Byzantine Fault-Tolerant Causal Ordering

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

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

ABSTRACT

Byzantine fault-tolerant causal ordering of messages in asynchronous systems is useful to many applications. Although the problem has been studied for broadcast communication, it has not been examined for unicasts or multicasts in asynchronous systems. In this paper, we use execution histories to prove that it is impossible to solve causal ordering for both unicasts and multicasts in an asynchronous system with one or more Byzantine processes. In view of these impossibility results, we propose the Channel Sync Algorithms to provide causal order of unicasts and multicasts under the Byzantine failure model in synchronous systems, which have a known upper bound on message latency. The Channel Sync Algorithms operate under the synchronous system model, but are inherently asynchronous and offer a high degree of concurrency as lock-step communication is not assumed.

CCS CONCEPTS

• Computing methodologies → Distributed algorithms; • Networks → Network algorithms.

KEYWORDS

byzantine fault-tolerance, causal order, unicast communication, causal multicast, asynchronous message-passing

ACM Reference Format:


1 INTRODUCTION

Causality provides important application-level semantics to distributed programs. Causality is defined by the happens before [14] relation on the set of events, and by extension, on the set of messages. If message m1 causally precedes m2 and both are sent to pi, then m2 cannot be delivered before m1 at pi to enforce causal order [3]. Causal ordering ensures that causally related updates to data occur in a valid manner respecting that causal relation. Applications of causal ordering include distributed data stores, fair resource allocation, and collaborative applications such as multiplayer online gaming, social networks, event notification systems, group editing of documents, and distributed virtual environments.

It is important to solve causal ordering under the Byzantine failure model because it mirrors the real world. Byzantine-tolerant causal ordering/causal consistency, which considered/relied on only broadcast communication, was studied in [1, 11, 12, 24]. Byzantine-tolerant causal ordering for unicasts or multicasts has not been considered besides the recent analysis in [19].

This paper makes the following contributions.

(1) We prove using execution histories that causal ordering of unicasts in an asynchronous system with even one Byzantine process is impossible.

(2) In view of the above impossibility result for asynchronous systems, we show that a solution can be designed in a synchronous system. The strengthening is in the form of a known upper bound δ on message latency, and also a known upper bound ψ on the relative speeds of processors.

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

(3) Based on the impossibility result of Byzantine-tolerant causal ordering of unicasts in an asynchronous system, we prove that it is impossible to solve for Byzantine-tolerant causal multicasts in an asynchronous system. We then give an extension of the Channel Sync algorithm for multicasts in a synchronous system and prove its correctness.

Roadmap. Section 2 reviews related work. Section 3 gives the system model. Section 4 gives the impossibility result of being unable to solve Byzantine causal unicast in an asynchronous system. For a synchronous system where there is a known upper bound on the message latency, Section 5 presents the Channel Sync algorithm for Byzantine causal unicast. Section 6 analyzes Byzantine causal multicast and proves that it is impossible to solve it in an asynchronous system. Section 7 gives the extension of the Channel Sync algorithm to Byzantine causal multicast in the synchronous system model. Section 8 gives a discussion and Section 9 concludes.

2 RELATED WORK

Algorithms for causal ordering of unicast messages in an asynchronous setting under a fault-free model have been proposed, e.g., in [23]. These extend to implement causal multicasts in a failure-free setting [13, 22]. The above algorithms append control information to application messages. The algorithm in [10] for the same setting...
The feasibility of solving systems as in [6], which makes them inappropriate for truly asynchronous as hidden/implicit assumptions such as bounds on message latency, such a system is given in [18]. Recently the Byzantine fault model was used to implement causal consistency in distributed shared memory and replicated databases [11, 12, 24]; these approaches relied on broadcast communication. To the best of our knowledge, no paper has attempted to solve causal ordering of unicasts and multicasts in an asynchronous system with Byzantine failures, besides our analysis of solvability [19].

3 SYSTEM MODEL

This paper deals with a distributed system having Byzantine processes which are processes that can misbehave [15, 21]. A correct process behaves exactly as specified by the algorithm whereas a Byzantine process may exhibit arbitrary behaviour including crashing at any point during the execution. A Byzantine process cannot impersonate another process or spawn new processes.

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

The system is first assumed to be asynchronous, i.e., there is no known fixed upper bound \( \delta \) on the message latency, nor any known fixed upper bound \( \psi \) on the relative speeds of processors [7]. In contrast, a synchronous system has been defined as one in which both \( \delta \) and \( \psi \) are known. [7] We prove that it is impossible to solve Byzantine-tolerant causal message ordering for unicasts and multicasts in an asynchronous system. In light of this impossibility result, we give an algorithm for a system where \( \delta \) is known and used by the algorithm; the algorithm relies on timeouts which can use knowledge of \( \psi \) for accuracy. Thus, it can be said that the algorithm assumes a synchronous system. Another algorithm for such a system is given in [18].

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 [6], which makes them inappropriate for truly asynchronous systems.

Let \( e_i^x \), where \( x \geq 1 \), denote the \( x \)-th event executed by process \( p_i \). An event may be an internal event, a message send event, or a message receive event. Let the state of \( p_i \) after \( e_i^x \) be denoted \( s_i^x \), where \( x \geq 1 \), and let \( s_i^0 \) be the initial state. The execution at \( p_i \) is the sequence of alternating events and resulting states, as \( (s_i^0, e_i^1, \ldots, s_i^n) \). The \( \text{happens before} \) [14] relation, denoted \( \rightarrow \), is an irreflexive, asymmetric, and transitive partial order defined over events in a distributed execution that is used to define causality.

**Definition 1.** The \( \text{happens before} \) relation on events consists of the following rules:

1. **Program Order:** For the sequence of events \( (e_1^1, e_2^1, \ldots) \) 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 \( e \rightarrow e' \land e' \rightarrow e'' \) then \( e \rightarrow e'' \).

Next, we define the \( \text{happens before} \) relation on the set of all application-level messages \( R \).

**Definition 2.** The \( \text{happens before} \) 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 \( m \rightarrow m' \) and \( m' \rightarrow m'' \), then \( m \rightarrow m'' \).

