causal ordering of unicasts and multicasts. It is important to solve this problem under the Byzantine failure model as opposed to a failure-free setting because it mirrors the real world.

M.-A. Koulali and M. Mezini (Eds.): NETYS 2022, LNCS 13464, pp. 87–103, 2022.
https://doi.org/10.1007/978-3-031-17436-0_7

Table 1. Solvability of Byzantine causal unicast, broadcast, and multicast in a fully asynchronous setting. Results for multicasts are the same as for unicasts, see Sect. 7. FIP = Full-Information Protocol.

Problem	Model	Liveness + Weak safety	Weak safety − Liveness	Strong safety + Liveness	Strong safety − Liveness
Unicast	No signatures	(1) no	(2) yes	(3) no	(4) no
Sect. 5		Theorem 1	Theorem 2	Theorem 3	Theorem 4
	w/ signatures	(5) no[a]	(6) yes	(7) no	(8) no
		Theorem 5	implied by Theorem 2	Theorem 6	Theorem 7
	FIP	(9) yes	(10) yes	(11) no	(12) no
		Sect. 8	implied by Theorem 2	Sect. 8	Sect. 8
Broadcast	No signatures	(13) yes	(14) yes	(15) no	(16) no
Sect. 6		algorithm in [1]	algorithm in [1]	Theorem 8	Theorem 9
	w/signatures	(17) yes	(18) yes	(19) no	(20) no
		implied by (13)	implied by (14)	Theorem 10	Theorem 11
	FIP	(21) yes	(22) yes	(23) no	(24) no
		implied by (13)	implied by (14)	Sect. 8	Sect. 8

[a] This is "yes" if the Byzantine processes are rational, see Sect. 8.

The main contributions of this paper are as follows:

1. The RST algorithm [23] provides an abstraction of causal ordering of point-to-point and multicast messages, and all other (more efficient) algorithms can be cast in terms of this algorithm. We describe an attack on liveness, that we call the artificial boosting attack, that can force all communication to stop when running the RST algorithm.
2. We prove that causal ordering of unicasts and multicasts in an asynchronous system with even one Byzantine node is impossible because liveness cannot be guaranteed. We define weak safety and strong safety and prove that if liveness is to be guaranteed, then weak safety cannot be guaranteed. Further, we prove that strong safety cannot be guaranteed. We also prove these results assuming digital signatures are allowed.
3. We prove that for causal ordering of broadcasts under Byzantine faults, if liveness is to be guaranteed, then weak safety can be guaranteed. Further, we prove that strong safety cannot be guaranteed. We also prove these results assuming digital signatures are allowed.
4. We show that for unicasts, multicasts, and broadcasts, a Full-Information Protocol (FIP) [2,9] can provide liveness + weak safety, but no strong safety. We also show that for rational processes, which act Byzantine only if they cannot be detected/suspected, the unsolvability results remain except for liveness + weak safety for unicasts and multicasts, with digital signatures.

Table 1 summarizes the main results about the solvability of the related problems of unicast, broadcast, and multicast in an asynchronous system with Byzantine faults.

2 Previous Work

Algorithms for causal ordering of point-to-point messages under a fault-free model have been described in [23,24]. These point-to-point causal ordering algorithms extend to implement causal multicasts in a failure-free setting [6,7,13,14,22]. The RST algorithm [23] is a canonical algorithm for causal ordering.

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 in [21] 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 [1]. There has been recent interest in applying the Byzantine fault model to implement causal consistency in distributed shared memory and replicated databases [11,12,25]. These rely on broadcasts, e.g., on Byzantine reliable broadcast [3] in [12] and on PBFT (total order broadcast) [5] in [11]. To the best of our knowledge, no paper has examined the feasibility of or solved causal ordering of unicasts and multicasts in an asynchronous system with Byzantine failures.

