2015 IEEE International Parallel and Distributed Processing Symposium Workshop

Causal Consistency for Geo-Replicated Cloud Storage under Partial Replication

Min Shen, Ajay D. Kshemkalyani, and Ta-yuan Hsu
University of Illinois at Chicago
Chicago, IL 60607, USA
Email: {mshen6, ajay, thsu4}@uic.edu

***Abstract*—Data replication is a common technique used for fault-tolerance in reliable distributed systems. In geo-replicated systems and the cloud, it additionally provides low latency. Recently, causal consistency in such systems has received much attention. However, all existing works assume the data is fully replicated. This greatly simplifies the design of the algorithms to implement causal consistency. In this paper, we propose that it can be advantageous to have partial replication of data, and we propose two algorithms for achieving causal consistency in such systems where the data is only partially replicated. This is the first work that explores causal consistency for partially replicated geo-replicated systems. We also give a special case algorithm for causal consistency in the full-replication case.**

***Keywords*-causal consistency; causality; cloud computing; distributed computing; geo-replicated storage; replication**

I. Introduction

Data replication is commonly used for fault tolerance in reliable distributed systems. It also reduces access latency in the cloud and geo-replicated systems. With data replication, consistency of data in the face of concurrent reads and updates becomes an important problem. There exists a spectrum of consistency models in distributed shared memory systems [19]: linearizability (the strongest), sequential consistency, causal consistency, pipelined RAM, slow memory, and eventual consistency (the weakest). These consistency models represent a trade-off between cost and convenient semantics for the application programmer.

Recently, consistency models have received attention in the context of cloud computing with data centers and geo-replicated storage, with product designs from industry, e.g., Google, Amazon, Microsoft, LinkedIn, and Facebook. The CAP Theorem by Brewer [16] states that for a replicated, distributed data store, it is possible to provide at most two of the three features: Consistency of replicas, Availability of Writes, and Partition tolerance. In the face of this theorem, most systems such as Amazon's Dynamo [12] chose to implement eventual consistency [7], which states that eventually, all copies of each data item converge to the same value. Besides the above three features, two other desirable features of large-scale distributed data stores are: low Latency and high Scalability [22]. Causal consistency is the strongest form of consistency that satisfies low Latency [22], defined as the latency less than the maximum wide-area delay between replicas. Causal consistency in distributed shared memory systems was proposed by Ahamad et al. [1]. Causal consistency has been studied by Baldoni el al. [5], Mahajan et al. [24], Belaramani et al. [6], and Petersen et al. [25]. More recently, in the past four years, causal consistency has been studied and/or implemented by numerous researchers [2], [3], [4], [13], [14], [20], [22], [23]. Many of these do not provide scalability as they use a form of log serialization and exchange to implement causal consistency. More importantly, all the works assume Complete Replication and Propagation (CRP) based protocols. These protocols assume full replication and do not consider the case of partial replication. This is primarily because full replication makes it easy to implement causal consistency.

Case for Partial Replication

Our proposed protocols for causal consistency are designed for partial replication across the distributed shared memory. We now make a case for partial replication. (1) Partial replication is more natural for some applications. Consider the following example. A user U's data is replicated across multiple data centers located in different regions. If user U's connections are located mostly in the Chicago region and the US West coast, the majority of views of user U's data will come from these two regions. In such a case, it is an overkill to replicate user U's data in data centers outside these two regions, and partial replication has very small impact on the overall latency in this scenario. With p replicas placed at some p of the total of n data centers, each write operation that would have triggered an update broadcast to the n data centers now becomes a multicast to just p of the n data centers. This is a direct savings in the number of messages and p is a tunable parameter. (2) For write-intensive workloads, it naturally follows that partial replication gives a direct savings in the number of messages without incurring any delay or latency for reads. (3) Recent researchers have explicitly acknowledged that providing causal consistency under partial replication is a big challenge. For example, Lloyd et al. [22] and Bailis et al. [3] write: "While weaker consistency models are often amenable to partial replication, allowing flexibility in the number of datacenters required in causally consistent replication remains an interesting aspect of future work." (4) The supposedly higher cost of tracking dependency metadata, which has deterred prior researchers from considering partial replication, is relatively small for applications such

978-1-4673-7684-6/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPSW.2015.68

as Facebook, where photos and large files are uploaded. In addition, the protocols for causal consistency that we present are efficient and have relatively low meta-data overheads.

Contributions

We present the first algorithms for implementing causal consistency in partially replicated distributed shared memory systems.

1) Algorithm Full-Track is optimal in the sense defined by Baldoni et al. [5], viz., the protocol can update the local copy as soon as possible while respecting causal consistency. This reduces the false causality in the system.
2) Algorithm Full-Track can be made further optimal in terms of the size of the local logs maintained and the amount of control information piggybacked on the update messages, by achieving minimality. The resulting algorithm is Algorithm Opt-Track.
3) As a special case of Algorithm Opt-Track, we present Algorithm Opt-Track-CRP, that is optimal in a fully replicated shared memory system. This algorithm is optimal not only in the sense of Baldoni et al. but also in the amount of control information used in local logs and on update messages. The algorithm is significantly more efficient than the Baldoni et al. protocol for the complete replication case.

A causal distributed shared memory is implemented using message passing under the covers. An added contribution of our algorithms is that they integrate ideas from the message passing paradigm with ideas from the shared memory paradigm and show how to attain optimality in time overhead, space overhead, and message overhead for causal memories.

A short announcement of these results appears as [27].

Organization

Section II gives the causal memory model. Section III presents two algorithms that implement causal consistency under partial replication and a special case algorithm dealing with full replication. Section IV analyses the complexity of the algorithms. Section V gives a discussion. Section VI concludes.

II. System Model

A. Causally Consistent Memory

The system model is based on that proposed by Ahamad et al. [1] and Baldoni et al. [5]. We consider a system which consists of n application processes $ap_1, ap_2, \ldots, ap_n$ interacting through a shared memory $\mathcal{Q}$ composed of q variables $x_1, x_2, \ldots, x_q$. Each application process ap_i can perform either a *read* or a *write* operation on any of the q variables. A *read* operation performed by ap_i on variable x_j which returns value v is denoted as $r_i(x_j)v$. Similarly, a *write* operation performed by ap_i on variable x_j which writes the value u is denoted as $w_i(x_j)u$. Each variable has an initial value $\perp$.

