Resilient Distributed Causal Memory in Client-Server Model

Lewis Tseng†
Boston College
lewis.tseng@bc.edu

Zezhi Wang*
Brown University
zezhi_wang@brown.edu

Yajie Zhao*
Columbia University
yz3231@columbia.edu

***Abstract*—We study distributed causal shared memory (or key-value pairs) in an *asynchronous* network under *crash* failures. Causal memory, introduced by Ahamad et al. in the context of multi-processor environment in 1994, is an abstraction which ensures that nodes agree on the relative ordering of read and write operations that are *causally related* on key-value pairs. Inspired by the recent interests in geo-replicated causal storage systems (e.g., COPS, Eiger, Bolt-on), we systematically study the fault-tolerance property of the causal shared memory in the *client-server model* in this work.**

We identify that $2f+1$ servers is both necessary and sufficient to build a resilient causal memory in the presence of up to f crashed servers. We provide both the necessity proof and a new optimal algorithm that matches the bound. For evaluation, we implement our algorithm in Golang and compare the performance with state-of-the-art fault-tolerant algorithms that ensure strong consistency in the Google Cloud Platform.

Keywords – distributed storage system, causal memory, evaluation, asynchrony, crash faults

I. Introduction

This paper considers the problem of implementing distributed shared memory (or shared key-value pairs) over *asynchronous* message-passing networks. Different from most prior theory works (e.g., [5], [4]), we adopt the client-server paradigm in which clients are the ones accessing the shared key-value pairs through read and write operations, and the servers are the ones that manage the data so that the desired consistency property is satisfied. No communication among clients is assumed. In this paper, we focus on providing the *causal consistency* [4], [35] in the presence of crash-prone nodes. In other words, we are interested in resilient (or crash-tolerant) causal shared memory.

Resilient shared memory has been extensively studied in both crash fault model (e.g., [5], [50], [28]), semi-Byzantine fault model (e.g., [44], [43], [20]), and Byzantine fault model (e.g., [31], [29]). Nonetheless, to the best of our knowledge, there is very few study on the fault-tolerance aspect of *causal memory* [4]. Especially, we are *not* aware of any work that studied the *tight resilience bound* of causal memory. Existing works either rely on consensus protocols (e.g., [39], [10]) or did not present tight result (e.g., [11], [52]). In this paper, we systematically study the fault-tolerance property of causal memory. In particular, we identify that it is both necessary and sufficient to have at least $2f+1$ servers when any clients and up to f servers may crash. The necessity proof is based on the well-known indistinguishability argument [24], [7]. For sufficiency, we constructively provide an optimal algorithm that matches the $2f+1$ bound.

On one hand, the identified resilience bound is *not* entirely surprising, since $2f+1$ is also the tight bound for resilient shared memory that provides stronger notion of consistency, e.g., atomicity [5], linearizability [28], or regularity [36]. On the other hand, we believe that our work is the first step towards understanding *resilient causal memory in the client-server model* more thoroughly. First, our results show that it is necessary to consider the *convergence* property for causal memory to derive a meaningful bound. This is different from other resilient shared memory with stronger consistency; hence, the tradeoff between convergence and resilience is a new topic to be explored in the context of shared memory. Second, our algorithm does not depend on other expensive primitive such as fault-tolerant consensus or broadcast, allowing a more efficient and simpler implementation in practice.

A. Motivation

Large-scale distributed storage systems are a critical infrastructure of many Internet services nowadays. Most user-facing Internet services are extremely sensitive to latency. It is observed that a slight increase in user-perceived latency results into significant revenues loss (e.g., [39], [23]). Hence, recent practical replicated storage systems (e.g., [1], [19], [38]) adopt weak consistency models to achieve higher performance (low latency and high throughput). Among the proposed weak consistency models, *causal consistency* is one of the more appropriate models, because it provides useful consistency guarantees for application developers [38], [39], [10], and is one of the strongest consistency models achievable [42] in network partition (or more precisely, under the framework of CAP theorem [15], [40]).

As the system grows larger, machine and hardware failures and soft errors (caused by defects, bugs, and/or configuration errors) are inevitable. Practical systems often rely on consensus protocols and state-machine replication to tolerate machine failures [38], [39], [10]. These systems replicate each

†This research is supported in part by National Science Foundation award CNS1816487. 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. *Work was done while the authors were with Boston College.

2473-3105/19/$31.00 ©2019 IEEE
DOI 10.1109/PRDC47002.2019.00035

machine using consensus so that a server (based on multiple machines) would be more resilient. However, such an approach is prohibitively expensive in terms of message complexity and end-to-end latency if one needs to tolerate many failures, since consensus protocols, e.g., Paxos [37], do not scale well. These observations motivate us to explore causal memory *without* using any expensive primitive such as consensus or atomic broadcast protocol.

B. Causal Consistency and Convergence

We briefly discuss the properties of causal memory. These properties will be formally defined in Section III. Causal memory provides read and write interfaces to access multiple key-value pairs that are stored and maintained at servers. Specifically, client c_i can invoke two operations:

- *write operation* $w_i(x)v$ that stores the value v onto the variable with key x.[1]
- *read operation* $r_i(x)v$ that reads from the variable with key x which later returns the value v to client c_i.

Causal memory provides causal consistency, which ensures that the values returned by read operations observe the *causality* (or happens-before relation [35]). That is, if a write operation $w(x)a$ causally precedes (or happens before) $w(x)b$, then the value returned by any client should respect the ordering. Recall that write operation w is said to causally precedes another write w' if (i) both writes are invoked by the same client and w is invoked first, (ii) a client reads the value written by w, then invokes w', or (iii) transitivity.

Causal consistency is found to be useful in many Internet services [38], [39], [10], because causality is natural for application developers to reason with. We will use social network examples for motivation. If an underlying storage system provides only eventual consistency, then it is possible that (i) after Alice creates a post, she might not see that post after refreshing the page several times; and (ii) Alice and Bob are commenting on Carol's profile picture, but Carol might see random subset of that conversation – which might appear in any order – and miss the logical implication between the comments. Causal memory ensures that these two scenarios would not occur, because a client will always observe events respecting causality. In these two examples, Alice will see her more recent post, and Carol will observe all comments that "happens-before" the newest comment she saw, and the comments will appear in a causal order. More importantly, causal memory can be implemented in a low-latency fashion, by not contacting a majority of nodes.