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

We require an extension of the \( \text{happens before} \) relation on messages to accommodate the possibility of Byzantine behaviour. We present a partial order on messages called Byzantine \( \text{happens before} \), denoted as \( \text{B-} \rightarrow \), 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 \( \text{happens before} \) relation 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 \( \text{B-} \rightarrow \).
2. If \( p_i \) is a correct process and \( p_i \) sent or delivered message \( m \) before sending message \( m' \), then \( m \rightarrow m' \).
3. If \( m \rightarrow m' \) and \( m' \rightarrow m'' \), then \( m \rightarrow m'' \).

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

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

The correctness of Byzantine causal order unicast/multicast/broadcast is specified on \( (R, \rightarrow) \) and \( (S, \text{B-} \rightarrow) \) as follows.

**Definition 6.** A causal ordering algorithm for unicast/multicast/broadcast messages must ensure the following:
An algorithm to solve causal ordering collects the execution history \( m \) for all correct processes that sent messages along that chain. As specified by \( CO(p) \), message passing originated at \( E \) from process \( p \) can cause \( E \) to match \( F \) in bounded time. But when \( m = B \), then there exists a causal chain from \( m \) to \( m' \) along correct processes that sent messages along that chain.

## 4 IMPOSSIBILITY RESULT

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

Let \( m_1 \rightarrow m_2 \mid E \) and \( m_1 \rightarrow m_2 \mid F \) be the evaluation \( (1 \lor 0) \) of \( m_1 \rightarrow m_2 \) using \( E \) and \( F \), respectively.

We rephrase the causal ordering problem (Definition 6) as \( CO(E, F, m_2) \) that returns 1 iff \( \forall m_1, m_1 \rightarrow m_2 \mid E = m_1 \rightarrow m_2 \mid F \). When 1 is returned, the algorithm output matches God’s truth and solves \( CO \) correctly. Thus, returning 1 indicates that the problem has been solved correctly by the algorithm using \( F \). 0 is returned if either

1. \( \exists m_1 \) such that \( m_1 \rightarrow m_2 \mid E = 1 \) and \( m_1 \rightarrow m_2 \mid F = 0 \), denoting a strong safety violation,
2. \( \exists m_1 \) such that \( m_1 \rightarrow m_2 \mid E = 0 \) and \( m_1 \rightarrow m_2 \mid F = 1 \), denoting a liveness violation.

To determine whether \( CO \) is solved correctly, we have to evaluate \( \forall m_1, m_1 \rightarrow m_2 \mid E = m_1 \rightarrow m_2 \mid F \) even if \( m_1 \in (M(E) \cup M(F)) \setminus M(E) \) because such an \( m_1 \) is recorded by the algorithm as part of \( F \).

The key observation we make is that in \( CO \), a single Byzantine process \( p_b \) can cause \( F \) (as recorded by the algorithm) to be different from \( E \). This is not just a mismatch between \( E_b \) and \( F_b \) but also between other \( E_a \) and \( F_a \) by contaminating \( F_a \) via direct and transitive message passing originated at \( p_b \).

Theorem 1. It is impossible to solve causal ordering (Definition 6) as specified by \( CO(E, F, m_2) \) of unicast messages in an asynchronous message passing system with one or more Byzantine processes.

Proof. We prove the impossibility of solving the \( CO \) problem by showing:

1. a reduction (denoted \( \leq \)) from \( Black\_Box \) to \( CO \), where \( Black\_Box \) is defined below,
2. a reduction from the \( Consensus \) problem (which by the FLP result [9] is unsolvable in the presence of a single Byzantine process) to the \( Black\_Box \) problem.

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

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

\[
\begin{align*}
  w &= \begin{cases} 
  0 & \text{if each correct } p_i \text{ has } V[i] = 0 \\
  1 & \text{if each correct } p_i \text{ has } V[i] = 1 
\end{cases} \\
  \land m_1 \rightarrow m_2 \mid E = m_1 \rightarrow m_2 \mid F 
\end{align*}
\]

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

- A Byzantine process may attempt to insert a fake entry in \( E_b \) and contaminate the reporting of histories in \( F \), leading to a liveness violation because \( M(F) \setminus M(E) \neq \emptyset \). Therefore, either contamination of \( F \) has to be prevented or malicious entries have to be filtered out from \( F \) in bounded time. But due to unicasting, a message from a potentially Byzantine process \( p_b \) to \( p_g \) in \( E \), cannot be verified in bounded time by other processes while collecting the reported execution history as the message itself cannot be broadcast or communicated to any process other than \( p_g \) to keep it private. Therefore, identification of Byzantine processes, their actual execution histories, and causal chains from them is required.

- Let there be a message \( m \) sent by \( p_b \) in \( E \). During the collection of \( E_b \) for reporting \( F_b \), Byzantine processes may delete information about \( m \) from \( E_b \), leading to a strength safety violation because \( M(E) \setminus M(F) \neq \emptyset \). Therefore, either deletion of information from \( E \) in \( F \) has to be prevented, or such deletions from \( E \) when presented with \( F \) have to be recognized in bounded time. This requires identification of the Byzantine processes, their actual execution histories, and causal chains from them.

If there were an algorithm to make \( F \) match \( E \), it requires identifying whether each of the processes that input their execution histories is correct or Byzantine (to trace and deal with the impact of contamination via message passing by the Byzantine processes from those Byzantine sources on the execution histories of other processes). Thus, \( Black\_Box \leq CO \).

In the \( Consensus \) problem, each process has an initial value and all correct processes must agree on a single value. The solution needs to satisfy the following three conditions [15, 21].

- Agreement: All non-faulty processes must agree on the same single value.
- Validity: If all non-faulty processes have the same initial value, then the agreed-on value by all the non-faulty processes must be that same value.
- Termination: Each non-faulty process must eventually decide on a value.
When $\text{Consensus}(\mathcal{V})$ is to be solved, it invokes the black box for $\text{Black\_Box}(\mathcal{V}, E, F, m_2)$. Each correct process outputs as its consensus value the value that it receives from $p_1$ and terminates. Agreement, Validity, and Termination clauses of $\text{Consensus}$ can be seen to be satisfied. So $\text{Consensus} \leq \text{Black\_Box}$.

