Partially Replicated Causally Consistent Shared Memory: Lower Bounds and An Algorithm *

Zhuolun Xiang
University of Illinois at Urbana-Champaign
xiangzl@illinois.edu

Nitin H. Vaidya
Georgetown University
nitin.vaidya@georgetown.edu

ABSTRACT

The focus of this paper is on causal consistency in a *partially replicated* distributed shared memory (DSM) system that provides the abstraction of shared read/write registers. Maintaining causal consistency in distributed shared memory systems has received significant attention in the past, mostly on *full replication* wherein each replica stores a copy of all the registers in the shared memory. To ensure causal consistency, all causally preceding updates must be performed before an update is performed at any given replica. Therefore, some mechanism for tracking causal dependencies is required, such as vector timestamps with the number of vector elements being equal to the number of replicas in the context of full replication. In this paper, we investigate causal consistency in *partially replicated systems*, wherein each replica may store only a subset of the shared registers. Building on the past work, this paper makes three key contributions:

- We present a necessary condition on the metadata (which we refer as a *timestamp*) that must be maintained by each replica to be able to track causality accurately. The necessary condition identifies a set of directed edges in a *share graph* that a replica's timestamp must keep track of.
- We present an algorithm for achieving causal consistency using a timestamp that matches the above necessary condition, thus showing that the condition is necessary and sufficient.
- We define a measurement of timestamp space size and present a lower bound (in bits) on the size of the timestamps. The lower bound matches our algorithm in several special cases.

CCS CONCEPTS

• **Theory of computation → Shared memory algorithms**.

KEYWORDS

distributed shared memory; causal consistency; tight conditions; lower bounds

*A brief announcement summarizing the results appeared at PODC 2018 [38]. This research is supported in part by National Science Foundation award 1409416, and Toyota InfoTechnology Center. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. government.

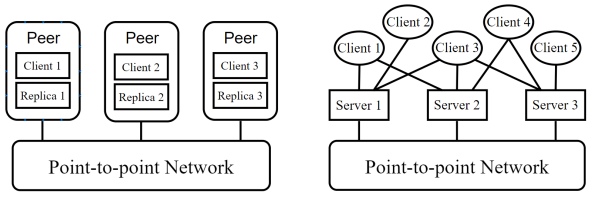
PODC '19, July 29-August 2, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6217-7/19/07…$15.00
https://doi.org/10.1145/3293611.3331600

ACM Reference Format:
Zhuolun Xiang and Nitin H. Vaidya. 2019. Partially Replicated Causally Consistent Shared Memory: Lower Bounds and An Algorithm . In *2019 ACM Symposium on Principles of Distributed Computing (PODC '19), July 29-August 2, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3293611.3331600

1 INTRODUCTION

Distributed shared memory systems maintain multiple replicas of the shared memory locations, which we refer to as shared registers. In recent years, the *causal consistency* model for the shared memory has received significant attention due to its emerging applications, such as social networking. Intuitively, causal consistency ensures that before an update is applied to a shared register, all the causally preceding updates must be applied at the same replica. This paper mainly focuses on the architecture illustrated in Figure 1a, which we refer to as the *peer-to-peer* architecture. Each peer has a *client* that issues read/write operations to the shared memory and a *replica* that helps implement the shared memory abstraction. We focus on the case when each replica is *partial* and may store a copy of just a subset of the shared registers. *Full replication* is obtained as a special case when each replica stores a copy of each shared register.



(a) Peer-to-peer architecture **(b) Client-server architecture**

We primarily present the results for the peer-to-peer architecture. The results can be extended to the client-server architecture in Figure 1b where each client may be accessing replicas stored at an arbitrary subset of the servers, as briefly discussed in Section 5.

In the context of *full replication*, several causally consistent shared memory systems have been designed, including Lazy Replication [22], COPS [24], GentleRain [12], Orbe [11], SwiftCloud [39], Occult [27], and Causalspartan [35]. Recently, there is also growing interest in *partial replication* due to the potential storage efficiencies that can be attained [2, 4, 6, 7, 9, 16, 17, 19, 25, 27]. For *full replication*, it suffices to use a vector timestamp [8, 14, 26] of length equal to the number of replicas [22] to achieve causal consistency.

Several researchers have observed that *partial replication* requires larger amount of metadata to track causal dependencies [2, 9, 16, 24]. For partial replication, in general, the timestamp (or

metadata) overhead is expected to be larger than that for full replication in order to avoid *false dependencies* as will be explained below. One straightforward method to implement partial replication is by adding "virtual registers" at each replica to simulate full replication. The virtual registers do not store actual data and cannot be accessed by clients. Then solutions for full replication such as vector clocks can be easily adapted for partial replication. However, there are several issues: (1) Every update message with metadata will be sent to all replicas in full replication, which is not necessary for partial replication. This may result in high bandwidth usage. (2) Simulating full replication introduces unnecessary dependencies (which we call *false dependencies*) among the update messages. For instance, if update u_x on register x depends on update u_y on register y, i.e. u_x can only be applied after u_y is applied, then on any replica who received u_x first will wait for the receipt of u_y, even if register y is virtual and not stored locally. However, there is no reason for such delay, since the virtual register y will not be accessed by any client from this replica, and thus u_x can be applied without the receipt of u_y. Therefore this simulation approach may result in stale versions.

Partial replication yields a trade-off between the flexibility of replication, number of false dependencies on update messages, and overhead of the metadata for tracking causality. A goal of our work is to characterize this trade-off. Intuitively, in our solution, each replica maintains an *edge-indexed vector timestamp* which keeps counters for a subset of edges in a *"share graph"* that characterizes how registers are shared among the replicas. We show that our timestamp is optimal in the sense that the subset of share graph edges tracked is necessary for correctness (Theorem 1). Also, there is no false dependency introduced in our solution. Our main contributions are as follows:

- We present a necessary condition on the metadata that must be maintained by each replica to be able to track causality accurately. The necessary condition identifies a set of directed edges in a share graph that a replica's timestamp must keep track of.
  In deriving the necessary condition, we make improvements over results presented in prior work of Hélary and Milani [16, 30].
- We present an algorithm for achieving causal consistency using a timestamp that matches the above necessary condition, thus showing that the condition is necessary and sufficient.
- We define a measurement of timestamp space size and present a lower bound (in bits) on the size of the timestamps. The lower bound matches our algorithm in several special cases.

2 PRELIMINARIES

We assume an asynchronous system, and the replicas communicate using reliable point-to-point message-passing channels. The communication channels are not necessarily FIFO. In Sections 2 through 4, we assume the peer-to-peer architecture in Figure 1a. Each peer contains a client and a replica. There are R peers, and hence there are R replicas. The replicas are numbered 1 through R. Replica i stores copies of a subset of shared registers named X_i. With full replication, $X_i = X_j$ for all replicas i, j. With partial replication, it is possible that $X_i \neq X_j$ for $i \neq j$. We define $X_{ij} = X_i \cap X_j$, the set of registers stored at replicas i and j both. For instance,in partial replication with four replicas, we may have $X_1 = \{x\}$, $X_2 = \{x, y\}$, $X_3 = \{y, z\}$, and $X_4 = \{z\}$, where x, y, z are registers. In this case, $X_{23} = \{y\}$ and $X_{14} = \emptyset$. In practice, set X_r for replica r may change dynamically, however, we consider the *static* case in this paper and leave the *dynamic* case for future work.

Hélary and Milani [16] introduced the notion of a *share graph* to represent a partially replicated system. Similar notions of *graph of groups* are introduced in causal multicast literature as well [5]. We will use the share graph when obtaining results for the *peer-to-peer* architecture in Section 3 and 4. To extend these results to the *client-server* architecture, in Section 5, we will introduce an *augmented* version of the share graph.