Roughly speaking, convergence defines the *eventual state* at each server. Original causal memory definition [4] does *not* require convergence to satisfy safeness. In practical systems, it is necessary to consider the convergence property to ensure correct application logic. This paper adopts the *persistence* property [11], [26], which requires that the value written by a completed write operation will *never* be lost if there is no other concurrent write.

[1]If i is clear from the context, then we may omit i in the notation.

C. Contributions

- In Section IV, we identify why prior solutions (e.g., [4]) can tolerate any number of failures in a multi-processor environment, whereas in the client-server model, such solutions do not work. In other words, we identify the need for new resilient causal memory algorithms.
- In Section V, we present an algorithm for causal memory under the case when up to f servers and any number of clients may crash. Our algorithm requires at least $2f+1$ servers for correctness.
- In Section VI, we prove that it is necessary to have $2f+1$ servers. This shows that $2f+1$ is the *tight* bound, i.e., the proposed algorithm is *optimal* in resilience.
- We implement our algorithm and compare its performance with ABD algorithm [5], the well-known crash-tolerant algorithm for atomic memory, and SBQ (Small Byzantine Quorum) algorithm [44], state-of-the-art algorithm providing safety (a form of strong consistency) when servers may suffer semi-Byzantine failures.

II. Related Work

There is a long history of study on Distributed Shared Memory (DSM). We only discuss the most relevant ones here. Resilience is well studied in quorum systems and DSMs ensuring atomicity, sequential consistency, regularity, and linearizability, e.g., [3], [5], [50], [2], [25], [32], [28]. These algorithms typically require clients to contact a quorum or even a majority of servers, and are *not* feasible if there is a network partition [40], [15]. Therefore, practitioners have strong interests in weaker consistency models. Among them, causal consistency is one of the more appropriate models, because it provides strong enough guarantees for application developers while ensuring low latency.

Most works on causal DSM were studied in the MCS (Memory-Consistency System) model (e.g., [30], [4], [12], [42], [34], [33]), which is significantly different from the client-server model in the presence of faults. James and Singh [30] pointed out that causal memory is $(n-1)$-resilient (or wait-free [27]) in the MCS model. We will explain in Sections IV and VI, respectively, why these algorithms are not correct, and why no $(n-1)$-resilient causal memory is possible in the client-server model with crash-prone nodes.

The closest works are [52] and [11], which also focus on resilient causal memory. SwiftCloud's [52] implementation also ensures some form of convergence and relies on a variation of reliable broadcast. Since SwiftCloud's focus is on client-side mergeable transactions, it relies on more complicated mechanisms such as causal broadcast between servers, exactly-once delivery and execution, global committing protocol, server reconnection protocol, etc. It is not clear if SwiftCloud can be adapted to our model in a straightforward way. Plus, there is no formal analysis on the tradeoff achieved by SwiftCloud in [52]. The final difference is that client (called scout in [52]) is always connected to a single server (called datacenter in [52]), whereas in our algorithm, a client can communicate with any server. Baldoni et al. [11] studied a slightly different problem,

resilient causal memory with dynamic servers, and presented a beautiful algorithm; however, their algorithm is *not* optimal in our model (the static setting). Their algorithm requires $3f$ servers, more than the optimal bound $2f + 1$ presented in the paper. Moreover, Baldoni et al. [11] did not consider the convergence property.

Recently, there is also a rich study on different aspects of causal memory, including various practical systems (e.g., [38], [39], [10], [22], [21], [14], [48]), partial replication (e.g., [47], [46], [17], [51]), and eventual property [8]. These works did not focus on the fault-tolerance aspect. Please refer to [49] for more details.

III. Preliminaries

A. System model

The client-server model consists of two types of nodes: (i) *server nodes* (or simply servers) that store data in the form of key-value pairs, and (ii) *client nodes* (or simply clients) which invoke operation to read and write the key-value pairs stored on the servers. The term "nodes" refer to both server and client nodes. If there is no failure, clients can communicate with any server, and servers can communicate among themselves. Clients do *not* communicate among themselves.

The asynchronous communication channel is modeled by a fair-loss point-to-point link [41], [11]. In the fair-loss channel, if the two ends are both fault-free, then the message can eventually be delivered; however, if one end becomes faulty, then there is *no* guarantee that a message will be delivered. This channel is both authenticated and reliable. That is, a correct node receives a message from another correct node if and only if the other correct node sent it. We do *not* assume known bounds on message transmission times. In other words, the communication is *asynchronous*.

The set of servers is static, and we do not assume the existence of a failure detector [16] or a membership service [18]. Consequently, in our system, faulty nodes are indistinguishable from slow nodes. The system consists of $n \ \ (n \geq 2)$ servers and $n_c \ \ (n_c \geq 2)$ clients, since Distributed Shared Memory (DSM) is trivial when $n = 1$ or $n_c = 1$. We will use $S = \{s_1, \cdots, s_n\}$ and $C = \{c_1, \cdots, c_{n_c}\}$ to denote the set of servers and clients, respectively. We assume that up to f servers and any number of clients may crash.

In the crash model, any clients and up to f servers may crash. If a node i crashes, it stops the execution possibly without noticed by other nodes, and other nodes may receive different set of messages from node i because of the assumption of a fair-loss point-to-point link and asynchronous delivery. If a node never fails throughout the execution, it is called a *fault-free* node; otherwise, it is said to be faulty.

B. Memory model and Consistency

Key-value pairs are replicated to each server. We assume that each write operation is univocally identifiable, and each client is sequential. As the object can be concurrently accessed by clients, the memory must guarantee consistency property that defines the semantics of the shared data, i.e., the value returned by each read operation. We adopt the causal consistency model in [4], which defines the consistency model in terms of histories. Let L_i denote the *local history* at client i, i.e., the sequence of read/write operations. A history H (of the system) is then the union of all local histories.