If $CO$ is (correctly) solvable, it returns $1$ for $\forall m_1, m_1 \rightarrow m_2|E = m_1 \rightarrow m_2|F$, and implicitly for all $m_2$. We now have

$$\text{Consensus} \leq \text{Black\_Box} \leq CO$$

This implies that if the $CO$ problem is solvable, then $\text{Consensus}$ is also solvable. That contradicts the FLP impossibility result for a Byzantine process system, hence $CO$ is not solvable.

**Digression 1.** It is worth observing that under the crash-failure model, even though $\text{Consensus} \leq \text{Black\_Box}$, we have that $\text{Black\_Box} \not\leq CO$. This latter relation $\not\leq$ is because solving $CO$ does not require identifying the crashed processes; their (correct) execution histories can be faithfully transmitted to other processes (transitively) via the execution messages sent in the execution history itself as it grows and be present at the other (correct) processes’ execution histories and in in-transit messages. As $m_1$ and $m_2$ must have been sent, the execution histories of their senders can transitively propagate to other non-crashed processes. In other words, the execution history of any prefix can be represented by that execution. Therefore, $M(E) = M(F)$. Hence, it suffices to consider the execution histories $E_i$ of non-crashed processes (that include $p_f$) to determine $m_1 \rightarrow m_2$ without having to identify the crashed processes.

**Digression 2.** We outline the logic that $CO$ (Definition 6) cannot be solved for Byzantine Causal Broadcast. For Byzantine Causal Broadcast, $F$ cannot be made to match $E$.

- By running the causal ordering layer above the Byzantine Reliable Broadcast (BRB) [4, 5] layer, liveliness violation can be prevented by ensuring $M(F) \setminus M(E) = \emptyset$. If a Byzantine process $p_b$ attempts to insert a fake entry about broadcast of $m$ by $p_x$ in $E_x (x = b$ or $x \neq b)$ at a correct process $p_y$, $p_y$ can verify whether or not this insertion is valid as based on the Reliability/ Termination properties of BRB, $m$ must be delivered by the BRB layer at all correct processes including $p_y$. Therefore, no message from a correct process to another correct process will wait indefinitely for causal delivery.

- However, a Byzantine process $p_b$ can delete from $F_x$ information about a broadcast of $m_1$ by $p_{b'}$ that it has received, where $p_{b'}$ may be a correct process, despite running the causal ordering layer above the BRB layer. A message $m_2$ then broadcast, where $m_1 \rightarrow m_2$ and the message chain passes through a message broadcast by $p_{x'}$, can be delivered by a correct process $p_r$ before $m_1$ is, if $p_r$ is not to wait indefinitely. Thus, $M(E) \setminus M(F) \neq \emptyset$ and strong safety violations may occur.

Thus, to solve $CO$, it is necessary to identify Byzantine processes, their actual execution histories, and causal chains from them. Then $\text{Black\_Box} \leq CO$ and hence $\text{Consensus} \leq CO$.

We now show a similar result to Theorem 1 with strong safety (Definition 6) defined in terms of the $\rightarrow$ relation replaced by weak safety (Definition 7) defined in terms of the $B$ relation in the correctness criteria for causal ordering.

**Theorem 2.** It is impossible to solve causal ordering (Definition 7) of unicast messages in an asynchronous message passing system with one or more Byzantine processes.

**Proof.** We rephrase the causal ordering problem (Definition 7) as $CO(E, F, m_2)$ that returns $1$ if $\forall m_1, m_1 \rightarrow m_2|E = m_1 \rightarrow m_2|F$. The problem is solved correctly iff $1$ is returned.

Observe, $m_1 \rightarrow m_2$ is equivalent to: $m_1 \rightarrow m_2$ and there is a causal path from send event of $m_1$ to send event of $m_2$ going through correct processes in the execution. Define $m_1 \rightarrow m_2|E$ as $m_1 \rightarrow m_2|F \land$ there is a causal path from send event of $m_1$ to send event of $m_2$ going through correct processes in the execution. (Likewise for $m_1 \rightarrow m_2|E$.) The algorithm to solve $CO$ does not have to determine whether the path through correct processes exists.

Note that $m_2$ is necessarily sent by a correct process when $m_1 \rightarrow m_2$ holds. The proof of Theorem 1 carries identically, subject to the following changes. In the specification of $\text{Black\_Box}$, the definition $\forall m_1, m_1 \rightarrow m_2|E = m_1 \rightarrow m_2|F$ instead of $\forall m_1, m_1 \rightarrow m_2|E$ is used.

That $\text{Consensus} \leq \text{Black\_Box}$ still holds is self-evident. $\text{Black\_Box} \leq CO$ still holds because solving $CO$ correctly still requires using the execution histories of Byzantine processes as recorded by the algorithm in $F$, similar to the proof for Theorem 1. In order for any algorithm to correctly solve $CO$, it must ensure that $F$ matches $E$. For this, the following must hold.

- Due to uncasting, a message $m$ from a potentially Byzantine $p_x$ to $p_y$ in $F_x$, cannot be verified in bounded time by other processes while collecting the reported execution history as the message itself cannot be broadcast or communicated to any process other than $p_y$ to keep it private. Thus, a fake entry may be inserted in $F_x$ by a Byzantine process, even if there exists some path through correct processes from sender of $m_1$ to sender of $m_2$, leading to a liveliness violation because $M(F) \setminus M(E) \neq \emptyset$. (Note, liveliness of $m_2$ is not with respect to a $m_1$ sent by a correct process but all $m_1$.) Therefore, either contamination of $F$ has to be prevented or malicious entries have to be filtered out from $F$ in bounded time. This requires identifying Byzantine processes, their actual execution histories, and causal chains from them.

- Let there be a message $m_1$ sent by correct process $p_x$ in $E_x$. During the collection of $E_x$ for reporting $F_x$, if there are no Byzantine processes along some path from $p_x$ to sender $p_y$ of $m_2$, (hence $p_k$ must be a correct process), it is possible to ensure that no Byzantine processes can cause deletion of information about $m_1$ from $F_x$, thus $(M(E)_x \setminus M(F) = \emptyset$, where $(M(E))_x$ is the messages of $M(E)$ sent by correct processes. Thus, weak safety violation of $m_2$ (with respect to $m_1$ sent by correct processes) can be prevented.