Definition 1 (**Share Graph [16]**). *We denote e_{ij} as a directed edge from i to j. Share graph is defined as a directed graph $G = (V, E)$, where $V = \{1, 2, \cdots, R\}$, and vertex $i \in V$ represents replica i. There exist directed edges e_{ij} and e_{ji} in E if and only if $X_{ij} \neq \emptyset$.*

As such, if $e_{ij} \in E$ then $e_{ji} \in E$, and G may be defined as an undirected graph as originally defined in [16]. However, as seen later, it is convenient to represent the sharing using directed edges.

X_{ij} will be referred to as the label of edges e_{ij} and e_{ji}. In this paper, we assume that *each replica has the knowledge of the share graph including the labels on each edge.*

Recall that each peer contains a client and a replica, and the client can issue *read* or *write* operations on a shared register stored at the local replica. Define $read(x)$ to be a read operation on register x, and $write(x, v)$ to be a write operation on register x that writes value v. When performing $read(x)$ or $write(x, v)$ operation on register $x \in X_i$, client i sends a request to replica i, and awaits the replica's response. The response to a write operation is an acknowledgement, and the response to a read operation is a returned value. Define update to be a tuple of the form $update(i, T, x, v)$, where i is the sender of the update, T is the timestamp attached with the update, x is the register being updated and v is the value. As illustrated in Figure 2, upon receiving *write* operations from the client, the replica will *issue updates* to some other replicas, i.e., sending $update(i, T, x, v)$ to other replicas who also replicate x in order to update their registers. Upon receiving $update(i, T, x, v)$ from other replica, the replica can decide when to *apply the update*, i.e., write the new value v into the register x. An *execution* is defined to be a sequence of clients' read/write operations and replicas' operations in issuing/applying updates. With our definition of replica-centric causal consistency in the next section, we will often construct executions by declaring the replicas' operations on updates without explicitly mentioning the clients' operations.

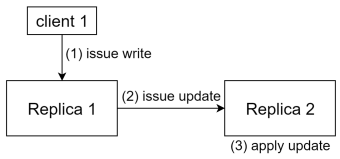


Figure 2: Illustration

2.1 Replica-centric Causal Consistency

In this section, we will use the notions of *issuing an update* and *applying an update* mentioned above. A client i may only read/write registers in X_i. Thus, replica i may only issue updates to registers in X_i. For convenience, each *write* operation on a given register is assumed to write a unique value.

In past work, several variations of causal consistency have been explored. One of the commonly used definitions of causal consistency is defined from clients' viewpoint, which we refer to as *client-centric causal consistency* below.

Definition 2 (Happened-before relation $\rightarrow$ for operations [23]). *Let o_1, o_2 be two operations of the client. o_1 happened-before o_2, denoted as $o_1 \rightarrow o_2$, if and only if at least one the following conditions is true: (1) Both o_1 and o_2 are performed by the same client, and o_1 occurs before o_2. (2) o_1 is a write operation, and o_2 is a read operation that returns the value written by o_1. (3) There exists an operation o_3 such that $o_1 \rightarrow o_3$ and $o_3 \rightarrow o_2$.*

Client-centric causal consistency is defined based on the relation $\rightarrow$ for operations above.

Definition 3 (Client-centric Causal Consistency). *Client-centric causal consistency is achieved if the following two properties are satisfied:*

- **Safety:** *If a read operation o_3 on some register x returns the value written by write operation o_1 on register x, then there* must not exist *another write operation o_2 on register x such that $o_1 \rightarrow o_2 \rightarrow o_3$.*
- **Liveness:** *For a write operation o_1 by some client that writes value v in register x, all replicas that store copies of register x should be updated with the value v within a finite time.*

The causal consistency model addressed in our work is inspired by replicated shared memory systems such as Lazy Replication [22]. We refer to this model as the *replica-centric* causal consistency model. We define the *happened-before* relation [23] between updates as follows.

Definition 4 (Happened-before relation $\hookrightarrow$ for updates). *Given updates u_1 and u_2, $u_1 \hookrightarrow u_2$ if and only if at least one of the following conditions is true:*

(1) u_1 is applied at a replica on any of its register sometime before the same replica issues u_2 on any of its register.
(2) There exists an update u_3 such that $u_1 \hookrightarrow u_3$ and $u_3 \hookrightarrow u_2$.

Intuitively, relation $\hookrightarrow$ is analogous to the happened-before relation between events in the context of causal multicast. That is, an update issued by replica i is considered causally dependent on any updates that were previously applied at that replica, regardless of whether the previously updated registers were read by the client or not.

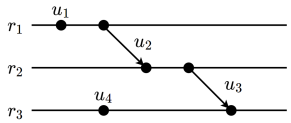


Figure 3: Relation $\hookrightarrow$

We give an example of relation $\hookrightarrow$ in Figure 3. In this example, there are 3 replicas r_1, r_2, r_3, where r_1 issues updates u_1 and u_2, r_2 issues update u_3 and r_3 issues update u_4. u_1 is applied at r_1, u_2 is applied at r_1, r_2, u_3 is applied at r_2, r_3 and u_4 is applied at r_3. In the figure, the send event of arrow with label u_2 depicts when update u_2 is issued at r_1, and the receive event of that arrow depicts the time when r_2 applies update u_2. By condition (1) of the $\hookrightarrow$ definition, we have $u_1 \hookrightarrow u_2$ and $u_2 \hookrightarrow u_3$, and by condition (2) we have $u_1 \hookrightarrow u_3$. Also, u_1 and u_4 are concurrent, i.e. $u_1 \not\hookrightarrow u_4$ and $u_4 \not\hookrightarrow u_1$. Similarly, u_2 and u_4 are concurrent.

Definition 5 (Replica-centric Causal consistency). *Replica-centric causal consistency is achieved if the following two properties are satisfied:*

- **Safety:** *If an update u_1 for register $x \in X_i$ has been applied at a replica i, then there* must not exist *update u_2 for some register in X_i such that (i) $u_2 \hookrightarrow u_1$, and (ii) replica i has not yet applied u_2.*
- **Liveness:**[1] *Any update u issued by a replica i for a register $x \in X_i$ should be applied at each replica j such that $x \in X_j$ within a finite time after all dependencies of u have been applied at j, i.e., all u' for some register $y \in X_j$ such that $u' \hookrightarrow u$ have been applied.*

For three reasons, we consider the replica-centric causal consistency in this paper. (i) First, the necessary conditions presented in Section 3.1 and 4 for the replica-centric causal consistency also applies to the client-centric causal consistency, because the proof construction in this paper can be adapted for client-centric causal consistency by the client reading all registers in its replica before issuing any write operation. (ii) Second, the algorithms for replica-centric causal consistency also implement the client-centric causal consistency, but with possible false dependencies. (iii) Third, in practice, maintaining the replica-centric causal consistency is efficient in metadata size, since it only uses a single timestamp per replica (as compared to, for instance, a timestamp per register per replica for the client-centric causal consistency). Many practical systems, including Lazy Replication [22], ChainReaction [1] and SwiftCloud [39], in fact, conform to the replica-centric view.

Relation with Causal Group Multicast. As we mentioned earlier, the definition of replica-centric causal consistency is analogous to the requirement for causal group multicast [5], where the messages need to be delivered to the processes in a causal order. The following correspondence can be obtained. Replicas sharing the same register x correspond to processes belonging to the same multicast group G_x. Any update to register x by replica i results in a multicast to group G_x by replica i. In the case of partial replication, our algorithm later in section 3.2 can essentially be viewed as causal group multicast with *overlapping groups* [5, 20, 32], where each process may belong to multiple groups (determined by how they share registers) and the multicast within a group is only received by members in that group. Hence our results below in Section 3.1 and 3.2 also apply to causal multicast with overlapping groups. For the sake of the consistency of presentation, we state our results in the context of distributed shared memory. Related work on causal group multicast and the comparison with our work are discussed in Section 6.