We now introduce the definition of the causal order (or happens-before relation) in DSM [4]. Let $\xrightarrow[\mathbf{CC}]{}$ represent the causal order induced by a history H and the corresponding reads-from order defined over H. For brevity, we ignore H in the notation. Formally, we say that $o_1 \xrightarrow[\mathbf{CC}]{} o_2$ if any of the following conditions holds:

- *Program-order*: $o_1 \xrightarrow{L_i} o_2$ for some c_i (o_1 precedes o_2 in the local history L_i),
- *Reads-from*: $o_1 \xrightarrow[\mathbf{read}]{} o_2$ (o_2 returns the value written by o_1), or
- *Transitivity*: there is some other operation o' such that $o_1 \xrightarrow[\mathbf{CC}]{} o' \xrightarrow[\mathbf{CC}]{} o_2$.

If $o_1 \xrightarrow[\mathbf{CC}]{} o_2$ in history H, we will say o_1 *happens before* o_2 in H. Two writes w_1 and w_2 are said to be *concurrent* if $w_1 \not\xrightarrow[\mathbf{CC}]{} w_2$ and $w_2 \not\xrightarrow[\mathbf{CC}]{} w_1$, and is denoted by $w_1 || w_2$.

Let $H|(c_i + W)$ denote the set of all operations in local history L_i and all write operations (by any client) in history H. Then, we define causal consistency in the presence of faults below. Note that it is a generalization of the original definition in the MCS model [4].

Definition 1. (Causal Consistency [4]) *A history H is* causally consistent *if there exists a* complete *history H' such that (i) H' completes H, and (ii) for each client c_i, there exists a serialization $\mathbb{S}_i$ of $H'|(c_i+W)$ that respects the causal order $\xrightarrow[\mathbf{CC}]{}$.*

Since we consider faulty clients, we need to consider only the *complete* history. Roughly speaking, a complete history is obtained from removing some of the incomplete operations and appending the response event to the other incomplete operations in any given history. There is a known technique to perform the transformation. Due to page limit, we present the details in our technical report [49]. In this paper, we assume that all histories are *complete*.

An algorithm is said to be *safe* if any history is causally consistent as per Definition 1, and satisfies *liveness* if in any history, *every* operation can be completed in finite amount of time given that the client does not crash and at most f servers crash. As also observed in [11], most prior work on causal memory only considered safety and liveness. However, for practical usages, we need to rule out a trivial solution that is not useful for most practical usages: returning initial value $\perp$ for each read and ignoring any write. Thus, we also need the *convergence* property.[2] We adopt persistence [11], [26] for convergence, which rules out the trivial solution.

[2]In the system literature, convergence property belongs to a subset of more general notion of "liveness." To focus on the tradeoff, we adopt the liveness property from traditional DSM literature (e.g., [6], [11], [30]), which only requires an operation to complete in finite amount of time.

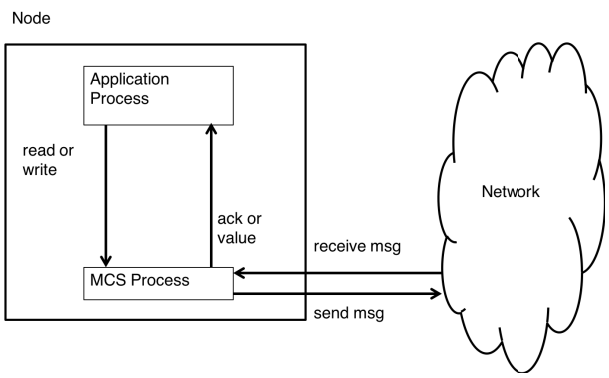


Fig. 1: Memory-consistency system model.

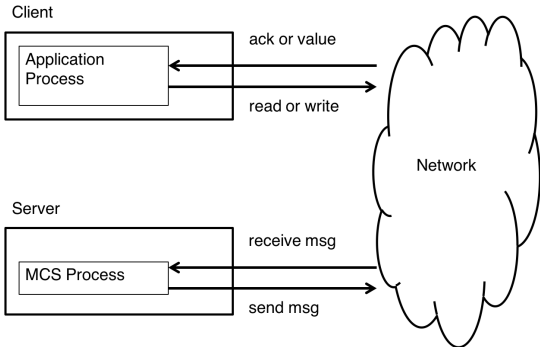


Fig. 2: Client-server model.

Definition 2 (Persistence of a value v)**.** *A value v of key k is persistent, if v is written into key x and there is no successive nor concurrent write operations, a client that reads infinitely many times on key x will eventually read v forever.*

Successive and concurrent writes are defined with respect to real time (or wall-clock time) of the event. Note that real time is only used for definitions. Each node does not have a notion of real time in our model, as typical in most prior works, e.g., [4], [5].

Definition 3 (Persistence)**.** *An algorithm satisfies persistence if for all the key-value pairs (k, v), every value written to key k is persistent.*

IV. Resilient Causal Memory: An Overview

In this section, we discuss why many prior causal memory implementations (e.g., [4], [34], [33]) are not resilient in the client-server model.

A. Causal Memory in MCS Model

Causal consistency was first proposed by Lamport [35] to order messages and events. Later, Ahamad et al. [4] integrated causal property with DSM, and studied it under the context of *multi-processor environment*, which is modeled as the Memory-Consistency System (MCS). In the MCS model, each node contains two processes: an application process that needs to access the key-value pairs, and an MCS process that is responsible to maintain the consistency guarantees through accessing local memory and communicating with other MCS processes via network. Pleaser refer to Figure 1 (adapted from a figure in [9]) for an illustration. One important feature of the MCS model is that if a node crashes, *both* application and MCS processes crash; hence, one does not need to worry about the application reading inconsistent data due to failures of the MCS process. It is shown in [30] that Ahamad's causal memory implementation [4] can tolerate any number of node failures in the MCS model.

B. Causal Memory in Client-Server Model

Recently there is a renewed interest in implementing causal memory in geo-replicated storage systems (e.g., COPS, Eiger, Bolt-on). The more appropriate model for these geo-replicated stores is the *client-server model* where servers posses all the key-value pairs, and clients only store values that they have recently queried. Typically, clients and servers usually reside on different physical machines in these storage systems. Please refer to Figure 2 for an illustration.

Resilience is much harder to achieve in the client-server model than in the MCS model. Consider the following example: a client c reads some value v for key k from server p, and later p crashes, then the system must ensure that the value v is stored at some place other than server p; otherwise, either client c cannot complete a new read operation on key k, violating liveness (i.e., the operation cannot terminate), or client c reads some (old) value from other servers that violates causal consistency. A similar dilemma also occurs when we allow clients to cache recent data. Our lower bound proof is a generalization of this observation, which is presented in Section VI.