If there were an algorithm to make $F$ match $E$, it still requires identifying whether each of the processes that input their execution histories is correct or Byzantine (to trace and deal with the impact of contamination via message passing by the Byzantine processes from those Byzantine sources on the execution histories of other processes). Hence $\text{Black\_Box} \leq CO$. The theorem follows. \qed
Digression 3. We outline the logic that CO (Definition 7) can be solved for Byzantine Causal Broadcast. For Byzantine Causal Broadcast, $F$ can be made to match $E$.
- $M(F) \setminus M(E) = 0$, hence liveness violations cannot occur. Same reasoning as in first bullet in Digression 2.
- If there is a path through correct processes along $m_1 \xrightarrow{B} m_2$, which can faithfully propagate information about $m_1$, when $m_2$ arrives at $p_r$ it will wait for $m_1$ which must arrive at $p_r$ because of the Reliability/ Termination properties of the BRB layer over which the causal ordering layer is run. Thus $M(E)_c \setminus M(F)_c = 0$ and weak safety violations cannot occur. (This holds even if broadcaster of $m_1$ is Byzantine.)

Thus to solve CO for broadcasts under Definition 7, it is not necessary to identify whether each process is Byzantine, hence $\text{Black\_Box} \notin CO$ and hence $\text{Consensus} \notin CO$.

5 CHANNEL SYNC ALGORITHM

As a result of Theorems 1, 2 we know that it is impossible to maintain both (strong and weak) safety and liveness while trying to causally order unicast messages in an asynchronous system with Byzantine faults. However, it is possible to develop a solution based on timeouts in the synchronous system model. Under the assumption of a network guarantee of an upper bound $\delta$ on message latency, we prevent the Byzantine nodes from making non-faulty nodes wait indefinitely resulting in a liveness attack.

Algorithm 1 presents a solution that assumes that the underlying network guarantees that all messages are delivered within $\delta$ time. As long as this assumption holds, Algorithm 1 can guarantee both weak safety and liveness. Each process maintains FIFO queues for each other process where it stores incoming messages from the concerned process. Application messages are delivered immediately after getting popped from the queue. However, control messages are not processed immediately; the algorithm checks to make sure that it is safe to deliver the next message in the queue before completing processing. Whenever a process sends a message it informs every other process about the send event via a control message. Whenever a process delivers a message, it also informs every other process via a control message. Whenever process $p_1$ receives a control or application message from process $p_j$, it pushes it into $\mathcal{Q}_j$. All control messages have timers associated with them to time them out in case of Byzantine behaviour of the sender and/or receiver. When $p_1$ pops a receive control message from any queue $\mathcal{Q}_x$ it waits for either the corresponding send control message to reach the head of its queue (be dequeued), or the receive control message gets timed out in case the send control message does not arrive. This ensures that causality is not violated at $p_1$, while ensuring progress. We also need to ensure that in case of non-Byzantine behaviour on part of both the sender and receiver, both the send control message and receive control message do not time out before the other arrives. In order to achieve this, the timer for receive control messages has to be set to at least $\delta$ as shown in Lemma 1 while the timer for send control messages can be varied (see discussion below). The timer for send control messages can be reduced (it can be set to 0 without compromising weak safety) to implement different behaviours in the system, but the timer for receive control message has to be at least $\delta$, and increasing it will only result in sub-optimal behaviour.

**Algorithm 1: Channel Sync Algorithm**

```
Data: Each $p_i$ maintains a FIFO queue $Q_i$ for every process $p_j$

1. when the application is ready to send message $m$ to $p_j$
   2. send($m, j, \text{app}$) to $p_j$
   3. for all $x \neq i, j$ do
      4. send(($i, j, \text{sent}$), $x$, control) to $p_x$

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

15. when the application is ready to process a message from $p_j$ and
   $Q_j \neq \emptyset$
   16. $(m, i, \text{type}) = Q_j$.pop()
   17. if $\text{type} = \text{control}$ \& $m[2] = \text{delivered}$ then
      18. while timeout period not exceeded \& timer not stopped
         19. wait in a non-blocking manner
      20. if timer is stopped then
         21. while matching send control message not reached head of
            22. $Q[m[1]]$
            23. wait in non-blocking manner
            24. delete $m$
      25. if $\text{type} = \text{control}$ \& $m[2] = \text{sent}$ then
         26. while timeout period not exceeded \& timer not stopped
            27. wait in a non-blocking manner
         28. if timer stopped then
            29. delete the matching receive control message (popped/in
               $Q[m[1]]$ if present)
            30. delete $m$
      31. if $\text{type} = \text{app}$ then
         32. deliver $m$
         33. for all $x \neq i, j$ do
            34. send(($i, j, \text{delivered}$), $x$, control) to $p_x$
```

Therefore, the timer for receive control messages should always be $\delta$.

**Lemma 1.** Under the assumption of a network guarantee of delivering messages within a finite time period $\delta$, no receive control message with a timer greater than or equal to $\delta$ can get processed before the matching send control message at any process when both the sender and receiver processes are correct, during the execution of Algorithm 1.
Algorithm 1, stopping the send control message timer (lines 11,14) spends in the queue is only because of latency induced by control message from a queue is considered to be \( \delta \) delivering messages within a finite time period \( \delta \) events are denoted \( \text{cm}_s \) and testing if it was stopped (lines 25,27) can be replaced by setting \( \text{cm}_s \) upwards while keeping \( \delta \) fixed at \( \delta \) as per Lemma 1.

If \( \delta_s = 0 \) (effectively, no timer for send control messages), then in Algorithm 1, stopping the send control message timer (lines 11,14) and testing if it was stopped (lines 25,27) can be replaced by setting and testing a boolean \text{flag}_\text{timer}_\text{stopped}.