By performing a series of *read* and *write* operations, an application process ap_i generates a local history h_i. If a local operation o_1 precedes another operation o_2, we say o_1 precedes o_2 under *program order*, denoted as $o_1 \prec_{po} o_2$. The set of local histories h_i from all n application processes form the global history H. Operations performed at distinct processes can also be related using the *read-from order*, denoted as $\prec_{ro}$. Two operations o_1 and o_2 from distinct processes ap_i and ap_j respectively have the relationship $o_1 \prec_{ro} o_2$ if there are variable x and value v such that $o_1 = w(x)v$ and $o_2 = r(x)v$, meaning that *read* operation o_2 retrieves the value written by the *write* operation o_1. It is shown in [1] that

- for any operation o_2, there is at most one operation o_1 such that $o_1 \prec_{ro} o_2$;
- if $o_2 = r(x)v$ for some x and there is no operation o_1 such that $o_1 \prec_{ro} o_2$, then $v = \perp$, meaning that a read with no preceding write must read the initial value.

With both the *program order* and *read-from order*, the *causality order*, denoted as $\prec_{co}$, can be defined on the set of operations O_H in a history H. The causality order is the transitive closure of the union of local histories' program order and the read-from order. Formally, for two operations o_1 and o_2 in O_H, $o_1 \prec_{co} o_2$ if and only if one of the following conditions holds:

1) $\exists ap_i$ s.t. $o_1 \prec_{po} o_2$ (program order)
2) $\exists ap_i, ap_j$ s.t. o_1 and o_2 are performed by ap_i and ap_j respectively, and $o_1 \prec_{ro} o_2$ (read-from order)
3) $\exists o_3 \in O_H$ s.t. $o_1 \prec_{co} o_3$ and $o_3 \prec_{co} o_2$ (transitive closure)

Essentially, the causality order defines a partial order on the set of operations O_H. For a shared memory to be causal memory, all the write operations that can be related by the causality order have to be seen by each application process in the order defined by the causality order.

B. Underlying Distributed Communication System

The shared memory abstraction and its causal consistency model is implemented on top of the underlying distributed message passing system which also consists of n sites connected by FIFO channels, with each site s_i hosting an application process ap_i. Since we assume a partially replicated system, each site holds only a subset of variables $x_h \in \mathcal{Q}$. For application process ap_i, we denote the subset of variables kept on the site s_i as X_i. If the replication factor of the shared memory system is p and the variables are evenly replicated on all the sites, then the average size of X_i is $\frac{pq}{n}$.

To facilitate the read and write operations in the shared memory abstraction, the underlying message passing system provides several primitives to enable the communication between different sites. For the write operation, each time

an application process ap_i performs $w(x_1)v$, it invokes the $\texttt{Multicast}(m)$ primitive to deliver the message m containing $w(x_1)v$ to all sites that replicate the variable x_1. For the read operation, there is a possibility that an application process ap_i performing read operation $r(x_2)u$ needs to read x_2's value from a remote site since x_2 is not locally replicated. In such a case, it invokes the $\texttt{RemoteFetch}(m)$ primitive to deliver the message m containing $r(x_2)u$ to a pre-designated site replicating x_2 to fetch its value u. This is a synchronous primitive, meaning that it will block until returning the variable's value. If the variable to be read is locally replicated, then the application process simply returns the local value.

The read and write operations performed by the application processes also generate *events* in the underlying message passing system. The following list of events are generated at each site:

- *Send event.* The invocation of $\texttt{Multicast}(m)$ primitive by application process ap_i generates event $send_i(m)$.
- *Fetch event.* The invocation of $\texttt{RemoteFetch}(m)$ primitive by application process ap_i generates event $fetch_i(f)$.
- *Message receipt event.* The receipt of a message m at site s_i generates event $receipt_i(m)$. The message m can correspond to either a $send_j(m)$ event or a $fetch_j(f)$ event.
- *Apply event.* When applying the value written by the operation $w_j(x_h)v$ to variable x_h's local replica at application process ap_i, an event $apply_i(w_j(x_h)v)$ is generated.
- *Remote return event.* After the occurrence of event $receipt_i(m)$ corresponding to the remote read operation $r_j(x_h)u$ performed by ap_j, an event $remote\_return_i(r_j(x_h)u)$ is generated which transmits x_h's value u to site s_j.
- *Return event.* Event $return_i(x_h, v)$ corresponds to the return of x_h's value v either fetched remotely through a previous $fetch_i(f)$ event or read from the local replica.

To implement the causal memory in the shared memory abstraction, each time an update message m corresponding to a write operation $w_j(x_h)v$ is received at site s_i, a new thread is spawned to check when to locally apply the update. The condition that the update is ready to be applied locally is called activation predicate in [5]. This predicate, $A(m_{w_j(x_h)v}, e)$, is initially set to $false$ and becomes $true$ only when the update $m_{w_j(x_h)v}$ can be applied after the occurrence of local event e. The thread handling the local application of the update will be blocked until the activation predicate becomes $true$, at which time the thread writes value v to variable x_h's local replica. This will generate the $apply_i(w_j(x_h)v)$ event locally. Thus, the key to implement the causal memory is the activation predicate.

C. Activation Predicate

To demonstrate the activation predicate, Baldoni et al. [5] defined a new relation, $\rightarrow_{co}$, on *send events* generated in the underlying message passing system. Let $w(x)a$ and $w(y)b$ be two write operations in O_H. Then, for their corresponding send events in the underlying message passing system, $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b})$ iff one of the following conditions holds:

1) $i = j$ and $send_i(m_{w(x)a})$ locally precedes $send_j(m_{w(y)b})$
2) $i \neq j$ and $return_j(x, a)$ locally precedes $send_j(m_{w(y)b})$
3) $\exists send_k(m_{w(z)c})$, such that $send_i(m_{w(x)a}) \rightarrow_{co} send_k(m_{w(z)c}) \rightarrow_{co} send_j(m_{w(y)b})$

Notice that the relation defined by $\rightarrow_{co}$ is actually a subset of Lamport's "happened before" relation [21], denoted by $\rightarrow$. If two send events are related by $\rightarrow_{co}$, then they are also related by $\rightarrow$. However, the other way is not necessarily true. Even though $send_i(m_{w(x)a}) \rightarrow send_j(m_{w(y)b})$, if there is no return event that occurred and $i \neq j$, these two send events are concurrent under the $\rightarrow_{co}$ relation. The difference between these two relations is essential under the context of causal memory. The $\rightarrow_{co}$ relation better represents the causality order in the shared memory abstraction as it prunes the "false causality"[1] introduced in the underlying message passing system, where message receipt events may causally relate two send events while their corresponding write operations in the shared memory abstraction are concurrent under the $\prec_{co}$ relation. Actually, in [5], the authors have shown that $send_i(m_{w(x)a}) \rightarrow_{co} send_j(m_{w(y)b}) \Leftrightarrow w(x)a \prec_{co} w(y)b$.