> In this paper, we consider algorithms that implement causal consistency by storing and attaching metadata (timestamps) with update messages, where the metadata is some encoding of the information about the execution history. When to apply a received update at a replica is determined only using the replica's local metadata and the metadata attached with the update.

[1] Note that our definition of Liveness implies no false dependencies.

3 TIMESTAMPS FOR REPLICA-CENTRIC CAUSAL CONSISTENCY

In this section, we consider partially replicated shared memory systems, which satisfy the replica-centric causal consistency model in Section 2.1 using an algorithm under the assumptions mentioned in Section 2. In particular, we identify a necessary and sufficient condition on the timestamp τ_i maintained by each replica i. Intuitively, our condition identifies a subset of directed edges in the share graph that are *necessary and sufficient* to "keep track" of for each replica in order to achieve replica-centric causal consistency.

For a replica i, and directed edge e_{jk} (from j to k) in the share graph, Definition 6 defines an (i, e_{jk})-loop as illustrated in Figure 4. We will use $-$ to denote the set difference, i.e., $A - B = \{x \in A \mid x \notin B\}$. After introducing the definition below, we provide intuition behind the definition.

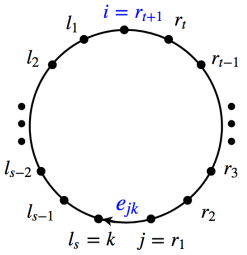


Figure 4: (i, e_{jk})**-loop**

Definition 6 (**(i, e_{jk})-loop**). *Given replica i and edge e_{jk} ($j \neq i \neq k$) in share graph G, consider a simple loop of the form $(i, l_1, l_2, \cdots, l_s = k, j = r_1, r_2, \cdots, r_t, i)$, where $s \geq 1$ and $t \geq 1$. Define $i = r_{t+1}$. The simple loop is said to be an (i, e_{jk})-loop provided that:*

(i) $X_{jk} - \left(\cup_{1 \leq p \leq s-1} X_{l_p}\right) \neq \emptyset$,

(ii) $X_{jr_2} - \left(\cup_{1 \leq p \leq s-1} X_{l_p}\right) \neq \emptyset$, *and*

(iii) for $2 \leq q \leq t$, $X_{r_q r_{q+1}} - \left(\cup_{1 \leq p \leq s} X_{l_p}\right) \neq \emptyset$.

As shown later, when there exists an (i, e_{jk})-loop, replica i needs to keep information regarding updates on edge e_{jk} in order to achieve causal consistency.

Intuition: This discussion refers to Figure 4. The definition of (i, e_{jk})-loop will be used to characterize the timestamp used by replica i in our algorithm. As we will show later in the proof of Theorem 1, the timestamp at replica i must reflect information regarding updates on edge e_{jk} if an (i, e_{jk})-loop exists. Consider the following execution. Let u be an update issued by replica j which is sent to replica k (i.e., an update on edge e_{jk}) since replica k stores the register that u is updating. Also suppose that there is a sequence of causally dependent updates propagated along the path $(j, r_2, \cdots, i, \cdots, l_{s-1}, k)$. Denote the update from l_{s-1} to k as u', so that we have $u \hookrightarrow u'$. Then the timestamps attached with u' should contain enough information about the $u \hookrightarrow u'$ relation for replica k to apply these two updates in the correct order, or postpone the application of u' if u' is received before u. Thus, it is necessary for replicas such as replica i to "keep track of" causally preceding updates that have taken place on edge e_{jk}. This allows replica i to propagate the dependency information to other replicas in the above loop that need it (particularly, replica l_1 to l_s). If condition (i) is not true, update u will also be sent to some replica l_p where $1 \leq p \leq s-1$, since all registers shared by j, k are also shared by j and some replica l_p. Similarly, if condition (ii) or (iii) is not true, the updates along the path $(j, r_2, ..., i)$ will also be sent to some replica l_p. Since the timestamps of these updates sent to some l_p may contain the information about $u \hookrightarrow u'$, it is not necessary for replica i to "keep track of" the causality for updates on edge e_{jk}. On the other hand, when all three conditions are true, replica i has to maintain such information to ensure causal consistency. For more details, the reader may refer to the proofs for the necessary and sufficient condition in later sections.

Example: Figure 5a shows a share graph for a system of 4 replicas. Suppose that $X_1 = \{a, y, w\}$, $X_2 = \{b, x, y\}$, $X_3 = \{c, x, z\}$ and $X_4 = \{d, y, z, w\}$. The label on edges between replicas i and j in Figure 5a corresponds to the registers in X_{ij}. For instance, $X_{34} = \{z\}$. By Definition 6, $(1, 4, 3, 2)$ is *not* a $(1, e_{34})$-loop since $X_{21} - X_4 = \emptyset$ which violates condition (iii). Similarly, $(1, 4, 3, 2)$ is *not* a $(1, e_{23})$-loop due to a similar reason. On the other hand, $(1, 2, 3, 4)$ is a $(1, e_{43})$-loop. Due to the existence of register w in X_{14}, $X_{14} - X_2 \neq \emptyset$, and the reader can easily check that all three conditions in Definition 6 are satisfied. Similarly, $(1, 2, 3, 4)$ is a $(1, e_{32})$-loop.

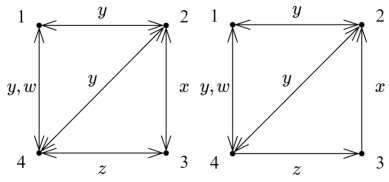
To help present the necessary condition in Section 3.1, we now define a *timestamp graph*. Intuitively, timestamp graph G_i consists of directed edges that are *necessary and sufficient* for replica i to keep track of in its timestamp, as we will show later in Section 3.1 and 3.2.

Definition 7 (**Timestamp graph G_i of replica i**). *Given share graph $G = (V, E)$, timestamp graph of replica i is defined as a directed graph $G_i = (V_i, E_i)$, where*

$$E_i = \{e_{ij} \mid e_{ij} \in E\} \cup \{e_{ji} \mid e_{ji} \in E\} \cup \{e_{jk} \mid \exists\, (i, e_{jk})\text{-loop in } G,\ j \neq i \neq k,\ e_{jk} \in E\}$$

$$V_i = \{u, v \mid e_{uv} \in E_i\}$$

Thus, E_i consists of all directed edges incident at i, and each edge $e_{jk} \in E$ such that an (i, e_{jk})-loop exists. Consider the share graph example in Figure 5(a) again. Figure 5(b) shows the *timestamp graph* for replica 1. Observe that the edge e_{43} is in G_1 but e_{34} is not in G_1, due to the fact that $(1, 2, 3, 4)$ is a $(1, e_{43})$-loop but $(1, 4, 3, 2)$ is *not* a $(1, e_{34})$-loop, as we explained earlier for the example of (i, e_{jk})-loop. By the example above and the definition of timestamp graph, we make the following three observations:



(a) G **(b)** G_1

Figure 5

1. Timestamp graphs may be different from the share graph. 2. Different replicas may have different timestamp graphs. 3. Edges in the timestamp graph are not necessarily bidirectional.

3.1 A Necessary Condition for Timestamps

As briefly stated in Section 2, each replica maintains a timestamp. To achieve replica-centric causal consistency, the timestamp must contain enough information. In this section, we obtain a necessary condition on the timestamps. In particular, Theorem 1 below shows that, if e_{jk} is in the timestamp graph of replica i, then it is necessary

for replica i to "keep track of" updates performed by replica j to registers in X_{jk}. To present the result formally, we introduce some additional terminology.

