Byzantine Fault-Tolerant Causal Broadcast on Incomplete Graphs

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

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

***Abstract*—Causal ordering of broadcasts is a widely used requirement in collaborative distributed software systems. We consider Byzantine-tolerant causal ordering of broadcasts in a replicated data store implemented over an incomplete graph network topology, wherein messages of the broadcast are sent via flooding. We propose two protocols to achieve this. The incomplete graph topology also occurs naturally in wireless networks and in overlay peer-to-peer networks. We identify four properties – safety, liveness, no impersonation, and no avatars – that a Byzantine-tolerant causal broadcast algorithm for a replicated data store over an incomplete graph must satisfy. We also reformulate the traditional properties – validity, integrity, self-delivery, and reliability (or termination) – specified for a complete graph in the literature for a replicated data store system over an incomplete graph topology. We then analyze whether Byzantine processes can mount attacks on these properties in our two protocols. We show results for the classical communication model and the local broadcast model.**

***Index Terms*—Byzantine fault-tolerance, Causal order, Causal broadcast, Incomplete graph, Causality, Wireless network, Peer-to-peer, Flooding, Asynchronous message-passing**

I. Introduction

Causality is widely used in providing useful application-level semantics in distributed systems [1]. Causality is defined by the *happens before* [2] relation on the set of events. 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 [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 social networks, multiplayer online gaming, group editing of documents, event notification systems, and distributed virtual environments.

Causal ordering under the Byzantine failure model is recently studied by Auvolat et al. [4] which considered Byzantine-tolerant causal broadcasts. The solvability of Byzantine-tolerant causal ordering problems is also considered in [5]–[7]. The works in [8]–[11] for causal consistency of replicated data under Byzantine failures relied on broadcasts. All these except [10] assumed the complete graph topology. Our interest is in incomplete graph topologies and arbitary (incomplete) graph peer-to-peer (P2P) overlay topologies because they are the ones that occur in practice rather than complete graph topologies. Complete (point-to-point) graph topology for which many of the results in the literature on Byzantine fault-tolerance exist require a higher (hidden) cost because the point-to-point channels abstraction is provided over underlying incomplete topologies and there is much redundancy in their provisioning.

979-8-3503-9730-7/22/$31.00 ©2022 IEEE

Incomplete graph topologies also occur naturally in wireless communication. Here hop-by-hop transmissions occur to achieve broadcast via flooding in the network. Furthermore, incomplete graph topologies also appear as P2P overlays. The modern real-time interactive applications mentioned above that depend on causal ordering use geo-replication. If using a small number of servers, the client-server delays in large systems are significant. Hence it is preferred that clients locally replicate state and synchronize among themselves over an incomplete graph (peer-to-peer) overlay. An update generated at a client peer has to be broadcast/propagated to all other client peers over the P2P overlay and the only way to achieve this is through flooding. This requires solving the Byzantine-tolerant causal broadcast problem over an incomplete graph.

In the *classical communication model*, it is possible for a Byzantine node to equivocate, i.e., transmit conflicting information to its neighbours. Such behaviour is possible if the message sent by a node to one neighbour is not heard by the other neighbours. In contrast, for the *local broadcast model* [12], [13], a message sent by a node is received identically by all its neighbours. Therefore in this model, an attempt by a node to equivocate can be detected by its neighbours. This model is more pertinent to wireless networks.

We also consider two types of Byzantine behaviour. First, a process may be *rational*, i.e., it behaves in a Byzantine way and mounts an attack only if it knows it will not be detected and identified as having mounted the attack. The second type of behaviour is *irrational*, i.e., a process mounts an attack even if it may be suspected or detected and identified as having mounted the attack.

The main contributions of this paper are as follows:

1) We identify four properties – safety, liveness, no impersonation, and no avatars – that a Byzantine-tolerant causal broadcast algorithm for a replicated data store over an incomplete graph must satisfy. Traditional properties – Validity, Integrity, Self-Delivery, and Reliability (or Termination) – specified for Byzantine Causal Broadcast based on Byzantine Reliable Broadcast [14], [15] for a complete graph exist in the literature; we

reformulate these for a replicated data store system over an incomplete graph topology.

2) We formulate the No Dependency Tracking protocol and the Dependency Tracking protocol for peer-to-peer overlays, based on folklore flooding algorithms [1], [16]. The former has the advantage that it is a no-frills lightweight protocol and works fine when there are no Byzantine processes. The latter tracks causal dependencies in its meta-data, which is more expensive, but is more resilient to some types of Byzantine process attacks. These protocols are developed in the literature and we present them as lightweight solutions to Byzantine-tolerant causal broadcast.
3) For each of the properties identified in Item 1 above, We analyze the solvability of Byzantine-tolerant causal broadcasts under the No Dependency Tracking and the Dependency Tracking protocols for the classical communication model. Table I summarizes these results. A superscript R (I) next to a result denotes that the annotated result holds when the Byzantine processes are rational (irrational). We also analyze how these results for the classical communication model vary in the local broadcast model.