With the $\rightarrow_{co}$ relation defined, Baldoni et al. gave an optimal activation predicate in [5] as follows:

$$A_{OPT}(m_w, e) \equiv \nexists m_{w'} : (send_j(m_{w'}) \rightarrow_{co} send_k(m_w) \wedge apply_i(w') \notin E_i|_e)$$

where $E_i|_e$ is the set of events happened at the site s_i up until e (excluding e).

This activation predicate cleanly represents the causal memory's requirement: a write operation shall not be seen by an application process before any causally preceding write operations. It is optimal because the moment this activation predicate $A_{OPT}(m_w, e)$ becomes true is the earliest instant that the update m_w can be applied. The activation predicate used in the original paper describing causal memory [1] uses the happened before relation. This activation predicate is given below as A_{ORG}. As shown by Baldoni et al. [5], it does not achieve optimality.

$$A_{ORG}(m_w, e) \equiv \nexists m_{w'} : (send_j(m_{w'}) \rightarrow send_k(m_w) \wedge apply_i(w') \notin E_i|_e)$$

[1]False causality was identified by Lamport [21] and later discussed by others [3], [4], [8], [11], [15], [20].

III. ALGORITHMS

We design two algorithms – Algorithm Full-Track and Algorithm Opt-Track – implementing causal memories in a partially replicated distributed shared memory system, both of which adopt the optimal activation predicate A_{OPT}. Algorithm Opt-Track is a message and space optimal algorithm for a partially replicated system. Subsequently, as a special case of this algorithm, we derive an optimal algorithm – Algorithm Opt-Track-CRP – for the fully replicated case, that is optimal and has lower message size, time, and space complexities than the Baldoni et al. algorithm [5].

A. Full-Track Algorithm

Since the system is partially replicated, each application process performing a write operation will only write to a subset of all the sites in the system. Thus, for an application ap_i and a site s_j, not all write operations performed by ap_i will be seen by s_j. This makes it necessary to distinguish the destinations of ap_i's write operations. The activation predicate A_{OPT} requires tracking the number of updates received that causally happened before under the $\rightarrow_{co}$ relation. In order to do so in a partially replicated scenario, it is necessary for each site s_i to track the number of write operations performed by every application process ap_j to every site s_k. We denote this value as $Write_i[j][k]$. Application processes also piggyback this clock value on every outgoing message generated by the $\texttt{Multicast}(m)$ primitive. The $Write$ matrix clock tracks the causal relation under the $\rightarrow_{co}$ relation, rather than the causal relation under the $\rightarrow$ relation.

Another implication of tracking under the $\rightarrow_{co}$ relation is that the way to merge the piggybacked clock with the local clock needs to be changed. In Lamport's happened before relation $\rightarrow$, a message transmission generates a causal relationship between two processes. However, under the $\rightarrow_{co}$ relation, it is reading the value that was written previously by another application process that generates a causal relationship between two processes. Thus, the $Write$ clock piggybacked on messages generated by the $\texttt{Multicast}(m)$ primitives should not be merged with the local $Write$ clock at the message reception. It should be delayed until a later read operation which reads the value that comes with the message.

The formal algorithm is listed in Algorithm 1. At each site s_i, the following data structures are maintained:

1) $Write_i[1 \ldots n, 1 \ldots n]$: the $Write$ clock (initially set to 0s). $Write_i[j,k] = a$ means that the number of updates sent by application process ap_j to site s_k that causally happened before under the $\rightarrow_{co}$ relation is a.
2) $Apply_i[1 \ldots n]$: an array of integers (initially set to 0s). $Apply_i[j] = a$ means that a total number of a updates written by application process ap_j have been applied at site s_i.

Algorithm 1: Full-Track Algorithm (Code at site s_i)

```
WRITE(x_h, v):
1  for all sites s_j that replicate x_h do
2      Write_i[i, j] + +;
3  Multicast[m(x_h, v, Write_i)] to all sites s_j (j ≠ i) that
   replicate x_h;
4  if x_h is locally replicated then
5      x_h := v;
6      Apply_i[i] + +;
7      LastWriteOn_i⟨h⟩ := Write_i;

READ(x_h):
8  if x_h is not locally replicated then
9      RemoteFetch[f(x_h)] from predesignated site s_j that
       replicates x_h to get x_h and LastWriteOn_j⟨h⟩;
10     ∀k, l ∈ [1 ... n], Write_i[k, l] :=
       max(Write_i[k, l], LastWriteOn_j⟨h⟩.Write[k, l]);
11 else
12     ∀k, l ∈ [1 ... n], Write_i[k, l] :=
       max(Write_i[k, l], LastWriteOn_i⟨h⟩.Write[k, l]);
13 return x_h;

On receiving m(x_h, v, W) from site s_j:
14 wait until
   (∀k ≠ j, Apply_i[k] ≥ W[k, i] ∧ Apply_i[j] = W[j, i] − 1);
15 x_h := v;
16 Apply_i[j] + +;
17 LastWriteOn_i⟨h⟩ := W;

On receiving f(x_h) from site s_j:
18 return x_h and LastWriteOn_i⟨h⟩ to s_j;
```

3) $LastWriteOn_i\langle \text{variable id}, Write\rangle$: a hash map of $Write$ clocks. $LastWriteOn_i\langle h\rangle$ stores the $Write$ clock value associated with the last write operation on variable x_h which is locally replicated at site s_i.

We can see from this algorithm that, instead of merging the piggybacked $Write$ clock at message reception, it is delayed until a later read operation at line (10) and (12). This implements tracking causality under the $\rightarrow_{co}$ relation. Furthermore, the activation predicate A_{OPT} is implemented at line (14).