The main idea of consensus-based implementations in prior causal storage system [39], [10] is using a cluster of machines to jointly act as the server; hence, no "server" will ever crash in their systems. In this work, we do not use consensus to replicate the data across machines. Each server is executed on one (virtual) machine, and may suffer crash failure. This design choice significantly reduces the coordination overhead.

V. Our Algorithm: RCM

One solution is to augment algorithm by Ahamad et al. [4] so that it is fault-tolerant in the client-server model. The trick is to use the reliable broadcast primitive [13] for *all* the server-server and client-server communication. The solution is possible, because $n \geq 2f + 1$ is sufficient for implementing reliable broadcast on top of fair-loss links [13], [45]. However, such a simple solution incurs high message complexity, since to deliver one message, at least $f+1$ servers need to be involved. The main contribution here is to devise a cheap and simple algorithm. Specifically, our algorithm has lower message complexity and lower latency than both the augmented algorithm and ABD [5]. Note that ABD provides atomicity, which is stronger than causal consistency.

We first describe a key element on tracking dependency (of the happens-before relation). Then we present our algorithm RCM (Resilient Causal Memory) followed by its correctness proof.

A. Dependency Tracking

Dependency tracking is the key to implement causal memory, e.g., [35], [4], [34], [33]. RCM uses vector timestamp [35] of size n_c to keep track of potential causal ordering, where n_c is the number of clients. Having a vector timestamp of size n_c might seem prohibitively expensive for a system with large number of clients. In the context of geo-replicated systems, n_c is usually moderate, because "clients" actually are proxies or coordinators within the same system. Hence, the number of clients are typically small in this scenario – in most cases, $n_c \leq 10$ (e.g., [1], [19]). It is left as an interesting future work to reduce the size of the vector timestamp.

Each vector timestamp is a vector of n_c entries: $t = \langle t[1], \quad \cdots \quad , t[n_c] \rangle$. Each entry is a non-negative integer that represents to the number of *writes* invoked and completed by each client. Two timestamps t and t' are partially ordered naturally: $t \leq t'$ if for all $1 \leq i \leq n_c, \quad t[i] \leq t'[i]$. A careful usage of the vector timestamp ensures that no *stale* value (with respect to the causal ordering) can be read by a client. A client uses the following function to merge its local timestamp and the timestamp learned from servers. For two timestamps t and t', $\text{MERGE}(t, t') = \langle \max\{t[1], t'[1]\}, \cdots, \max\{t[n_c], t'[n_c]\} \rangle$.

B. RCM (Resilient Causal Memory)

The pseudo-code of RCM is presented in Algorithm 1 (for server) and Algorithm 2 (for client). The algorithm uses two communication primitives: *send* and *multicast*. Send is simply a unicast to a specific receiver. Multicast sends a message to each server. If the sender is a server, multicast will send the message to itself as well. Recall that we assume a fair-loss channel; hence, if both sender and the intended receiver is fault-free, then the message will eventually be delivered. However, if one end fails, then there is *no* guarantee that the message is delivered. Furthermore, a message may be delivered out of order.

1) Variables: Each server j keeps an array M in the local memory, and $M[k]$ stores the value corresponding to key k. Server j also stores incoming update messages in a message queue Q. Additionally, each server has a vector timestamp t_s^j of size n_c. When the identity of j is clear, we often neglect j in the notation of timestamp for simplicity.

Initially, the timestamp has 0 in each entry, and all the variables $M[k]$ are set to $\perp$ at the servers. For each write, we also use an integer variable $\texttt{witness}(*, *)$, which is initialized to 0, to keep track of how many servers have seen this write request so far. The main purpose is to ensure that enough servers have seen a request so that the request can be safely written to the local memory. Essentially, this part resembles reliable broadcast, and ensures that whenever a server applies the write request, eventually all other fault-free servers will apply the write request too.

Each client i has its local vector timestamp t_c^i to keep track of dependency. We will use t_c if i is clear from the context. Client i also has a non-negative integer `counter` that is used to count the number of local operations at client i. The purpose of `counter` is to extract the information from the response messages corresponding to the `counter`-th operation. Finally, client i has two buffers `resp` and `writeBuf` that store the read responses and write acknowledgments from the servers, respectively.[3]

2) Algorithm Flow: The algorithm is event-driven as prior algorithms [4], [34], [33]. Servers are expecting three types of messages: message with read and write operations from the client and message with an update operation from other servers. RECEIVING(*) function specifies the steps to take when receiving the corresponding messages. When receiving a read request, a server waits until it is *safe* to return the value from its local storage (line 1 to line 3 in Algorithm 1). When receiving a write request w, a server has to wait until it applies w to its local storage through a lazy update mechanism, and all the writes that happen before w (line 4 to line 7 in Algorithm 1). The server compares the vector timestamps to check whether these conditions have been met. Line 8 to line 15 in Algorithm 1 implement a variation of reliable broadcast by forwarding messages and collecting enough witnesses.

UPDATE(*) function is used to apply the write operation to local storage and update local timestamp t_s and is executed *infinitely often*. Moreover, it is also executed in parallel to address the possibilities that messages might be added to the Q in an arbitrary order. This part is similar to the lazy update mechanism in prior works [4], [34], [33]. The key innovation here is to integrate the update function with the codes in line 8 to line 15 in Algorithm 1, which ensures the following properties when a message m is added to Q at server j: (i) m will eventually be applied at server j if j has not crashed, and (ii) other fault-free servers will eventually add m to Q. This is because that we require $\texttt{witness}(*, *) \geq f + 1$, which means that at least one fault-free server has "witnessed" the message, and will eventually deliver this message to other fault-free servers.

Clients provide read and write functions that specify how to interact with the servers and how to update the local timestamp t_c. As discussed earlier, clients use MERGE to merge timestamps from servers and its own local timestamp. RECEIVING(*) function is also event-driven, which handles messages received from servers.

C. Correctness

We first briefly discuss the intuition behind the proof. Causal consistency follows from the usage of vector timestamp. This part of the proof is inspired by the proof in the original causal memory paper [4]. Our proof is more involved, since a client's local history is essentially a combination of different "views"

[3]These two buffers could be eliminated in pseudo-code; however, we choose the event-driven presentation to mimic a common practical implementation which has a separate process that handles the receiving messages in the background.