Roadmap. Section II reviews previous work. Section III gives our system model and the definitions of correctness criteria for Byzantine-tolerant causal broadcasts over an incomplete graph topology (or overlay). Section IV gives the No Dependency Tracking and the Dependency Tracking protocols. Section V gives the main results about satisfiability of safety, liveness, no impersonation, and no avatars correctness properties by the two protocols. Section VI gives additional correctness properties and results about their satisfiability by the two protocols. Section VII gives the conclusions.

II. Previous Work

Besides a large body of work on causal order broadcasts in failure-free settings (e.g., [16], [17] and references therein), there has been only some work on causal broadcasts under various failure models. Causal ordering of broadcast messages under crash failures in asynchronous systems was introduced in [3]. This algorithm required each message to carry the entire set of messages in its causal past as control information. The algorithm in [18] implements crash fault-tolerant causal broadcast in asynchronous systems with a focus on optimizing the amount of control information piggybacked on each message.

The first algorithm for causally ordering broadcast messages in an asynchronous system with Byzantine failures is proposed in [4]. The solvability of Byzantine-tolerant causal ordering problems is considered in [5]–[7]. The paper [5] showed that although the algorithm in [4] claims to satisfy safety of $m_1 \rightarrow m_2$ where $\rightarrow$ is the causality relation on messages, that can be considered only a weak form of safety that holds if there is a causal path from the sending of message m_1 to sending of message m_2 going through only correct processes. However, this weak form of safety may not always be useful in practice. The results in [5] went on to prove that unconditional or strong safety is not possible in an asynchronous message-passing system with even a single Byzantine process. The above works used the complete graph topology.

There has also been recent interest in applying the Byzantine fault model to implement causal consistency in distributed shared memory and replicated databases [8], [9], [11]. These rely on broadcasts, e.g., on Byzantine reliable broadcast [15] in [9] and on PBFT (total order broadcast) [19] in [8]. In [11], Byzantine reliable broadcast is used to remove misinformation induced by the combination of asynchrony and Byzantine behaviour. These use the client-server model or complete graph and do not apply to an incomplete graph topology (or peer-to-peer overlay) over processes. The impact of rational misbehaving clients on causal consistency of replicated data in a P2P overlay has been studied in [10]. Algorithms for three models of secure causal consistency were outlined. Some of the attacks could be dealt with by relying on cryptographic primitives, a trusted central server, and trusted hardware. Experimental results evaluating the costs such as latency imposed by the algorithms were given. In contrast, our work considers all possible attacks by Byzantine processes as identified by traditional correctness properties, on causal broadcast, and analyzes whether/how they can be prevented by our simple protocols.

Causal order in mobile wireless networks has been considered in [20], [21], but this considers the cellular network model and not the peer-to-peer model. Furthermore, it does not consider Byzantine processes.

Raynal and Cao [22] discussed how to implement Byzantine-tolerant broadcast assuming a complete graph topology over a network which is incomplete, using a bbai_broadcast primitive. This is possible if the vertex-connectivity of the underlying incomplete graph is $2f+1$ and $f < n/3$, where f is the maximum number of Byzantine processes and n is the total number of processes in the graph (system). Using a layered approach, Byzantine Causal Broadcast (which cannot really give strong safety as shown in [5]) can be run above Byzantine Reliable Broadcast (both of which assume a complete graph topology) over the incomplete graph topology running bbai_broadcast. Drawbacks of this approach are: (i) inherently it cannot provide true safety or strong safety [5], (ii) it incurs an added latency for Byzantine Reliable Broadcast and $O(n)$ control message broadcasts per application broadcast. Further overheads/restrictions are: (iii) to implement Byzantine-tolerant broadcast over an incomplete graph requires complete knowledge of the graph topology and of $2f+1$ node-disjoint paths to each other p_j, and (iv) when p_i broadcasts, it needs to send messages along $2f+1$ node-disjoint paths to each other node j, which incurs its message overheads and latency overheads for the multi-hop paths to each other p_j. In contrast, our lightweight approach explores Byzantine tolerant causal broadcast without any of these overheads/restrictions (ii), (iii), and (iv), nor the requirement that $f < n/3$.