3 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 $|P| = n$. 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. For a message send event at time t_1, the corresponding receive event occurs at time $t_2 \in [t_1, \infty)$. 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. Besides authenticated channels and use of signatures, we do not consider the use of other cryptographic primitives.

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* [16] 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, $\forall\ k, j$ such that $k < j$ we have $e_i^k \rightarrow e_i^j$.*
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**: Given events e and e'' in execution trace α, if $\exists\ e' \in \alpha$ such that $e \rightarrow e' \wedge e' \rightarrow e''$ 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 on messages 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 $m \rightarrow m' \wedge m' \rightarrow m''$ 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 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', then $m \xrightarrow{B} m'$.*
3. *If $m \xrightarrow{B} m' \wedge m' \xrightarrow{B} m''$ 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 $(R, \rightarrow)$ and $(S, \xrightarrow{B})$. We now define the correctness criteria that a causal ordering algorithm must satisfy. Ideally, strong safety and liveness should be satisfied because, as we show for application semantics, strong safety is desirable over weak safety.

Definition 6. ***Weak Safety:*** *$\forall m' \in BCP(m)$ such that m' and m are sent to the same correct process(es), no correct process delivers m before m'.*

Definition 7. ***Strong Safety:*** *$\forall m' \in CP(m)$ such that m' and m are sent to the same correct process(es), no correct process delivers m before m'.*

Definition 8. ***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. This definition is different from $m \rightarrow_M m'$ [1], 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 6) – which we show as result (2) in Table 1 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 [1]. Our definition of $\xrightarrow{B}$ and $\rightarrow_M$ [1] 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.

4 Attacks Due to Byzantine Behaviour

All existing algorithms for implementing causal order for point-to-point messages in asynchronous systems use some form of *logical timestamps*. This principle is abstracted by the RST algorithm [23]. Each message m sent to p_i is accompanied by a *logical timestamp* in the form of a matrix clock providing information about send events 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 maintains (a) a vector $Delivered_i$ of size n with $Delivered_i[j]$ storing a count of messages sent by p_j and delivered by p_i, and (b) a matrix M_i of size $n \times n$, where $M_i[j,k]$ stores the count of 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 not delivered until the following *delivery condition* is met: $\forall k$, $M^m[k,i] \leq Delivered_i[k]$.
4. After delivering a message m, p_i merges the logical timestamp associated with m with its own matrix clock, as $\forall j,k$, $M_i[j,k] = \max(M_i[j,k], M^m[j,k])$.

A Byzantine process may fabricate values in the matrix timestamp in order to disrupt the causal ordering of messages in an asynchronous execution. The attacks are described in the following subsections.

4.1 Artificial Boosting Attack

A Byzantine process p_j may increase values of $M_j[x,*]$ beyond the number of messages actually sent by process x to one or more processes. When p_j sends a message with such a Byzantine timestamp to any correct process p_k, it will result in p_k recording Byzantine values in its M_k matrix. These Byzantine values will get propagated across correct processes upon further message passing. This will finally result in correct processes no longer delivering messages from other correct processes because they will be waiting for messages to arrive that have never been sent.

As an example, consider a single malicious process p_j. p_j forges values in its M_j matrix as follows: if p_j knows that p_i (where i may be j) has sent x messages to p_l, it can set $M_j[i,l] = (x+d)$, $d > 0$. When p_k delivers a message from p_j, it sets $M_k[i,l] = (x+d)$. Finally, when p_k sends a message m to p_l, p_l will wait for messages to arrive from p_i (messages that p_i has never sent) before delivering m. This is because $(Delivered_l[i] \leq x) \wedge (M^m[i,l] = x+d) \implies (Delivered_l[i] < M^m[i,l])$. Therefore, p_l will never be able to deliver m. A single Byzantine process p_j has effectively blocked all communication from p_k to p_l. This attack can be replicated for all pairs of processes by p_j. Thus, a single Byzantine process can block all communication (including between each pair of correct processes), thus mounting a liveness attack.