Definition 8 (**Causal past and Causal dependency graph [29]**). *Causal dependency graph $\mathcal{R}$ of a replica that has applied updates in set U consists of vertices in $S = U \cup \{u' \mid u \in U,\ u' \hookrightarrow u\}$ and directed edges in $\{(u_1, u_2) \mid u_1 \hookrightarrow u_2$ and $u_1, u_2 \in S\}$. Set S is referred as the* causal past *of the replica [29].*

Theorem 1 will use the following terminology:

- We define relation for causal dependency graph $\mathcal{R}_0, \mathcal{R}_1$ of replica i as follows: $\mathcal{R}_0 < \mathcal{R}_1$ if there exists an execution in which replica i's causal dependency graph equals $\mathcal{R}_0$ at some point of time, and $\mathcal{R}_1$ subsequently.
- Two causal dependency graphs with vertex sets S_1 and S_2 are said to differ only in updates on e_{jk} if and only if (i) all the updates in $(S_1 - S_2) \cup (S_2 - S_1)$ are issued by replica j for registers in X_{jk} and (ii) the edges between any vertices in $S_1 \cap S_2$ are identical in both the causal dependency graphs.
- We will say that replica i with causal dependency graph $\mathcal{R}_0$ is *oblivious* to updates on e_{jk}, if the replica's timestamp is identical for every pair of causal dependency graphs $\mathcal{R}_1$ and $\mathcal{R}_2$ such that (i) $\mathcal{R}_0 < \mathcal{R}_1$ and $\mathcal{R}_0 < \mathcal{R}_2$, and (ii) $\mathcal{R}_1$ and $\mathcal{R}_2$ differ only in updates to X_{jk}.

Intuitively, a replica that is oblivious to updates on e_{jk} does not keep track of updates to registers in X_{jk} by replica j.

Theorem 1. *Consider a partially replicated shared memory system that implements replica-centric causal consistency. Any replica i must not be oblivious to update on any edge $e \in E_i$, where E_i is the edge set in the timestamp graph of replica i.*

The proof of Theorem 1 is provided in Appendix A. Intuitively, the theorem states that replica i's timestamp needs to be dependent on the updates performed on edge e_{jk} for each $e_{jk} \in E_i$. For instance, a vector timestamp whose elements are indexed by edges in E_i, and count updates performed on the corresponding edges, satisfies the requirements in Theorem 1. Indeed, in Section 3.2 we present an algorithm that uses precisely such a timestamp, proving that the necessary condition in Theorem 1 is sufficient as well. Later in Section 4 we obtain a lower bound on the size of the timestamps in the unit of bits. The necessary condition of Theorem 1 does not provide a measure of the size of the timestamps, whereas Theorem 2 provides lower bound on the size.

3.2 Sufficiency of Tracking Edges in Timestamp Graph

We propose an algorithm for implementing causally consistent shared memory in this section. The algorithm is for peer-to-peer architecture where each client only issues operations to one corresponding replica. Recall that $G_i = (V_i, E_i)$ is the timestamp graph of replica i.

Timestamps: Each replica i maintains an *edge-indexed* vector timestamp τ_i that is indexed by the edges in E_i. For edge $e_{jk} \in E_i$, $\tau_i[e_{jk}]$ is an integer, initialized to 0.

Client's Algorithm:

(1) Upon *read* operation on register x: send $read(x)$ to the replica, wait for the value returned by the replica.
(2) Upon *write* operation on register x with value v: send $write(x, v)$ to the replica, wait for the acknowledgement from the replica.

Replica's Algorithm:

(1) Upon receiving a $read(x)$ request from the client: replica i responds with the value of the local copy of register x.
(2) Upon receiving a $write(x, v)$ request from the client: replica i performs the following operations atomically:
   (a) write v into the local copy of register x,
   (b) for each $e_{jk} \in E_i$, update timestamp τ_i as
   $$\tau_i[e_{jk}] := \begin{cases} \tau_i[e_{jk}] + 1, & \text{if } j = i \text{ and } x \in X_{ik}, \\ \tau_i[e_{jk}], & \text{otherwise} \end{cases}$$
   (c) send $update(i, \tau_i, x, v)$ message to each other replica $k \in V$ such that $x \in X_k$,
   (d) return *ack* to the client.
(3) Upon receiving a message $update(k, \tau_k, x, v)$ from replica k:
   replica i adds $update(k, \tau_k, x, v)$ to a local data structure named $pending_i$.
(4) For any $update(k, \tau_k, x, v) \in pending_i$, when $\tau_i[e_{ki}] = \tau_k[e_{ki}] - 1$ and $\tau_i[e_{ji}] \geq \tau_k[e_{ji}]$ for each $e_{ji} \in E_i \cap E_k$, $i \neq j \neq k$, replica i performs the following operations atomically:
   (a) writes value v to its local copy of register x,
   (b) for each $e \in E_i$, updates timestamp τ_i as
   $$\tau_i[e] := \begin{cases} \max(\tau_i[e], \tau_k[e]), & \text{for each } e \in E_i \cap E_k, \\ \tau_i[e], & \text{for each edge } e \in E_i - E_k \end{cases}$$
   (c) removes $update(k, \tau_k, x, v)$ from $pending_i$.

The proof for the correctness of the algorithm is provided in the full version [37]. Note that the timestamp used by the algorithm implies replica i is not oblivious to update on any edge $e_{jk} \in E_i$, indicating the necessary condition in Theorem 1 is also sufficient.

Intuition for correctness: Our algorithm is similar to standard causal multicast algorithms [5]. The novelty of our algorithm lies in the *edge-indexed* vector timestamp, which contains a counter for each edge in the timestamp graph of the replica. Intuitively, keeping track of edges incident at i ensures FIFO delivery of update messages to/from i, and keeping track of the other edges in E_i guarantees that causal dependencies are carried when a chain of causally dependent update messages are propagated along a cycle. Although maintaining counters for all the edges in cycles for the second part is sufficient, it is not always necessary – our (i, e_{jk})-loop characterizes precisely which subset of edges in the cycle is necessary and sufficient for maintaining causal consistency.

Optimizations: We briefly discuss some mechanisms to reduce the timestamp size (details in the full version [37]). (1) *Timestamp Compression:* We observe that, in our algorithm, the different elements of the vector τ_i at replica i are not necessarily independent.

For instance, suppose that $e_{j1}, e_{j2}, e_{j3}, e_{j4} \in E_i$ for some $j \neq i$, with $X_{j1} = \{x\}$, $X_{j2} = \{y\}$, $X_{j3} = \{z\}$ and $X_{j4} = \{x, y, z\}$. Observe that the number of updates performed to registers corresponding to these four edges is *not independent*. Thus, it is possible to compress the timestamp to reduce its space requirement. (2) *Allowing False Dependencies:* A false dependency occurs when the application of an update u_1 is delayed at some replica, waiting for some update u_2 to be applied, even though $u_2 \not\hookrightarrow u_1$. We can introduce a "dummy" copy of some register at replicas to change the share graph and thus reduce the timestamp size possibly, but at the cost of extra update messages. (3) *Restricting Inter-Replica Communication Patterns:* It is known that restricted communication graphs can allow dependency tracking with a lower overhead [7, 21, 28] in the message-passing context. A similar observation applies in the case of partial replication too.

3.3 Relation to previous results

Our results make an improvement over previous results [16] regarding the timestamp size. Hélary and Milani [16] identify a larger set of edges (compared to E_i) that replica i needs to "track", however their result, although sufficient, does not always yield the necessary set of edges to track. Note that Hélary and Milani's results [16] consider the client-centric causal consistency. As mentioned when introducing the replica-centric causal consistency, our necessary conditions presented in Section 3.1 and 4 applies to their settings. The definition of the minimal x-hoop in [16, 30] states the following.