Protocol	Safety	Liveness	No Impersonation	No Avatars	Validity	Integrity	Self-Delivery	Reliability (or Termination)
No Dependency Tracking	no^{R,I}	Theorem 3 no^{I}, yes^{R}	yes^{R,I}	no^{I}, yes^{R}; yes^{I,R} (with TC)	yes^{R,I}	yes^{R,I}	yes^{R,I}	no^{I}, yes^{R}; no^{I}, yes^{R} (Th. 3) (with TC)
Dependency Tracking	no^{R,I}	Theorem 3 no^{I}, yes^{R}	yes^{R,I}	no^{I}, yes^{R}; yes^{I,R} (with TC)	yes^{R,I}	yes^{R,I}	yes^{R,I}	no^{I}, yes^{R}; no^{I}, yes^{R} (Th. 3) (with TC)

TABLE I: Solvability of Byzantine causal broadcast over an incomplete graph topology (or a peer-to-peer overlay) in a fully asynchronous system for the classical communication model. All results hold for both protocols for the local broadcast model except that the no^{I} changes to yes^{I} for the No avatars property and the Reliability (or Termination) property, without TC. TC = trusted component. Results are specified for TC when different from the default without TC.

III. System Model

The distributed system is modelled as an undirected graph $G = (P, C)$. Here P is the set of processes (peers) 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 assumed to be an arbitrary graph and in particular, we do not assume a complete graph. A process can only send a message to its neighbors to which it has an edge in C. 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 spawn new processes. This model subsumes the ad-hoc wireless network model where the peers communicate over channels to the peers in their radio range. The system model also assumes that messages can be authenticated with signatures (or with some other form of authenticators such as MAC vectors).

In addition to this, we also consider the possibility of the addition of trusted components (TC) [23], [24] as part of a separate analysis for the satisfiability or unsatisfiability of the properties of Byzantine causal broadcast. In this analysis, we consider each node containing a trusted component in the form of specialized hardware. This specialized hardware continues to function normally even in the case where the node is compromised and becomes Byzantine. For the purposes of this paper, the trusted component is abstracted as a public key–secret key pair (P_i, S_i) residing at each process p_i. The secret key is used to sign messages and assign monotonically increasing sequence numbers to messages. Every process knows every other process's public key, ensuring that processes can determine whether incoming messages and their sequence numbers are legitimate. As the TC facility may not be available, we analyze the properties of Byzantine causal broadcast separately without the TC and with the TC – the default is a system without the TC.

Processes locally replicate state and synchronize among themselves over the incomplete graph topology (or P2P incomplete graph overlay) C. An update or an *operation* that is *generated* at a process p_i is identified by a tuple (i, seq_i) and is broadcast over C via flooding. This flooding is initiated by p_i by sending the operation to its neighbors in C. When a process p_j receives an operation (that it has not seen before), it *executes* the operation on the local replica and floods the same operation to its other neighbors in C atomically.

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* [2] relation, denoted $\rightarrow$, is an irreflexive, asymmetric, and transitive partial order defined over events in a distributed execution that captures causality. When applied to replicated data stores, the $\rightarrow$ relation on operations $o, o' \in O$, the set of operations in the entire execution, is as follows [25].

Definition 1. *$o \rightarrow o'$ if and only if o' is generated in a replica where o has already been executed.*

$(O, \rightarrow)$ is an irreflexive, symmetric, and transitive partial order.

Definition 2. *The causal past of operation o is denoted as $CP(o)$ and defined as the set of operations in O that causally precede operation o under $\rightarrow$.*

A linear extension $(O, <)$ of a partial order set $(O, \rightarrow)$ is defined to be a serialization S_x =$(O, <)$. Thus, $\forall o, o' \in O$, $o \rightarrow o' \Rightarrow o < o'$. A history is causally consistent, i.e., enforces causal order, if and only if for each process p_i there is a serialization S_i that respects the order $(O, \rightarrow)$ [25]. Thus, a history is causally consistent if and only if all processes execute operations as per some causal serialization.

We identify four properties that should be satisfied by causal broadcast over an incomplete graph topology in the face of Byzantine attacks.

Definition 3. ***Safety:*** *$\forall o' \in CP(o)$, no correct process executes o before o'.*

Definition 4. ***Liveness:*** *Each operation o generated by a correct process is eventually executed at each correct process.*

Definition 5. ***No impersonation:*** *An operation associated with identifier (i, seq_i) that is executed at a process p_j should have been generated by process p_i.*

No impersonation corresponds to the terms authenticity and non-repudiation used in the literature.

Definition 6. ***No avatars:*** *The operation o associated with identifier (i, seq_i) when executed at p_j should be identical*

to the operation o' associated with the same identifier when executed at p_k for all correct p_j, p_k.

Additional properties for the replicated data store system over incomplete graphs, that are counterparts of Byzantine Causal Broadcast and Byzantine Reliable Broadcast in message-passing systems over a complete overlay topology, are formulated and the protocols are analyzed for these reformulated properties in Section VI.

IV. Protocols for Causal Ordering over Incomplete Graph

The No Dependency Tracking protocol and Dependency Tracking protocols for flooding in an incomplete graph given in this section are based on folklore and literature [1], [16]; we present them in the context of Byzantine-tolerant causal broadcast.