4.2 Safety Violation Attack

A Byzantine process p_j may decrease values of $M^m[*,k]$ to smaller values than the true causal past of message m and send it to a correct process p_k. This may cause m to get delivered out of order at p_k resulting in a causal violation. Furthermore, if p_j decreases the values of $M^m[*,*]$ to smaller values than the true causal past of message m then, once m is delivered to p_k and p_k sends a message m' to correct process p_l, there may be a further causal violation due to a lack of transitive causal data transfer from m to p_k prior to sending m'. These potential causal violations are a result of the possibility of a message getting delivered before messages in its causal past sent to a common destination.

As an example, consider a single malicious process p_j. p_j forges values in the M^m matrix as follows: if p_j knows that p_i has sent x messages to p_k, p_j can set $M^m[i,k] = x-1$ and send m to p_k. If m is received at p_k before the x^{th} message m' from p_i is delivered, m may get delivered before m' resulting in a causal violation of strong safety at p_i. In another attack, if p_j knows that p_i has sent y messages to p_l, it can reduce $M^m[i,l] = y-1$ and send m to p_k. Assume p_k delivers m and sends m' to p_l. If m' arrives at p_l before m'', the y^{th} message from p_i to p_l, arrives at p_l, m' may get delivered before m'' resulting in a causal violation of strong safety at p_l. In this way, a malicious process may cause violations of strong safety (but not weak safety) at multiple correct processes by sending a single message with incorrect causal control information.

5 Results for Unicasts

Causal order of messages can be enforced by either: (a) performing appropriate actions at the receiver's end, or (b) performing 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 order messages (or events) by utilizing timestamps at the receiving process. Traditionally, logical clocks use transitively collected control information attached to each incoming message for this purpose. The RST abstraction [23] (refer Sect. 4) is used. However, in case there is a single Byzantine node p_j in an asynchronous system, it can change the values of M_j at the time of sending m to p_i. This may result in safety or liveness violations when p_i communicates with a third process p_k as explained in Sect. 4. Lemma 1 proves that transitively collected control information by a receiver can lead to liveness attacks in asynchronous systems with Byzantine nodes.

As it is not possible to ensure causal delivery of messages by actions at the receiver's end, therefore, constraints on when the sending process can send messages need to be enforced to maintain causal delivery of messages. Each sender process would need to wait to get an acknowledgement from the receiver before sending the next message. Messages would get delivered in FIFO order at the receiver. While waiting for an acknowledgment, each process would continue to receive and deliver messages. This is important to maintain concurrency and avoid deadlocks. 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. However, Lemma 2 proves that even this approach would fail in the presence of one or more Byzantine nodes. Theorem 1 puts these results together and proves that it is impossible to causally order unicast messages in an asynchronous system with one or more Byzantine nodes.

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 causal order under weak safety 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 [23], described in Sect. 4. 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 in Sect. 4 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 weak safety between m' and m, p_k implements the delivery condition in rule 3 of the RST abstraction (Sect. 4), 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 $M^m[i,k]$. 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 node. □

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

Proof. The only way that a sending process p_i can ensure weak 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 weak 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 sender or at a receiver can prevent a liveness attack (while maintaining weak safety). The theorem follows. □

Theorem 2. *It is possible to guarantee weak safety without a liveness guarantee while causally ordering point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. The theorem that weak safety can be maintained without liveness guarantees was indirectly proved in the proofs of Lemma 1 and Lemma 2. □

Theorem 3. *It is impossible to guarantee strong safety (while guaranteeing liveness) while causally ordering point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. Consider $m' \in CP(m)$ sent to common destination p_r, where m' and m are sent by p_i and p_k, respectively. If p_i sends the next messages after m' only when it is safe to do so (as described in the proof of Lemma 2), an attack on strong safety can be mounted because a Byzantine p_i may not follow the above rule; it may send a subsequent message before getting an ack for message m', and message m along the causal chain beginning with a subsequent message may be delivered to the common destination p_r before m' is delivered. Thus, this option cannot be used to guarantee strong safety while guaranteeing liveness.