Algorithm 1 RCM for server j

1: **function** RECEIVING(READ, $k, i, \texttt{counter}_i, t_i$)
2: **wait until** $t_s^j[l] \geq t_i[l], \quad \forall l$
3: Send $(\text{RESP}, \texttt{counter}_i, M[k], t_s^j)$ to client i
4: **function** RECEIVING(WRITE, $k, v, i, \texttt{counter}_i, t_i$)
5: Multicast $(\text{UPDATE}, k, v, i, \texttt{counter}_i, t_i, j)$
6: **wait until** $t_s^j[l] \geq t_i[l], \quad \forall l$
7: Send $(\text{ACK}, \texttt{counter}_i, t_s^j)$ to client i
8: **function** RECEIVING(UPDATE, $k, v, i, \texttt{counter}_i, t_i, s$)
9: * witness$(*,*)$ is initialized to 0 *\
10: **if** this is the first UPDATE message from s **then**
11: $\texttt{witness}(i, \texttt{counter}_i) \leftarrow \texttt{witness}(i, \texttt{counter}_i) + 1$
12: **if** j has *not* sent this UPDATE message **then**
13: Multicast $(\text{UPDATE}, k, v, i, \texttt{counter}_i, t_i, j)$
14: **if** $\texttt{witness}(i, \texttt{counter}_i) = f + 1$ **then**
15: Enqueue $\langle k, v, i, t_i \rangle$ to Q
16: * this function is executed infinitely often *\
17: **function** UPDATE( )
18: **if** Q is non-empty **then**
19: $\langle k, v, i, t_i \rangle \leftarrow$ dequeue Q
20: **wait until** $t_s^j[i] = t_i[i] - 1$ **and** $t_s^j[l] \geq t_i[l], \quad \forall l \neq i$
21: * update timestamp and write to local memory *\
22: $t_s^j[i] \leftarrow t_i[i]; \quad M[k] \leftarrow v$

Algorithm 2 RCM for client i

1: **function** READ(k)
2: Multicast $(\text{READ}, k, i, \texttt{counter}, t_c^i)$
3: **wait until** $\langle \texttt{counter}, v, t \rangle$ is in resp
4: $t_c^i \leftarrow \text{MERGE}(t_c^i, t)$
5: $\texttt{counter} \leftarrow \texttt{counter} + 1$
6: **return** v
7: **function** WRITE(k, v)
8: * increment timestamp *\
9: $t_c^i[i] \leftarrow t_c^i[i] + 1$
10: Multicast $(\text{WRITE}, k, v, i, \texttt{counter}, t_c^i)$
11: **wait until** $\langle \texttt{counter}, t \rangle$ is in writeBuf
12: $t_c^i \leftarrow \text{MERGE}(t_c^i, t)$
13: $\texttt{counter} \leftarrow \texttt{counter} + 1$
14: **return** WRITE-ACK
15: **function** RECEIVING(RESP, $\texttt{counter}', v, t$)
16: Add $\langle \texttt{counter}', v, t \rangle$ to resp
17: **function** RECEIVING(ACK, $\texttt{counter}', t$)
18: Add $\langle \texttt{counter}', t \rangle$ to writeBuf

of the servers that the client has interacted with. This makes the construction of a serialization of the client's local history more complicated. Liveness and the two convergence properties follow from the usage of witness and the two aforementioned properties in Section V-B2.

Correctness Proof: Let H be a history of any execution. Based on the observations that the timestamp is always incremented, and for a write operation, the writer client increments its corresponding entry in the timestamp, the following lemma can be proved by induction. Following the notation in [4], $t_c^i(o)$ denotes the timestamp of operation o at any client i (timestamps used at line 10 in Algorithm 2).

Lemma 1. *Let o_1 and o_2 be two operations such that $o_1 \xrightarrow[\mathbf{CC}]{} o_2$. Then for any clients i and j, we have (i) $t_c^i(o_1) \leq t_c^j(o_2)$; and (ii) if o_2 is a write operation, then $t_c^i(o_1) < t_c^j(o_2)$. The same conditions hold if $i = j$.*

Lemma 1 is then used to prove the safety property as stated below. The proof is inspired by the one in [4] that cleverly used the property of vector timestamps. There are two major differences in our proof: (i) we need to find a complete history H'; and (ii) the construction of a serialization $\mathbb{S}$ of $H'|(c_i+W)$ is different, as clients do not see all the writes (unlike the case in MCS model where each node will see all the write operations eventually).

Theorem 1 (Safety)**.** *H satisfies causal consistency.*

Proof of Theorem 1. Due to space constraint, we assume the history H' is complete and consists of only completed operations. Please refer to [49] on how to obtain a complete history H' from H. Then, the next step is to find a serialization $\mathbb{S}$ of $H'|(c_i + W)$ that respects the causal ordering. $\mathbb{S}$ is constructed as follows:

- Let $o_1, o_2, \ldots$ denote the sequence of client i's operations.
- For each i's operation o, let $S(o)$ be any of the servers from which the client i received a reply message (write acknowledgment or read response).
- *Rule 1*: The serialization is concatenating all writes at client i as they are applied (line 14 of Algorithm 2) and all reads as they occur (line 6). That is, all the operations at client i follows the same ordering of the local history L_i.
- *Rule 2*: For the remaining write operations, we insert them to $\mathbb{S}$ one-by-one in the order of increasing timestamps (breaking tie arbitrarily). For each remaining write w, iterate through $\mathbb{S}$ and find the first operation o such that $t_c^j(w) \leq t_c^l(o)$ for some clients j, l. Insert w right before o.