A. No Dependency Tracking Protocol

Processes locally replicate state and synchronize among themselves over the incomplete graph topology (or the P2P incomelete graph overlay) C. An update or an *operation* that is *generated* at a process p_i is identified by a tuple (i, seq_i) and is broadcast over C via constrained flooding. This flooding is initiated by p_i by sending the operation to its neighbors in C. A process p_i maintains a $next\_seq\_no\_expected[j]$ for each other process p_j which is $1+$ the sequence number of the latest operation from p_j that it has seen and executed. When a process p_i receives an operation (j, seq_j) (that it has not seen before) generated by p_j from neighbor p_k:

1) If $seq_j > next\_seq\_no\_expected[j]$, the operation is buffered and not executed locally nor forwarded.
2) If $seq_j = next\_seq\_no\_expected[j]$, p_i *executes* the operation on the local replica and floods/forwards the same operation to all neighbors in C except p_k atomically. If there are other buffered operations in a continuous window generated by p_j, those are also executed on the local replica and forwarded in sequence. $next\_seq\_no\_expected[j]$ is updated to $1+$ the upper edge of the window of operations executed and forwarded.

Theorem 1. *In a fault-free system, the No Dependency Tracking protocol guarantees safety and liveness of causal order.*

Proof. **Safety:** Let $o_i \rightarrow o'_j$. This implies that once o_i was generated at p_i, it was flooded and reached p_j where it was executed and forwarded onwards for flooding, after which o'_j was generated at p_j and then forwarded onwards as part of flooding o'_j. Due to FIFO channels, the nature of flooding, and the assumption of no faulty processes which guarantees that all operations are forwarded in sequential order of sequence number seq_l for each (l, seq_l) identifier, operation o_i will reach and be executed at each process p_k before o'_j reaches and is executed at p_k.

Liveness: Due to FIFO channels, the nature of flooding, and the assumption of no faulty processes which guarantees that all operations are forwarded in sequential order of sequence number seq_l for each (l, seq_l) identifier, an operation o_i will reach and be executed at each process in the system. Hence each operation generated is eventually executed at each other process. □

As a further modification to the protocol, as explained in Section V-A, each process should also sign the message containing its operation that it generates, to thwart certain attacks on safety.

B. Dependency Tracking Protocol

Processes locally replicate state and synchronize among themselves over the incomplete graph topology (or the incomplete graph P2P overlay) C. An update or an *operation* that is *generated* at a process p_i is identified by a tuple (i, seq_i) and is broadcast over C via constrained flooding. This flooding is initiated by p_i by sending the operation to its neighbors in C. When a process p_j receives an operation (that it has not seen before), if it is *safe* (defined below) to execute the operation on the local replica, it *executes* the operation on the local replica and floods the same operation to its other neighbors in C atomically.

Each operation that is generated is associated with a set of dependencies to enforce causal ordering. The standard ways to represent these dependencies are using version vectors [26] or direct (immediate) dependencies [27], [28]. When an operation arrives at a process via the flooding, it is executed only when it is *safe* to do so, i.e., the dependencies as specified in the version vector or direct dependency set associated with the operation are satisfied. Thus, the operations corresponding to those dependencies have also been locally received and executed before this operation is executed. There are two options about the forwarding of an operation – either (i) it can be forwarded once it is received and possibly before its dependencies are satisfied and the operation is executed, or (ii) only after its dependencies are satisfied and the operation is executed. We assume option (ii) because if a dependency o_j is not satisfied when an operation o_i is received, the receiver can pinpoint the sender peer as a Byzantine process that is not following the protocol. Thus, a rational Byzantine process will not mount such an attack of forwarding an operation whose dependencies have not been forwarded. Further, there is no advantage forwarding o_i as the next process to receive it will not be able to execute it either without receiving o_j which is yet to be forwarded.

Theorem 2. *In a fault-free system, the Dependency Tracking protocol guarantees safety and liveness of causal order.*

Proof. **Safety:** Let $o_i \rightarrow o'_j$. This implies that once o_i was generated at p_i, it was flooded and reached p_j where it was executed once its causal dependencies were satisfied, and forwarded onward for flooding, after which o'_j was generated at p_j and then forwarded onward as part of flooding o'_j. o_i could be a direct causal dependency of o'_j. Due to FIFO channels, the nature of flooding, and the assumption of no faulty processes

which guarantees that an operation is forwarded after its direct causal dependencies and hence transitively after its indirect dependencies are satisfied, locally executed, and forwarded, operation o_i will reach and be executed at each process p_k before o'_j reaches and is executed at p_k.