The only other way for safe delivery of m is for p_r to rely on transitively propagated control information about $CP(m)$. There exists a chain of messages ordered by $\rightarrow$ from m' to m and sent by processes along this path H. We use the RST abstraction for the transmission of control information about $CP(m)$. Let $M_i[i,r]$ be x when m' is sent. A Byzantine process along H, that sends m'', can set $M^{m''}[i,r]$ to a lower value x' than x and thereby propagate x' instead of x along H. $M_k[i,r]$ that is piggybacked on m as $M^m[i,r]$ will be less than x. Hence, a strong safety attack can be mounted at p_r.

Thus, no action at the sender or at the receiver can prevent a strong safety attack. □

Theorem 4. *It is impossible*[1] *to guarantee strong safety (even without guaranteeing liveness) while causally ordering point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. The proof of Theorem 3 showed strong safety can never be satisfied. This result was independent of liveness attacks. The same result holds even if liveness attacks can be mounted, and Theorem 1 showed liveness attacks could be mounted on weak safety requirements, which implies they can also be mounted on strong safety requirements. □

5.1 Results for Unicasts Allowing Digital Signatures

Theorem 5. *It is impossible to guarantee liveness while satisfying weak safety using digital signatures while causally ordering point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. Lemma 2 (sending a message only when a receiver is guaranteed not to have a weak safety attack) can be seen to hold even with the use of digital

[1] Here in Theorems 4, 7, 9, and 11, we rule out the trivial solution of not delivering any messages to guarantee strong safety.

signatures. So the only remaining option to guarantee liveness (while satisfying weak safety) is to try to use transitively received control information.

In the RST abstraction, a sending process p_i will sign its row of M^m whereas row s ($\forall s \in P$) is signed by p_s. This allows the receiver process p_j to do the max of its row $M_j[s, *]$ and $M^m[s, *]$ ($\forall s \in P$), both of which were signed by P_s, and update its M_j matrix.

The same liveness attack (while satisfying weak safety), as shown in the proof and scenario in Lemma 1, can be mounted when $y = x$ (i.e., using $M^{m''}[y = x, k]$ in that proof), even with the use of digital signatures. This is because a Byzantine process p_x can always sign its inflated row x entries of M_x. Although this allows the receiver to be reassured that entries in the xth row of M^m were not forged by anyone, it does not help in avoiding the indefinite wait of the liveness attack mounted by p_x.

Thus, liveness cannot be guaranteed while satisfying weak safety despite using digital signatures. □

Theorem 6. *It is impossible to guarantee strong safety while satisfying liveness using digital signatures while causally ordering point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. Consider $m' \in CP(m)$ sent to common destination p_r, where m' and m are sent by p_i and p_k, respectively. If p_i relies on sending the next messages after m' only when it is safe to do so (as described in the proof of Lemma 2), a Byzantine p_i can cause strong safety to be violated by not following the above rule, as shown in the proof of Theorem 3. Thus, this option cannot be used.

The only other way for safe delivery of m while satisfying liveness is for p_r to rely on transitively propagated control information about $CP(m)$; for this we assume the RST abstraction. Consider the following sequence: correct process p_i sends a message m' to p_r, then sends a (signed) message m'' (containing the rows of M_i as $M^{m''}$, where row s is signed by p_s) to p_j. p_j, a Byzantine process, delivers message m'', acts on the message, and then sends a message m_1 to p_k. However, on receiving the message m'' from p_i, p_j does not update $M_j[i, *]$ with the most recently signed row $M^{m''}[i, *]$ received but uses an older row, also signed (earlier) by p_i, pretending as though p_i's message m'' had never been delivered and processed. p_k uses this (older) row of $M_j[i, *]$ received on m_1 as $M^{m_1}[i, *]$ and sets $M_k[i, *]$ to this older value which does not get replaced by p_i's signed row that was piggybacked on m''. p_k now forwards this older row, signed by p_i, as part of M^m it piggybacks on m it sends to p_r. p_r can deliver m even if m' from p_i has not been received. Here, p_i, p_k, and p_r are all correct processes and m' (sent by p_i to p_r) $\rightarrow m$ (sent by p_k to p_r), yet p_r may deliver m before m', thus violating strong safety. The use of digital signatures does not help in preventing such a violation. Hence, a strong safety attack can be mounted at p_r. □