Claim 1. *Using the construction above, $\mathbb{S}$ is a serialization.*

Proof. Suppose it is not, which means that there are consecutive operations w_1, w_2 and r on the same variable k in $\mathbb{S}$ such that (i) w_1 precedes w_2 and w_2 precedes r in $\mathbb{S}$; and (ii) r returns the value written by write w_1. If w_2 is client i's write, then the scenario is impossible due to (i) $\mathbb{S}$ respects the program order (by Rule 1), and (ii) $t_c^i(w_1) < t_c^i(w_2) \leq t_c^i(r)$ (by Lemma 1). Now consider the case when w_2 is some other client j's write. Since w_2 is inserted *before* r in $\mathbb{S}$, we have $t_c^j(w_2) \leq t_c^i(r)$. By construction, there must be some server s, from which client i received a response to complete read r or some operation prior to read r, letting client i know the existence of w_2, since $t_c^j(w_2)[j] \leq t_c^i(r)[j]$. As a result, read r must return the value written by w_2 or some value "newer" than the one by w_2, a contradiction. In both cases, we derive a contradiction, proving the claim. □

Using a similar argument in [4] which is based on Lemma 1 and the property of the timestamps, we can show that $\xrightarrow[\mathbf{CC}]{}$ is a partial order in H', since $\xrightarrow[\mathbf{CC}]{}$ is acyclic by the definition of vector timestamp. Finally, we use the following claim to complete the proof of Theorem 1.

Claim 2. *Let o_1 and o_2 be two operations in $H'|(c_i + W)$ such that $o_1 \xrightarrow[\mathbf{CC}]{} o_2$, o_1 precedes o_2 in $\mathbb{S}$.*

Proof. The proof is by contradiction. Suppose that for some o_1 and o_2, $o_1 \xrightarrow[\mathbf{CC}]{} o_2$ and o_2 precedes o_1 in $\mathbb{S}$. By Lemma 1, we have $t_c^j(o_1) \leq t_c^l(o_2)$ for some j, l. Then, o_1 cannot be inserted due to Rule 2 of the construction of $\mathbb{S}$. This means that both o_1 and o_2 are operations by client i. And by Rule 1, o_2 occurs before o_1 in L_i. This contradicts with the assumption that $o_1 \xrightarrow[\mathbf{CC}]{} o_2$. □

Observe that due to lines 8 to 15 in Algorithm 1, if $m = \langle k, v, i, t_i \rangle$ is in any server's Q, then eventually m will be in any other fault-free server's Q. We can then use this observation to prove liveness.

Theorem 2 (Liveness)**.** *Any operation o can be completed eventually at fault-free clients.*

Proof. The claim below follows from the observations: (i) $|\texttt{witness}(*)| \geq f+1$; (ii) among these $f+1$ servers, at least one is fault-free, say s; and (iii) every other fault-free server will receive s's multicast message at line 13 of Algorithm 1.

Claim 3. *If $m = \langle k, v, i, t_i \rangle$ is in any server's Q, then eventually m will be in any other fault-free server's Q.*

Consider a server s with m inside its Q. By induction and Claim 3, it is easy to see that server s can increment each entry until the point where $t_s^j[i] = t_i[i] - 1$ and $t_s^j[l] \geq t_i[l], \forall l \neq i$. To see this, suppose client i increments the l-th entry of its timestamp, which means some server (in the write quorum) has applied the corresponding write operation from client l, say w_l. Then, by Claim 3, server s will also be able to receive w_l, and eventually increment the l-th entry of t_s and catch up to the point of t_i. □

Finally, the following theorem can be proved based on the observation that each completed write operation will eventually be propagated to all the fault-free servers.

Theorem 3 (Persistence)**.** *H satisfies persistence as per Definition 3.*

Theorems 1, 2, and 3 together prove that RCM is correct given that $n \geq 2f + 1$.

VI. Lower Bound

In the section, we show that $2f + 1$ is the lower bound on the number of servers for an f-resilient causal memory, i.e., a causal shared memory that tolerates up to f server crashes and any client crash. Due to space limit, we only present the theorem and the intuition behind the proof. Pleaser refer to [49] for details.

Theorem 4. *If there exists an f-resilient DSM that satisfies liveness and persistence, then the system consists of at least $2f + 1$ servers, i.e., $|S| = n \geq 2f + 1$.*

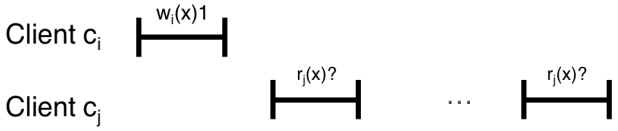


Fig. 3: Intuition for Theorem 4.

In Theorem 4, we do not include the safety condition, since it turns out that liveness and persistence together already imply the lower bound. The proof is an adaption of the proof showing that an atomic shared memory is *not* possible when there are strictly less than $2f + 1$ servers [5]. Our proof is more complicated due to the "eventual" feature in the definition of persistence.

In our proof, we formally show that if $n \leq 2f$, then the scenario in Figure 3 is possible. Particularly, after c_i completes a write $w_i(x)1$ and crashes, then it is possible that another client c_j might *never* be able to read the value 1 on the same key x. While the intuition is simple, the full proof relies on some delicate lemmas and notions that were introduced in prior work [24], [7], [27] such as execution, legal execution, A-free execution for some set of nodes A, and extended history. The key tool to show contradiction in our proof is based on the indistinguishability argument [24], [7]. More precisely, we create two executions that are *indistinguishable* to the client node c_j, because it observes the same message pattern in both executions. Then we show that either liveness or persistence is violated in an execution. Hence, no such DSM implementation exists if $n \leq 2f$. Please refer to [49] for complete discussion on the notions and tools, and the full proof.

VII. Evaluation

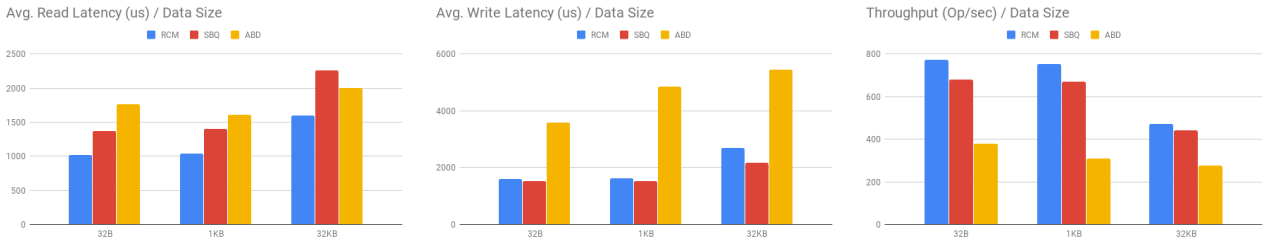
In this section, we first describe our implementation and the tools used, then report our evaluation results.