A send event and a receive event are referred to as \( s \) and \( r \), respectively. The control messages we use for send and receive events are denoted \text{cm}_s \text{ and } \text{cm}_r \text{.}

Theorem 3. Under the assumption of a network guarantee of delivering messages within a finite time period \( \delta \), queued messages in Algorithm 1 will be dequeued in at most \( \delta_s + \max(\delta_r, \delta_s) \) time.

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

(1) There may be no control messages in front of \( m \) in \( Q_k \). Since the latency induced by application messages that may be in front of \( m \) is 0, \( m \) will be popped and delivered immediately. The waiting time in the queue for \( m \) is 0.

(2) There may be one or more send control messages before \( m \) in \( Q_k \). Each of the control messages will take at most \( \delta_r \) time to get processed. Since the timers for all of those control messages are ticking concurrently, \( m \) will have to wait for at most \( \delta_r \) time.

(3) There may be one receive control message \text{cm}_r \ in front of \( m \) in \( Q_k \). \text{cm}_r \ is for application message \( m_1 \) sent from \( i_1 \) (before time 0) to \( i_0 \) (received before time 0). Note, if there are multiple receive control messages ahead, the analysis can be independently made for each of them.

(a) \text{cm}_s \ does not arrive in \( \delta_s \). \text{cm}_r \ times out at \( \delta_s \). So total delay is \( \delta_r \).

(b) Otherwise \text{cm}_s \ is inserted in \( Q_k \) in time \( \delta_r \) from time 0.

(i) It may be blocked by \text{cm}_r \( \delta_r \). This time out in \( \delta_r \) time. Total delay is therefore \( \delta_r + \delta_r \).

(ii) It may be blocked by \text{cm}_r \ for application message \( m_2 \) from \( i_2 \) sent before time 0 to \( i_1 \) received before time 0, ahead in \( Q_k \). Therefore \text{cm}_r \ arrived within time \( \delta_r \) from time 0. It waits for \text{cm}_s \( \delta_s \).

(4) Reasoning for the delay introduced by wait for \text{cm}_s \, corresponding to application message \( m_2 \) in \( Q_k \) as follows.

(a) \text{cm}_s \ does not arrive in \( \delta_r \) and \( \text{cm}_r \) times out in \( \delta_r \) after its arrival which was latest at \( \delta_r \) from time 0. Total delay is therefore \( \delta_r + \delta_r \).

(b) Otherwise \text{cm}_s \ arrived within \( \delta_r \) from time 0 because \( m_2 \) was sent before time 0 due to transitive chain \( m_2 \rightarrow m_1 \) and \( m_1 \) was received before time 0. Therefore \text{cm}_s \ is inserted in \( Q_k \) in \( \delta_r \) from time 0.

(i) It may be blocked by \text{cm}_r \( \delta_r \). This time out in \( \delta_r \) time. Total delay is therefore \( \delta_r + \delta_r \).

(ii) It may be blocked by \text{cm}_r \ for application message \( m_3 \) from \( i_3 \) sent before time 0 to \( i_2 \) received before time 0, ahead in \( Q_k \). Therefore \text{cm}_r \ arrived within time \( \delta_r \) from time 0. It waits for \text{cm}_s \( \delta_s \).

(5) The reasoning for the delay introduced by wait for \text{cm}_s \, in \( Q_k \) is identical to the reasoning for the wait introduced by \text{cm}_s \, in the previous item. In particular, \text{cm}_s \ is inserted in \( Q_k \) at \( \delta_r \) from time 0.

We generalize the above analysis as follows. Define \( \rightarrow \) as the “waits for” or “succeeds in time” relation on control messages in the queues at \( p_j \). Then, there exists a chain of control messages 

\[
\text{cm}_r \leftarrow \text{cm}_s \leftarrow \text{cm}_r \leftarrow \text{cm}_s \leftarrow \text{cm}_r \leftarrow \ldots \leftarrow \text{cm}_s
\]

each of which must have arrived in the corresponding \( Q_k \) within time \( \delta_r \) from time 0 (see \( \rightarrow \) below). This chain corresponds to the following chain of application messages:

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

We prove that \( \rightarrow \) \( \text{cm}_r \leftarrow \text{cm}_s \) is inserted in \( Q_k \) within time \( \delta_r \) from time 0, \text{cm}_s \ was inserted in \( Q_k \) within time \( \delta_r \) from time 0.”
Assume the induction hypothesis is true for \( x, x \geq 2 \). We show the result (\( \ast \)) for \( x + 1 \). As \( cmr_{i_0} \) arrives in \( Q_i \) before \( cmsg_{i_0} \), from the induction hypothesis for \( x, cmr_{i_0} \) is inserted in \( Q_i \) within \( \delta_x \) from time 0. It waits for \( cmsg_{i_0} \). \( cmsg_{i_0} \) arrived within \( \delta_x \) from time 0, because \( m_{x+1} \) was sent before time 0 due to transient change \( m_{x+1} \rightarrow m_x \rightarrow \ldots \rightarrow m_1 \) and \( m_1 \) was received before time 0 (because \( cmr_{i_0} \) was received in \( Q_b \) before time 0). Therefore \( cmsg_{i_0} \) is inserted in \( Q_{i_{x+1}} \) within \( \delta_x \) from time 0. (end of proof of (\( \ast \)).

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

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

1. There is no receive control message ahead of \( cmsg_{i_k} \) in \( Q_{i_k} \). Total delay this queue contributes is \( \delta_t + \delta_i \).
2. Total overall delay contributed by queues \( Q_{i_k} \), \( i \in [1, k - 1] \) combined is considered next. Send control messages ahead of \( cmsg_{i_k} \) on timing out contribute up to \( \delta_t \) combined delay. Receive control messages ahead of \( cmsg_{i_k} \) on timing out contribute up to \( \delta_i \) combined delay. The combined contribution of such send and receive control messages is up to \( \delta_t + \delta_i \) delay. The combined delay contributed by queues \( Q_{i_k} \) through \( Q_{i_{k-1}} \).
3. Send (or receive) control messages ahead of \( m \) in \( Q_{i_k} \) contribute a delay of \( \max(\delta_t, \delta_i) \).