Note that size n^2 matrices were previously used for causal message ordering in message-passing systems [26].

B. Opt-Track Algorithm

Algorithm Full-Track achieves optimality in terms of the activation predicate. However, in other aspects, it can still be further optimized. We notice that, each message corresponding to a write operation piggybacks an $O(n^2)$ matrix, and the same storage cost is also incurred at each site s_i. Kshemkalyani and Singhal proposed the necessary and sufficient conditions on the information for causal message ordering and designed an algorithm implementing these optimality conditions [17], [18] (hereafter referred to as the KS algorithm). The KS algorithm aims at reducing the message size and storage cost for causal message ordering

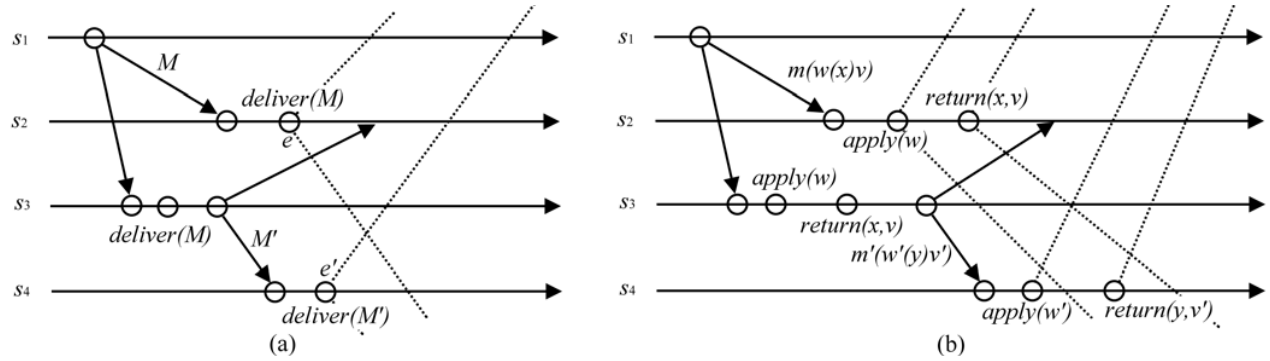

Figure 1. Illustration of the two conditions for destination information to be redundant. (a) For causal message ordering algorithms, the information is "s_2 is a destination of M". The causal future of the relevant message delivery events are shown in dotted lines. (b) For causal memory algorithms, the information is "s_2 is a destination of m". The causal future of the relevant *apply* and *return* events are shown in dotted lines.

algorithms in message passing systems. The ideas behind the KS algorithm exploit the transitive dependency of causal deliveries of messages. In the KS algorithm, each site keeps a record of recently received messages from each other site. The list of destinations of the message is also kept in each record (the KS algorithm assumes multicast communication) and is progressively pruned, as described below. With each outgoing message, these records are also piggybacked. The KS algorithm achieves another optimality, in the sense that no redundant destination information is recorded. There are two situations when the destination information can become redundant. These are illustrated in Fig 1(a).

1) When message M is delivered at site s_2 (we denote this event as e), then the information that s_2 is part of message M's destination no longer needs to be remembered in the causal future of e. This is because the delivery of M at s_2 is guaranteed at events in the causal future of e.
   In addition, we *implicitly* remember in the causal future of e that M has been delivered to s_2, to clean the logs at other sites.
2) Consider two messages M and M' such that M' is sent in the causal future of sending M and both messages have site s_2 as the receiver. Then the information that s_2 is part of message M's destinations no longer needs to be remembered in the causal future of the delivery events (denoted as e') of message M' at all recipient sites s_k. (In fact, the information need not even be transmitted on M' sent to sites s_k, other than to site s_2.) This is because by ensuring message M' is causally delivered at s_2 with respect to any message M'' that is also sent to s_2 in the causal future of sending M', it can be inferred using a transitive argument that message M should have already been delivered at s_2 before M'' is delivered.
   In addition, we *implicitly* remember in the causal future of events of type e' that M has been delivered to s_2, to clean the logs at other sites.

Remembering *implicitly* means inferring that information from other later or more up to date log entries, without storing that information.

Although the KS algorithm is for message passing systems, its ideas of deleting unnecessary dependency information still apply to distributed shared memory systems. We can adapt the KS algorithm to a partially replicated shared memory system to implement causal memory there. Now, each site s_i will maintain a record of the most recent updates received from every site, that causally happened before under the $\rightarrow_{co}$ relation. Each such record also keeps a list of destinations representing the set of replicas receiving the corresponding update. When performing a write operation, the outgoing update messages will piggyback the currently stored records. When receiving an update message, the optimal activation predicate A_{OPT} is used to determine when to apply the update. Once the update is applied, the piggybacked records will be associated with the updated variable. When a later read operation is performed on the updated variable, the records associated with the variable will be merged into the locally stored records to reflect the causal dependency between the read and write operations. Similar to the KS algorithm, we can prune redundant destination information using the following two conditions. These are illustrated in Fig 1(b).

- **Condition 1**: When an update m corresponding to write operation $w(x)v$ is applied at site s_2, then the information that s_2 is part of the update m's destinations no longer needs to be remembered in the causal future of the event $apply_2(w)$.
  In addition, we *implicitly* remember in the causal future of event $return_2(x, v)$ that m has been delivered to s_2, to clean the logs at other sites.
- **Condition 2**: For two updates $m_{w(x)v}$ and $m'_{w'(y)v'}$ such that $send(m) \rightarrow_{co} send(m')$ and both updates are sent to site s_2, the information that s_2 is part of

update m's destinations is irrelevant in the causal future of the event $apply(w')$ at all sites s_k receiving update m'. (In fact, it is redundant in the causal future of $send(m')$, other than m' sent to s_2.) This is because, by transitivity, applying update m' at s_2 in causal order with respect to a message m'' sent causally later to s_2 will infer the update m has already been applied at s_2. In addition, we *implicitly* remember in the causal future of events $return_k(y, v')$ that m has been delivered to s_2, to clean the logs at other sites.