Liveness: We prove by contradiction. Assume o_i is the first operation that is not executed when received by some process, say p_k, from some process, say p_j, because its causal dependencies are not satisfied. Therefore when p_j received o_i, it was executed and forwarded to p_k because the causal dependencies were satisfied, i.e., those causal dependency operations were previously received, executed, and forwarded. As channels are FIFO, p_k would also have received those operations before receiving o_i, contradicting the assumption that such an o_i exists. □

As a further modification to the protocol to thwart certain attacks on safety, which we explain and justify in Section V-A, each process should sign the message containing the operation it generates and its direct dependencies. In addition, it should also include a hash (cryptographic summary) of its direct causal dependencies.

V. Analysis of Protocols and Attacks

A. Safety

1) Dependency Tracking protocol: We identify several types of safety attacks by a Byzantine process on the Dependency Tracking protocol, and analyze whether they can be countered. Let x be an operation generated at a Byzantine process. Let z, y, and w be other operations. Let $z \rightarrow y \rightarrow x \rightarrow w$. Here z is in the causal past of y and both are in the causal past of x which is at the Byzantine process. w is in the causal future of x and by transitivity, of z and y. The Byzantine process can mount attacks as follows.

1) Delete dependency of y on z. To mount this attack, operation z should not be forwarded by the Byzantine process until after y arrives, y's dependency on z is deleted, and y is forwarded.
   As a result, y can get executed before z at some correct process p_q, resulting in a safety violation.
2) Add to dependencies of z the dependency on y. To mount this attack, the dependency of y on z should also be deleted, so that z will be executed before y. Further, forwarding of operation z, which arrives first at the Byzantine process, should wait until operation y arrives and the Byzantine process learns of it so dependency y can be added to dependencies of z.
   As a result, y can get executed before z at some correct process p_q, resulting in a safety violation.
3) Add to dependencies of y the dependency on x. To mount this attack, the dependency of x on y should also be deleted, so that x will be executed before y. Further, forwarding of operation y, which arrives first, should wait until operation x is executed and the Byzantine process generates x so dependency x can be added to dependencies of y.
   As a result, x can get executed before the causally earlier operation y at some correct process p_q, resulting in a safety violation.
4) Delete dependency of x on y. As x is generated locally at the Byzantine process, this deletion or non-insertion of the dependency on y in the dependencies of x is entirely under the local control of the Byzantine process. As part of the attack, the propagation of y, which arrives at the Byzantine process before it generates x, is postponed to after propagating x.
   This attack can be mounted and there cannot exist any counter to this attack. The deletion of this dependency makes x concurrent to y and x can get executed before its causally earlier operation y at some correct process p_q, resulting in a safety violation.
5) Add to dependencies of y/x the dependency on w. To mount this attack, the Byzantine process will need a way to learn about or guess the future operation w and add it as a dependency to dependencies of y/x before forwarding y/x. (If w cannot be guessed, the Byzantine process waits to receive w and do this before forwarding y/x.) Also, the dependency of w on y/x should be deleted when w arrives at and is executed at the Byzantine process and before forwarding w, so that w may be executed before y/x.
   This addition of a dependency on a future operation w causes y/x to be executed after the future operation is executed at some correct process p_q, resulting in a safety violation.

Attacks (1)-(3) can be prevented by requiring each process to sign the message containing the operation it generates and its direct dependencies. A Byzantine process cannot forge the signature of another process.

Attack (5) can be prevented by requiring a message containing an operation and its direct dependencies to also include a hash (cryptographic summary) of its direct causal dependencies. A Byzantine process cannot create a valid hash of its direct causal dependencies in which it wants to include w because w is a future yet-to-occur operation. If it attempts to do so, the invalid hash can be detected by a correct process receiving the message that includes the hash along with the operation.

Attack (4) cannot be prevented as it is entirely within the control of the Byzantine process. By deleting the dependency of x on y and delaying the propagation of y to after that of x, x can get executed before y at some other processes p_q. Even though $y \rightarrow x$ in actuality, the attack makes x logically concurrent to y. Process p_q has no way of identifying or even suspecting the process that mounted the attack, and hence even rational processes can fearlessly mount this attack. This attack cannot be prevented by using a TC.

Local broadcast model: The above results hold for the local broadcast model because mounting the attack requires addition/deletion of dependencies and/or operation-forwarding order swapping actions that can be implemented in this model.

Type of safety attack	No Dependency Tracking protocol	Dependency Tracking protocol
Delete dependency of y on z	no^{R,I}	yes^{R,I}
Add to dependencies of z the dependency on y	no^{R,I}	yes^{R,I}
Add to dependencies of y the dependency on x	no^{R,I}	yes^{R,I}
Delete dependency of x on y	no^{R,I}	no^{R,I}
Add to dependencies of y/x the dependency on w	no^{R,I}	yes^{R,I}

TABLE II: Guarantee of safety of Byzantine causal broadcast over an incomplete graph topology (or a peer-to-peer overlay) in a fully asynchronous system. x is an operation at a Byzantine process. w is a future operation. $z \rightarrow y \rightarrow x \rightarrow w$. These results hold for both the classical communication model and the local broadcast model.