Total overall delay contributed by all queues \( Q_{i_k} \) to \( Q_{i_0} \) is thus

\[
\max(\delta_t, \delta_i) + \max(\delta_t, \delta_i) + \max(\delta_t, \delta_i) + \max(\delta_t, \delta_i) = \delta_t + \max(\delta_t, \delta_i).
\]

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

Combining \( k = 0 \) and \( k > 0 \) cases, the total overall delay of \( m \) is bounded by \( \max(\delta_t, \delta_i) + \max(\delta_t, \delta_i) = \delta_t + \max(\delta_t, \delta_i). \)

Since the amount of time each message spends in the message queue is bounded by a finite quantity, every application message will eventually be delivered. Therefore liveness is maintained by Algorithm 1.

**Corollary 1.** Algorithm 1 guarantees liveness.

**Theorem 4.** Under the assumption of a network guarantee of delivering messages within a finite time period \( \tau \), Algorithm 1 can guarantee weak safety by setting timers for control messages as a function of \( \delta \).

**Proof.** In order to ensure weak safety, prior to delivering any message \( m' \) at process \( p_j \), we need to ensure that if \( 3m \in BCP(m') \) such that \( m \) is sent to \( p_j \), then \( m \) is delivered before \( m' \) at \( p_j \).

Algorithm 1 ensures weak safety at any process as follows:

- **Program Order:** Since we assume FIFO channels, messages from \( p_i \) to \( p_j \) get enqueued in \( Q_i \) in program order and get delivered in program order.
- **Transitive Order:** Let \( m \) be sent by \( p_i \) to \( p_j \) at send event \( s_i^v \). Consider a causal chain of \( b \) messages starting at \( s_i^v \) from \( i = i_0 \) and ending at \( j = i_b \) through correct processes and having these events:
  
  \[
  (s_i^v = s_i^v \rightarrow s_{i_1} \rightarrow s_{i_2} \rightarrow \ldots \rightarrow s_{i_{b-1}} \rightarrow s_{i_b} \rightarrow r_i^v)
  \]

  Let \( s_i^v \rightarrow s_i^v \) and \( m' = (s_{i_{b-1}} \rightarrow r_i^v) \) be the last message of the causal chain. This implies that \( m \in BCP(m') \) by transitivity. The control messages for all the events in the causal chain above will reach \( p_j \).

We make the following observations at \( p_j \):
1. In \( Q_{i_0} \), \( cmsg_{i_0} \) (control message for \( s_i^v \)) waits for \( m \) (sent at \( s_i^v \)) to get delivered.
2. From Lemma 1, in \( Q_{i_0} \) (\( 1 \leq \alpha \leq (b - 1) \)), \( cmr_{i_\alpha} \) waits for \( cmsg_{i_{\alpha+1}} \) in \( Q_{i_{\alpha+1}} \) to be processed.
3. In \( Q_{i_\alpha} \) (\( 1 \leq \alpha \leq (b - 2) \)), \( cmr_{i_\alpha} \) waits for \( cmr_{i_{\alpha+1}} \) to be processed.
4. In \( Q_{i_{b-1}} \), \( m' \) (sent at \( s_{i_{b-1}} \)) waits for \( cmr_{i_{b-1}} \) to be processed. Hence, message \( m' \) waits for message \( m \) to get delivered.

Algorithm 1 therefore ensures weak safety: “that \( \forall m \in BCP(m') \) sent to the same \( p_j \), \( m \) gets delivered before \( m' \) at \( p_j \)” under a network guarantee of delivering messages within a fixed time.

**Complexity for Unicasts.** The Channel Sync algorithm uses \( 2(n - 2) \) control messages of size \( O(1) \) each per application message and does not inhibit concurrency (beyond what is necessary to enforce causal order). Any delay up to the maximum in Theorem 3 is essential for causal order in the face of Byzantine processes. The algorithm has a very high degree of concurrency but each process has to manage \( n \) queues and a timer per control message.

Note that in contrast to the Channel Sync algorithm, the algorithm in [1] for causal ordering of broadcasts under weak safety requires \( O(n) \) broadcasts (control message broadcasts) of size \( O(n) \) each per application message broadcast. It also has an added latency equivalent to \( 3\delta \) due to the underlying Bracha’s BRB protocol [4].

**6 BYZANTINE CAUSAL MULTICAST**

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 \( 2^{\lceil \log P \rceil} - 1 \) groups. Although there are several algorithms for causal ordering of messages under dynamic groups, such as [13, 22], none of them consider the Byzantine failure model.

Byzantine Reliable Multicast (BRM) [16, 17] has traditionally been defined based on Bracha’s Byzantine Reliable Broadcast (BRB) [4, 5]. These algorithms require that in every multicast group \( G \), less than \( \lceil |G|/3 \rceil \) 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 \( br\_deliver \)'s message \( m \) from a correct process \( p_i, p_j \) 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_Viability, 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 3.

**Theorem 5.** It is impossible to guarantee liveness and strong safety/weak safety (Definition 6 / 7) while causally ordering multicast messages in an asynchronous message passing system with one or more Byzantine processes.

*Proof.* From Theorems 1 and 2, it is impossible to causally order unicasts with even one Byzantine process in the system. As unicast is a special case of multicast where the group size is 1 (or 2 if you include the sender in the group) and the special case cannot be solved, the general case of multicast also cannot be solved. □

7 CHANNEL SYNC ALGORITHM FOR MULTICAST

The Channel Sync algorithm idea can be extended to work for causal ordering of multicast messages in the face of Byzantine failures, under the assumption of the network guarantee of an upper bound \( \delta \) on the message latency. The modifications to adapt this algorithm are given next. Observe that weak safety (+ liveness) needs to hold only for the \( \rightarrow_B \) relation on messages, which are the messages sent by and received by only correct processes.

7.1 Channel Sync Algorithm for Multicast over BRM Layer

Let \( \delta_{BRM} \) be the latency of the BRM protocol. (This is \( \delta \) for Bracha’s BRM.) The major changes to Algorithm 1 to get Algorithm 2 that runs on top of the BRM layer and also guarantees BC_Viability, BC_Integrity, BC_Self-delivery, BC_Reliability, are as follows. (1) The application messages are sent via the BRM layer whereas send and receive control messages are sent directly. As the BRM layer does not maintain source-order delivery, to ensure source order of multicasts, sequence numbers are used and a non-blocking wait of up to \( \delta_{BRM} \) is introduced when a message is br_delivered (lines 3-4).