Definition 9 (Hoop [16, 30]). *Given a register x and two replicas r_a and r_b in $C(x)$ where $C(x)$ is the set of the replicas that stores x, we say that there is a x-hoop between r_a and r_b, if there exists a path $(r_a = r_0, r_1, ..., r_k = r_b)$ in the share graph G such that: i) $r_h \notin C(x)$ $(1 \leq h \leq k-1)$ and ii) each consecutive pair (r_{h-1}, r_h) shares a register x_h such that $x_h \neq x$ $(1 \leq h \leq k)$*

Definition 10 (Minimal Hoop [16, 30]). *An x-hoop $(r_a = r_0, r_1, ..., r_k = r_b)$ is said to be minimal, if and only if i) each edge of the hoop can be labelled with a different register and ii) none of the edge label is shared by replica r_a and r_b.*

The following result in [16, 30] intended to be a tight condition for achieving causal consistency.

Lemma 1 ([16, 30]). *A replica has to transmit some information about a register x if and only if the replica stores x or belongs to a "minimal x-hoop"*

As we show with an example now, the condition in Lemma 1 from [16, 30] is not, in fact, tight [31].

Consider the share graph in Figure 6 (we omit the direction of each edge in the figure for brevity). In the figure, the label on edges e_{uv}, e_{vu} shows set X_{uv}. The share graph consists of replicas $i, a_1, a_2, k, j, b_1, b_2$. Replicas j and k share register x, replicas b_1, b_2, a_1 share register y, and replicas b_2, a_1, a_2 share register z. Labels on other edges are unique and distinct from x, y, z.

The loop $(j, b_1, b_2, i, a_1, a_2, k)$ is considered a "minimal x-hoop" by Definition 10 from [16, 30] because (i) the label on each edge in the loop is distinct, (ii) none of the edge labels is shared by replica j and replica k. The result in Hélary and Milani [16, 30] implies that replica i must transmit (or keep) information about updates to register x by replicas j, k. However, it can be shown that presence of the two edges labeled y (and the manner they are situated) makes it unnecessary for replica i to be aware of updates to register x issued by replica j.

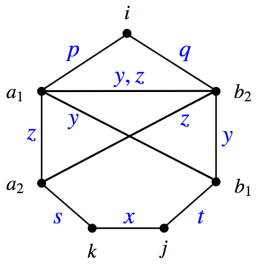


Figure 6: Example

For instance, consider the execution where there is an update by replica j on x and then a sequence of causally dependent updates propagating along the hoop $j, b_1, b_2, i, a_1, a_2, k$ on registers t, y, q, p, z, s respectively. Since the update by replica i on register p is also causally dependent on the update issued by replica b_1 on register y, replica a_1 will apply the update on y before the update on p. Since the update on y already record the dependency of the update on x, there is no need for replica i to be aware of updates to x issued by replica j. More details can be found in the correctness proof of the algorithm. Similarly, replica i does not need to transmit information regarding updates to x issued by replica k. Our necessary condition (Theorem 1) does not require replica i to keep track of these updates.

In general, our definition of the timestamp graph (Definition 7) identifies a necessary and sufficient set of edges for each replica which is a subset of the edge set identified in [16].

4 LOWER BOUND ON TIMESTAMP SIZE

Section 3.1 obtained a necessary condition on the timestamps assigned to the replica. In this section, we obtain a lower bound (in bits) on the size of the timestamps. From the definition of the causal dependency graph it should be apparent that two different causal dependency graphs may possibly correspond to the same *causal past* (i.e., set S in Definition 8). In order to derive the lower bound, we impose the following constraint[2].

Constraint 1. *For any replica i, its timestamp at any given time is a function of its causal past at that time.*

Definition 11. **Timestamp space size $\sigma^i(m)$ of replica i under Constraint 1:** *Consider the set of executions $\mathcal{E}$ in which each replica issues up to m updates. The timestamp space size of replica i under Constraint 1, denoted as $\sigma^i(m)$, is the minimum number of distinct timestamps that replica i must assign over all the executions in $\mathcal{E}$.*

Note that replica i may not use all the distinct $\sigma^i(m)$ timestamps in the same execution. However, over all possible executions, replica i will need to use at least $\sigma^i(m)$ distinct timestamps.

Let S be a causal past, which is a set of updates as per Definition 8. Recall that $G = (V, E)$ denotes the share graph. For $e_{jk} \in E$, let $S|_{e_{jk}}$ denote the set of updates in S that are issued by replica j on registers in X_{jk}. For $e_{jk} \notin E$, define $S|_{e_{jk}} = \emptyset$ for convenience.

Definition 12 (Conflict). *Given share graph $G = (V, E)$, and two possible causal pasts S_1, S_2 of replica i from executions in $\mathcal{E}$, we say that S_1 and S_2 conflict if following conditions hold:*

[2]Note that our proposed algorithm in Section 3.2 actually satisfies this constraint.

(1) $\forall e \in E, S_1|_e \neq \emptyset \neq S_2|_e$, *and*
(2) $\exists e \in E$ *such that* $S_1|_e \subset S_2|_e$, *where*
  (a) $e = e_{ij}$ *or*
  (b) $e = e_{ji}$ *or*
  (c) $\exists$ *a simple loop* $(i, l_1, \cdots, l_s, r_1, \cdots, r_t, i = r_{t+1}) \in G$ *where* $e = e_{r_1 l_s}$ *such that*
    (1) $S_1|_{e_{r_p l_q}} = S_2|_{e_{r_p l_q}}$ *for* $1 \leq p \leq t+1, 1 \leq q \leq s$ *and* $e_{r_p l_q} \neq e_{r_1 l_s}$, *and*
    (2) $S_x|_{e_{r_p r_{p+1}}} - \bigcup_{1 \leq q \leq s} S_x|_{e_{r_p l_q}} \neq \emptyset$ *for* $1 \leq p \leq t$ *and* $x = 1, 2$

Explanation: Condition (1) means that both causal pasts S_1, S_2 have at least one update on every edge in the share graph, which allows us to construct executions where some replica i's causal past can equal to S_1 or S_2. More specifically, using this property we can construct executions where the updates in S_1 (or S_2) are issued and propagated via a spanning tree rooted at replica i in the share graph, thus leading to causal past S_1 (or S_2) in replica i. Condition (2) means that the set of updates in S_1 on some edge is a strict subset of those in S_2 on the same edge. As we will show in the proof, the set difference above ensures that replica must distinguish S_1 from S_2, otherwise causal consistency may be violated. Three kinds of edges are listed in the definition (see Figure 7), (a) outgoing edges of replica i, (b) incoming edges of replica i, and (c) edges $e_{r_1 l_s}$ that are in a loop which satisfies two conditions stated in the definition: (1) the set of updates on any "chord edges" in the loop except $e_{r_1 l_s}$ are identical for S_1 and S_2, and (2) for both causal pasts S_1 and S_2, for replica r_p in $r_1, ..., r_t$, there exists some update sent to r_{p+1} that is not sent to any of $l_1, ..., l_s$. All conditions above will be used in the proof of Lemma 2.