Theorem 7. *It is impossible (see footnote 1) to guarantee strong safety (even without guaranteeing liveness) using digital signatures while causally ordering* *point-to-point messages in an asynchronous message passing system with one or more Byzantine processes.*

Proof. The proof of Theorem 6 showed strong safety can never be satisfied even using digital signatures. This result was independent of liveness attacks. This same result holds even if liveness attacks can be mounted, and Theorem 5 showed liveness attacks could be mounted on weak safety requirements, which implies they can also be mounted on strong safety requirements. □

6 Results for Broadcasts

Byzantine Reliable Broadcast (BRB) has traditionally been defined based on Bracha's Byzantine Reliable Broadcast (BRB) [3,4]. For this algorithm to work, it is assumed that less then $n/3$ processes are Byzantine. When a process does a broadcast, it invokes `br_broadcast()` and when it is to deliver such a message, it executes `br_deliver()`. In the discussion below, it is implicitly assumed that a message is uniquely identified by a (sender ID, sequence number) tuple. BRB 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_broadcast(m)`.
- Integrity: For any message m, a correct process executes `br_deliver` at most once.
- Self-delivery: If a correct process executes `br_broadcast(m)`, then it eventually executes `br_deliver(m)`.
- Reliability (or Termination): If a correct process executes `br_deliver(m)`, then every other correct process also (eventually) executes `br_deliver(m)`.

As causal broadcast is an application layer property, it runs on top of the BRB layer. Byzantine Causal Broadcast (BCB) is invoked as `BC_broadcast`(m) which in turn invokes `br_broadcast`(m') to the BRB layer. Here, m' is m plus some control information appended by the BCB layer. A `br_deliver`(m') from the BRB layer is given to the BCB layer which delivers the message m to the application via `BC_deliver(m)` after the processing in the BCB layer. The control information is abstracted by the *causal barrier* [1,10] which tracks the immediate or direct dependencies and is bounded by $O(n)$. In addition to the BCB-layer counterparts of the properties satisfied by BRB, BCB must satisfy safety and liveness. Liveness and weak safety can be satisfied as given by the protocol in [1]. Next, we analyze the possibility of strong safety and liveness, and all four combinations (refer Table 1) if digital signatures can be used.

Theorem 8. *It is impossible to guarantee strong safety and liveness while causally ordering broadcast messages in an asynchronous message passing system with one or more Byzantine process.*

Proof. Strong safety (along with liveness) cannot be ensured by requiring the sender to wait for acknowledgements *ack*1 to its broadcast that the message has been `BC_delivered`, and for receivers to wait for an ack $ack2$ from the sender that the message has been `BC_delivered` to all recipients, before broadcasting further messages. This is because a Byzantine process p_x may read a message m before it is `br_delivered`, and broadcast m_1 without waiting for *ack2*. A third correct process p_y may then `br_deliver` and `BC_deliver` m_1 before m. So no action at the sender can enforce strong safety.

The only option left is for the receiver to use transitively propagated information. So we assume the causal barrier abstraction for tracking (transitive) dependencies for broadcasts. Consider a Byzantine process p_j that reads message m broadcast from a correct process p_i while it is being processed by the BRB layer before `br_delivery` at p_j, takes action based on it and broadcasts m_1 (thus, $m \rightarrow m_1$ semantically) but excludes m from the causal barrier of m_1. A correct process p_k may `BC_deliver` m_1 before m. It then broadcasts m' which may be `BC_delivered` by a correct process p_l before m, thus violating strong safety.