Further to ensure that a send control message is enqueued in \( Q_j \) after the matching application message, it waits for \( \delta_{BRM} \) on arrival before further processing. Further, to ensure FIFO enqueuing of receive control messages w.r.t. send control messages sent before them, a receive control message is also required to wait for \( \delta_{BRM} \) on arrival before further processing (lines 9-10). (2) The send control message contains the group members instead of the receiver \( j \) in the second parameter (line 7). (3) When this send control message is received, the parameter \( G \) is manipulated to track the matching receive control messages in their queues and its timer is stopped if all matching receive control messages from members in \( G \) are in their queue or popped (lines 14,17,20). (4) When popped, a send control message deletes the matching receive control messages if they are present in their queue or are popped (lines 33-34).

Consider a correct sender \( p_s \) and a correct receiver \( p_r \). The earliest that an observer \( p_r \) can enqueue the receive control message is \( \delta_{BRM} \) before the app message from \( p_s \) to \( p_r \) and 0 wait for it, 0 for the receive control message from \( p_s \) to \( p_r \), and a wait of \( \delta_{BRM} \) for the receive control message before enqueuing. In contrast, the latest that the send control message from \( p_s \) to \( p_r \) can get enqueued at \( p_s \) is \( 2\delta_{BRM} + \delta \) max of \( \delta_{BRM} \) for \( p_s \)’s app message to be br_delivered to \( p_r \) and any ensuing wait before it sends the send control message to \( p_r \), \( \delta \) for the send control message to reach \( p_r \), and a wait of \( \delta_{BRM} \) for the send control message before enqueuing. So \( \delta_r = \delta + \delta_{BRM} \), the difference. \( \delta \) can be 0 or larger with the same trade-offs as discussed for Algorithm 1. If \( \delta_r = 0 \) (effectively no timer for send control messages), receive control messages in their queues may wait \( \delta_r \) until they time out as they may not get deleted when the send control message gets popped and deleted (because the receive control message arrived after that time).

It is not necessary to send the group members \( G \) in the second parameter of the send control message and we can eliminate this space overhead and corresponding time overhead for processing \( G \). In this case, stopping the send control message timer is not useful (because we cannot track the matched receive control messages in order to delete them (lines 33-34)) nor is it possible (lines 17,20).

This implies that a send control message must not need a timer (i.e., \( \delta \) is effectively set to 0), and the send control message’s second parameter \( G \) is to be replaced by \( x \) (line 7). Correctness of the algorithm is not impacted.

The proofs of correctness (weak safety and liveness) are almost identical to those for Algorithm 1. BC_Viability, BC_Integrity, BC_Self-delivery, BC_Reliability can easily be seen to follow from the corresponding BRB layer properties. This leads to:

**Theorem 6.** Under a network guarantee of delivering messages within \( \delta \) time, Algorithm 2 ensures BC_Viability, BC_Integrity, BC_Self-delivery, BC_Reliability, weak safety and liveness.

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

**BC_Viability:** If a correct process executes bc_deliver(m) from correct process \( p_x \), it must have executed br_deliver(m) from \( p_x \) (lines 3-8, 21-37). Since the BRM layer guarantees validity, if a correct process executes br_deliver(m) from correct process \( p_x \), \( p_x \) must have executed br_multicast(m). From lines 1-2 and the algorithm, correct process \( p_x \) executes br_multicast(m) only when it has executed bc_multicast(m).

**BC_Integrity:** At a correct process, the BRM layer delivers messages to the BCM layer by pushing messages into a FIFO queue.
Algorithm 2: Channel Sync Algorithm for Multicast Using BRM Layer

Data: Each $p_i$ maintains a FIFO queue $Q_j$ for every process $p_j$

1. when the application is ready to send message $m$ to group $G$ via $\text{bc_multicast}(m, G)$:
2. $\text{br_multicast}(m, G)$ to each $p_j \in G$
3. when $\text{br_deliver}(m, G)$ from $p_j$:
4. wait non-blocking for $\min(\delta_{\text{BRM}} \text{ since highest arrived seq_num, until no lower seq_num existing})$
5. if $j = i$
6. for all $x \in D$
7. $\text{send}(i, G, \text{sent}, x, \text{control})$ to $p_x$
8. $Q_j.\text{push}(m, G, \text{app})$
9. when $(m, i, \text{control})$ arrives from $p_j$:
10. wait non-blocking for $\delta_{\text{BRM}}$
11. $Q_j.\text{push}(m, i, \text{control})$
12. start timer for message $m$
13. if $m[2] = \text{sent}$
14. $D = m[1]$
15. for all $x \in D$
16. if matching receive control message is in $Q_x$ or popped then
17. stop timer of matching receive control message:
18. $D = D \setminus \{x\}$; stop timer if $D = \emptyset$
19. if $m[2] = \text{delivered}$
20. if matching send control message $c$ is in $Q_m[1]$ or popped then
21. stop timer; $c.D = c.D \setminus \{m[0]\}$; stop timer of $c$ if $c.D = \emptyset$
22. when the application is ready to process a message from $p_j$ and $|Q_j| \neq 0$:
23. Only one instance of this block is executed at a time for a particular $Q_x$
24. $(m, *, \text{type}) = Q_j.\text{pop}()$
25. if $\text{type} = \text{control} \land m[2] = \text{delivered}$ then
26. while timeout period not exceeded \& timer not stopped do
27. wait in a non-blocking manner
28. if timer is stopped then
29. while matching send control message not reached head of $Q_m[1]$ do
30. wait in non-blocking manner
31. delete $m$
32. if $\text{type} = \text{control} \land m[2] = \text{sent}$ then
33. while timeout period not exceeded \& timer not stopped do
34. wait in a non-blocking manner
35. for all $x \in m[1] \setminus D$ do
36. delete the matching receive control message (popped/in $Q_x$ if present)
37. delete $m$
38. if $\text{type} = \text{app}$ then
39. $\text{bc_deliver}(m, \text{app})$
40. for all $x$ do
41. $\text{send}(i, j, \text{delivered}, x, \text{control})$ to $p_x$

as seen in line 8. Since the BRM layer executes $\text{bc_deliver}(m)$ 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 21-37.