Notice that, in the KS algorithm, even if the destination list in a message M's record becomes $\varnothing$ at a certain event e in site s_i, that record still needs to be kept until a later message from message M's sender is delivered at s_i. This is because although M's destination list becomes $\varnothing$ at s_i, it might still be non-empty at other sites. Thus, by piggybacking M's record with an empty destination list, we can prune M's destination list at other sites in the causal future of event e. This is illustrated in Fig 2. Notice that, after the deliveries of message M_2 and M_3, the destination list in the message M_1's record at site s_4 becomes empty. If we delete M_1's record at this time, then when message M_4 is later sent to site s_3, there is no way for s_3 to know that it can delete site s_2 from its local record of message M_1's destination list. M_1's record can be deleted at site s_4 after another message from s_1 is delivered at s_4. This technique is important for achieving optimality of no redundant destination information of already delivered messages (Condition 1) and of messages guaranteed to be delivered in causal order (Condition 2). However, note that such records with $\varnothing$ destination lists may accumulate. Only the latest such record per sender needs to be maintained; the presence of other such records can be implicitly inferred.

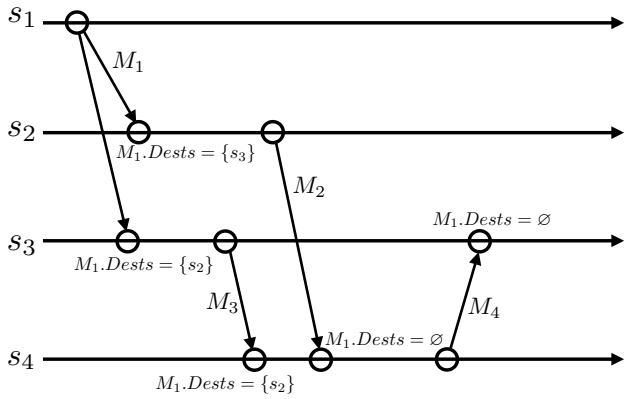

Figure 2. Illustration of why it is important to keep a record even if its destination list becomes empty.

With the above discussion, we give the formal algorithm in Algorithm 2. The following data structures are maintained at each site:

1) $clock_i$: local counter at site s_i for write operations performed by application process ap_i.
2) $Apply_i[1 \ldots n]$: an array of integers (initially set to 0s). $Apply_i[j] = a$ means that a total number of a updates written by application process ap_j have been applied at site s_i.
3) $LOG_i = \{\langle j, clock_j, Dests\rangle\}$: the local log (initially set to empty). Each entry indicates a write operation in the causal past. $Dests$ is the destination list for that write operation. Only necessary destination information is stored.
4) $LastWriteOn_i\langle$variable id, $LOG\rangle$: a hash map of LOGs. $LastWriteOn_i\langle h\rangle$ stores the piggybacked LOG from the most recent update applied at site s_i for locally replicated variable x_h.

Notice that lines (4)-(6) and lines (10)-(11) prune the destination information using condition 2, while lines (29)-(30) use condition 1 to prune the redundant information. Also, in lines (7)-(8) and in the `PURGE` function (see Algorithm 3), entries with empty destination list are kept as long as and only as long as they are the most recent update from the sender. This implements the optimality techniques described before. The optimal activation predicate A_{OPT} is implemented in lines (24)-(25).

Algorithm 3 gives the procedures used by Algorithm Opt-Track (Algorithm 2). Function `PURGE` removes old records with $\varnothing$ destination lists, per sender process. On a read operation of variable x_h, function `MERGE` merges the piggybacked log of the corresponding write to x_h with the local log LOG_i. In this function, new dependencies get added to LOG_i and existing dependencies in LOG_i are pruned, based on the information in the piggybacked data L_w. The merging implements the optimality techniques described before.

At the expense of slightly larger message overhead, we can distribute the *Write* processing in lines (3)-(8) of Algorithm 2 to the receivers' sites after line (27). Instead of the loop in line (4), send the LOG; and on its receipt, for each entry o in L_w, subtract $x_h.replicas$ from $o.Dests$. This reduces the time complexity of a write operation from $O(n^2p)$ to $O(n^2)$.

C. Opt-Track-CRP: Adapting Opt-Track Algorithm to Fully-Replicated Systems

Algorithm Opt-Track can be directly applied to fully replicated shared memory systems. Furthermore, since in the full replication case, every write operation will be sent to exactly the same set of sites, namely all of them, there is no need to keep a list of the destination information with each write operation. Each time a write operation is sent, all the write operations it piggybacks as its dependencies will share the same set of destinations as the one being sent, and their destination list will be pruned by condition 2. Also, when a write operation is received, all the write operations it piggybacks also have the receiver as part of their destinations.

Algorithm 2: Opt-Track Algorithm (Code at site s_i)

WRITE(x_h, v):
1 $clock_i + +$;
2 **for** *all sites* $s_j (j \neq i)$ *that replicate* x_h **do**
3 $L_w := LOG_i$;
4 **for** *all* $o \in L_w$ **do**
5 **if** $s_j \notin o.Dests$ **then** $o.Dests := o.Dests \setminus x_h.replicas$;
6 **else** $o.Dests := o.Dests \setminus x_h.replicas \cup \{s_j\}$;
7 **for** *all* $o_{z,clock_z} \in L_w$ **do**
8 **if** $o_{z,clock_z}.Dests = \varnothing \wedge (\exists o'_{z,clock'_z} \in L_w | clock_z < clock'_z)$ **then** remove $o_{z,clock_z}$ from L_w;
9 send $m(x_h, v, i, clock_i, x_h.replicas, L_w)$ to site s_j;
10 **for** *all* $l \in LOG_i$ **do**
11 $l.Dests := l.Dests \setminus x_h.replicas$;
12 PURGE;
13 $LOG_i := LOG_i \cup \{\langle i, clock_i, x_h.replicas \setminus \{s_i\}\rangle\}$;
14 **if** x_h *is locally replicated* **then**
15 $x_h := v$;
16 $Apply_i[i] + +$;
17 $LastWriteOn_i\langle h\rangle := LOG_i$;

READ(x_h):
18 **if** x_h *is not locally replicated* **then**
19 RemoteFetch$[f(x_h)]$ from predesignated site s_j that replicates x_h to get x_h and $LastWriteOn_j\langle h\rangle$;
20 MERGE$(LOG_i, LastWriteOn_j\langle h\rangle)$;
21 **else** MERGE$(LOG_i, LastWriteOn_i\langle h\rangle)$;
22 PURGE;
23 return x_h;