Effectively, by p_j dropping m from the causal barrier of m_1, the relation $m \rightarrow m_1$ (and hence $m \rightarrow m'$) was changed to $m \not\rightarrow m_1$ (and $m \not\rightarrow m'$). As this action of logically swapping the order of the semantic "`BC_deliver(m)`" and `BC_broadcast(`m_1`)` was solely under the local control of a Byzantine process, no protocol can exist to counter this action. □

Examples of strong safety violations in real-world applications:

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 get an advantage over correct processes in winning the game.

Theorem 9. *It is impossible (see Footnote 1) to guarantee strong safety even without liveness guarantees while causally ordering broadcast messages in an asynchronous message passing system with one of more Byzantine process.*

Proof. The proof of Theorem 8 showed strong safety can never be satisfied. This result was independent of liveness attacks. So even if liveness attacks cannot be mounted on broadcasts (refer algorithm in [1]), strong safety cannot be guaranteed. □

Theorem 10. *It is impossible to guarantee strong safety (while satisfying liveness) using digital signatures while causally ordering broadcast messages in an asynchronous message passing system with even one Byzantine processes.*

Proof. The same proof of Theorem 8 applies because the action by a Byzantine process that causes the strong safety attack is local to that process and signing messages and/or causal barriers will not help because it only authenticates the messages and/or causal barriers. □

Theorem 11. *It is impossible (see Footnote 1) to guarantee strong safety (even without satisfying liveness) using digital signatures while causally ordering broadcast messages in an asynchronous message passing system with even one Byzantine processes.*

Proof. The proof of Theorem 10 showed strong safety can never be satisfied even using digital signatures. This result was independent of liveness attacks. So even if liveness attacks cannot be mounted on broadcasts, strong safety cannot be guaranteed. □

7 Byzantine Causal Multicast (BCM)

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. There can exist overlapping multicast groups. In the general case, there are $2^{|P|} - 1$ groups. Although there are several algorithms for causal ordering of multicasts under dynamic groups, such as [6,7,13,14,22], none consider the Byzantine failure model.

Byzantine Reliable Multicast (BRM) [18,19] has traditionally been defined based on Bracha's Byzantine Reliable Broadcast (BRB) [3,4]. For these algorithms to work, it is assumed 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, sequence number) 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. In addition to the BCM-layer counterparts of the properties satisfied by BRM, BCM must satisfy safety and liveness (Sect. 3).

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 [10,22] or in the optimal encoding of the theoretically minimal control information as in [6,7,13,14]. 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 Sect. 4 can be mounted on the causal multicast algorithms. Furthermore, all the results and proofs given in Sect. 5 for unicasts, and summarized in Table 1, apply to multicasts with straightforward adaptations. The intuitive reason is given below.

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, 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 [1], the underlying Bracha's 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.

8 Discussion

On Broadcast vs. Unicast. Byzantine causal broadcast is solvable [1]. Then why is Byzantine fault-tolerant causal order for point-to-point communication impossible? The problem is that a single Byzantine adversary can launch a liveness attack by artificial boosting. In Byzantine causal broadcast, all messages are sent to every process in the system and the underlying Byzantine reliable broadcast layer [3] ensures that every correct process receives the exact same set of messages. Upon receiving m, the receiving process simply waits for its logical clock to catch up with m's timestamp (each broadcast delivered will increment one entry in the logical clock) and deliver m once it is safe to do so. After delivering message m, the receiving processes' logical clock is greater than or equal to m's timestamp. This means that a receiving process does not need to merge message m's timestamp into its own logical clock upon delivering m. Hence no amount of artificial boosting can result in a liveness attack in Byzantine causal broadcast. In case of causal ordering for unicasts and multicasts, every process receives a different set of messages. When a process p_i delivers message m, it means that

p_i has delivered all messages addressed to it in the causal past of m. However, it requires the timestamp attached to m to ascertain the messages in the causal past of m that are not addressed to p_i. Therefore, the receiving process needs to merge the timestamp of the delivered message into its own logical clock so that subsequent messages sent by it can be timestamped with their causal past.