2) Safety of the No Dependency Tracking Protocol: As this protocol does not track direct dependencies, except implicitly the dependencies on operations generated by the same initiator, we examine how the equivalents of the safety attack types for the Dependency Tracking protocol can be executed on the No Dependency Tracking protocol.

As before, let x be an operation at a Byzantine process. Let $z \rightarrow y \rightarrow x \rightarrow w$. The Byzantine process can mount attacks as follows.

1) Delete dependency of y on z. As z arrives before y, to mount this attack, operation z should not be forwarded by the Byzantine process until after y arrives and is forwarded.
   As a result, y can get executed before z at some correct process p_q, resulting in a safety violation.
2) Add to dependencies of z the dependency on y. As z arrives before y, to mount this attack, operation z should not be forwarded by the Byzantine process until after y arrives and is forwarded.
   As a result, y can get executed before z at some correct process p_q, resulting in a safety violation.
3) Add to dependencies of y the dependency on x. As y arrives before x is generated, to mount this attack, the Byzantine process should not forward y until after x is generated and forwarded.
   As a result, x can get executed before the causally earlier operation y at some correct process p_q, resulting in a safety violation.
4) Delete dependency of x on y. As x is generated locally at the Byzantine process, this logical deletion or non-insertion of the dependency on y in the dependencies of x is entirely under the local control of the Byzantine process. To mount the attack, the propagation of y, which arrives at the Byzantine process before it generates x, is postponed to after generating and propagating x.
   The deletion of this dependency makes x concurrent to y and x can get executed before its causally earlier operation y at some correct process p_q, resulting in a safety violation.
5) Add to dependencies of y/x the dependency on w. To mount this attack, the Byzantine process will need a way to learn about the future operation w. When x is generated or y is received, the Byzantine process delays forwarding y/x until after the future operation w arrives, is executed at, and is forwarded by the Byzantine process.
   This addition of a dependency on a future operation w causes y/x to be executed after the future operation is executed at some correct process p_q, resulting in a safety violation.

None of these attacks can be prevented. Signing messages does not help because the Byzantine process is not tampering with them, but is merely swapping the order of forwarding arrived operations. A process has no way of identifying or even suspecting the process that mounted the attack, and hence even rational processes can fearlessly mount this attack. The use of a TC cannot prevent these attacks.

Observe that swapping the order of forwarding arrived operations generated by a common process does not amount to an attack because a correct process executes operations in sequence number order as generated by the generator process; and signing messages prevents tampering with the sequence number field. Hence, messages need to be signed.

Local broadcast model: The above results hold for the local broadcast model because mounting the attack requires swapping the order of forwarding operations and this can be implemented in this model.

B. Liveness

For a liveness attack to be successful, an operation generated by a correct process should be forever prevented from being executed at some other correct process.

Theorem 3. *In a system with up to f Byzantine processes, a liveness attack can be mounted if and only if the vertex connectivity k of the overlay graph topology is at most f.*

Proof. By definition, a graph is k-connected if $n > k$ and the removal of less than k nodes does not disconnect the graph. From Menger's Theorem [29], a graph is k-connected if and only if for any two nodes $u, v \in P$, there exist k node-disjoint uv-paths.

If $k \leq f$, there are at most f node-disjoint paths connecting some correct node u that generates an operation and some correct node v on which the liveness attack is to be mounted. If there is at least one Byzantine process on each of these at most f node-disjoint paths, they can each block the forwarding of u's operation towards v. Thus, if $k \leq f$, a liveness attack can be mounted.

If $k > f$, there is at least one (more precisely, there are $\geq k-f$) node-disjoint uv-paths on which there is no Byzantine process and hence nodes along this path will forward u's operation towards and to v. Thus, u's operation can get executed at v. Hence, if $k > f$, a liveness attack cannot be mounted. □

Liveness subject to Theorem 3 can be seen to be satisfied by both the No Dependency Tracking protocol and the Dependency Tracking protocol.

If the Byzantine processes collectively mount a liveness attack on a particular operation, they simply withold forwarding that operation to any of the neighbours. Even though the attack is successful (subject to the above theorem), they will be detected when they forward (messages signed by the generator process containing) other later operations whose dependency set contains the operation that was withheld in the liveness attack (Dependency Tracking protocol) or whose sequence number is not the next expected sequence number for the same source (No Dependency Tracking protocol). Thus, rational Byzantine processes will not mount a liveness attack but irrational Byzantine processes may; the use of a TC cannot prevent this attack.

If a Byzantine process adds a dependency of operation o on a fake operation x that was never generated, a liveness attack can be mounted, but this can be prevented by requiring that each process sign each operation it generates and associated dependency meta-data. This prevents addition of fake dependencies.

Local broadcast model: The results above also hold for the local broadcast model because mounting the attack is simply to withold forwarding the operation to any of the neighbours, which can be implemented in this model.