On receiving $m(x_h, v, j, clock_j, x_h.replicas, L_w)$ from site s_j:
24 **for** *all* $o_{z,clock_z} \in L_w$ **do**
25 **if** $s_i \in o_{z,clock_z}.Dests$ **then** wait until $clock_z \leq Apply_i[z]$;
26 $x_h := v$;
27 $Apply_i[j] := clock_j$;
28 $L_w := L_w \cup \{\langle j, clock_j, x_h.replicas\rangle\}$;
29 **for** *all* $o_{z,clock_z} \in L_w$ **do**
30 $o_{z,clock_z}.Dests := o_{z,clock_z}.Dests \setminus \{s_i\}$;
31 $LastWriteOn_i\langle h\rangle := L_w$;

On receiving $f(x_h)$ from site s_j:
32 return x_h and $LastWriteOn_i\langle h\rangle$ to s_j;

Algorithm 3: Procedures used in Algorithm 2, Opt-Track Algorithm (Code at site s_i)

PURGE:
1 **for** *all* $l_{z,t_z} \in LOG_i$ **do**
2 **if** $l_{z,t_z}.Dests = \varnothing \wedge (\exists l'_{z,t'_z} \in LOG_i | t_z < t'_z)$ **then**
3 remove l_{z,t_z} from LOG_i;

MERGE(LOG_i, L_w):
4 **for** *all* $o_{z,t} \in L_w$ *and* $l_{s,t'} \in LOG_i$ *such that* $s = z$ **do**
5 **if** $t < t' \wedge l_{s,t} \notin LOG_i$ **then** mark $o_{z,t}$ for deletion;
6 **if** $t' < t \wedge o_{z,t'} \notin L_w$ **then** mark $l_{s,t'}$ for deletion;
7 delete marked entries;
8 **if** $t = t'$ **then**
9 $l_{s,t'}.Dests := l_{s,t'}.Dests \cap o_{z,t}.Dests$;
10 delete $o_{z,t}$ from L_w;
11 $LOG_i := LOG_i \cup L_w$;

So, when checking for the activation predicate at line (24)-(25) in Algorithm 2, all piggybacked write operations need to be checked. With these additional properties in the full replication scenario, we can represent each individual write operation using only a 2-tuple $\langle i, clock_i\rangle$, where i is the site id and $clock_i$ is the local write operation counter at site s_i when the write operation is issued. In this way, we bring the cost of representing a write operation from potentially $O(n)$ down to $O(1)$. This improves the algorithm's scalability when the shared memory is fully replicated.

In fact, Algorithm 2's scalability can be further improved in the fully replicated scenario. In the partially replicated case, keeping entries with empty destination list as long as they represent the most recent applied updates from some site is important, as it ensures the optimality that no redundant destination information is transmitted. However, this will also require each site to almost always maintain a total of n entries. In the fully replicated case, we can also decrease this cost. We observe that, once a site s_3 issues a write operation $w'(x_2)u$, it no longer needs to remember any previous write operations, such as $w(x_1)v$, stored in the local log. This is because all the write operations stored in the local log share the same destination list as w'. Thus, by making sure the most recent write operation is applied in causal order, all the previous write operations sent to all sites are guaranteed to be also applied in causal order. Similarly, after the activation predicate becomes true and the write operation w' is applied at site s_1, only w' itself needs to be remembered in $LastWriteOn_1\langle 2\rangle$. This is illustrated in Fig 3.

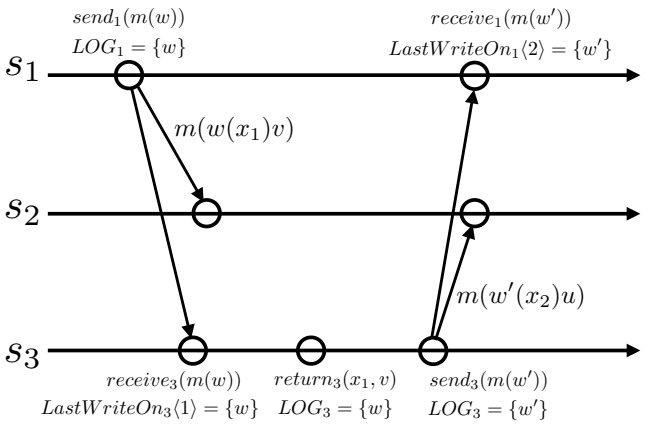

Figure 3. In fully replicated systems, the local log will be reset after each write operation. Also, when a write operation is applied, only the write operation itself needs to be remembered. For clarity, the *apply* events are omitted in this figure.

This way of maintaining local logs essentially means that each site s_i now only needs to maintain $d+1$ entries at any time with each entry incurring only an $O(1)$ cost. Here, d is the number of read operations performed locally since the most recent local write operation. This is because the local log always gets reset after each write operation, and each read operation will add at most 1 new entry into the local log. Furthermore, if some of these read operations read variables that are updated by the same application process, only the entry associated with the very last read operation needs to be maintained in the local log. Thus, the number of entries to be maintained in the full replication scenario will be at most n.

Furthermore, if the application running on top of the shared memory system is write-intensive, then the local log will be reset at the frequency of write operations issued at each site. This means, each site simply cannot perform enough read operations to build up the local log to reach a number of n entries. Even if the application is read-intensive, this is still the case because read-intensive applications usually only have a limited subset of all the sites to perform write operations. Thus, in practice, the number of entries that need to be maintained in the full replication scenario is much less than n.

With the above discussion, we give the formal algorithm of a special case of Algorithm 2, optimized for the fully replicated shared memories. The algorithm is listed in Algorithm 4. Each site still maintains the same data structures as in Algorithm 2, the only difference lies in that there is no need to maintain the destination list for each write operation in the local log, and hence the format of the log entries becomes the 2-tuple $\langle i, clock_i \rangle$. Algorithm 4 assumes a highly simplified form. However, it is very systematically derived by adapting Algorithm 2 to the fully replicated case. Algorithm 4 is significantly better than the algorithm in [5] in multiple respects, as we will show in Section IV.