A. Implementation

We chose Golang[4] (Go) to implement our system. It is an open source language developed by Google, which aims for efficiency and ease-of-use for developing concurrent and distributed applications. Moreover, it is a lightweight, statically typed, and compiled language, and its concurrency mechanisms (such as channels) makes development of distributed systems much easier. Finally, there are many open-sourced Golang projects, which simplifies our development. Specifically, we use the following open-sourced projects in our implementation:

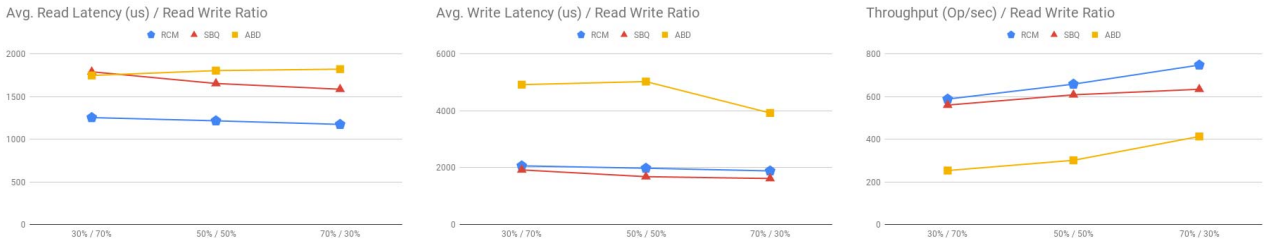
- For writing and reading data into and from disk, we use *diskv*,[5] a persistent disk-backed key-value storage system.
- For point-to-point communication among servers, and between servers and clients, we use ZeroMQ (ZMQ).[6] ZMQ is a high performance and asynchronous distributed messaging system. We chose ZMQ over raw TCP because

[4]https://golang.org/
[5]https://github.com/peterbourgon/diskv
[6]https://zeromq.org



(a) Average read latency (μs) (b) Average write latency (μs) (c) Throughput (Ops/Sec)

Fig. 4: LAN Performance vs. Data Size



(a) Average read latency (μs) (b) Average write latency (μs) (c) Throughput (Ops/Sec)

Fig. 5: LAN Performance vs. R/W Ratio

of its simple socket management. Moreover, it provides a thread-safe message queue, enabling us to remove a number of mutex locks that are blocking some concurrent operations. Compared to the TCP-based design, ZMQ speeds up the efficiency by a considerable amount. We also use a third-party Go-binding library *zmq4*[7] to dynamically control ZMQ in our system.

ZMQ provides several scalable messaging patterns to for different application scenarios. We adopt the *Publish-Subscribe (Pub-Sub)* pattern for the Multicast primitive, and the *Extended Request-Reply* pattern for replying message for the Send primitive. The *Pub-Sub* pattern is a data distribution pattern which connects set of publishers and set of subscribers. This type of one-way architecture satisfies the need of multi-casting messages and listening for messages. It is particularly useful when the sender does not expect the response a priori. The *Extended Request-Reply* pattern is a non-blocking request-response pattern that allows us to build the asynchronous communication pattern.

B. Experiment Setup

All evaluations were performed on virtual machines (VM) in Google Cloud Platform (GCP). Each node (both client and server) is executed on a separate VM of type n1-standard-2 with 2 virtual cores and 7.5 GB RAM. In all the experiments, we have three servers and two clients. Our workload generator at the client continuously sends out read or write requests to the servers. There are 10 keys, and in each run of the experiments, we generate $10,000$ operations uniformly random on the keys. It is known that the bottleneck of distributed replicated storages is the concurrent operation on the same key-value pairs. Therefore, we have high ops/key ratio to understand the worse scenarios. We expect to see performance improvement if there are more keys accessed.

We test our implementation in both local area network (LAN) case and wide-area network (WAN) case:

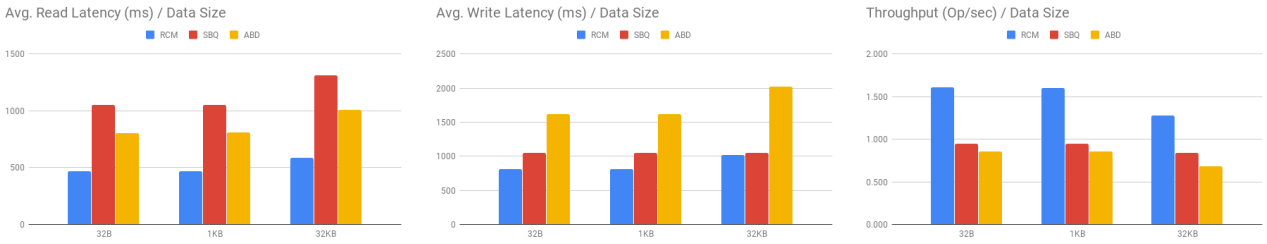
- *LAN*: all servers and clients are in the same datacenter in GCP. The measured average round-trip time (RTT) between any two VMs is around $250\mu s$.
- *WAN*: all nodes are in different continent. We have one server in South Carolina (North America), one server in London (Europe), one server in Tokyo (Asia), one client in Sao Paulo (South America), and one client in Sydney (Australia). The measured average RTT is around 200ms.

In addition to the location of the VMs, we also control two variables in our experiments:

- *Data size*: the size of value in each key-value pair. We test three sizes: 32 Bytes, 1024 Bytes (1KB), and 32 KB.
- *Read/Write ratio*: the ratio between the number of read operation and the number of write operation. We test 30/70, 50/50, and 70/30, where first number represents the percentage of read operations among total operation.

The reason that these variables are of interest is the results would demonstrate the impact of higher concurrency (due to higher write ratio) or higher network latency (due to larger value to be transmitted over the network). We report the average of five runs for each experiment configuration.

[7]https://github.com/pebbe/zmq4



(a) Average read latency (ms) (b) Average write latency (ms) (c) Throughput (Ops/Sec)

Fig. 6: WAN Performance vs. Data Size

C. Evaluation Results

We compare RCM with the well-known fault-tolerant algorithms ABD [5] and SBQ [44]. ABD achieves atomicity in the presence of crash faults, whereas SBQ safeness in the presence of semi-Byzantine faults. SBQ assumes fault-free clients and the timestamp is always correct (i.e., incorruptible); hence, it does not tolerate full Byzantine faults. For SBQ, we set Read and Write quorums to 3; For all three algorithms (ABD, SBQ, and RCM), we set f to 1.