C. No impersonation

An impersonation attack can be mounted if a process p_j generates an operation and associates it with identifier (i, seq_i) and this gets executed at a correct process p_k by being attributed to p_i. Such an attack can be prevented by requiring each process to sign the operation (and its direct causal dependencies) that it generates and forwards via flooding. A Byzantine process p_j will be unable to sign using p_i's signature, any operation it creates but wants to associate with identifier (i, seq_i).

The above analysis holds for both the No Dependency Tracking protocol and the Dependency Tracking protocol.

Local broadcast model: The above results hold for the local broadcast model because no new form of attack can be mounted in the model (compared to the classical communication model), and preventing the attack by signing can be done in both models.

D. No avatars

An avatar attack can be mounted if a process p_i propagates two different operations associated with its own identical identifier (i, seq_i) and causes these two different operations with the same identifier to be executed at different correct processes p_j and p_k.

It is not explicitly possible to prevent such an attack. Requiring signed messages does not help because the Byzantine process signs the two different operations as generated by itself. However, when a correct processes receives the two different signed operations with the same identifier along different paths in the overlay topology, it can detect that an avatar attack has been mounted and pinpoint the process that has mounted the attack. This detection prevents a rational process from mounting the attack as its identity as a Byzantine process will become known in the system. The attack can be mounted by an irrational Byzantine process.

However, if we assume a TC that can provide unique sequence numbers, then signing such messages can prevent avatar attack.

The above analysis holds for both the No Dependency Tracking protocol and the Dependency Tracking protocol.

Local broadcast model: Mounting the attack requires sending different messages with the same sequence number to different neighbours – and this is not allowed in this model. Thus, neither rational nor irrational processes will be able to mount the attack.

VI. Additional Correctness Properties

In addition to the correctness properties – safety, liveness, no impersonation, and no avatars – for a replicated data store over an incomplete graph topology that we defined, there can be stated four additional properties which have counterparts for the complete graph overlay in message-passing systems. For the complete graph (overlay) for message-passing systems, the following properties have been defined for Byzantine causal order reliable broadcast [4].

- Validity [15]: If a correct process delivers a message m from a correct process p_s, then p_s must have executed broadcast(m).
- Integrity [15]: For any message m, a correct process executes deliver(m) at most once.
- Self-delivery [15]: If a correct process executes broadcast(m), then it eventually executes deliver(m).
- Reliability (or Termination) [15]: If a correct process executes deliver(m), then every other correct process also (eventually) executes deliver(m).
- Causal order [5]: Let the $\rightarrow$ relation on messages $m \rightarrow m'$ be such that the send event of m happens before the send event of m'. If $m \rightarrow m'$ then no correct process delivers m' before m.

The equivalents of these above properties can be stated for the incomplete graph topology (or the incomplete graph P2P overlay) for our replicated data store model. Next, we state these equivalents and show that they (except the causal order and Reliability/Termination properties) are satisfied by No Dependency Tracking and Dependency Tracking protocols.

- Validity: If a correct process p_i executes operation o generated by another correct process p_j, then o must have been generated, executed, and flooded by p_j.

It is evident that Validity is satisfied for both No Dependency Tracking and Dependency Tracking protocols.

- Integrity: For an operation with identifier (i, seq_i), (signed by the generator process p_i), the operation is executed at correct process p_j at most once.
  It is evident that Integrity is satisfied by both the No Dependency Tracking and Dependency Tracking protocols.
- Self-delivery: An operation generated by a correct process is also eventually executed by that correct process.
  It is evident that Self-delivery is satisfied by both the protocols.
- Reliability (or termination): If a correct process executes an operation with identifier (i, seq_i), then every other correct process (eventually) executes the same operation with the same identifier.
  From the liveness property (Section V-B) for both the protocols, if a correct process executes an operation, it is forwarded via flooding and exposed to a liveness attack by irrational processes subject to Theorem 3. However, as the No Avatars property is not satisfied even if a liveness attack cannot be mounted when irrational Byzantine processes generate operations in the classical communication model, different correct processes may receive and execute different operations with the same identifier and hence Reliability/termination is also not guaranteed. In the local broadcast model, the No Avatars property is satisfied by rational and irrational Byzantine processes, hence Reliability/termination is exposed to a liveness attack by irrational processes subject to Theorem 3.
  If a TC is assumed, the No avatars property is satisfied but irrational processes may mount a liveness attack, subject to Theorem 3, for both the classical communication and the local broadcast models for both protocols.
- Causal order: This is covered in our definition of safety (Definition 1) for replicated data stores and was analyzed in Section V-A.