Algorithm 4: Opt-Track-CRP Algorithm (Code at site s_i)

WRITE(x_h, v):
1 $clock_i ++$;
2 send $m(x_h, v, i, clock_i, LOG_i)$ to all sites other than s_i;
3 $LOG_i := \{\langle i, clock_i \rangle\}$;
4 $x_h := v$;
5 $Apply_i[i] := clock_i$;
6 $LastWriteOn_i\langle h \rangle := \langle i, clock_i \rangle$;

READ(x_h):
7 MERGE$(LOG_i, LastWriteOn_i\langle h \rangle)$;
8 return x_h;

On receiving $m(x_h, v, j, clock_j, L_w)$ from site s_j:
9 **for** *all* $o_{z,clock_z} \in L_w$ **do**
10 wait until $clock_z \leq Apply_i[z]$
11 $x_h := v$;
12 $Apply_i[j] := clock_j$;
13 $LastWriteOn_i\langle h \rangle := \langle j, clock_j \rangle$;

MERGE$(LOG_i, \langle j, clock_j \rangle)$:
14 **for** *all* $l_{s,t} \in LOG_i$ *such that* $s = j$ **do**
15 **if** $t < clock_j$ **then** delete $l_{s,t}$ from LOG_i;
16 $LOG_i := LOG_i \cup \{\langle j, clock_j \rangle\}$;

IV. Complexity

Four metrics are used in the complexity analysis:

- message count: count of the total number of messages generated by the algorithm.
- message size: the total size of all the messages generated by the algorithm. It can be formalized as $\sum_i$*(# type i messages * size of type i messages)*.
- time complexity: the time complexity at each site s_i for performing the write and read operations.
- space complexity: the space complexity at each site s_i for storing local logs and the $LastWriteOn$ log.

The following parameters are used in the analysis:

- n: the number of sites in the system
- q: the number of variables in the shared memory system
- p: the replication factor, i.e., the number of sites at which each variable is replicated
- w: the number of write operations performed in the shared memory system
- r: the number of read operations performed in the shared memory system
- d: the number of write operations stored in local log (used only in the analysis of Opt-Track-CRP algorithm)

Table I summarizes the results. A detailed complexity analysis is given in [28].

In the KS algorithm, although the upper bound on the size of the log and the message overhead is $O(n^2)$ [18], Chandra et al. [9], [10] showed through extensive simulations that the amortized log size and message overhead size is $O(n)$. This is because the optimality conditions implemented ensure that only necessary destination information is kept in the log. This also applies to the Opt-Track algorithm because the same optimization techniques are used. Therefore, although the total message size complexity of the Opt-Track algorithm is $O(n^2pw + nr(n-p))$, this is only the asymptotic upper bound. The amortized message size complexity of the Opt-Track algorithm is $O(npw + r(n-p))$. Similarly, although the space complexity of the Opt-Track algorithm is $O(npq)$, this is only the asymptotic upper bound. The amortized space complexity will be $O(pq)$ [9], [10].

V. Discussion

Compared with the existing causal memory algorithms, our suite of algorithms has advantages in several aspects. Similar to the complete replication and propagation causal memory algorithm, *OptP*, proposed by Baldoni et al., our algorithm also adopts the optimal activation predicate. How-

Table I
COMPLEXITY MEASURES OF CAUSAL MEMORY ALGORITHMS.

Metric	Full-Track	Opt-Track	Opt-Track-CRP	*OptP* [5]
Message count	$pw + 2r\frac{(n-p)}{n}$	$pw + 2r\frac{(n-p)}{n}$	nw	nw
Message size	$O(n^2pw + nr(n-p))$	$O(n^2pw + nr(n-p))$ amortized $O(npw + r(n-p))$	$O(nwd)$	$O(n^2w)$
Time Complexity	write $O(n^2)$ read $O(n^2)$	write $O(n^2p)$ read $O(n^2)$	write $O(n)$ read $O(1)$	write $O(n)$ read $O(n)$
Space Complexity	$O(npq)$	$O(npq)$ amortized $O(pq)$	$O(max(n,q))$	$O(nq)$

ever, compared with the Opt-Track-CRP algorithm, the *OptP* algorithm incurs a higher cost in the message size complexity, the time complexity for read and write operations, and the space complexity. This is because the *OptP* algorithm requires each site to maintain a $Write$ clock of size n, and does not take advantage of the optimization techniques in the KS algorithm.

Compared with other causal consistency algorithms [2], [3], [4], [6], [13], [14], [20], [22], [23], [24], [25], our algorithms have the additional ability to implement causal consistency in partially replicated distributed shared memory systems. Further, the algorithms in [6], [24], [25] do not provide scalability as they use a form of log serialization and exchange to implement causal consistency.

The benefit of doing partial replication compared with full replication lies in multiple aspects. First, this reduces the number of messages sent with each write operation. Although the read operation may incur additional messages, the overall number of messages can still be lower than the case of full replication if the replication factor is low and readers tend to read variables from the local replica instead of remote ones. Hadoop HDFS and MapReduce is one such example. The HDFS framework usually chooses a small constant number as the replication factor even when the size of the cluster is large. Furthermore, the MapReduce framework tries its best to satisfy data locality, i.e., assigning tasks that read only from the local machine. In such a case, partial replication generates much less messages than full replication.

Of the four metrics in Table I, message count is the most important. As the formulas indicate, partial replication gives a lower message count than full replication if

$$pw + 2r\frac{(n-p)}{n} < nw \Longrightarrow w > 2\frac{r}{n}.$$

This is equivalently stated as: partial replication has a lower message count if the write rate (defined as $w_{rate} = \frac{w}{w+r}$) is such that $w_{rate} > \frac{2}{2+n}$.

Fig 4 plots message count as a function of w_{rate} for $n = 10$ and various replication factors (p). The $p = 10$ plot corresponds to the full replication case. Even for this low n, partial replication has lower message count for $w_{rate} > \frac{2}{12}$.

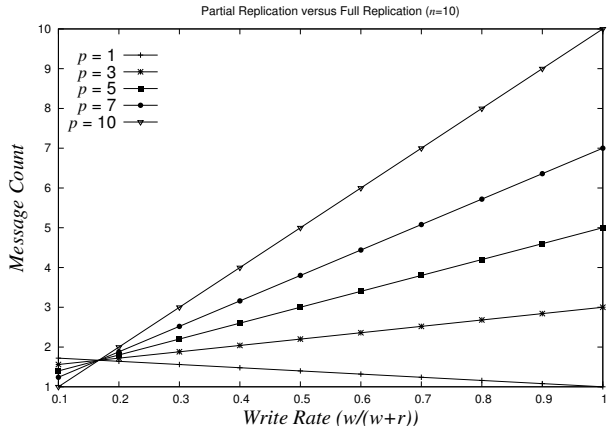

Figure 4. The graph illustrates message count for partial replication vs. full replication, by plotting message count as a function of w_{rate}.