The first set of figures show the read latency, write latency, and throughput per client of the three algorithms in LAN under different data size with a write ration of 0.1 in Figures 4a, 4b, and 4c respectively. The second set of figures show the read latency, write latency, and throughput per client of the three algorithms in LAN under different write ratio with data size 64 Bytes in Figures 5a, 5b, and 5c respectively. The third set of figures show the read latency, write latency, and throughput per client of the three algorithms in WAN under different data size with a write ration of 0.1 in Figures 6a, 6b, and 6c respectively. We present more evaluation results in [49].

We summarize a few interesting points below:

- One interesting observation is that as the read ratio increases, all the algorithms have higher throughput. This might be owing to the effect of concurrent writes and the fact that write operation takes longer to complete as all the data has to be written to the disk.
- Results in WAN and LAN have similar patterns. However, due to much higher network latency, the difference among the latency of each algorithm is more significant. Our algorithms show significant improvement in latency over ABD and SBQ.
- While ABD and SBQ are relatively stable with different read-write ratio, RCM has lowest performance when read-write ratio is 30/70. This may be due to the increasing buffer size when most operations are writes, which leads to a slowdown in local computation.
- In all cases, RCM has higher throughput than ABD and SBQ, indicating that it is a good alternative for latency-sensitive applications if atomicity can be sacrificed.

Note that the client is single-threaded, so the throughput is unreasonably low for practical usage. This can be easily addressed by increasing the parallelism of the client workload generator. The main purpose of the evaluation is to test the performance under different network scenarios. Such an optimization is left as an interesting future work.

VIII. Summary

In this paper, we first identify the difference between MCS model (multi-processor environment) and client-server model. The main result is showing that $2f + 1$ servers is both necessary and sufficient to implement causal shared memory in asynchronous networks with crash-prone nodes. We also perform evaluation in GCP and demonstrate that RCM is faster than ABD and SBQ.

There are several interesting open problems. We list two important ones below:

- A more efficient mechanism to track dependency.
- A weak enough but still useful convergence property to reduce the required number of servers.

References

[1] Cassandra. http://cassandra.apache.org/.
[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, Dec. 1996.
[3] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *J. ACM*, 58(2):7:1–7:32, Apr. 2011.
[4] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
[5] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
[6] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, Jan. 1995.
[7] H. Attiya and F. Ellen. *Impossibility Results for Distributed Computing*. Morgan & Claypool, June 2014.
[8] H. Attiya, F. Ellen, and A. Morrison. Limitations of highly-available eventually-consistent data stores. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 385–394, 2015.
[9] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, 1994.
[10] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, NY, USA, 2013. ACM.
[11] R. Baldoni, M. Malek, A. Milani, and S. Piergiovanni. Weakly-persistent causal objects in dynamic distributed systems. In *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on*, pages 165–174, Oct 2006.
[12] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, Aug. 1991.

[13] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[14] M. Bravo, L. E. T. Rodrigues, and P. V. Roy. Saturn: a distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pages 111–126, 2017.

[15] E. A. Brewer. Towards robust distributed systems (Invited Talk). In *Proc. ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, 2000.

[16] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[17] T. Crain and M. Shapiro. 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*, PaPoC '15, pages 6:1–6:4, New York, NY, USA, 2015. ACM.

[18] F. Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[20] S. Dolev, T. Petig, and E. M. Schiller. Brief announcement: Robust and private distributed shared atomic memory in message passing networks. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 311–313, 2015.

[21] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[22] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014*, pages 4:1–4:13, 2014.

[23] Y. Einav. Amazon found every 100ms of latency cost them 1% in sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, January 2019.

[24] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[25] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Comput. Archit. News*, 18(2SI):15–26, May 1990.

[26] R. Guerraoui and R. Levy. Robust emulations of shared memory in a crash-recovery model. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 400–407, 2004.

[27] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[28] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[29] D. Imbs, S. Rajsbaum, M. Raynal, and J. Stainer. Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems. *J. Parallel Distrib. Comput.*, 93-94:1–9, 2016.

[30] J. James and A. K. Singh. Fault tolerance bounds for memory consistency. In J. E. Burns and H. Attiya, editors, *PODC*, page 285. ACM, 1997.

[31] K. Kanjani, H. Lee, W. L. Maguffee, and J. L. Welch. A simple byzantine-fault-tolerant algorithm for a multi-writer regular register. *IJPEDS*, 25(5):423–435, 2010.

[32] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *SIGARCH Comput. Archit. News*, 20(2):13–21, Apr. 1992.

[33] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 43–57, New York, NY, USA, 1990. ACM.

[34] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

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

[36] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[37] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[38] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[39] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2013.

[40] N. Lynch and S. Gilbert. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[41] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[42] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. Technical report, The University of Texas at Austin, 2011.

[43] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[44] J. Martin, L. Alvisi, and M. Dahlin. Small byzantine quorum systems. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*, pages 374–388, 2002.

[45] M. Raynal. *Fault-Tolerant Message-Passing Distributed Systems - An Algorithmic Approach*. Springer, 2018.

[46] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu. Causal consistency for geo-replicated cloud storage under partial replication. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 509–518, 2015.

[47] M. Shen, A. D. Kshemkalyani, and T. Y. Hsu. OPCAM: optimal algorithms implementing causal memories in shared memory systems. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN 2015, Goa, India, January 4-7, 2015*, pages 16:1–16:4, 2015.

[48] K. Spirovska, D. Didona, and W. Zwaenepoel. Wren: Nonblocking reads in a partitioned transactional causally consistent data store. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 1–12, 2018.

[49] L. Tseng. Resilient causal memory in client-server model. Technical report, Boston College, 2019 https://drive.google.com/file/d/1y_1GcC5HVFgQTE2iTk7H1kVNGmqw-4CS/view?usp=sharing.

[50] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

[51] Z. Xiang and N. H. Vaidya. Brief announcement: Partially replicated causally consistent shared memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 273–275, 2018.

[52] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, New York, NY, USA, 2015. ACM.