BC Self-Delivery: If a correct process $p_i$ executes $\text{bc_multicast}(m)$, and it is present in the multicast group $G$, it will $\text{br_multicast}(m, G)$ to $G$ (lines 1-2), then $\text{br_deliver}(m)$ message $m$ into the FIFO queue at $p_i$ (and at other processes in $G$) (lines 3-8) and eventually $\text{bc_deliver}(m)$ from the queue at $p_i$ (lines 21-37).

BC Reliability: When a correct process executes $\text{bc_deliver}(m)$ for any message $m$, it means that it has delivered $\text{bc_deliver}(m)$ as seen in lines 21-37. By the reliability property provided by the BRM layer, all correct processes in the group $G$ must have executed $\text{br_deliver}(m)$ (lines 3-8) and placed $m$ in their FIFO delivery queues. This means that they will eventually execute $\text{bc_deliver}(m)$ as seen in lines 21-37.

Liveness: Follows from Corollary 1 to Theorem 3.

Weak Safety: Follows from Theorem 4. The use of multicast instead of unicast does not affect the correctness of weak safety. To see this, consider $p_x$ uncasting $m_1$ to $p_y$, let $m_1 \rightarrow m_2$ via a message chain which ends with $m_2$ received by $p_y$ and then $m_2$ is sent by $p_y$ to $p_z$. From the logic in Theorem 4, at $p_z$, the send control message for $m_1$ will be popped from $Q_x$ before the receive control message for $m_2$ from $Q_y$, which is ahead of $m_2$ in $Q_y$. As the algorithm sends the application message to its destinations before broadcasting the corresponding send control message, even if $p_x$ had multicast $m_1$ to $\{p_y, p_z\}$, $m_1$ would be ahead of and be popped before the send control message for $m_1$ from $Q_x$ at $p_z$. Therefore it is guaranteed that even if multcasts are used, $m_1$ is delivered before $m_2$ at $p_z$ and weak safety is satisfied. (Further, as indicated above, the send control message can have timer $\delta_s = 0$ and the group $G$ need not be transmitted on the send control messages; correctness is not affected.)

Complexity for Multicasts. The algorithm uses $n(|G| + 1)$ point-to-point control messages of size $O(1)$ each per multicast to $G$ and does not inhibit concurrency (beyond what is necessary to enforce causal order); any delay up to the maximum in Theorem 3 is essential for causal order in the face of Byzantine processes.

7.2 Channel Sync Algorithm for Multicast without BRM Layer

The Channel Sync Algorithm can also be extended to causally order multicast messages without using the BRM layer. This is useful when (a) the application wants to avoid the $\delta_{\text{BRM}}$ delays until delivery, (b) the objective is only to thwart attacks on the causal ordering property (weak safety and liveness) and not the four properties of BRM, and/or (c) the restriction that the number of Byzantine processes in each group $G$ has to be less than $|G|/3$ cannot be met. The choice between using or not using a BRM primitive depends on the application’s requirements. The following changes will need to be made to Algorithm 2. (i) The $\text{bc_multicast}(m, G)$ in line (2) would be replaced by “send $(m, G)$ to each $x \in G$”. (ii) The $\text{bc_deliver}(m, G)$ in line (3) would be replaced by arrival of a message $(m, G)$. (iii) Lines (4) and (10) would be eliminated because there is no need to wait as the BRM primitive is not used. Further, $\delta_s$ would reduce from $\delta_{\text{BRM}} + \delta$ to $\delta$. 

108
8 DISCUSSION

Byzantine causal broadcast under weak safety is solvable in an asynchronous system [1, 19]. Byzantine causal unicast or multicast are not. One cannot use a Byzantine fault-tolerant (BFT) causal broadcast protocol to implement point-to-point or multicast abstraction by adding recipient-ID and filtering on arrival only those messages intended for the local node because the filtering mechanism at the local node can be voided/compromised if the local node is Byzantine. 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 node and it can read a message not intended for it. A $p_i$ to $p_j$ unicast must be kept private to the two. This is not possible without the use of cryptograph primitives, which are not considered as mentioned in the system model.

We rule out full-information protocols (FIP) [8] where the entire transitive 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. Item (i) can also be understood as follows: a FIP performs a form of flooding which essentially implements broadcasting, which is not permitted for maintaining privacy.

Synchronization mechanism in the algorithms. In view of the impossibility result, the algorithms we presented are in a synchronous system model. Here, processes are not required to execute in lock-step rounds. In a step of lock-step execution, a process first sends messages and then receives messages sent by others in that very step. After receiving a message in a step, it has to wait for the start of the next step to send messages. (Lock-step execution can be provided by synchronizers [2] in an asynchronous system, and is useful when the application program is synchronous, i.e., written assuming lock-step execution. It is not possible to design synchronizers under Byzantine failures.) Indeed, if lock-step execution is emulated by a synchronous system or can be simulated, causal order is naturally satisfied because only one message hop is traversed in one step or round. Our algorithms are designed for asynchronous applications that do not use lock-step in their code (see list of applications listed in Section 1, e.g., social networking). If lock-step were used, an additional delay of at least the time needed to emulate a step which would be at least $\delta$, 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 Channel Sync algorithm, $2\delta$ is an upper bound on the delay when there is Byzantine behavior whereas the total delay can be as low as 0. We are not aware of any work which provides lock-step that tolerates Byzantine faults.

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

9 CONCLUSION

This paper showed that it is impossible to implement Byzantine causal order of unicasts and multicasts in an asynchronous system. The Channel Sync algorithms for unicasts and for multicasts were then presented to implement causal order in a synchronous system which has a network guarantee of an upper bound on message latency. The Channel Sync algorithms have a non-trivial cost of implementation but have a very high degree of concurrence.

REFERENCES