Full-Information Protocols (FIP). The system model rules out full-information protocols (FIP) [9] where the entire transitively collected message history is used as control information – because (i) a message from p_i to p_j or to G needs to be kept private to those two processes or to G, and (ii) a FIP obviates the need for causal ordering. Encrypting messages from p_i to p_j or to G, on which is superimposed the FIP, can provide (liveness + weak safety), but not strong safety, for unicasts and multicasts – however, the cost of a FIP is prohibitively high and as noted in (ii), a FIP obviates the need for causal ordering which rules out this approach. Note, liveness (+ weak safety) can be provided because a Byzantine process must send the messages contained in any inflated message advertisement, in the message history. Also, strong safety cannot be provided because attacks analogous to those in Theorems 3, 6 and 4, 7 proofs can be mounted, wherein a Byzantine process selectively suppresses the message history. A FIP can neither provide strong safety for broadcasts – using reasoning similar to Theorems 8, 10 and 9, 11 proofs, it is seen that a Byzantine process has local control over selectively suppressing message history.

Strong Safety vs. Weak Safety. It is impossible to guarantee strong safety for broadcasts (and unicasts). The Byzantine causal broadcast algorithm in [1] provides only weak safety but this is not always useful in practice because it requires $\xrightarrow{B}$ to hold but correct processes cannot identify whether $\xrightarrow{B}$ or just $\rightarrow$ holds when processing an arrived message. In the absence of strong safety, the examples given after Theorem 8 demonstrate that a Byzantine causal broadcast algorithm is not useful to users of certain applications. (Weak safety suffices to prevent double-spending in the money-transfer algorithm [1] using `BC_broadcast`, because actually Byzantine causal order is not required for this application; source order is sufficient [8] and weak safety does not violate source order.)

Rational vs. Irrational Byzantine Agents. A *rational* Byzantine agent will mount an attack only if it is not detected/suspected. It can be seen that all impossibility results for strong safety (cases in Table 1) hold even for rational agents because deleting entries, whether signed or in a FIP or neither, from the causal past is entirely local to the Byzantine agent and undetectable by others. Only Theorem 5 for liveness + weak safety under signed messages will not hold for rational agents because in the proof of Lemma 1 on which it depends, the attacker p_x that inflates $M^{m''}[y,k]$ can do so only for $y = x$ – as it cannot sign for p_y and can sign only for $p_y = p_x$. The attack victim p_k can suspect p_x when p_k continues waiting for the delivery condition to be satisfied (or does not receive the *ack* soon enough). Note, in the proof of Lemma 1 (for Theorem 1), if p_k does not get the messages (or the *ack* to *probe*) from p_y in reasonable time, p_k can suspect p_y (as p_y may be Byzantine) and stop waiting for it, although Byzantine p_x mounted the attack and goes unsuspected. Further, in the proof of Lemma 2 on which Theorems 1 and 5 depend, if the sender p_i does not get an *ack* from receiver p_j in a reasonable amount of time, p_i can suspect p_j as being Byzantine; however, Lemma 2 is essentially using some elements of synchronous communication and so it cannot be said that a possibility result holds for rational agents in a truly asynchronous system.

In view of the impossibility result of Theorem 1, algorithms for liveness + weak safety in a stronger asynchrony model are given in [20].