Partial replication can also help to reduce the total size of messages transmitted within the system. Although the two partial replication causal memory algorithms proposed might have a higher message size complexity compared with their counterparts for full replication, this complexity measurement is only for the control messages and does not take into consideration the size of the data that is actually being replicated. In modern social networks, multimedia files like images and videos are frequently shared. The size of these files is much larger than the control information piggybacked with them. Doing full replication might improve the latency for accessing these files from different locations, however it also incurs a large overhead on the underlying system for transmitting and storing these files across different sites.

Further, in the scenario depicted in Section I, where most accesses to a user's file are located within certain geographical regions, or the workload is write-intensive, the improvement in the latency brought by full replication is less significant compared to the cost it imposes on the underlying system.

Some of the algorithms for fully replicated data stores provide causal+ consistency, or *convergent consistency* [2], [3], [4], [22], [23]: here, once updates cease, all processes will eventually read the same value (or set of values) of each

variable. This provides liveness guarantees. We can provide causal+ consistency for our partially replicated system as follows: periodically, run a global termination detection algorithm [19]; once termination is detected, determine the final set of values of each variable, and use that set to provide convergent causal consistency.

In our algorithms for partially replicated systems, a read may be non-local. This can affect availability if the process read-from is down. If a non-local read does not respond in a timeout period, then a secondary process is contacted. This provides better availability in light of the CAP Theorem.

VI. Conclusion

We considered the problem of providing causal consistency in large-scale geo-replicated storage under the assumption of partial replication. This is the first such work that explores the causal consistency problem for partially replicated systems and fills in a gap in the literature on causal consistency in shared memory systems. We proposed two algorithms to solve the problem. The first algorithm is optimal in the sense that that each update is applied at the earliest instant while removing false causality in the system. The second algorithm is additionally optimal in the sense that it minimizes the size of meta-information carried on messages and stored in local logs. We discussed the conditions under which partial replication can provide less overhead (transmission and storage) than the full replication case. In addition, as a derivative of the second algorithm, we proposed an optimized algorithm that reduces the message overhead, the processing time, and the local storage cost at each site in the fully replicated scenario. This algorithm is better than the Baldoni et al. algorithm.

References

[1] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: definitions, implementation and programming. *Distributed Computing, 9, 1*, pages 37–49, 1995.

[2] S. Almeida, J. Leitao, and L. Rodrigues. ChainReaction: a causal+ consistent datastore based on chain replication. *In ACM Eurosys*, pp. 85-98, 2013.

[3] P. Bailis, A. Fekete, A. Ghodsi, J.M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. *In ACM SOCC*, 2012.

[4] P. Bailis, A. Ghodsi, J.M. Hellerstein, and I. Stoica. Bolt-on causal consistency. *ACM SIGMOD*, pp. 761-772, 2014.

[5] R. Baldoni, A. Milani, and S. Piergiovanni. Optimal propagation-based protocols implementing causal memories. *Distributed Computing, 18, 6*, pages 461–474, 2006.

[6] N. Belaramani, M. Dahlin, L. Gao, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. *In NSDI*, 2006.

[7] P. Bernstein and S. Das. Rethinking eventual consistency. *Proc. of the 2013 ACM SIGMOD International Conf. on Management of Data*, 2013.

[8] K. Birman. A response to Cheriton and Skeen's criticism of causally and totally ordered communication. *In ACM SIGOPS Operating Systems Review*, 28(1): 11-21, 1994.

[9] P. Chandra, P. Gambhire, and A.D. Kshemkalyani. Performance of the optimal causal multicast algorithm: A statistical analysis. *IEEE Transactions on Parallel and Distributed Systems, 15(1)*, pages 40–52, January 2004.

[10] P. Chandra and A.D. Kshemkalyani. Causal multicast in mobile networks. *Proc. of the 12th IEEE/ACM Symposium on Modelling, Analysis, and Simulation of Computer and Communication Systems*, pages 213–220, 2004.

[11] D.R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *In ACM SOSP*, 1993.

[12] 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. *Proc. of the 19th ACM SOSP*, pages 205–220, 2007.

[13] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable causal Consistency using dependency matrices and physical clocks. *In ACM SOCC*, 2013.

[14] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. GentleRain: Cheap and scalable causal consistency with physical clocks. *In ACM SOCC*, 2014.

[15] P. Gambhire and A.D. Kshemkalyani. Reducing false causality in causal message ordering. *Proc. 7th International High Performance Computing Conference (HiPC), LNCS 1970, Springer,* pp 61-72, 2000.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002.

[17] A.D. Kshemkalyani and M. Singhal. An optimal algorithm for generalized causal message ordering. *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, page 87, 1996.

[18] A.D. Kshemkalyani and M. Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing, 11, 2*, pages 91–111, 1998.

[19] A.D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2008.

[20] K. Lady, M. Kim, and B. Noble. Declared causality in wide-area replicated storage. *In Workshop on Planetary-Scale Distributed Systems*, 2014.

[21] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM, 21*, pages 558-564, 1978.

[22] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. *Proc. of the 23rd ACM SOSP*, pages 401–416, 2011.

[23] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Stronger semantics for low latency geo-replicated storage. *In NSDI*, 2013.

[24] P. Mahajan, L. Alvisi, and M. Dahlin. Consistency, availability, and convergence. *Tech. Rep. TR-11-22, Univ. Texas at Austin, Dept. Comp. Sci.*, 2011.

[25] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. *In SOSP*, 1997.

[26] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters, 39, 6*, pages 343–350, 1991.

[27] M. Shen, A.D. Kshemkalyani, and T. Hsu. OPCAM: Optimal algorithms implementing causal memories in shared memory systems. *In ICDCN*, Jan 2015.

[28] M. Shen, A.D. Kshemkalyani, and T. Hsu. OPCAM: Optimal algorithms implementing causal memories in shared memory systems. *Technical Report, Univ. Illinois at Chicago, Dept. of Computer Science*, 2014.