VII. Conclusions

We identified four properties – safety, liveness, no impersonation, and no avatars – that should be satisfied for causal consistency of a replicated data store communicating over an incomplete graph topology (or an incomplete graph P2P overlay) and considered two flooding-based protocols from the literature. The No Dependency Tracking protocol is prone to more types of safety attacks than the Dependency Tracking protocol; however the Dependency Tracking protocol is also subject to one type of safety attack. Liveness is satisfied if a certain relationship is satisfied by the vertex connectivity k of the overlay topology and the maximum number of Byzantine processes f, namely $k > f$. The No Impersonation property can be satisfied by the two protocols. The No Avatars property can be satisfied by both protocols if the Byzantine processes are rational. The above results for the classical communication model also hold, with some exceptions, for the local broadcast model which is more prevalent in wireless ad-hoc networks.

Although our protocols cannot provide safety in some of the cases analyzed due to the impossibility result of [5], neither can the approach of layering Byzantine Causal Broadcast over Byzantine Reliable Broadcast (both for complete graph overlays) over bbai_broadcast (which is over an incomplete graph) and our lightweight direct approach avoids the overheads of this layering approach. Our approach uses the bare minimum number of messages $|C|$ in each broadcast, eliminates the latency and message overheads of Byzantine Reliable Broadcast and the bbai_broadcast, and does not require knowledge of the graph topology. Further, liveness is guaranteed in a graph with vertex connectivity $(f + 1)$ and f is not restricted to be less than $n/3$. We do not require $2f + 1$ vertex connectivity in the incomplete graph, as is required by the bbai_broadcast primitive in the layered approach. Furthermore, standard approaches for dealing with the dependency vector/ direct dependency representation in the face of churn as well as a dynamically changing incomplete topology can be adapted. Thus our approach is suited to real-time collaborative applications such as social networking and massive multiplayer online gaming.

References

[1] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.

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

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

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

[5] A. Misra and A. D. Kshemkalyani, "Solvability of byzantine fault-tolerant causal ordering problems," in *Proc. 10th International Conference on Networked Systems, NETYS 2022, Virtual Event*, ser. Lecture Notes in Computer Science, M. Koulali and M. Mezini, Eds., vol. 13464. Springer, 2022, pp. 87–103. [Online]. Available: https://doi.org/10.1007/978-3-031-17436-0_7

[6] ——, "Causal ordering in the presence of byzantine processes," in *28th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2022.

[7] ——, "Byzantine fault-tolerant causal ordering," in *24th International Conference on Distributed Computing and Networking (ICDCN)*, 2023.

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

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

[10] A. van der Linde, J. Leitão, and N. M. Preguiça, "Practical client-side replication: Weak consistency semantics for insecure settings," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2590–2605, 2020.

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

[12] V. Bhandari and N. H. Vaidya, "On reliable broadcast in a radio network," in *Proc. 24th PODC*. ACM, 2005, pp. 138–147.

[13] C. Koo, V. Bhandari, J. Katz, and N. H. Vaidya, "Reliable broadcast in radio networks: the bounded collision case," in *Proc. 25th PODC*. ACM, 2006, pp. 258–264.

[14] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *J. ACM*, vol. 32, no. 4, p. 824–840, Oct. 1985. [Online]. Available: https://doi.org/10.1145/4221.214134

[15] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987. [Online]. Available: https://doi.org/10.1016/0890-5401(87)90054-X

[16] R. Friedman and S. Manor, "Causal ordering in deterministic overlay networks," Computer Science Department, Technion, Tech. Rep., 2004.

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

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

[19] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, M. I. Seltzer and P. J. Leach, Eds., 1999, pp. 173–186.

[20] S. Alagar and S. Venkatesan, "Causal ordering in distributed mobile systems," *IEEE Trans. Computers*, vol. 46, no. 3, pp. 353–361, 1997. [Online]. Available: https://doi.org/10.1109/12.580430

[21] P. Chandra and A. Kshemkalyani, "Causal multicast in mobile networks," in *Proc. of The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004)*, 2004, pp. 213–220. [Online]. Available: https://doi.org/10.1109/MASCOT.2004.1348235

[22] M. Raynal and J. Cao, "From incomplete to complete networks in asynchronous byzantine systems," in *Proc. of the 35th International Conference on Advanced Information Networking and Applications (AINA-2021)*, ser. Lecture Notes in Networks and Systems, vol. 225. Springer, 2021, pp. 102–112. [Online]. Available: https://doi.org/10.1007/978-3-030-75100-5_10

[23] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo, "Efficient byzantine fault-tolerance," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 16–30, 2011.

[24] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 189–204, 2007.

[25] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto, "Causal memory: Definitions, implementation and programming," *Distributed Computing*, vol. 9, no. 1, pp. 37–49, 1995.

[26] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, pp. 215–226, 1989.

[27] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," *ACM Trans. Comput. Syst.*, vol. 7, no. 3, pp. 217–246, 1989.

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

[29] D. B. West, *Introduction to Graph Theory*. Prentice Hall, 2nd edition, 2001.