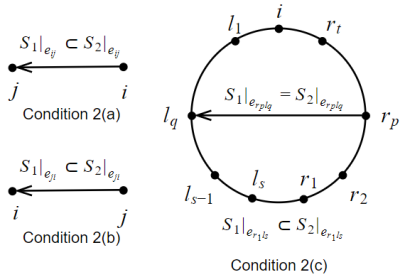


Figure 7: Condition 2 of Definition 12

Lemma 2. *Consider two possible causal pasts* S_1, S_2 *of replica* i. *If* S_1 *and* S_2 *conflict, then distinct timestamps must be assigned to them for ensuring the safety and liveness properties in Definition 5.*

Proof Sketch. The proof is presented in the full version [37]. Here we give some intuition of the proof. First we create two executions $\mathcal{E}_1$ and $\mathcal{E}_2$, after which replica i has causal past S_1 and S_2 respectively. The executions need to be created carefully such that they can be extended later to derive a contradiction as follows. If S_1 and S_2 conflict, but are assigned with the same timestamp, then replica i cannot distinguish whether it has causal past S_1 in $\mathcal{E}_1$ or S_2 in $\mathcal{E}_2$. Note that S_1 and S_2 differs in updates on some edge e. Suppose the difference is the update set U and $e = e_{jk}$. Then we can carefully extended the executions $\mathcal{E}_1$ and $\mathcal{E}_2$ such that replica k with identical local timestamps T_k receives an update u also with identical timestamps t_u in both extensions, and in one extension u is causally dependent on updates in U while in another it is not. Then replica k cannot distinguish between the two executions, and hence may violate either safety or liveness for causal consistency. □

Once we know all the pairs of conflicting causal pasts of a replica, we can easily derive the lower bound for the timestamp space size of that replica. For replica i, we define a *conflict graph* H_i with vertex set equal to the set of all possible causal pasts of replica i. An edge is added between any two causal pasts of replica i that conflict. Then, the *chromatic number* for the conflict graph is a lower bound on timestamp space size. Therefore, we have the following theorem.

Theorem 2. *Consider a partially replicated shared memory system that implements replica-centric causal consistency using an algorithm under Constraint 1. Let* $\chi(H_i)$ *denote the chromatic number of conflict graph* H_i. *Then,* $\sigma^i(m) \geq \chi(H_i)$ *for any replica* i.

Implication: Although our result does not explicitly imply a closed-form lower bound for the timestamp sizes, it can be shown that in several cases the lower bound has closed form and is tight.

- For instance, if the share graph is a tree, the timestamp lower bound is $2N_i \log m$ bits for replica i, where N_i is the number of i's neighbors in the share graph and m is the maximum number of updates that i will issue in the execution.
- When the share graph is a cycle of n replicas, the timestamp size for each replica has lower bound $2n \log m$ bits. Note that the timestamp sizes are tight in the above examples, since our algorithm will use timestamps of these sizes.
- In the case of *full replication* where the share graph is a clique and each edge shares identical set of registers, the above theorem implies the lower bound of the timestamp space size to be m^R where R is the total number of replicas. This lower bound is also tight, because the traditional vector timestamps satisfy this bound (similar to the timestamps used by Lazy Replication [22] when applied to the *peer-to-peer* architecture in Figure 1a).

One may relate the m^R lower bound for *full replication* to the classic lower bound on the vector clock size obtained by Charron-Bost [8] for determining happened-before relation in a message passing system. Although two bounds equal for full replication, however, it is not true in general for partial replication. *Timestamps for deciding happened-before relation between events cannot be directly used for maintaining causal consistency and vice versa.* One reason is that to achieve causal consistency, the timestamps should reflect information about whether there is any causally dependent update missing, but not false dependencies. Another reason is that the happened-before relation may have to be determined between any two events, while to achieve causal consistency, only for updates received by the same replica we need to determine the happened-before relation. As a result, our previous necessary and sufficient condition on the timestamps from Section 3.1 and 3.2 implies that the vector clock should have size equal to the number of edges in the timestamp graph (Definition 7), which may be larger than, smaller than or equal to n depending on the share graph.

5 EXTENDING RESULTS TO THE CLIENT-SERVER ARCHITECTURE

The results presented for the peer-to-peer architecture in Section 3 can be extended to the client-server architecture. The system model

of client-server architecture is illustrated in Figure 1b. There are C clients numbered 1 through C. Each client i is associated with an arbitrary subset of replicas R_i. Client i is restricted to perform read/write operations on registers in $\cup_{r \in R_i} X_r$.

Several natural extensions of the previous definitions are introduced in the full version [37] to obtain the results for the client-server architecture: (a) The algorithm is extended by taking into account the fact that a client may propagate dependencies across two replicas. In particular, in the client-server architecture, a client also needs to maintain a timestamp locally, and the timestamp will be included with the request to the replicas. (b) The share graph is augmented as shown below with additional edges that capture the causal dependencies propagation across the replicas due to the client accessing multiple replicas. (c) The definitions of (i, e_{jk})-loop and timestamp graph can then be suitably modified to apply to the client-server architecture.

Below we only present the definition of augmented share graph. Full details of the modifications for the client-server architecture can be found in the full version [37]. Recall that E is the set of edges in the share graph defined previously in Section 2.

Definition 13 (**Augmented Share Graph**). *Augmented share graph $\widehat{G}$ consists of vertices in $V = \{1, \cdots, R\}$ and directed edges in $\widehat{E} = E \cup \{e_{jk} \mid \exists$ client c such that $j, k \in R_c\}$.*

For replica j, k such that $X_{jk} = \emptyset$, there is no edge in E. However, if there exists client c such that $j, k \in R_c$, then directed edges between j and k exist in $\widehat{E}$.

Using the augmented share graph, we can obtain a necessary condition similar to Theorem 1, and an algorithm similar to that in Section 3.2, showing that the condition is also sufficient for achieving causal consistency in the client-server architecture.

6 RELATED WORK

Some of the relevant work is already discussed in Section 1, therefore, it is not included here.

Causal group communication: Several protocols [5, 20, 32] have been proposed for implementing causal group multicast with overlapping groups, and a simulation-based evaluation on causal group multicast is presented in [18]. Kshemkalyani [20] studied a causal group multicast protocol wherein each message M is piggybacked with metadata consisting of the list of messages that happened-before M and their corresponding destinations. They investigated the necessary and sufficient conditions on the destination information tracked in this piggybacked metadata. As a result, their algorithm can remove redundant information in the metadata at run-time. However, compared to our work, their result assumes a particular structure of the metadata, and the conditions do not express how the overlapping groups (or how replicas share registers in the context of shared memory) affect the size of the metadata. To the best of our knowledge, lower bound for metadata size required for causality tracking with *overlapping* multicast groups is not previously obtained.

Algorithms for message passing: The prior work on timestamps for capturing causality in message-passing is relevant here, in particular, several approaches for reducing timestamp size by exploiting communication topology information [21, 28, 34]. Charron-Bost proved the minimum size of the vector clock is the number of the processes in the message passing system in order to capture causality [8]. Lower bounds on non-structured timestamps for capturing causal dependencies between events have been studied previously [29], but the results do not directly apply to our problem setting. First, the events that satisfy happened-before relation in the message passing system may be false dependencies in our partial replication setting, if the event (or update) is sent to some different replica. Second, maintaining causal consistency only concerns the causality of the updates received by the same replica, not any pair of events as in message passing system in the previous works.

Algorithms for causal consistency: Hélary and Milani identified the difficulty of efficient implementation under causal consistency for partial replication [16, 30]. As discussed earlier, our work improves on the results of Hélary and Milani. Milani has systematically studied mechanisms to implement causal consistency, and presented a propagation-based protocol for partial replication [3]. Raynal [33] and Birman [5] studied protocols for implementing partially replicated causal objects, with an architecture similar to that in Figure 1a, but the size of the metadata is $O(mn)$ in general, where n is the number of replicas and m is the number of objects. Shen et al. [36] proposed two algorithms, *Full-Track* and *Opt-Track*, to achieve causal consistency for partial replication under relation $\rightarrow_{co}$ proposed by Milani [3]. Based on the previous work [20], their algorithm *Full-Track* carries metadata of size $O(n^2)$ and *Opt-Track* carries metadata of optimal size assuming a particular metadata structure. Kshemkalyani and Hsu's work on approximate causal consistency sacrifices accuracy of causal consistencies to reduce the meta-data [17, 19].