References

1. Auvolat, A., Frey, D., Raynal, M., Taïani, F.: Byzantine-tolerant causal broadcast. Theoret. Comput. Sci. **885**, 55–68 (2021)
2. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. (TOCS) **5**(1), 47–76 (1987)
3. Bracha, G.: Asynchronous byzantine agreement protocols. Inf. Comput. **75**(2), 130–143 (1987). https://doi.org/10.1016/0890-5401(87)90054-X
4. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. J. ACM **32**(4), 824–840 (1985). https://doi.org/10.1145/4221.214134
5. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Seltzer, M.I., Leach, P.J. (eds.) Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 173–186 (1999)
6. Chandra, P., Gambhire, P., Kshemkalyani, A.D.: Performance of the optimal causal multicast algorithm: a statistical analysis. IEEE Trans. Parallel Distrib. Syst. **15**(1), 40–52 (2004). https://doi.org/10.1109/TPDS.2004.1264784
7. Chandra, P., Kshemkalyani, A.D.: Causal multicast in mobile networks. In: 12th International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 213–220 (2004). https://doi.org/10.1109/MASCOT.2004.1348235
8. Collins, D., et al.: Online payments by merely broadcasting messages. In: 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, pp. 26–38 (2020)
9. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. MIT Press, Cambridge (1995). https://doi.org/10.7551/mitpress/5803.001.0001
10. Hadzilacos, V., Toueg, S.: A modular approach to fault-tolerant broadcasts and related problems. Technical report, 94-1425, Cornell University, p. 83 pages (1994)
11. Huang, K., Wei, H., Huang, Y., Li, H., Pan, A.: Byz-GentleRain: an efficient byzantine-tolerant causal consistency protocol. arXiv preprint arXiv:2109.14189 (2021)
12. Kleppmann, M., Howard, H.: Byzantine eventual consistency and the fundamental limits of peer-to-peer databases. arXiv preprint arXiv:2012.00472 (2020)
13. Kshemkalyani, A.D., Singhal, M.: An optimal algorithm for generalized causal message ordering (abstract). In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, p. 87. ACM (1996). https://doi.org/10.1145/248052.248064
14. Kshemkalyani, A.D., Singhal, M.: Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. Distrib. Comput. **11**(2), 91–111 (1998). https://doi.org/10.1007/s004460050044

15. Kshemkalyani, A.D., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, Cambridge (2011). https://doi.org/10.1017/CBO9780511805318
16. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
17. Liskov, B., Ladin, R.: Highly available distributed services and fault-tolerant distributed garbage collection. In: Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing, pp. 29–39 (1986)
18. Malkhi, D., Merritt, M., Rodeh, O.: Secure reliable multicast protocols in a WAN. In: Proceedings of the 17th International Conference on Distributed Computing Systems, pp. 87–94 (1997). https://doi.org/10.1109/ICDCS.1997.597857
19. Malkhi, D., Reiter, M.K.: A high-throughput secure reliable multicast protocol. J. Comput. Secur. **5**(2), 113–128 (1997). https://doi.org/10.3233/JCS-1997-5203
20. Misra, A., Kshemkalyani, A.D.: Byzantine fault tolerant causal ordering. CoRR abs/2112.11337 (2021). https://arxiv.org/abs/2112.11337
21. Mostefaoui, A., Perrin, M., Raynal, M., Cao, J.: Crash-tolerant causal broadcast in o (n) messages. Inf. Process. Lett. **151**, 105837 (2019)
22. Prakash, R., Raynal, M., Singhal, M.: An adaptive causal ordering algorithm suited to mobile computing environments. J. Parallel Distrib. Comput. **41**(2), 190–204 (1997)
23. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Inf. Process. Lett. **39**(6), 343–350 (1991)
24. Schiper, A., Eggli, J., Sandoz, A.: A new algorithm to implement causal ordering. In: Bermond, J.-C., Raynal, M. (eds.) WDAG 1989. LNCS, vol. 392, pp. 219–232. Springer, Heidelberg (1989). https://doi.org/10.1007/3-540-51687-5_45
25. Tseng, L., Wang, Z., Zhao, Y., Pan, H.: Distributed causal memory in the presence of byzantine servers. In: 2019 IEEE 18th International Symposium on Network Computing and Applications (NCA), pp. 1–8 (2019)