In a somewhat different line of research, concurrent timestamp systems for shared memory, which enable processes to order operations using bounded timestamps have been explored [10, 13, 15]; the problem addressed in our work is distinct from this prior work.

7 SUMMARY

This paper investigates partially replicated causally consistent shared memory systems. We present a tight necessary and sufficient condition on the replica timestamp and a lower bound on the size of the timestamps for implementing replica-centric causal consistency in a partially replicated system.

REFERENCES

[1] Sérgio Almeida, João Leitão, and Luís Rodrigues. 2013. ChainReaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 85–98.

[2] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2012. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 22.

[3] R. Baldoni, A. Milani, and S. T. Piergiovanni. 2006. Optimal propagation-based protocols implementing causal memories. *Distributed Computing* 18 (2006), 461–474.

[4] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. 2006. PRACTI replication. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation-Volume 3*. USENIX Association, 5–5.

[5] K. Birman, A. Schiper, and P. Stephenson. 1991. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)* 9, 3 (1991), 272–314.

[6] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2015. Towards a scalable, distributed metadata service for causal consistency under partial geo-replication. In *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*. ACM, 5.

[7] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. 2017. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 111–126.

[8] Bernadette Charron-Bost. 1991. Concerning the size of logical clocks in distributed systems. *Inform. Process. Lett.* 39, 1 (1991), 11–16.

[9] Tyler Crain and Marc Shapiro. 2015. Designing a causally consistent protocol for geo-distributed partial replication. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM, 6.

[10] Danny Dolev and Nir Shavit. 1997. Bounded concurrent time-stamping. *SIAM J. Comput.* 26, 2 (1997), 418–455.

[11] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. 2013. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 11.

[12] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.

[13] Cynthia Dwork and Orli Waarts. 1992. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible!. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*. ACM, 655–666.

[14] C. J. Fidge. 1987. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*.

[15] S. Haldar and P. Vitányi. 2002. Bounded concurrent timestamp systems using vector clocks. *Journal of the ACM (JACM)* 49, 1 (2002), 101–126.

[16] Jean-Michel Hélary and Alessia Milani. 2006. About the efficiency of partial replication to implement distributed shared memory. In *2006 International Conference on Parallel Processing (ICPP'06)*. IEEE, 263–270.

[17] T. Hsu and A. Kshemkalyani. 2016. Performance of Approximate Causal Consistency for Partially Replicated Systems. In *Proceedings of the Third International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. ACM, 7–13.

[18] Michael H Kalantar and Kenneth P Birman. 1999. Causally ordered multicast: the conservative approach. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No. 99CB37003)*. IEEE, 36–44.

[19] Ajay D Kshemkalyani and Ta-yuan Hsu. 2015. Approximate causal consistency for partially replicated geo-replicated cloud storage. In *Proceedings of the Fifth International Workshop on Network-Aware Data Management*. ACM, 3.

[20] Ajay D Kshemkalyani and Mukesh Singhal. 1998. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing* 11, 2 (1998), 91–111.

[21] Sandeep S Kulkarni and Nitin H Vaidya. 2017. Effectiveness of Delaying Timestamp Computation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 263–272.

[22] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst.* 10 (1992), 360–391.

[23] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

[24] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 401–416.

[25] Tariq Mahmood, Shankaranarayanan Puzhavakath Narayanan, Sanjay Rao, TN Vijaykumar, and Mithuna Thottethodi. 2016. Achieving causal consistency under partial replication for geo-distributed cloud storage. (2016).

[26] F. Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Workshop on Parallel and Distributed Algorithms*.

[27] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 453–468.

[28] Sigurd Meldal, Sriram Sankar, and James Vera. 1991. Exploiting locality in maintaining potential causality. In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*. ACM, 231–239.

[29] G. Melideo. 2001. *Tracking Causality in Distributed Computations*. Ph.D. Dissertation.

[30] A. Milani. 2006. *Causal Consistency in Static and Dynamic Distributed Systems*. Ph.D. Dissertation. Università di Roma.

[31] Alessia Milani. 2019. personal communication.

[32] Achour Mostefaoui and Michel Raynal. 1993. Causal multicasts in overlapping groups: Towards a low cost approach. In *Distributed Computing Systems, 1993., Proceedings of the Fourth Workshop on Future Trends of*. IEEE, 136–142.

[33] Michel Raynal and Mustaque Ahamad. 1998. Exploiting write semantics in implementing partially replicated causal objects. In *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing-PDP'98-*. IEEE, 157–163.

[34] Luis ET Rodrigues and Paulo Verissimo. 1995. Causal separators for large-scale multicast communication. In *Proceedings of 15th International Conference on Distributed Computing Systems*. IEEE, 83–91.

[35] Mohammad Roohitavaf, Murat Demirbas, and Sandeep Kulkarni. 2017. Causalspartan: Causal consistency for distributed data stores using hybrid logical clocks. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 184–193.

[36] Min Shen, Ajay D Kshemkalyani, and Ta-Yuan Hsu. 2015. Causal consistency for geo-replicated cloud storage under partial replication. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 509–518.

[37] Zhuolun Xiang and Nitin H Vaidya. 2017. Lower bounds and algorithm for partially replicated causally consistent shared memory. *arXiv preprint arXiv:1703.05424* (2017).

[38] Zhuolun Xiang and Nitin H Vaidya. 2018. Brief Announcement: Partially Replicated Causally Consistent Shared Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*. ACM, 273–275.

[39] Marek Zawirski, Nuno Preguiça, Sérgio Duarte, Annette Bieniusa, Valter Balegas, and Marc Shapiro. 2015. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*. ACM, 75–87.

ACKNOWLEDGEMENTS

The authors thank Alessia Milani for her feedback. The authors also thank the anonymous reviewers for the in-depth review and helpful suggestions.

Appendix

A PROOF OF THEOREM 1

We prove Theorem 1 by showing that either safety or liveness property in Definition 5 will be violated if replica i is oblivious to update on any edge $e_{jk} \in E_i$. Consider an execution $\mathcal{E}$ in which all issued updates have been applied at the relevant replicas, and replica i's causal dependency graph is $\mathcal{R}$. This can happen since the system satisfies liveness property of the replica-centric causal consistency and all updates are applied within a finite time. We will now extend the execution to show contradictions. *In the following extended executions, suppose any other message that is not explicitly mentioned is delayed indefinitely.* This is possible since the system is asynchronous. From Definition 7 of edge set E_i, there are three possible types of edges in E_i, as in the following three cases.

Case 1: $e = e_{ij} \in E_i$**:**

Let $\mathcal{E}_1$ be an extended execution where replica i issues update u_1 on edge e_{ij} (i.e., for a register in X_{ij}) after $\mathcal{E}$. Suppose that the causal dependency graph of i after issuing u_1 is $\mathcal{R}_1$. Let $\mathcal{E}_2$ be an extended execution where replica i issues update u_2 on edge e_{ij} after $\mathcal{E}_1$, and let $\mathcal{R}_2$ be the causal dependency graph of i after issuing u_2.

Since $\mathcal{R}_1$ and $\mathcal{R}_2$ only differ in update u_2 on edge $e = e_{ij}$, and replica i is oblivious to update on e_{ij}, the timestamp attached to u_1, u_2 that sent to j will be identical. Thus, replica j cannot determine the correct order in which to these two updates were sent (recall that the channel is not FIFO). Thus, causal consistency cannot be assured.

Case 2: $e = e_{ji} \in E_i$**:**

Let $\mathcal{E}_1$ be an extended execution where replica j issues update u_1 on edge e_{ji} (i.e., for a register in X_{ij}) after $\mathcal{E}$, but u_1 is not yet applied at replica i. Let the causal dependency graph of i before applying u_1 be $\mathcal{R}_1$. Let $\mathcal{E}_2$ be an extended execution where replica j issues update u_2 on edge e_{ji} after $\mathcal{E}_1$, and suppose that u_1 is applied at replica i but not u_2. Let the new causal dependency graph of i be $\mathcal{R}_2$.

Since $\mathcal{R}_1$ and $\mathcal{R}_2$ only differ in updates on edge $e = e_{ji}$, and replica i is oblivious to update on e_{ji}, replica i has identical timestamps after applying u_1 and before. Thus, when replica i receives update u_2, it cannot differentiate between the following two cases: (i) i has already received and applied update u_1, and thus, it can now apply update u_2. (ii) i has not yet received update u_1, so it must wait for that update message before applying u_2. If replica i applies u_2 when u_2 arrives (i.e., without waiting for another update message), but the situation is as in (ii), then safety requirement of causal consistency is violated. On the other hand, if replica i decides to wait, but the situation is as in (i), then another update may never be received from j, and liveness requirement of causal consistency is violated.

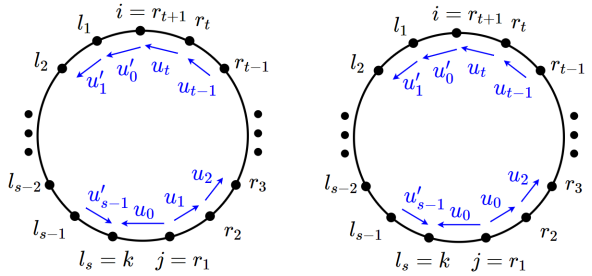
Case 3: $e = e_{jk} \in E_i$ **and there exists an** (i, e_{jk})**-loop** $(i, l_1, \cdots, l_s = k, j = r_1, \cdots, r_t, i)$**:**

By the definition of the (i, e_{jk})-loop, we have

(i) $X_{jk} - \left(\cup_{1 \le p \le s-1} X_{l_p}\right) \neq \emptyset$,

(ii) $X_{jr_2} - \left(\cup_{1 \le p \le s-1} X_{l_p}\right) \neq \emptyset$, and

(iii) for $2 \le q \le t$, $X_{r_q r_{q+1}} - \left(\cup_{1 \le p \le s} X_{l_p}\right) \neq \emptyset$.



(a) Illustration for Case 3.1

(b) Illustration for Case 3.2

Figure 8: Examples of Timestamp Graphs

Case 3.1: $X_{jr_2} - \left(\cup_{1 \le p \le s} X_{l_p}\right) \neq \emptyset$, that is, X_{jr_2} has a register w_1 that is not shared by any of replicas in $l_1, \cdots, l_s$.

Consider the following extension of $\mathcal{E}$ as the execution $\mathcal{E}_1$, as illustrated in Figure 8a.

- Initially, replica $r_1 = j$ issues an update u_0 on edge e_{jk} on register w_0, where $w_0 \in X_{jk} - \left(\cup_{1 \le p \le s-1} X_{l_p}\right)$, i.e. not shared by any replicas in $l_1, \cdots, l_{s-1}$. Such w_0 exists since $X_{jk} - \left(\cup_{1 \le p \le s-1} X_{l_p}\right) \neq \emptyset$. Thus, u_0 is sent to l_s but not any of $l_1, \cdots, l_{s-1}$.
- Replica $r_1 = j$ then issues update u_1 on edge $e_{r_1 r_2}$ on register $w_1 \in X_{jr_2} - \left(\cup_{1 \le p \le s} X_{l_p}\right)$, i.e. w_1 is not shared by any replicas in $l_1, \cdots, l_s$. Thus, u_1 is sent to r_2 but not any of $l_1, \cdots, l_s$. The corresponding update message is next received by r_2.
- For $p = 2$ to t: r_p receives an update message from r_{p-1} and applies the update. The update can be applied since all its causal dependencies have been applied. Then r_p issues an update on edge $e_{r_p r_{p+1}}$ on a register w_p that is not shared by any of $l_1, \cdots l_s$. Such w_p exists since for $2 \le q \le t$, $X_{r_q r_{q+1}} - \left(\cup_{1 \le p \le s} X_{l_p}\right) \neq \emptyset$. Let us call this update u_p. Thus, we have constructed a sequence of updates so far such that $u_0 \hookrightarrow u_1 \hookrightarrow u_2 \hookrightarrow \cdots \hookrightarrow u_t$, where $r_{t+1} = i$.
- Subsequently, i issues an update u'_0 on edge e_{il_1}. l_1 receives the update message, applies the update, and then issues an update u'_1 on edge $e_{l_1 l_2}$. The update can be applied since all its causal dependencies have been applied. Continuing in this manner, we build a sequence of updates such that $u'_0 \hookrightarrow u'_1 \hookrightarrow \cdots \hookrightarrow u'_{s-1}$, where update u'_p in this chain is issued by replica l_p on edge $e_{l_p l_{p+1}}$.
- Combining the two sequences of updates, we obtain the following sequence, $u_0 \hookrightarrow u_1 \hookrightarrow u_2 \hookrightarrow \cdots \hookrightarrow u_t \hookrightarrow u'_0 \hookrightarrow u'_1 \hookrightarrow u'_2 \hookrightarrow \cdots \hookrightarrow u'_{s-1}$.

Now consider an alternate extension of $\mathcal{E}$ as the execution $\mathcal{E}_2$ in which replica j does not initially perform update u_0, but the remaining sequence of updates above are performed. The timestamp of replica i when issuing update u'_0 will be identical in both executions, because the causal dependency graphs at i when issuing update u'_0 only differ by updates on edge e_{jk}.

By induction, we can easily show that the timestamp attached to the update u'_{s-1} received by l_s from l_{s-1} will be identical in both executions. In performing the induction, we make use of the assumption that any other message that is not explicitly mentioned in the above executions is delayed indefinitely, including those on edges in $\{e_{r_x l_y} | r_x l_y \neq r_1 l_s\}$. In the first execution $u_0 \hookrightarrow u'_{s-1}$, but this is not the case in the second execution. If the update message from r_1 to l_s is not delivered before l_s receives the update from l_{s-1}, then replica l_s cannot determine whether it should wait for an update from r_1 or not, and either safety or liveness condition may be violated.

Case 3.2: $X_{jr_2} - \left(\cup_{1 \le p \le s} X_{l_p}\right) = \emptyset$. Since by condition (ii), $X_{jr_2} - \left(\cup_{1 \le p \le s-1} X_{l_p}\right) \neq \emptyset$, $\exists w_1 \in X_{jr_2} \cap X_{jl_s} - \left(\cup_{1 \le p \le s-1} X_{l_p}\right)$, that is, X_{jr_2} has a register w_1 that is shared by l_s but not any of replicas in $l_1, \cdots, l_{s-1}$.

We build two extensions of $\mathcal{E}$, similar to Case 3.1. Figure 8b illustrates this case.

For the first execution, replica $r_1 = j$ issues an update u_0 on the register w_1. Since w_1 is also shared by r_2, u_0 will be also sent and delivered to r_2. Also, note that u_0 is sent to k, r_2 but not any of $l_1, \cdots, l_{s-1}$. Unlike Case 3.1, no other update is performed on edge $e_{r_1 r_2}$. The remaining sequence of updates is identical to Case 3.1. This results in the following happened-before relation.

$u_0 \hookrightarrow u_2 \hookrightarrow u_3 \hookrightarrow \cdots \hookrightarrow u_t \hookrightarrow u'_0 \hookrightarrow u'_1 \hookrightarrow \cdots \hookrightarrow u'_{s-1}$

For the second execution, replica r_1 does not issue update u_0, but the remaining sequence of updates are performed.

By similar argument as in Case 3.1, the timestamp attached to the update u'_{s-1} will be identical in both executions, and replica l_s cannot determine whether it should wait for an update from r_1 or not, and either safety or liveness condition may be violated.